Test Generation Using Event Sequence Graphs

Zur Erlangung des akademischen Grades

DOKTORINGENIEUR
(Dr.-Ing.)

der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn
vorgelegte Dissertation
von

Dipl.-Ing. **Christof J. Budnik**
aus Bielefeld

Referent:	Prof. Dr.-Ing. Fevzi Belli
Korreferent:	Prof. Dr.-Ing. Ina Schieferdecker
Tag der mündlichen Prüfung:	21. Dezember 2006

Paderborn, den 29.01.2007

Diss. 14/229

To my late grandfather
Johannes

Acknowledgement

I would like to thank my parents for their steady morally support and encouragement that enabled me to start and conclude my Ph.D. studies.

I also thank my supervisors, Professor Fevzi Belli for inspiring me into software testing and research, and Prof. Ina Schieferdecker for her valuable advices.

My thanks go also to Prof. Lee White (University of Michigan), Prof. Nimal Nissanke (London South Bank University), and Prof. Aditya P. Mathur (Purdue University) for fruitful discussions and comments that helped to shape this thesis.

Declaration

The work presented in this thesis has been drawn from research undertaken between January 2002 and October 2006 at the Department of Electrical Engineering, University of Paderborn.

Much of the work has been published elsewhere as follows:

- Belli, F., Budnik, C.J., Nissanke, N.: Modeling, Analysis and Testing of System Vulnerabilities. *Work-in-Progress Papers of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, 2003, p. 11.
- Belli, F., Budnik, C.J., Nissanke, N: Finite-State Modeling, Analysis and Testing of System Vulnerabilities. *Proceedings of the 17th International Conference on Architecture of Computing Systems (ARCS)*, Lecture Notes in Informatics (LNI), vol. 41, 2004, pp. 19-33.
- Belli, F., Budnik, C.J.: Minimal Spanning Set for Coverage Testing of Interactive Systems. *Proceedings of the first International Colloquium on Theoretical Aspects and Computing (ICTAC)*, Lecture Notes of Computer Science (LNCS), vol. 3407, Springer, 2004, pp. 220-234.
- Belli, F., Budnik, C.J.: Towards Optimization of the Coverage Testing of Interactive Systems., *Proceedings of the 28th International Computer Software and Applications Conference (COMPSAC)*, IEEE Computer Science, 2004, pp. 18-19.
- Belli, F., Budnik, C.J.: Towards Minimization of Test Sets for Coverage Testing of Interactive Systems. *Proceedings of the Conference on Software Engineering (SE)*, Lecture Notes in Informatics (LNI), vol. 64, 2005, pp. 79-90.
- Belli, F., Budnik, C.J.: Test Cost Reduction for Interactive Systems. *Proceedings of the Conference on Sicherheit*, Lecture Notes in Informatics (LNI), vol. 62, 2005, pp. 149-160.
- Belli, F., Budnik, C.J.: Towards Minimization of Test Sets for Human-Computer Systems. *Proceedings of the18th International Conference on Industrial & Engineering Applica-*

tions of Artificial Intelligence & Expert Systems (IEA/AIE), Lecture Notes in Computer Science (LNCS), vol. 3533, Springer, 2005, pp. 300-309.

- Belli, F., Budnik, C.J.: Towards Self-Testing of Component-Based Software. *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, IEEE Computer Society, vol. 2, 2005, pp. 205-210.
- Belli, F., Budnik, C.J., Hollmann, A.: A Holistic Approach to Testing of Interactive Systems using Statecharts. *Proceedings of the 2nd South-East European Workshop on Formal Methods (SEEFM)*, South-Eastern European Research Center SEERC, 2005.
- Belli, F., Budnik, C.J., Hollmann, A.: Holistic Testing of Interactive Systems Using Statecharts. *Journal of Mathematics, Computing & Teleinformatics (AMCT)*, vol. 1(3), 2005, pp. 54-64.
- Belli, F., Budnik, C.J., White, L.: Event-based Modeling, Analysis and Testing of User Interactions: Approach and Case Study. *Journal of Software Testing, Verification and Reliability (STVR)*, vol. 16(1), John Wiley & Sons, Ltd, 2006, pp. 3-32.
- Belli, F., Budnik, C.J., Hollmann, A.: Holistic Testing of Interactive Systems Using Statecharts. *Proceedings of the Conference on Sicherheit*, Lecture Notes in Informatics (LNI), vol. 77, 2006, pp. 345-356.
- Hollmann, A., Belli, F., Budnik, C.J.: Test Case Generation and Selection Based on Statecharts. *Student Program Paper of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, to appear, 2006.
- Belli, F., Budnik, C.J.: Test Minimization for Human-Computer Interaction. *International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies*, vol. 26(2), Springer, 2007.

I declare that the work in this thesis is original work I undertook between the dates of registration for the Degree of Doctor of Electrical Engineering at the University of Paderborn.

Table of Contents

Symbols and Notation

Set Theory

$\{x_1, x_2, \ldots, x_n\}$	Set of elements $x_1, x_2, \ldots, x_n$
$\varnothing$	The empty set
$x \in M$	x is an element of M
$M \subseteq N$	M is a subset of N
$M \cup N$	Union of sets M and N
$M \setminus N$	Difference of sets M and N
$M \times N$	Cartesian product of sets M and N
$[\![M]\!]$	A bag M as a set of elements where each element has an occurrence number
$M \uplus N$	Union of bags M and N

Basic Sets

D	Set of system dysfunctions
E	A finite set of edges
F	Set of system functions
$N^-(v)$	Set of predecessors of vertex v
$N^+(v)$	Set of successors of vertex v
R	The regular Set denoted by the regular expression r
V	A finite set of labeled vertices
V_{env}	Set of environmental events
V_{sys}	Set of system response
$\Xi(ESG)$	Set of entry nodes of an given ESG
$\Gamma(ESG)$	Set of exit nodes of an given ESG

Miscellaneous

diff(*v*)	The function denotes the number of predecessor events of *v* minus the number of its successor events
DM	Defense Matrix
$\overline{\text{ESG}}$	Inversion of an ESG
$\widehat{\text{ESG}}$	Completion of an ESG
ES_i	Event Sequences of length *i*
l(ES)	Function which determine the number of vertices of an given event sequence
l_cov	Sum of length of ESs up to a given length
l_max	Maximum length of an ES to be covered
α(ES)	Function that determine the initial vertex of an given event sequence
β	Weight factor
ω(ES)	Function that determine the end vertex of an given event sequence
ψ	Test costs
$\langle x_1, x_2, \ldots, x_m \rangle$	Finite sequence $x_1, x_2, \ldots, x_m$
$\oplus$	concatenation of event sequences
$\sqsubseteq$	Risk ordering relation

Abbreviations

CES	Complete Event Sequence
CESG	Completed ESG
CPP	Chinese Postman Problem
CTS	Complete Transition Sequence
EP	Event Pair
ES	Event Sequence
ESG	Event Sequence Graph
FCES	Faulty Complete Event Sequence
FCTS	Faulty Complete Transition Sequence
FEP	Faulty Event Pair
FES	Faulty Event Sequence
FSA	Finite State Automata
FTP	Faulty Transition Pair
FTS	Faulty Transition Sequence
GATE	Generation and Analysis of Test Event Sequences

GUI	Graphical User Interface
MSCES	Minimal Spanning Complete Event Sequence
MSFCES	Minimal Spanning Faulty Complete Event Sequence
SUT	System under Test
TP	Transition Pair
TS	Transition Sequence
UI	User Interface
UML	Unified Modeling Language
XML	Extended Markup Language

1 Introduction

This chapter describes the motivation, context and intended contribution of this thesis. Starting point is the existing work and terminology used; the goals are identified and the software quality features considered are outlined. A holistic view is introduced to generate and select adequate test cases from a behavior model of the system under consideration.

1.1 Software and its Features

In terms of behavioral patterns, the relationships between the system and its environment, i.e., the user, the natural environment, etc., can be described as proactive, reactive or interactive. In the case of *proactivity*, the system generates the stimuli, which evoke the activity of its environment. In the case of *reactivity*, the system behavior evolves through its responses to stimuli generated by its environment. Most systems are nowadays *interactive* in the sense that the user and the system itself can be both pro- and reactive.

When observing an interactive system, depending on the expectations of the user concerning the system behavior, a distinction is to be made between *desirable* and *undesirable* situations or events. The sum of desirable and undesirable events defines the critical system properties that are global quality factors, e.g., *reliability, correctness*, *safety*, *robustness*, *user-friendliness*, etc. Any deviation from the desirable behavior amounts to an undesirable situation; the fact that the system can be unintentionally transferred into such a state might be viewed as a *vulnerability* of the system. The consideration of vulnerabilities necessitates a clear understanding of the notion of undesirability.

A *failure* is a manifestation of a given form of vulnerability and, as a result, affects the ability of the system to perform its functions [IEEE90]. During the execution of the system, a failure can be realized, or triggered, by an *error* as an incorrect and thus, an undesirable, system state. An error can eventually be traced back to a human action, or inaction, leading to an incorrect or undesirable result and, thus, to a *fault*. In other words, a chain of incidences "*fault* → *error* → *failure*" emerges, where $A \rightarrow B$ means that A causes, or leads to, B [ALR04, Lap92]. The notion of "undesirability" is the key characteristic applicable to the terms fault, error and failure. In this context, the terms "fault" and "error" are often used synonymously as the cause of a "failure".

Achieving given software quality is an investigation for the existence of faults. In the following the undesirable behavior is discussed for the different quality aspects (see also [IEEE90]).

Correctness refers to the extent to which the system behavior corresponds to the requirements. A system is said to be correct if its behavior matches the intended, desirable behavior for all input data.

Reliability focuses on the actual use of the system over time. A system may have faults but still be highly reliable if the faults (as undesirable events) appear only on inputs that never occur in actual use.

In the case of *safety*, the vulnerability originates from within the system due to potential failures and its spill-over effects causing potentially extensive damage to its environment. In the face of such failures, the environment could be a helpless, passive victim. The goal of the system design in this case is to prevent faults that could potentially lead to such failures or, in worst cases, to mitigate the consequences at run-time should such failures ever occur.

In case of *robustness*, the system performs well not only under ordinary conditions but also under unusual conditions that stress its designers' assumptions. Interactive systems are usually too big and too complicated for a single human mind to comprehend in their entirety, and thus it is difficult for their developers to be able to discover and eliminate all the errors, or to even be certain as to what extent of errors exist. This is especially true with regard to subtle errors that only make their presence known in unusual circumstances. In this case the system should not lock up the computer, cause damage to data or send the user through an endless chain of dialog boxes without purpose.

In contrast to safety and robustness the lack of *user friendliness* of an interactive system is a milder form of system vulnerability. A failure to be user-friendly is still a failure to fulfill a system attribute, though it may typically cause an annoyance or an irritation to the user, possibly leading to confusion and complaints. However, disastrous consequences cannot be entirely ruled out under stressed conditions of the user if the interactive system is intended to mediate between a safety critical application and its operator.

1.2 Software Testing

Software quality is the dominant success criterion in software industry. As software applications are becoming more sophisticated, complex and expensive, it is obvious that software quality is growing.

Activities associated with ensuring correct and safe operation of software are known as verification and validation (V&V). *Verification* implies that *software under consideration* is checked during its development concerning design requirements, asking the question: "Does the implementer work correct?" [BBL76]. *Validation* ensures that the delivered system satisfies the customer requirements, asking the question: "Does the system work correct?" Both Verification as Validation of the requirements as early as possible avoids unnecessary and costly rework.

Validation activities usually require a behavioral model. This model is typically documented in a set of requirements specification documents that specify functional and non-functional behavior of the software under consideration. Functional requirements describe the actions of the system in terms of the operations that should take place in response to a series of external events. Non-functional requirements place constraints on this behavior, such as memory limitations, real-time deadlines and safety properties.

Testing is the traditional and still most common validation method in the software industry to achieve this goal. It is usually carried out applying *test cases* to the *system under test* (SUT). While constructing test cases, one generally has to produce meaningful test inputs and then to determine the expected system outputs for these inputs. A *test oracle* is a means, automated or manual, for checking the correctness of the outputs computed by a SUT. Hence, software testing requires the existence of such an oracle, i.e., an external source of correct information about the expected behavior and output of the software. Because testing requires the execution of the software, it is classified as a dynamic analysis technique.

The purpose of testing is to assure the system quality and along with this to reveal faults. However, the testing itself will rather identify the fault not the error. In fact the faults are to be fixed. For this purpose, an erroneous situation is to be developed that causes a failure and thus cannot be hidden from the environment, e.g., the user of the system. This thesis primarily applies testing to the validation problem.

1.3 Testing Interactive Systems

An interface that enables information to be passed between a human user and hardware or software components of a computer system is defined as a *user interface* (UI) [IEEE90].

With the growing complexity of interactive software systems, also their (UI), mostly realized graphically (*graphical user interfaces*, GUI), become more complex. Accordingly, the test and analysis process becomes more and more tedious and costly. In this particular kind of system, the user interface can have an environment, that is, the plant, the device or the equipment, which the UI is intended for and embedded in, controlling technical processes.

While developing interactive systems, construction of the user interactions deserves special care, and should be handled separately because it requires different skills, and maybe different techniques than construction of common software. This fact has been recognized very early, defining a *User Interaction Management System* that can be independent of the application, graphics package, etc. [TH83]. The design part of the UI development needs a good understanding of the user and his/her needs, while the implementation part requires familiarity with the technical equipment, e.g., programming platform, language, etc. [Shn98]. Testing requires both: a good understanding of user requirements, and familiarity with the technical equipment.

Testing of UIs is an important step in the development of interactive systems as it checks the compliance of the system with the user requirements. Thereby, the UI is considered as a set of system functions. *Black-box* testing of the UI means testing of the behavior of the system and thus, its system functions. Accordingly, to generate test cases for a UI, one has to first identify the test objects and test objectives. The test objects are the instruments for the input, e.g., windows, icons, menus, pointers, commands, function keys, alphanumerical keys, etc. The objective of a test is to generate the expected system behavior (i.e., desirable event) as an output. In a broader sense, the test object is the SUT; the objective of the test is to gain confidence in the SUT.

In order to meet the market demands and resource requirements, testing should be *automated* to become more effective and be more efficient (i.e., faster, cheaper), considering also deployment of *test tools*. For industrial black-box testing of UIs, most of the conventional tools include capture/replay features for recording and analyzing of test scenarios. In spite of the maturity and quality of these tools, a lot of manual effort is needed that is costly and error-prone.

1.4 Objectives, and Novelty of the Work

Software testing is a widely used method in practice for quality assurance. But the required test inputs are still not systematically generated in an efficient manner, and testing becomes an uncontrollable process and therefore unusable. The reasons are missing models from the design or a present informal specification so that adequate test inputs cannot be derived systematically. This drawback can be solved by the present approach with the help of a graph model called *Event Sequence Graph* (ESG) which has been introduced by F. Belli [Bel01]. An ESG is a simple albeit powerful formalism for capturing the behavior of a variety of interactive systems that include embedded systems and graphical user interfaces. A collection of ESGs is proposed as a model of an interactive system.

Based on ESG notion, this thesis introduces a *holistic* view of fault modeling that requires an additional, complementary step to system modeling. Thus, both the desirable and undesirable behavior of the system is specified at the same level of design granularity. In other words, intended and unintended usage scenarios will be generated to check both that system behavior is in according to the user expectations and faults are handled properly.

The 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. This is achieved by an ordering of system states according to the risks posed to that attribute and by selecting tests that address specific faults. 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 followed a faulty event pair when testing for any undesirable outcome.

The novelty of the approach is the modeling, analysis and testing of system behaviors, with respect to both correct and faulty behaviors, based on ESGs. The approach focuses on testing a system for correctness with respect to its behavioral requirements and for its robustness against incorrect usage in terms of user inputs, and analyzing the consequences under possible malfunction of the environment. It is based on two simple ideas. First, the desirable behavior of the user and the 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 analysis.

In brief, a model-based black-box test methodology for an efficient test case selection is introduced for testing interactive systems, mostly represented by their user interfaces. The main contributions of this work are:

(i) an ESG based formalism for the modeling and analysis of the behavior of discrete, event-based, sequential systems;

(ii) a test generation algorithm that takes an ESG as input and generates adequate tests for testing the behavior of a SUT under expected and unexpected conditions, and

(iii) an evaluation of the feasibility and effectiveness of the proposed modeling and test generation schemes using two case studies.

(iv) Tools that are necessary for an effective deployment of the approach are introduced and discussed.

(v) Lessons learned from intensive and extensive application of the approach to industrial projects are summarized.

1.5 Outline

The remainder of this thesis is organized as follows. The next chapter 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.

Chapter 3 is a rigorous introduction to Event Sequence Graph (ESG) which is a simple albeit powerful formalism for capturing the behavior of interactive systems that include embedded systems and graphical user interfaces.

In Chapter 4 the test process is described for the generation of tests from ESGs to check for the correctness of system behavior in the presence of expected and unexpected input event sequences. The 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.

Chapter 5 reports a case study that investigates the fault detection effectiveness and the cost of test generation.

An example in Chapter 6 demonstrates the analytical power of ESG based modeling and risk analysis.

An extending of the approach to statecharts is given in Chapter 7. Applying the modeling of functions and malfunctions leads to additional rules for the test case generation which are evaluated in a second case study.

Chapter 8 provides an overview of a toolkit developed to support ESG based modeling and test generation.

A discussion related to the current work and directions for further research appear in Chapter 9.

2 Related Work

This chapter reviews the existing work in related areas, i.e., on modeling, specification and test generation. Comparisons with the approach developed in the thesis are supposed to show its originality.

2.1 Testing Techniques

Testing is the execution of a program with the aim of causing failures and thus reveals faults. There is no justification, however, for any assessment on the correctness of the system under test based on the success or failure of a single test, because there can potentially be an infinite number of test cases, even for very simple programs.

To overcome this shortcoming of testing, formal methods have been proposed, which introduce models that represent the relevant features of the SUT. The modeled, relevant features are either functional behavior or the structural issues of the SUT, leading to *specification-oriented* testing or *implementation-oriented* testing, respectively. Once the model is established, it “guides” the test process to generate and select test cases, which form sets of test cases (also called *test suites*) [Bei90, Bin00]. The selection is guided by an *adequacy criterion,* which provides a measure of how effective a given set of test cases is in terms of its potential to reveal faults. Most of the existing adequacy criteria are *coverage-oriented* [ZHM97]. The ratio of the portion of the specification or code that is covered by the given test set in relation to the uncovered portion can then be used as a decisive factor in determining the point in time at which to stop testing (*test termina-*

tion) [Ham96]. Another problem that arises is the determination of the test outcomes (*oracle problem*) [Bin00, Ham96].

In [Bel01] a fault model is introduced this thesis is based on, which consider not only the desirable situations, but also the undesirable ones. A similar fault model is also used in the mutation analysis and testing approach which systematically and stepwise modifies the SUT using *mutation operations* [DLS78]. This approach has been well understood, is widely used and, thus, has become quite popular. Although originally applied to implementation-oriented unit testing, mutation operations have also been extended to be deployed at more abstract, higher levels, e.g., integration testing, state-based testing, etc. [DMM01]. Once applied to the graphical representation of a SUT, mutation operations can be viewed as elements of the complementing steps of the event sequence graphs, as introduced in Chapter 3. Such operations have also been independently proposed by other authors, e.g., "state control faults" for fault modeling by Bochmann and Petrenko [BP94], or for "transition-pair coverage criterion" and "complete sequence criterion" by Offutt et.al. [OLAA03]. However, the latter two notions have been precisely introduced in [Bel01] and [WA00], respectively, prior to the work of Offutt *et al.*, where they also appeared.

Several approaches have been proposed to assess the robustness of software-based systems. Fetzer and Xiao [FX02] have proposed techniques for increasing the robustness of C libraries using wrappers. Huhns and Holderfield [HH02] equate robustness to the resilience of software failures 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 [KKS98]. The proposed ESG approach differs from the approaches cited in that it allows modeling on incorrect behavior, which is often the cause of software systems' lack of robustness and provides an algorithmic approach to the test generation for testing the software-based system for robustness.

2.2 Finite-State-Based Test Generation

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 [Cho78]. There are, however, significant differences between the goals and assumptions of the W-method and the approach presented in this work. First, conformance testing generates tests to detect faults clearly defined by some hypotheses, e.g. transition errors, missing, or extra states. The approach in this thesis generates test cases to *cover* the ESG. Second, the W-method assumes two models are available, i.e. the system model and the design model. These two models are then compared. The first model is required to be correct; the second one is to be checked against the correct model. The approach in this work supposes the existence of only one model that describes the correct behavior of the system. This model is exploited to generate tests to check the real, implemented SUT, i.e. the SUT is tested against its specification. Third, conformance testing assumes the system under consideration behaves deterministically and is completely specified by a finite-state automaton. These assumptions are not necessarily fulfilled by UIs; therefore, they are not made by the approach presented in this work.

Chow's work was followed with the Wp-method [FBKA91], the Unique Input-Output Sequences method [SD88], Transition Tour method [ADLU91, NT81], Distinguishing Sequences method [Sar89], and several empirical evaluations [BP94, SS97]. Finite-state models have also been proposed as a means for the specification and analysis of system behavior [Par69, Sha80]. More recently, researchers have proposed ways to generate tests from a variety of UML specifications such as statecharts and sequence diagrams [FMMD94, OLAA03].

Most finite-state based test generation methods focus on some form of coverage, e.g., transition coverage [ADLU91, OLAA03, Rob00, SLD92, WA00], and state identification [Cho78, SD88]. The test generation algorithm introduced in this work achieves complete coverage of ESGs [Bel01, MB03] through the use of the Chinese Postman problem [EJ73, Kwa62] for managing round trips. In addition, the algorithm also formalizes and generalizes the notion of pair-wise testing [CDFP97, TL02] by including the ability to generate tests that cover all possible n-tuple of events for some $n>1$. While the number of generated tests can become impractically large for increasing *n*, 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. This is one of the advantages of the approach introduced in this work.

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 [GH99]. 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” [Pel01]. Although this approach considers the appearance of desirable and undesirable events by means of models with not only positive but also negative scenarios, the problem is that these events are constructed intuitively and are therefore neither systematic nor complete.

2.3 Digraphs for Modeling and Test Generation

Digraphs, with variations in terminology, have been used for modeling finite-state behavior since the work of Kleene and Myhill [Kle56, Myh57]. Chhikara et al. use event sequence diagrams to study dynamic probabilistic risk analysis [CH00]. Memon et al. use event flow graphs to model UI event sequences as test cases [MPS01]. Event graphs and timed event graphs are also used in other areas of research such as simulation [Sch95] and automatic control [LL95].

In this work ESGs are used for modeling system behavior, test generation, and robustness testing (see also [GBBD05]), which are based on a finite sequence of events. 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.”

2.4 Test Generation for GUIs

White and Almezen took a different approach to the problem of test generation using finite-state models [WA00, WAA01]. Their work is in the context of generating tests for testing GUIs. Rather than use the traditional Mealy or Moore machines, they propose an alternative representation of user-responsibilities using the idea of *complete interaction sequences*. A complete interaction sequence 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 complete interaction sequences. An advantage of the approach based on complete interaction sequences lies in its scalability and intuitiveness. Instead of creating a single composite finite-state model, multiple complete interaction sequences, each representing a user responsibility, are created thereby simplifying the task of model construction and test generation. This thesis instead introduces a refinement (see also [HI98, Mar97]) to decompose a system hierarchy in multiple ESGs, which in turn can be tested separately.

A different approach for GUI testing has been recently published by Memon *et al.* [MPS00, MPS01], which deploys methods of knowledge engineering to generate test cases, test oracles, etc., and to handle the test termination problem: from a knowledge engineering point of view, the testing of a GUI system represents a typical *planning* problem that can be solved using a goal-driven strategy [MPS01]. Given a set of operators, an initial state and a goal state, the planner is expected to produce a sequence of operators that will change the initial state to the goal state. For the GUI testing problem described above, this means that the test sequences have to be constructed dependent upon both the desirable, correct events and the undesirable, faulty events.

There have also been other GUI testing papers from this research group. Memon *et al.* [MPS01/2] have also written a more general paper on GUI testing that emphasizes coverage criteria involving events and event sequences. They decompose the GUI system into components and emphasize a hierarchy of interacting components in their generated tests. Thus, the components can be tested in isolation. They have provided a case study showing the correlation between this coverage criterion and that of the underlying code coverage. A more recent paper [MBHN03] has shown how a more streamlined GUI testing method can be used for regression testing. Both approaches, i.e. those of Memon *et al.* and of Gargantini and Heitmeyer, use some heuristic methods to cope with the state explosion problem.

Few research work can be found for test automation of interactive systems, especially concerning of their GUIs. The industrial state-of-the-art strategy for automated testing involves ad-hoc deployment of record/playback tools [Azu93, Kep94]. However, more research work has been done on regression testing of GUI to avoid obsolescence of the generated test cases whenever the structure of the GUI is modified [MS03, Whi96]. This work does not try to repair unusable test cases, because the model is maintained and not the test cases. Thus, changes in the system only have to be fixed in the model, where all test cases were generated again.

2.5 Conclusion of the Comparison with Related Work

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. 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 access to automata theory.

2.6 Summary

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

This chapter introduces the event sequence graphs (ESGs) to model both the desirable and undesirable behavior of a system under consideration which interacts with its environment through ordered pairs of environment stimuli/ system response. In this context, the environment can be one or more human users, a set of service seekers, also other technical systems, or any combination thereof. The terms "user" and "environment" is thereafter used interchangeably.

3.1 Formalization

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 [BBW06]. An event, an externally observable phenomenon, can be a user stimulus or a system response, punctuating different stages of the system activity.

Definition 3.1 (Event Sequence Graph): An *event sequence graph* $ESG = (V, E, \varXi, \varGamma)$ is a directed graph with

$V \neq \emptyset$: a finite set of labeled vertices (nodes),

$E \subseteq V \times V$: a finite set of edges (arcs),

$\varXi, \varGamma \subseteq V$: finite sets of distinguished vertices $\xi \in \varXi$, and $\gamma \in \varGamma$, called entry nodes and exit nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $\langle \xi, v_0 \ldots, v_k \rangle$ from one $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_{0,} \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to one $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

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

The set *V* is partitioned into two subsets V_{env} and V_{sys} such that

$$V = V_{env} \cup V_{sys},\ V_{env} \cap V_{sys} = \varnothing$$

where V_{env} is the set of *environmental events* (i.e., user inputs) and V_{sys} a set of *system responses*. The distinction between the sets V_{env} and V_{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. Entries are not necessarily to be contained in V_{env} and the exits in V_{sys} even if that holds in most cases. Thus, it is assumed that the system has already been running and that exits do not necessarily mean a system shut-down.

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

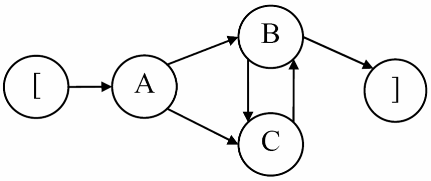


Figure 3.1: An ESG with *A* as entry and *B* as exit and pseudo vertices *[,]*

Example 3.1: For the ESG given in Figure 3.1: $V = \{A,\ B,\ C\}$, $\Xi = \{A\}$, $\Gamma = \{B\}$, and $E = \{(A,C),\ (A,B),\ (B,C),\ (C,B)\}$. Note that arcs from pseudo vertex *[* and to pseudo vertex *]* are not included in *E.*

For a vertex $v \in V$, $N^{+}(v)$ denotes the set of all *successors* of *v*, and $N^{-}(v)$ denotes the set of all *predecessors* of *v*.

Definition 3.2 (Refinement): Given an ESG, say $ESG_1 = (V_1, E_1, \Xi_1, \Gamma_1)$, a vertex $v \in V_1$, and an ESG, say $ESG_2 = (V_2, E_2, \Xi_2, \Gamma_2)$ with $V_1 \cap V_2 = \varnothing$. Then replacing *v* by *ESG₂* produces a *refinement* of *ESG₁*, say $ESG_3 = (V_3, E_3, \Xi_3, \Gamma_3)$ with $V_3 = V_1 \cup V_2 \setminus \{v\}$, and $E_3 = E_1 \cup E_2 \cup E_{pre} \cup E_{post} \setminus E_{1\,replaced}$ (\: set difference operation), wherein $E_{pre} = N^{-}(v) \times \Xi(ESG_2)$ (connections of the predecessors of *v* with the entry nodes of *ESG₂*), $E_{post} = \Gamma(ESG_2) \times N^{+}(v)$ (connections of exit nodes of *ESG₂* with the successors of *v*), and $E_{1\,replaced} = \{(v_i, v),\ (v, v_k)\}$ with $v_i \in N^{-}(v)$ and $v_k \in N^{+}(v)$ (replaced arcs of ESG₁).

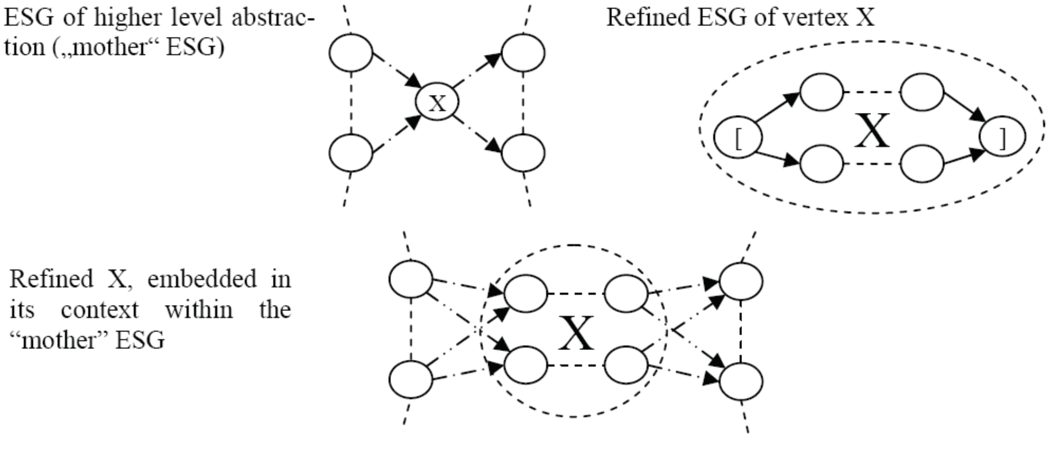

ESG of higher level abstraction („mother" ESG)
X
Refined ESG of vertex X
[
X
]
Refined X, embedded in its context within the "mother" ESG
X

Figure 3.2: Refinement of a vertex *v* and its embedding in the refined ESG

As Figure 3.2 illustrates, *every* predecessor of vertex *v* of the ESG of higher level abstraction points to the entries of the refined ESG. In analogy, *every* exit of the refined ESG points to the successors of *v*. The refinement of *v* in its context within the original ESG of higher level abstraction contains no pseudo vertices *[* and *]* because they are only needed for the identification of entries and exits of the ESG of a refined vertex.

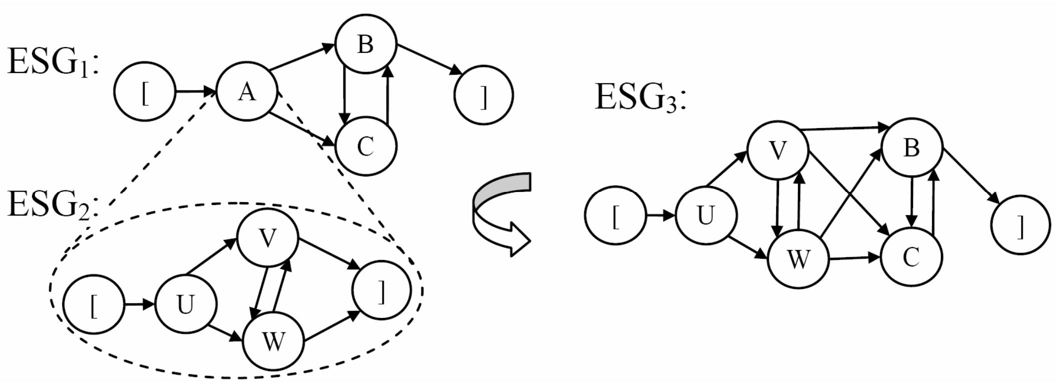


Figure 3.3: A Refinement of the vertex A of the ESG given in Figure 3.1

Example 3.2: In Figure 3.3 the refinement of the vertex A of ESG_1 is given as ESG_2. ESG_3 is the resulting refinement of ESG_1. Note that the pseudo vertices *[* and *]* of ESG_2 are not included in ESG_3.
More precisely, ESG_1 is given as $V_1=\{A, B, C\}$, $E_1=\{(A, B), (A, C), (B, C), (C, B)\}$. In the refinement, i.e., ESG_2 of A, the predecessors and successors are $N^-(v)=\{\}$, $N^+(v)=\{B, C\}$ and the refinement of ESG_2 is given by $V_2=\{U, V, W\}$, $E_2=\{(U, V), (U, W), (W, V), (V, W)\}$, $\Xi(ESG_2)=\{U\}$ and $\Gamma(ESG_2)=\{V, W\}$. The resultant ESG_3 is represented by
$V_3= V_1 \cup V_2 \setminus \{v\}=\{A, B, C\} \cup \{U, V, W\} \setminus \{A\}=\{B, C, U, V, W\}$ and
$E_3= E_1 \cup E_2 \cup E_{pre} \cup E_{post} \setminus E_{1replaced}$
$= \{(A, B), (A, C), (B, C), (C, B)\} \cup \{(U, V), (U, W), (V, W), (W, V)\} \cup \{\} \cup \{(V, B), (V,C), (W, B), (W, C)\} \setminus \{(A, B), (A, C)\}$
$= \{(B, C), (C, B), (U, V), (U, W), (V, W), (W, V), (V, B), (V, C), (W, B), (W, C)\}$

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 Figure 3.1 appears in Figure 3.4.

Definition 3.3 (Inversion): The inverse (or complementary) ESG is then defined as $\overline{ESG}=(V,\overline{E},\Xi,\Gamma)$ with $\overline{E}=V \times V \setminus E$.

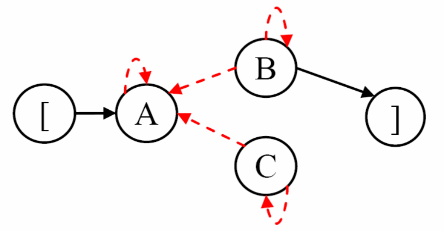


Figure 3.4: $\overline{ESG}$ – Complement of the ESG given in Figure 3.1

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 Figure 3.1 and its complement in Figure 3.4 leads to a CESG shown in Figure 3.5.

Definition 3.4 (Completion): For an $ESG = (V, E, \Xi, \Gamma)$, its *completion* is defined as $\widehat{ESG} = (V, \hat{E}, \Xi, \Gamma)$ with $\hat{E} = E \cup \overline{E}$.

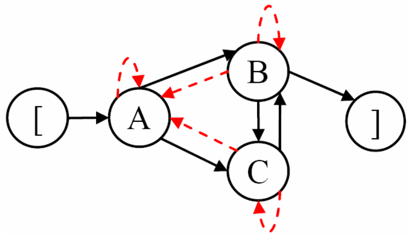


Figure 3.5: CESG – Completed ESG of the ESG given in Figure 3.1

Figure 3.5 illustrates $\widehat{ESG}$, which can systematically be constructed in three steps:

- Add arcs in the opposite direction wherever only one-way arcs exist.
- Add self-loops to vertices wherever none exist.
- Add arcs between vertices wherever no arcs connect them.

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

Interaction patterns implicit in an ESG, $\overline{ESG}$, and *CESG*, can also be expressed in terms of a regular expression [Pra97, Sal69; Sha80]. For example, the following regular expression

$$r = A(A^{+}B+C)^{*}B$$

captures all interaction patterns implicit in the CESG in Figure 3.5. The expression indicates that *A* can either be followed by a pattern of a sequence of one or more events of *A* followed by *B* or by the event *C* and, furthermore, this pattern can recur multiple times. Finally, the expression is terminated by the event *B*. 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*, it is referred 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 *V*. Often, it is a finite set of ESGs that model all interactions of concern with *M*. *R(M)* denotes 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 [Myh57], which are also adopted as computation schemes [Ian60], or as *syntax diagrams*, e.g., as used in [JW74] 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* [Kor96]). A flexible visualization at different abstraction levels is given through *view graphs* [Gos02].

3.2 Modeling Functions and Malfunctions

The term *system function*, or simply *function*, are used 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 *V*.

For a system *M*, event sequences over *V* that belong to *R(M$_e$)* denote system functions, while $\overline{R(M_e)}$ denote malfunctions. Let *F* denote the set of *system functions* and *D* the set of system *malfunctions*, where

F $\subseteq$ *R(M$_e$) and D* $\subseteq \overline{R(M_e)}$.

Let *Σ** be the set of all event sequences over *V*, then *Σ** represents the union of all functions and malfunctions given by $\overline{R(M_e)} \cup R(M_e)$. Thus, the complement of $R(M_e)$ is $\overline{R(M_e)} = \Sigma^* \setminus R(M_e) = \overline{R(M_e)} \cup R(M_e) \setminus R(M_e)$.

To test a system, one generally produces meaningful test *inputs* and the 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 test 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.

A sequence of *n* consecutive edges is an *event sequence (ES) of length n*+1, giving an *(n*+1*)-tuple* of events.

Definition 3.5 (Event Sequence): Let *V*, *E* be defined as in Definition 3.1. Then any sequence of vertices $\langle v_0,\ldots,v_k \rangle$ is called an *event sequence* (*ES*) if $(v_i, v_{i+1}) \in E$, for $i = 0,\ldots,k-1$.

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

Furthermore, α (*initial*) and ω (*end*) are functions to determine the initial vertex and end vertex of an ES, e.g., for *ES*=$\langle v_0,\ldots,v_k \rangle$, initial vertex and end vertex are $\alpha(ES) = v_0$, $\omega(ES) = v_k$, respectively.

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

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

Example 3.3: For the ESG given in Figure 3.1, *BCBC* is an ES of length 4 with the initial vertex *b* and end vertex *c*.

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

Definition 3.6 (Complete Event Sequence): An ES is a complete ES (or, it is called a *complete event sequence*, CES), if $\alpha\left(ES\right)=\xi\in\Xi$ is an entry and $\omega\left(ES\right)=\gamma\in\Gamma$ is an exit.

Example 3.4: *ACB* is a CES of the ESG given in Figure 3.1.

CESs represent walks from the entry of the ESG to its exit realized by the form (initial) user inputs→ (interim) system responses → … → (final) system response.

Note that a CES may invoke no interim system responses during user-system interaction, i.e., it may consist of consecutive user inputs that lead to the exit.

Analogous to the notion of EP, *faulty (or illegal) event pairs (FEP)* are introduced as the edges of the corresponding $\overline{ESG}$.

Definition 3.7 (Faulty Event Pair): Any EP of the $\overline{ESG}$ is a *faulty event pair* (*FEP*) for ESG.

Example 3.5: *CA* of the given $\overline{ESG}$ in Figure 3.4 is a FEP.

Further, an EP of the ESG can be extended to a *faulty*, or an *illegal*, *event triple* by adding a subsequent FEP (if there exists one) to this EP, e.g., *AB* and *BB* of Figure 3.5, resulting in *ABB*. Thus a faulty event triple 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.

Definition 3.8 (Faulty Event Sequence): Let $ES = \langle v_0, \ldots, v_k \rangle$ be an event sequence of length $k+1$ of an ESG and $FEP = \langle v_k, v_m \rangle$ a faulty event pair of the corresponding $\overline{ESG}$. The concatenation of the ES and FEP then forms a *faulty event sequence* $FES = \langle v_{0,\ldots,} v_k, v_m \rangle$.

Faulty event sequences have only one FEP at its end because of a simple reason: in the error state in which the system should lead it is not observable what kind of system changes have been evoked, if any. Thus, it is essential to reset the system before triggering another FEP. Error states are also not distinguished which lapse a superposition of FEPs, i.e., conducting more than one FEP immediately after another.

Example 3.6: For the ESG given in Figure 4, BCA is an FES of length 3.

Definition 3.9 (Faulty Complete Event Sequence): An FES is *complete* (or, it is called a *faulty complete event sequence*, *FCES*) if $\alpha\ (FES) = \xi \in \Xi$ is an entry. The ES as part of a FCES is called a *starter*.

Note that Definition 3.9 explicitly points out that a FCES does not finish at an exit, unlike a CES that must finish at an exit.

Example 3.7: For the ESG given in Figure 3.5, the FEP CA of the $\overline{ESG}$ can be completed to the FCES ACBCA by using the ES ACBC as a starter. Note that the [is not included in the FCES as it is a pseudo vertex.

The starter ACBC in Example 3.7 is arbitrarily chosen, and hence the variation in length of an FCES is always attributable to starters prior to this special FEP under consideration. The result is then FCESs of various lengths. Thus, the "length" in the test process primarily relates to the CESs.

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

(a) An FEP that starts at the entry of *e* is also an FCES.
(b) 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, the set of FCES is referred as the set *V*.

3.3 Fault Model

This approach assumes that a specification is given describing the functional system tasks to construct appropriate ESGs, in accordance with the user expectations.

Definition 3.10 (Test Case): A *test case* is an ordered pair of an input and expected output of the SUT. Any number of test cases can be extended to a *test set*.

Since a CES is supposed to successfully run the system it can be used as a test case. The test inputs are thereby determined by the events and the expected outputs as the system response that has been reached.

Definition 3.11 (Fault): A test fails if the CES starting at the entry node

(i) cannot reach the exit node due to a failure, e.g., system crash (*sequencing fault*), or

(ii) reaches the exit node, but does not deliver the expected outputs (*functional fault*).

Accordingly, a FCES is supposed to cause a failure, or if there is an *exception handling* mechanism [Goo75, RLT78], an error message about the impairment of the events; otherwise a sequencing fault occurs. In this context exception handling is not used concerning any specific programming languages but rather in general. A sequencing fault can also occur in the starter portion, i.e., in the ES as the prefix of the FCES. FCES-based functional faults do not make any sense as they are supposed to exclude the expected behavior of the system. This fault model assumes that there is no user error, i.e., upon a faulty user input the system has to inform the user, and, wherever possible, point him or her properly in the right direction in order to reach the anticipated desirable situation. Due to this requirement, the complementary view was introduced to consider potential user errors in the modeling of the system (see also [Gau95, KPY99]).

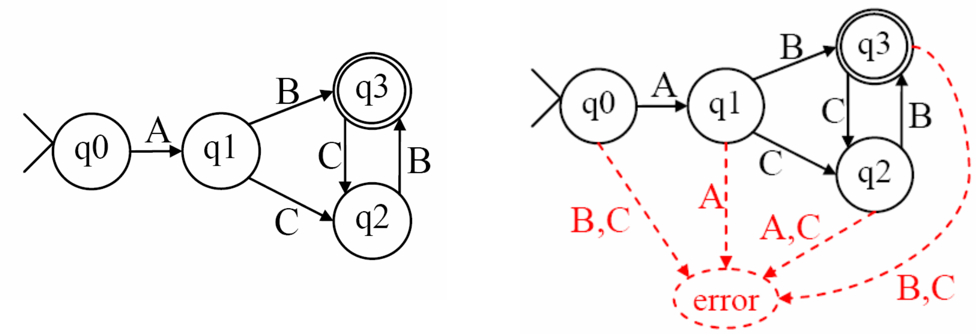
Although the fault model is very simple and makes clear how the oracle problem is handled: a CES-based test case is supposed to succeed a test by reaching the exit node whereby a FCES-based one passes the test when the system indicate a fault. In spite of its simplicity, the fault model is sufficiently 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 output has been obtained.

3.4 Handling Other Features

3.4.1 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 Figure 3.1 is represented as an FSA (Figure 3.6(a)) which then is completed by an error state (Figure 3.6(b)). Figure 3.5 is then compared with Figure 3.6(b) in order to illustrate the fault modeling features in FSAs.



(a) FSA which is equivalent to the ESG of Figure 3.1

(b) Completed FSA

Figure 3.6: Completed FSA of Figure 3.1, leading to a total of 7 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 [Wes96]. The ESG in Figure 3.1 has three events, leading to a total of 9 edges ($3^2 = 9$) of the CESG in Figure 3.5, without counting the entry and exit edges. Assuming that the corresponding FSA in Figure 3.6(a) has four states and an input alphabet of three symbols, *A, B* and *C*, the corresponding, *completed FSA* is given in Figure 3.6(b) with an extra state "error". For the sake of simplicity, edges are allowed to be associated with multiple inputs, e.g., with *B,C*. Evidently, a completed FSA 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 of the completed FSA in Figure 3.6 has a total of 12 edges (3·4 = 12); with the edge labeled with two inputs counted as a double edge. The approach presented in this thesis is different from the finite-state based approaches, in that ESGs are based on a finite sequence of events rather than states, and therefore disregards the detailed internal behavior of the system. Hence, an ESG is a more abstract representation compared to a state transition diagram of a FSA.

3.4.2 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.

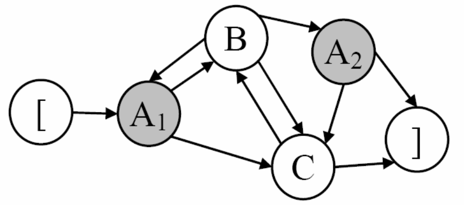


Figure 3.7: Interaction ambiguities caused by the double occurrence of *A*

As an example, Figure 3.7 depicts an ESG that has two different nodes (indicated by shading) 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.

3.4.3 Extension of the Fault Model

Based on past work related to fault modeling [BG91, DLS78, DMM01, DO91, EB84, FMMD94] and denoting inputs, outputs, states, or transitions as *elements*, the following fault model is obtained.

- *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. [Gil62, Glu63]. The hypotheses can be extended from single errors to multiple (*n*) errors [EB84]:

- 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, [CBCH+92, MS03, OLAA03].

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).

3.5 Summary

Event sequence graphs (ESG) visualize human-machine interaction in a formal, but nevertheless lucid and easy to understand way. It is clear that such a representation disregards the detailed internal behavior of the system, which is given by means of its different states and, hence, an ESG is a more abstract representation. Furthermore, the ESG representation is adequate first for the complementary view and thus for a precise fault model. Second, the notion of refinement, as introduced in this chapter, strongly supports to model and analyze also even very complex systems by hierarchically decomposing them.

4 Test Process, Test Generation and Test Execution

In this chapter the focus is on test generation from an ESG model. The method described in this section uses ESGs and their complements as inputs and primarily generates a test set that is complete with respect to a model-based coverage criterion introduced. Along with this, algorithms and results known from graph theory are adopted and extended to minimize the test set constructed, considering also structural features of the SUT. Parts of this chapter have been published in [BB04].

4.1 Objectives

As already mentioned when discussing the fault model (Section 3.3), 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 an error state and might invoke a fault detection/correction procedure, provided that an appropriate exception handling mechanism has been implemented. 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.

A test generation algorithm is sought that, given an ESG and the corresponding CESG, generates tests that satisfy the following *coverage criteria*.

Definition 4.1 (Coverage Criteria):

(a) Cover all event pairs in the ESG, and
(b) 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 [Bel01, BD97]. However, this criterion is more powerful than the transition or node coverage criterion [OLAA03] as, in addition, it requires the coverage of pairs of nodes in the ESG and its corresponding CESG, thus the coverage of the FEPs. Later in Section 4.3.3 it is shown how this pair-wise coverage can be generalized to n-tuple, $n > 2$, coverage and the cost and benefits of such an extension. Moreover, Case Study 1 (Section 5.5) demonstrates the fault detection capability of FCESs.

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.

Definition 4.2 (Constraints):

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

The constraints 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, the 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 satisfies (Definition 4.1(a)) and (Definition 4.2(a)) 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 (Definition 4.1(b)) and (Definition 4.2(b)) is referred to as the *minimal spanning set for the coverage of faulty event sequences* (*MSFCES*). 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, it must be checked if the system behaves as expected on a correct input. In the case an FCES is input, it must be checked 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.

The test process is summarized by the given Algorithm 4.1. The approach assumes that the system is modularly structured, e.g., having a set of well-defined functional tasks $f_1, \ldots, f_n$ given by the specification. Each of these tasks represents a system response and can be modeled by a set of ESGs which are constructed prior testing.

A major problem is the determination of the correct (i.e., desirable) and faulty (i.e., undesirable) behavior, also known as the oracle problem [Bin00, Ham94]. 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.

The coverage-oriented test process of the approach leads to test cases which exercise the specified functions of the implemented SUT with the goal to cover these functions. This coverage must be, of course, economical in terms of the number of test cases. Therefore, a stopping rule (*test termination*) of the test case generation is needed. This is given in the Algorithm 4.1 by the coverage of the event sequences of increasing length, which is determined by analyzing the model.

Algorithm 4.1 (Test Generation and Execution):

Input: A set of well defined functional tasks $f_1,\ldots,f_n$ represented by appropriate ESGs with:
n := number of the functional tasks of the system
length := required length of the event sequences to be covered
Output: A test report of failed and succeeded test cases

```
FOR i:=1 TO n DO
BEGIN
  Generate $\overline{ESG}$
  FOR k:=2 TO length DO
  BEGIN
    Cover all ESs of length k by means of CESs
    subject to minimizing the sum of lengths of
    the CESs                              //Section 4.3.1
  END
  Cover all FEPs by means of FCESs subject to
  minimizing the sum of the lengths of the FCESs
                                          //Section 4.3.2
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 Test Generation and Execution Algorithm

In this section the algorithms used for the generation of CESs and FCESs is sketched given an ESG and its complement. Algorithms presented here are extensions of the well-known algorithm for solving the Chinese Postman Problem (CPP) [EJ73].

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 Definition 4.2. However, such a solution might fail to satisfy the first criteria in Definition 4.1 which requires that all event pairs be also covered. It is the satisfaction of Definition 4.1 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.

4.3.1 Minimal Spanning Set for the Coverage of ESs

As mentioned earlier in Chapter 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 [Wes96], MSCESs can be constructed as follows:

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

For each of the three steps Figure 4.1 includes a corresponding ESG which illustrates different steps of the construction of MSCESs.

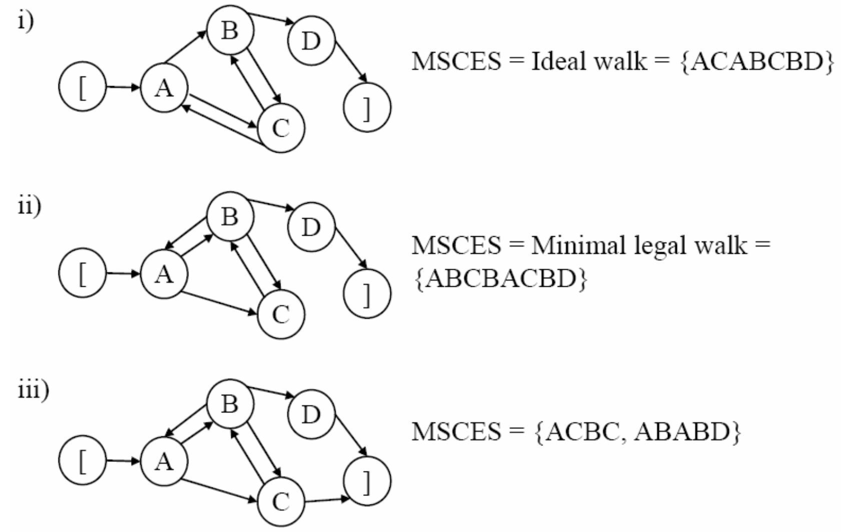
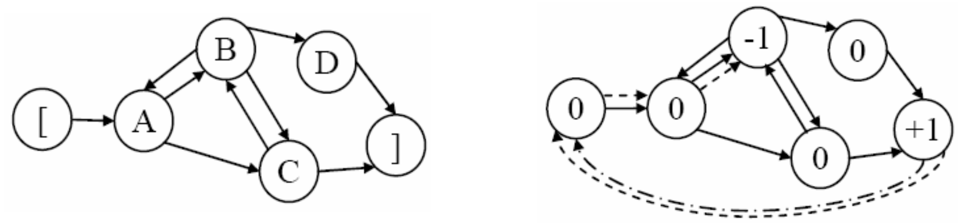

i)
MSCES = Ideal walk = {ACABCBD}
ii)
MSCES = Minimal legal walk = {ABCBACBD}
iii)
MSCES = {ACBC, ABABD}
[
A
B
C
D
]

Figure 4.1: Construction of MSCESs for different ESGs

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 the results relevant to the calculation of test costs are summarized and that make the test process scalable [Dre98].

An algorithm described in [Thi03] to solve the CPP determines a minimal tour that covers the edges of a given strongly connected graph. Transformation of an ESG into a strongly connected graph is illustrated in Figure 4.2. Addition of a backward edge, indicated as a chain dotted arrow from *]* to *[*, transforms the ESG in Figure 4.2(a) to a strongly connected graph in Figure 4.2(b).



(a) An example ESG

(b) Transferring walks into tours and balancing the nodes

Figure 4.2: ESG transformation for the coverage of ESs

The labels of the vertices in Figure 4.2(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 [Wes96]. This tour may have multiple occurrences of the backward edge indicating the number of walks.

Example 4.1: For the ESG given in Figure 4.2(a), the minimal tour (based on Figure 4.2(b)) and the minimal set of the legal walks (i.e., CESs) covering the EPs is given by:

Minimal Tour = *[ACBC][ABABD][*;
MSCES = {*ACBC, ABABD*}

Note that no entire walks exist. Therefore, an ideal walk cannot be constructed.

It is obvious that a large number of events can exist for large systems resulting in a large number of nodes. By means of the introduced refinement (see Definition 3.2), the system can be divided into several ESGs to reduce the nodes and therewith the calculation period of the algorithm. But to fulfill the coverage criteria in Definition 4.1, the additional EPs of a refined vertex also have to be considered for test generation. Therefore it is claimed that at least one CES for each entry and exit node as an initial and end vertex respectively has to exist. This problem is already be solved by considering the pseudo vertices *[* and *]*.

In Algorithm 4.2, ε and σ denote the pseudo nodes of the ESG, the symbol := the assignment command. Given an event $v \in V$, *diff*(v) denotes the number of predecessor events of v minus the number of its successor events, which enables the construction of the bags (or multisets) *A*, *B* in the FOR-loop. The notation $[\![\ ]\!]$ for bags and $\uplus$ for bag unions are introduced. They can be defined informally as follows. For instance, if *diff*(v)=3 in the first iteration step, assuming that A is initially empty, the bag A will consist of three instances of v, i.e., A = $[\![v, v_{,}\ v]\!]$ after the assignment there. Note that $[\![v, v_{,}\ v]\!] \neq \{v\}$, because the two entities on either side of the inequality sign $\neq$ are of different types; on the LHS is a bag (with three instances of v), whereas on the RHS is a singleton set with one element v. Turning to $\uplus$, note that $[\![v, v_{,}\ v]\!] \uplus [\![v]\!] = [\![v, v_{,}\ v, v]\!]$.

Theorem 4.1: MSCES can be constructed in time O($|V|^3$)

Proof (sketched; see also [Thi03]): The shortest paths from one node to all other ones can be determined by a depth-first-search in O($|E|$+$|V|$), as ESG under consideration is an unweighted graph. Furthermore, because $|E|$>>$|V|$+1 holds for a strongly connected graph, the complexity can be approximated to O($|E|$). Resulting in O($|V|$*$|E|$) for the shortest path of all nodes to all others. 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($|V|$*$|E|$) + O($|V|^3$) + O($|E|$*$|V|$) = O($|V|^3$). □

Algorithm 4.2 (Generation of MSCES):

Input: *ESG=(V, E, Ξ, Γ); ε=[, σ =]*
Output: *MSCES*

```
add_arc(ESG,($\sigma$,$\varepsilon$));
bags A, B, M := $[\![\,]\!]$ ;
set MSCES := $\emptyset$;                          //empty bags & set
FOREACH node $v \in V \cup \{\sigma,\varepsilon\}$ DO
BEGIN
  IF diff(v) > 0 THEN
    FOR i:=1 TO diff(v) DO
      A := A $\uplus$ $[\![v]\!]$ ;
  IF diff(v) < 0 THEN
    FOR i:=1 TO diff(v) DO
      B := B $\uplus$ $[\![v]\!]$
END;
m := |A| = |B|;                                   //cardinality
D[1 .. m][1 .. m];                                //distance matrix D
FOREACH node $v \in A$ DO
  compute_shortest_paths(v,B,D);
M := solveAssignmentProblem(D);
FOREACH $(i,j) \in M$ DO
BEGIN
  Path := get_shortest_path(i,j);
  FOREACH arc $e \in Path$ DO
    add_arc(ESG,e)
END;
EulerTourList := compute_Euler_tour(ESG);
start := 1;
FOR i:=2 TO length(EulerTourList)-1 DO
BEGIN
  IF (getElement(EulerTourList,i) = $\sigma$) THEN
```

```
      MSCES := MSCES ∪ getPartialList(
                                  EulerTourList,start,i);
    start := i+1
  END;
  RETURN MSCES;
```

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 [Dij59].

Example 4.2: The minimal set of the illegal walks for the graph in Figure 4.2(a) is

FCES={*AA, AD, ABA, ACA, ACC, ACD, ABDB, ABDA, ABDD*}

While constructing the MSCESs, it is taken 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 a failure to occur/recur.

Consider the following scenario: based on the ESG given in Figure 4.3, 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.

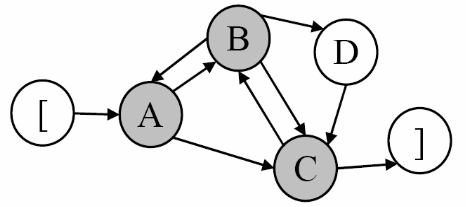


Figure 4.3: Static faults vs. dynamic faults (discussed events are shadowed)

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.

Figure 4.4 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.

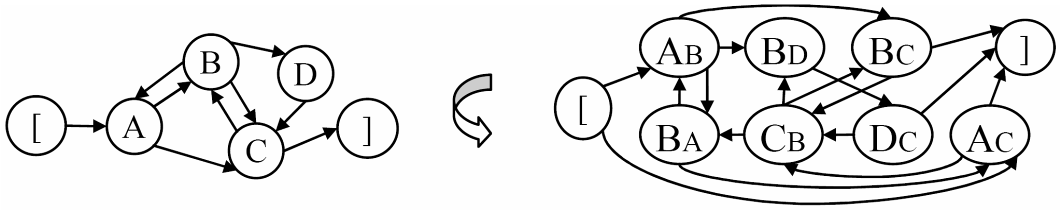


Figure 4.4: Extending the ESG for covering ESs of length=3

Theorem 4.2: At each transformation, starting with a completed ESG, the number of edges stepwise increases to $|V|^{k+1}$ where k is the length of the ES to be covered with $k \geq 2,\ k \in \mathbb{N}$.

Proof (by mathematical induction):

Initial Step. $k=2:\frac{\left(|V|^2\right)^2}{|V|}=|V|^3\overset{k=2}{\Rightarrow}|V|^{k+1}$

Inductive Step. $k=n:\frac{\left(|V|^n\right)^2}{|V|^{n-1}}=|V|^{2n-n+1}=|V|^{n+1}\overset{k=n}{\Rightarrow}|V|^{k+1}$

$$k=n+1:\frac{\left(|V|^{n+1}\right)^2}{|V|^{n+1-1}}=|V|^{2n-n+2}=|V|^{n+2}\overset{k=n+1}{\Rightarrow}|V|^{k+1}$$ □

The common procedure embodying this approach is given in Algorithm 4.3.

Algorithm 4.3 (Generating ESs and FESs with length > 2):

Input: $ESG=(V, E, \Xi, \Gamma)$; $\varepsilon =[$, $\sigma=]$, $ESG'=(V', E', \Xi', \Gamma')$ with $V'=\emptyset$, $\varepsilon'=[$, $\sigma'=]$;
Output: $ESG'=(V', E', \Xi', \Gamma')$, $\varepsilon'=[$, $\sigma'=]$;

```
FOREACH (i,j) $\in$ E DO
BEGIN
  addNode(ESG',(ES(ESG,i) $\oplus$ $\omega$(ES(ESG,j)));
  removeArc(ESG,(i,j))
END;
FOREACH i $\in$ V' DO
BEGIN
  FOREACH j $\in$ V' DO
    IF(ES(ESG',i) $\oplus$ $\omega$ (ES(ESG',j)) =
              $\alpha$ (ES(ESG',i)) $\oplus$ (ES(ESG',j)) THEN
      addArc(ESG',(i,j));
  FOREACH (k,l) with k=$\varepsilon$ DO
    IF(ES(ESG',i) =
                 ES(ESG,l) $\oplus$ $\omega$ (ES(ESG',i)) THEN
      addArc(ESG',($\varepsilon$',i));
  FOREACH (k,l) with l=$\sigma$ DO
    IF(ES(ESG',i) =
                 $\alpha$ (ES(ESG',i)) $\oplus$ ES(ESG,k) THEN
      addArc(ESG',(i,$\sigma$'));
RETURN ESG';
```

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

The approach has been applied to the testing and analysis of the GUIs of different kinds of systems, leading to a considerable amount of practical experience [Bel01]. A great deal of test effort could be saved considering the structural features of the SUT. Thus, there is further potential for the reduction of the cost of the test process.

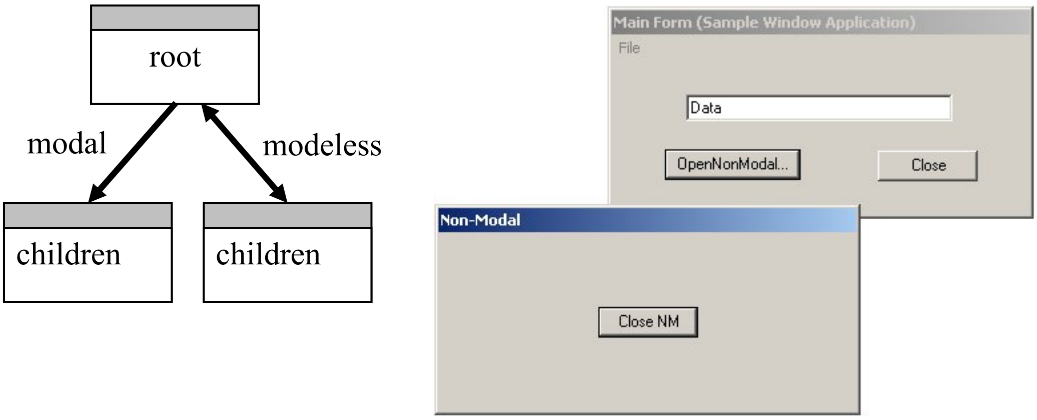


Figure 4.5: Modal windows vs. modeless windows and an example of an modeless opened window

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

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

Figure 4.5 represents these window types as a “family tree”. In this tree, a unidirectional edge indicates a modal parent-child relationship. A bidirectional edge indicates a modeless one.

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

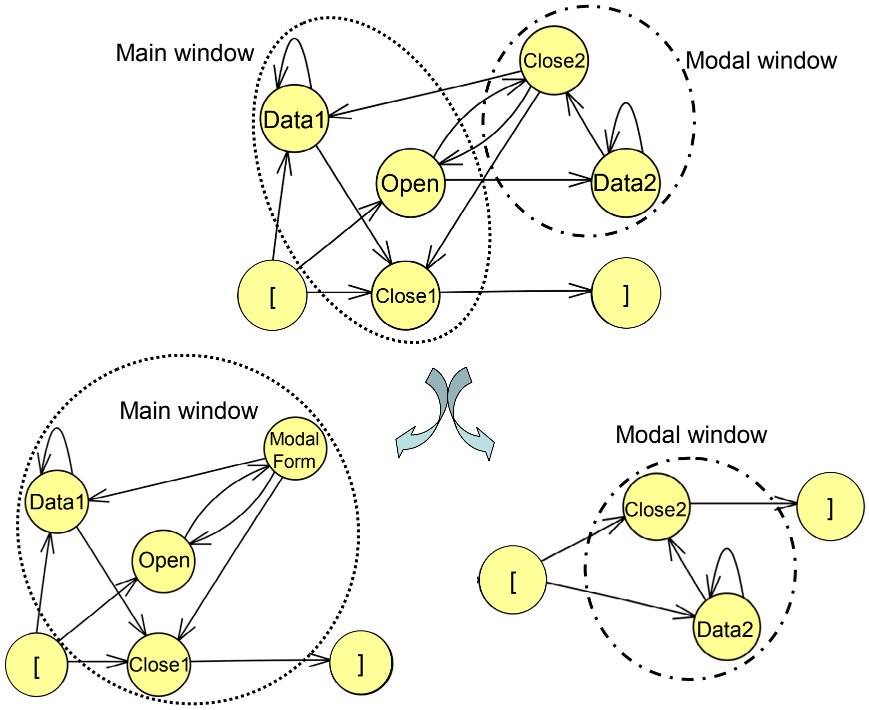


Figure 4.6: Modification of the ESG of the example in Figure 4.5 by considering the modality feature

Thus, similar to the strong-connectedness and symmetrical features [SS97], the modality feature is extremely important for testing since it avoids unnecessary test efforts. Figure 4.6 represents the modified ESG, which separates the event “Modal Form” takes the modality into account that avoids unnecessary FEPs.

Theorem 4.3: The separation of one node of an ESG with $|V|$ nodes leads in worst-case to $|V|^2+1$ test cases to cover all legal and illegal event pairs.

Proof: The total number of edges (including the self-loops) of a digraph with $|V|$ nodes is $|V|^2$ which determines the number of total number of legal and illegal event pairs. Let δ be the number of separated nodes with $0<\delta<|V|$ before the decomposition. After decomposition, the number of nodes of the original ESG is reduced by δ+1 because a virtual node has to represent the δ+1 nodes that had been separated. The new ESG possesses then exactly these δ+1 nodes. Therefore, after the decomposition two ESGs exist: the previous one with δ edges less plus one and a new one with δ edges. Thus, the number of test cases is given by the function:

$$f\left(|V|,\delta\right)=\left(|V|-\delta+1\right)^2+\delta^2 \Rightarrow f\left(|V|,\delta\right)\Big|_{\delta=1}=f\left(|V|,\delta\right)\Big|_{\delta=|V|}=|V|^2+1 \qquad \square$$

Theorem 4.4: The separation of $\frac{|V|+1}{2}$ nodes of an ESG with $|V|$ nodes leads in best-case to $2\left(\frac{|V|+1}{2}\right)^2$ test cases to cover all legal and illegal event pairs.

Proof: For the function $f\left(|V|,\delta\right)=\left(|V|-\delta+1\right)^2+\delta^2$ of the Theorem 4.3 above, there exists a minimum for $\delta=\frac{|V|+1}{2}$, because

$$\frac{\partial f\left(|V|,\delta\right)}{\partial\delta}=-2|V|+4\delta-2 \text{ with } -2|V|+4\delta-2=0 \Rightarrow \delta=\frac{|V|+1}{2}$$

Thus, resulting in $f\left(|V|,\delta\right)\Big|_{\delta=\frac{|V|+1}{2}}=\left(\frac{|V|+1}{2}\right)^2+\left(\frac{|V|+1}{2}\right)^2=2\left(\frac{|V|+1}{2}\right)^2$. □

4.5 Test Configuration and Test Cost

The number and length of the event sequences are the primary factors that influence the cost of the test process. In order to compare the costs of different test configuration by a case study (see Section 5.5), a precise definition of these notions is necessary.

Definition 4.3 (Test Configuration): A *test configuration* is defined as a quintuple $(l_{\max}, \#ES_i, l_{\text{cov}}, \#CES, \beta)$ with

- $l_{\max}$ as the given maximum length of the ESs to be covered,
- $\#ES_i$ as the number of ESs of length *i*, with *i* = 2, 3, ..., $l_{\max}$,

- l_{cov} as the sum of lengths of *CES*s to cover all ESs of up to a given maximum length $l_{\max}$,
- *#CES* as the number of the CESs, and
- $\beta \in \mathbb{R}$ as the weight factor for conducting multiple tests

which determine the *test costs for ESs* as $\Psi_{ES} := \beta \cdot \sum_{i=2}^{l_{max}} \# ES_i + \sum_{i=2}^{l_{max}} \# ES_i \cdot i$,

and *test costs for CESs* as $\Psi_{CES} := \beta \cdot \# CES + l_{cov}$

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

The costs formulae Ψ_{ES} and Ψ_{CES} in the Definition 4.3 have two terms. The first term determines the maximum number of ESs (in Ψ_{ES}) and the maximum number of CESs (in Ψ_{CES}); the latter is the maximum number of tests to be run. Thus, this first term reflects the test costs caused by restarting the system before initiating another test (*test multiplicity*). However, a single test can cover several ESs, even of different length. To adjust to a specific situation, the tester can vary the weight factor β . Typically, β has the value 1 if the tester has no hint about the multiplicity of a planned series of tests. The weight factor can be greater than 1 in order to reflect a situation with disproportionate costs for restarting the system before a new test. The value of β can be also zero if the costs of the restart process are negligible.

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

For the deployment of these and Ψ_{CES} in the case study (Chapter 5), the assumption is made that each event and every restart have the same test costs, i.e., β =1.

4.6 Summary

Based on the test process, the algorithms introduced determine a minimal number of legal and illegal test cases to fulfill a well-defined coverage criterion. The test costs are given by the total length of the CESs and FCESs which are necessary for the coverage. The length of the ESs can be increased stepwise that enables a scalability of the test costs which increase proportional with the length of the ESs to be covered.

5 Case Study 1: The RealJukebox - RJB

The case study presented in this chapter demonstrates the ease-of-use of the approach and validates the effectiveness of the algorithms developed and analyzed in the previous section. These algorithms were applied to various components of the public domain software RealJukebox of the RealNetworks in order to generate tests. There was no access to the source code and no specification of the application was available, except an on-line user manual. Hence all ESGs required for test generation were derived from the GUI of RealJukebox. Parts of this chapter have been published in [BBW06].

5.1 Objectives

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

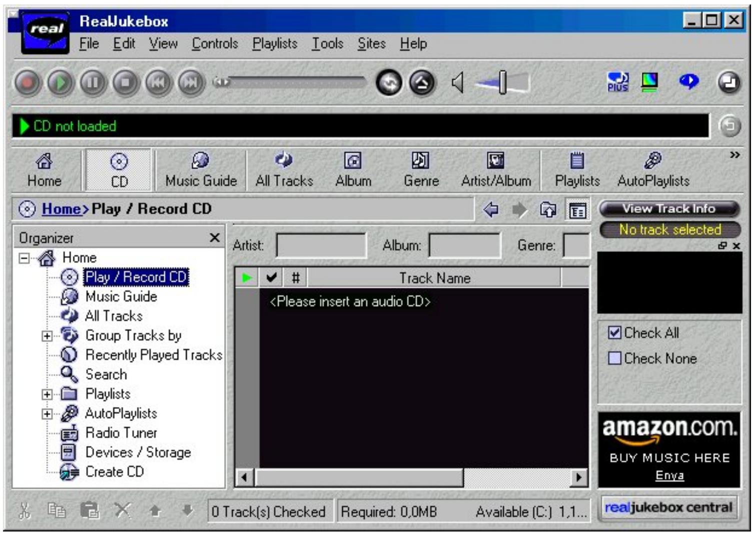
- event-tuple coverage, i.e., length of the ES to be covered, subsequently referred to as *n-tuple coverage*, and
- the number of the event sequences

on the fault detection effectiveness of the CESs and FCESs using the algorithms sketched in Section 4.3 (see [BSH86, WRHO+00]).

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. The fault detection effectiveness of the generated test set for *n*=2, 3, and 4, are studied 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. Figure 5.1 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>

No track selected

0 Track(s) Checked

Required: 0,0MB

Available (C:) 1,1...

Figure 5.1: Example of a GUI (RealJukebox of RealNetworks)

In the absence of a manufacturer’s system specification, namely, a functional description of the RJB, the help facilities and the handbook of the RJB are used to produce the references for construction of the test cases and test scripts, based on CES as desirable events. Those functions describe the steps as to how to reach situations the user wants, i.e., desirable events in terms of system functions (responsibilities). For this case study, 12 different functions of the SUT (Table 5.1) were identified.

Table 5.1: System functions as responsibilities of the system to interact with the user

1. *Play and Record a CD or Track*
2. *Create and Play a Playlist*
3. *Edit Playlists and/or AutoPlaylists*
4. *Views Lists and/or Tracks*
5. *Edit a Track*
6. *Visit the Sites*
7. *Visualization*
8. *Skins*
9. *Screen Sizes*
10. *Different Views of Windows*
11. *Find Music*
12. *Configure RJB*

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 Figure 5.2 represents the top-level GUI to produce the desired interaction *Play and Record a CD or Track* via the main menu in Figure 5.1. 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. Figure 5.2 illustrates all sequences of user-system interactions to realize the likely operations that the user might launch when using the system.

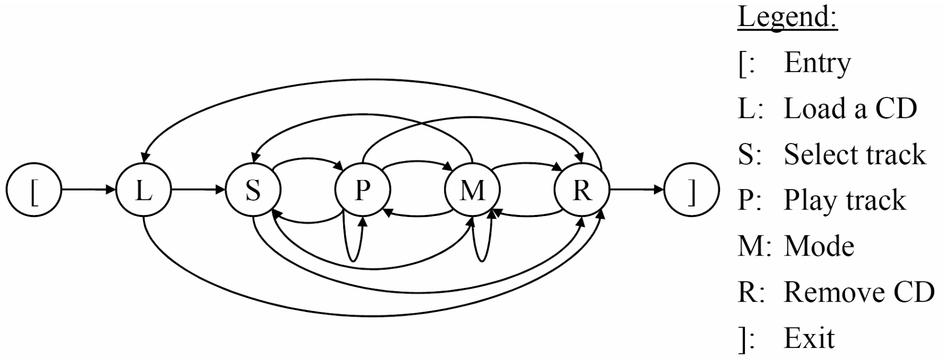


Figure 5.2: The system function *Play and Record a CD or Track* represented as an ESG

Each of the correct interactions, denoted by the nodes in Figure 5.2, defines a system sub-function that must be refined and represented in a corresponding sub-ESG, as done in Figure 5.3 for the node *P* (*Play track*).

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.

The ESGs of Figure 5.3 refine the vertices of the graph depicted in Figure 5.2. The refinement *Select track* describes the alternative ways to select a track. *Play track* describes the variety of the functions similar to those encountered in an ordinary cassette player. The sub-graph *Mode* describes the different ways to control the operation, i.e., playing of the tracks. Any of the design levels can be used to generate CESs, thus test cases. As refined levels include more information, more test cases can be generated by analyzing those refined ESGs.

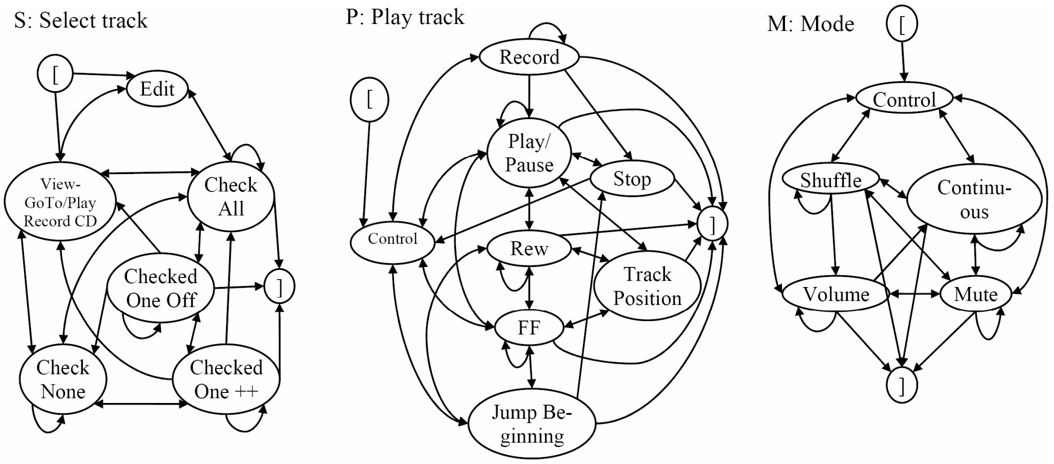


Figure 5.3: Refinement of the vertices *S*, *P* and *M* of the ESG in Figure 5.2

Each of the vertices of an ESG in Figure 5.2 and Figure 5.3 represents user inputs which interact with the system, leading eventually to events as system responses that are expected, i.e., correct situations. Thus, each edge of the ESG represents a pair of subsequent legal events which was defined as an event pair.

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 V_{env}, see Section 3.1), or a system response (an element of the set V_{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

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

which defines an ES as introduced in Chapter 3. An ES may consist of no interim system responses but only user inputs and a final system response as, for example, in Figure 5.2. Note that these event sequences are similar to those used by White et al. [WA00].

Given the ESG in Figure 5.2, test generation begins with an analysis of the system function *Play and Record a CD or Track*. This analysis leads to

EP={ *LS*, *LR*, *SP*, *SM*, *SR*, *PS*, *PP*, *PR*, *PM*, *MP*, *MS*, *MM*, *MR*,
RL, *RM* }

as the set of EPs.

In the next step CESs are generated which cover the set of EPs. As explained in Section 3.1, CES is a walk obtained by extending the EPs by appropriate prefixes and/or suffixes. The list

CES={ *LSR*, *LR*, *LSPR*, *LSMR*, *LSR*, *LSPSR*, *LSPPR*, *LSPR*,
LSPMR, *LSMPR*, *LSMSR*, *LSMMR*, *LSMR*, *LRLR*,
LRMR }

gives the CESs as test inputs. For each EP there is a corresponding CES listed which by definition has to be started with *L* and finished by *R*.

Some CESs, e.g., *LSR*, occur more than once. 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

CES_{new}={ *LSR*, *LR*, *LSPR*, *LSMR*, *LSPSR*, *LSPPR*, *LSPMR*,
LSMPR, *LSMSR*, *LSMMR*, *LRLR*, *LRMR* }.

The new set of the CESs ensures that all EPs are covered. However, it is not optimized yet using the minimal length criteria given by MSCES.

Next the set of FCESs are constructed. To do so the set of FEPs is examined. The dashed edges of the CESG in Figure 5.4 represent the FEPs of the function *Play and Record a CD or Track* of the RJB. These edges are listed in the set

FEP={ *LL*, *SL*, *LP*, *PL*, *LM*, *ML*, *SS*, *RP*, *RR*, *RS* }.

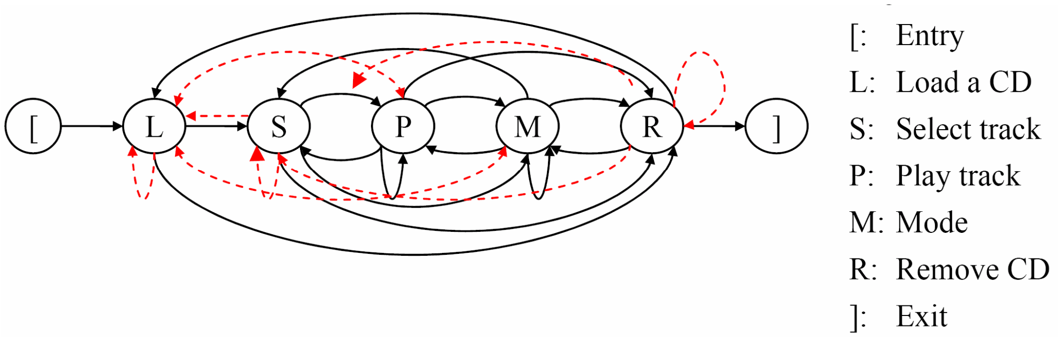


Figure 5.4: CESG (Completed ESG) of Figure 5.2 (dashed lines: FEPs)

Based on the definitions of the FESs and the FEPs, now the FCESs are systematically established in two forms:

- FEPs that start at the entry are complete test inputs to trigger undesirable situations, e.g., *LL*, *LP*, *and LM* of the set FEP.
- FEPs which do not start at the entry, e.g., *SL*, *PL*, *ML*, *SS*, *RP*, *RR*, *RS* of the set FEP, need prefixes resulting in

FCES={ *LL*, *LP*, *LM*, *LSL*, *LSPL*, *LSPML*, *LSS*, *LSPMRP*, *LSPMRR*, *LSPMRS* }.

Together with these test inputs, the test process can be carried out as described in Algorithm 4.1.

5.4 Test Generation

As mentioned in Section 5.2, 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 Chapter 8) 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. Tests were generated for pairwise, 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.

RJB was tested on two different PCs with different processors in order to detect and filter likely permanent or transient errors within the hardware and/or system software in any one of these test environments. An assumption is made that it is very unlikely for the same system failure to occur in both PCs.

By means of running the tests on two different platforms, random errors are excluded that cannot be reproduced. Thus, the reported errors are permanent ones and have been detected on both platforms (and not on only one stochastically). Furthermore, they are reproducible in an effective manner, provided that the system requirements specified in Table 5.2 are fulfilled.

Table 5.2: Different test systems and their equipment

	System Requirements of the Manufacturer	System 1 (Test Platform 1)	System 2 (Test Platform 2)
Operating system	Windows NT with Service Pack 4	Windows NT with Service Pack 4	Windows NT with Service Pack 4
Computer processing speed	Intel Pentium 200 MHz	Intel Pentium 266 MHz	Intel Pentium 233 MHz
RAM (available)	32 MB	64 MB (9 MB)	64 MB (16 MB)
Graphics	16 bit color video card (800x600 resolution)	32 bit color video card (1024x768 resolution)	32 bit color video card (1024x768 resolution)

The studied platforms are described in Table 5.2. Both these platforms fulfill the system requirements of the manufacturer; thus compatibility and conformance problems can be excluded. Significant characteristics are taken into account by considering the different options of run mode that are listed in the system description. The options differ from each other in the RJB settings: *AutoPlay* activated/deactivated, *AutoRecord* activated/deactivated. Any inconsistencies that might occur during the testing have been carefully analyzed taking adjusted settings of the RJB into account.

5.5 Results

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

Table 5.3 depicts the generated number of test cases for each function from Table 5.1 sorted by length.

Table 5.3: Test cases for the functions in Table 5.1

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

In Table 5.4 different values are assigned to the weight factor β to take different levels of costs of resuming the test process after performing a test into account (see Section 4.5, Definition 4.3). It becomes apparent that the test cost reduction increases with

- increasing length of covered ES, and
- increasing weight factor

if CESs are used as test inputs instead of ESs because CESs frequently cover several ESs of a given length by one single test (see the last column of Table 5.4).

Table 5.4: Test costs of ES and CES and their percentage reduction

Length of covered ES	Weight factor β	Test Cost Ψ_{ES}	Test Cost Ψ_{CES}	Reduction
2	0	938	826	11.94 %
	1	1407	985	29.99 %
	2	1876	1144	39.02 %
3	0	4706	3384	28.09 %
	1	5962	3854	35.36 %
	2	7218	4324	40.09 %
4	0	16414	10865	36.00 %
	1	19341	11890	38.52 %
	2	22268	12915	42.00 %

An overview of faults found subject to their sequence length is illustrated in Table 5.5 that summarizes the number of detected faults in the ESs to be covered by the total test cases from Table 5.3. It clearly shows that the detected faults by covering ESs of length 2 is a subset of the detected faults by covering ESs of length 3, which is, in turn, a subset of the detected faults by covering ESs of length 4.

Table 5.5: Overview of faults found, test cases and their distribution over sequence length covered

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

Table 5.6 shows the number of additional faults detected in different functions in terms of the length of ESs that are to be covered. In Table 5.6, it is interesting to note that many faults were detected in function 1, and yet not so many faults were detected in any of the other functions. Most notable were functions 2, 9 and 10, in which no faults were detected at all. First of all, function 1, *Play and Re-*

cord a CD or Track, corresponds to very high complexity, as can be seen from the tests required in Table 5.3 and also in Figure 5.2 and Figure 5.3. Yet by far, the most faults (28) were detected by just ESs of length 2; no other comparable number of faults is seen by any other function. Also many other functions required more tests of length 2, but did not discover nearly this many faults (only 5 faults for function 3 is the next largest number of faults for ES of length 2). Function 2 has nearly as many tests as function 1; but since this is logic to just create a playlist, there are clearly not as many faults as in actually playing a CD or track. As for functions 3, 9 and 10, they are of much lower complexity, and require far fewer tests than function 1.

Table 5.6: Additional faults by function

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

Table 5.7 summarizes the length of covered ESs and the corresponding faults, classified as functional fault and sequencing fault (for Definition 3.11, see Section 3.3).

Table 5.7: Test case costs and detected faults depending on the event length

Length of covered ES	Detected Faults by *CES*	Detected Faults by *FCES*	Total nb. Detected Faults	Sequencing Fault	Functional Fault
2	24	20	44	16	28
3	24+7	20+5	56	16+8	28+4
4	31+4	25+8	68	24+10	32+2

It is worth mentioning that additional 12 transient faults have been found during the testing procedure. These faults have been detected randomly and on only one of the test systems; thus they could not be reproduced on both test platforms (see Table 5.2). Therefore, they are not included in Table 5.5 and Table 5.6, which contain only faults that can be reproduced anytime on either platform exercising the same test case.

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

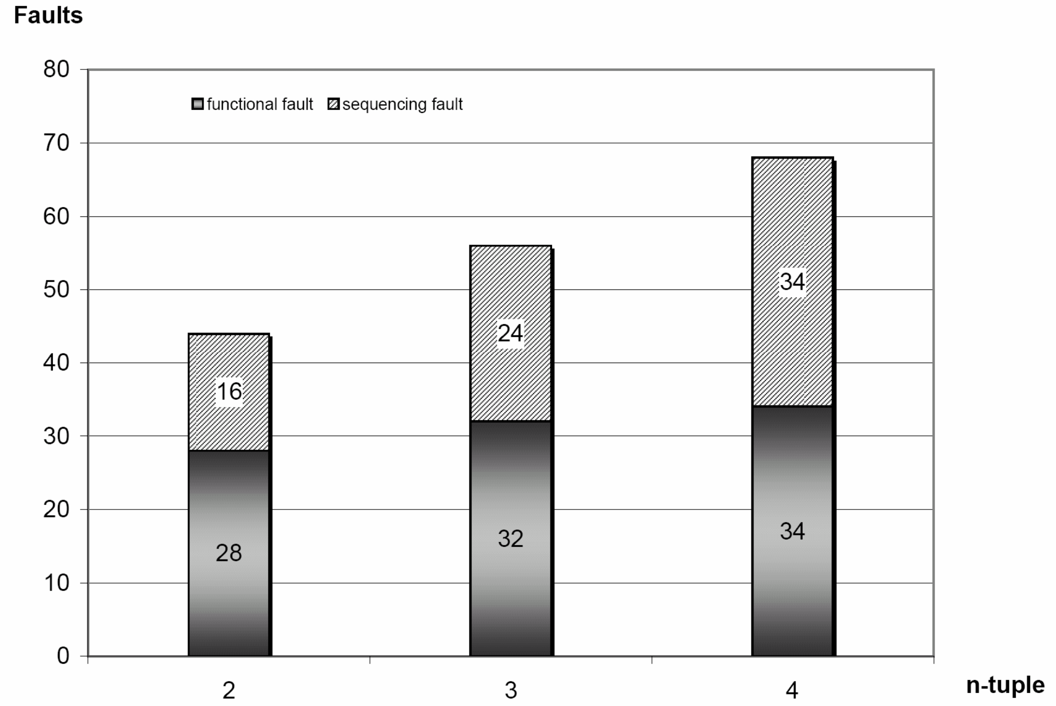


Figure 5.5: Detected sequencing faults and functional faults depending on the event length

Figure 5.5 shows the detected sequencing faults and functional faults with test length. Generally there are more functional faults than sequencing faults, specifically for test lengths of 2 and 3. Only at length 4 are there an equal number of functional faults and sequencing faults (see also Table 5.7).

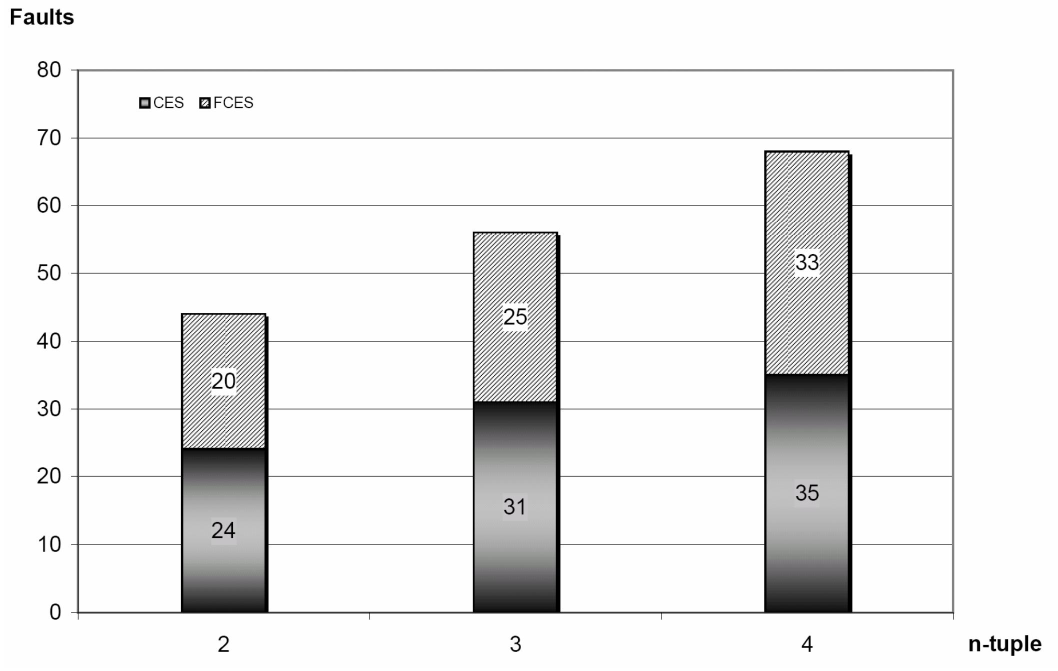


Figure 5.6: Detected faults by CES and FCES

It is also interesting to note that FCES-based test cases really did deliver approximately the same number of faults as the CES-based test cases did (Figure 5.6). Again, it is concluded that the reason is a poor design strategy of the GUI which concentrates on the realization of the desirable events and neglects the handling of the undesirable ones.

Figure 5.7 combines and refines the results found in Figure 5.5 and Figure 5.6. It can be observed that the tests based on the CESs of length 4 and on FCESs of length 4 are very beneficial in detecting defects: 19 defects have been detected by tests based on FCES of length 4 in relation to only 6 based on FCESs of length 2! Thus, a clear tendency can be observed that an increasing number and length of CES-based and FCES-based test cases lead to the detection of an increasing number of defects. Note, however, that Figure 5.7 does not consider the number of necessary tests, i.e., test costs.

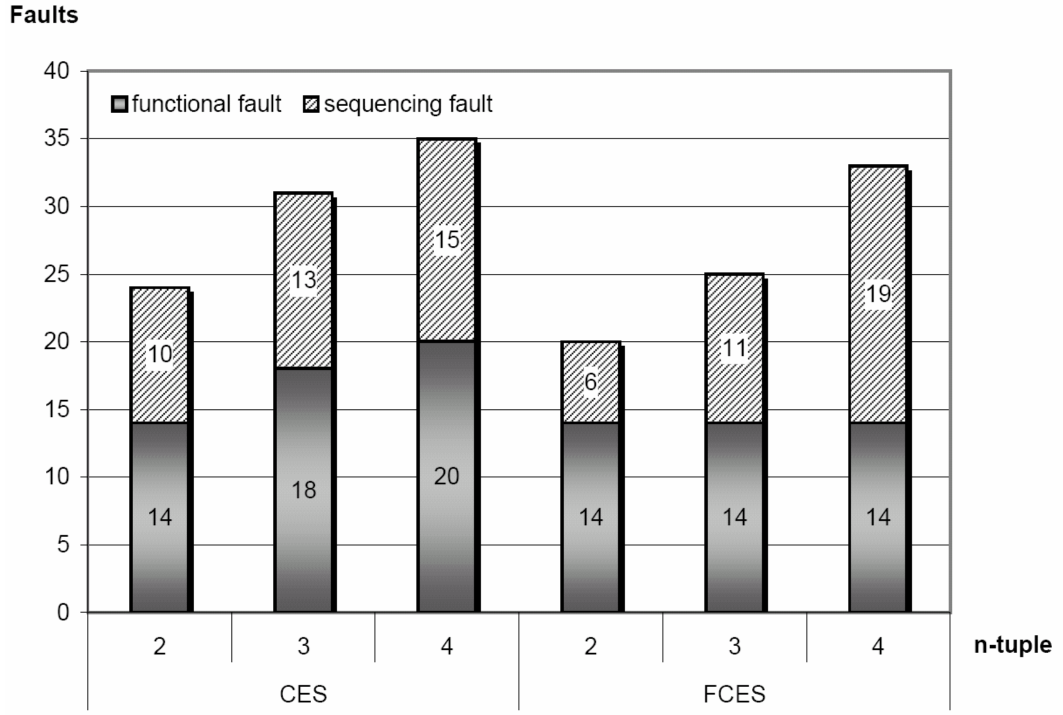


Figure 5.7: Sequencing faults and functional faults based on CES/FCES, depending on the event length

5.6 Analysis of the Results

Table 5.8 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. Figure 5.8 shows a plot of the cumulative count of faults detected versus the count of tests generated for *n*-tuple event coverage.

Table 5.8: 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

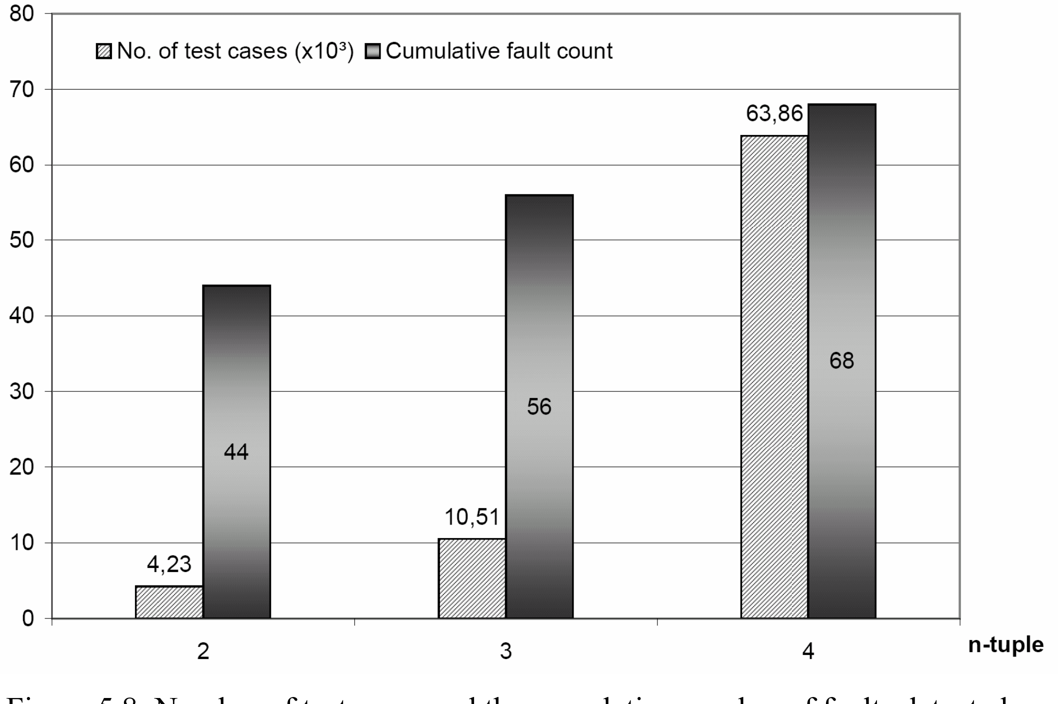


Figure 5.8: Number of test cases and the cumulative number of faults detected vs. length of test cases

The following observations from the data in Table 5.8 and Figure 5.8 are observed.

- 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.
- 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.

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 illustrate the fault detection capability of the approach, the function Play track is analyzed. This function is represented by the ESG in Figure 5.9 as a refinement of node *P* in Figure 5.2 and Figure 5.3.

Some of the detected faults are listed in Table 5.9. The fault detection process is simple. As an example, to detect fault 1 in Table 5.9, one starts with the Control option of the Main Menu of the RJB as in Figure 5.3 and sequentially pushes the button Rec and then the button Rew, or alternatively, the button FF, as shown in Figure 5.9 as alternative edges labeled with No. 1. The other faults in Table 5.9, labeled accordingly in Figure 5.9 with fault numbers, can be detected similarly. Fault numbers caused by an EP are marked by dotted arcs in Figure 5.9.

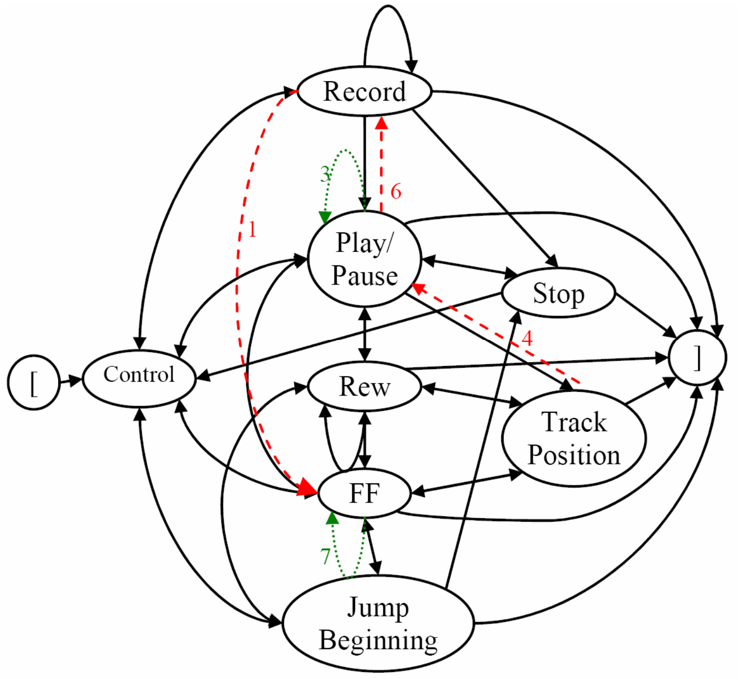


Figure 5.9: Completed ESG as a refinement of Figure 5.2

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 3.3.

Table 5.9: Detected faults related to the system function Play track (node *P* in Figure 5.3)

No.	Detected Faults	Test Case
1	While recording, pushing the forward button or rewind button stops the recording process without a due warning.	*Control Record FF*
2	If a track is selected but the pointer refers to another track, pushing the play button invokes playing the selected track; i.e., the situation is ambiguous.	*SelectTrack Play*

No.	Detected Faults	Test Case
3	Menu item Play/Pause does not lead to the same effect as the control buttons that will be sequentially displayed and pushed via the main window. Therefore, pushing play on the control panel while the track is playing stops the playing.	*Control Play/Pause Play/Pause*
4	Track position could not be set before starting the play of the file.	*Control Play/Pause FF Trackposition Play/Pause*
5	Record Shuffle does not activate shuffling, i.e., tracks will be processed sequentially.	*CheckOne++ Shuffle Record*
6	If the track is in Pause and Record button is pushed, then the track will be played.	*Control Play/Pause Play/Pause Record*
7	The system jumps to a track that was not selected and terminates the play-back although the selected tracks have not been completely played.	*Control Play/Pause FF FF FF*

Upon the detection of a fault, it was analyzed and categorized (sequencing/functional fault, ES-/FES-based, length). Then the system was restarted, i.e., recovered from the error state and the test process continued. Thus, about 432,000 sec were spent to execute 78,611 test cases, resulting in 5.5 sec. per test.

In order to produce approximating curves for the time between failures over the failure number, well-known software reliability models [Lyu96] have been deployed. Those models specify among other things the dependence of the failure process on fault detection. Using the reliability tool CASRE [LN92] it was found that the models Musa-Okumoto, Musa-Basic, Geometric, and Jelinski-Moranda correlated well with the data. Note that each of those models is characterized by a set of (more or less realistic) assumptions, e.g., that failures independently occur, or perfect debugging, i.e., fixing an error will not create new fault(s), or errors fixed earlier have a bigger effect than the ones fixed later, etc. [Lyu96]. One usually has to combine several models in order to compensate the assumptions that are not suitable for the SUT. For the case study, the best “goodness of fit” could be achieved by Geometric model, the results of which are depicted in Figure 5.10 and Figure 5.11 for the CESs and FCESs, respectively.

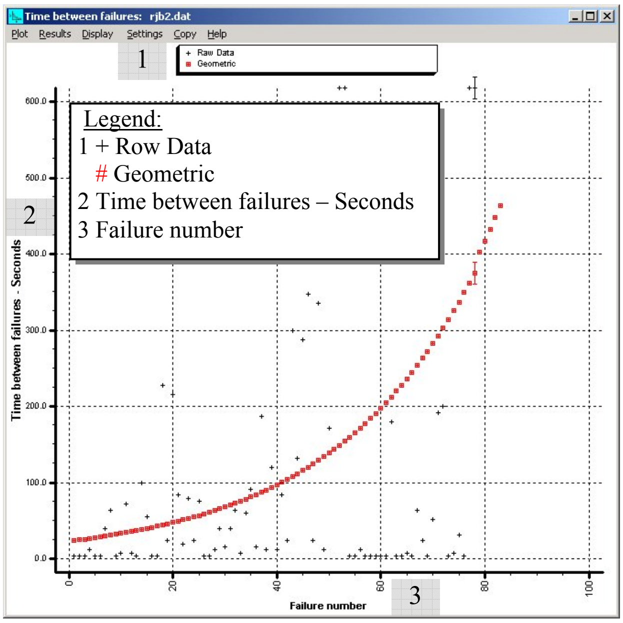

Time between failures: rjb2.dat
Plot Results Display Settings Copy Help
1
+ Raw Data
■ Geometric
Legend:
1 + Row Data
# Geometric
2 Time between failures – Seconds
3 Failure number
2
Time between failures - Seconds
600.0
500.0
400.0
300.0
200.0
100.0
0.0
0
20
40
60
80
100
Failure number
3

Figure 5.10: Time between failures based on the CESs of Figure 5.9

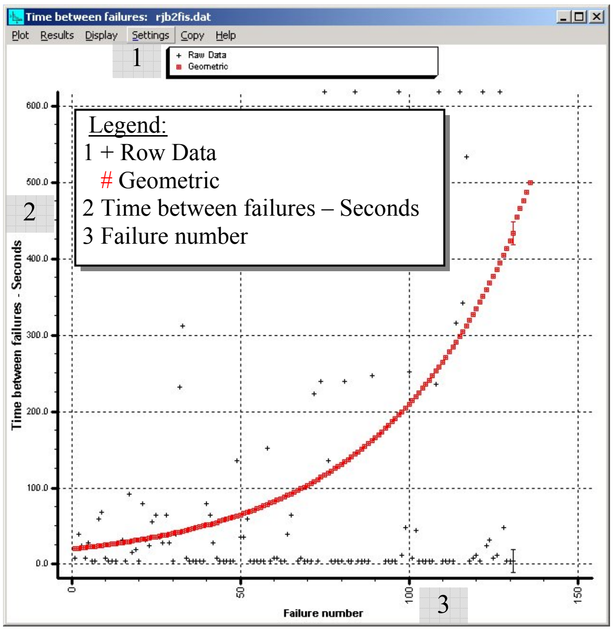

Time between failures: rjb2fis.dat
Plot Results Display Settings Copy Help
1
+ Raw Data
■ Geometric
Legend:
1 + Row Data
# Geometric
2 Time between failures – Seconds
3 Failure number
2
Time between failures - Seconds
600.0
500.0
400.0
300.0
200.0
100.0
0.0
0
50
100
150
Failure number
3

Figure 5.11: Time between failures based on the FCESs of Figure 5.9

The number of faults in Figure 5.10 and Figure 5.11 is greater than the real number because the repetitive occurrence of faults could not be eliminated. The SUT could also not be corrected before testing it further.

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

5.8 Defense Mechanism

Once the system has been transferred into an erroneous 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 [Goo75, RLT78].

The construction of the FCESs as described in Section 3.2 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* of the FCES set in Section 5.3 is "less risky" in terms of flexibility in fault correction than the sequence *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 fault correction capability can be implemented by adopting a conventional parsing algorithm known in compiler construction [AHU77]. 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 3.4.3) can be extremely useful for forming fault hypotheses that take the individual risk ratios into account. See also Section 6.1 for a safety related view of this notion.

5.9 Discussion

At first glance, it seems that only 68 faults are detected upon executing approximately 80,000 test cases (Table 5.8) – a huge test effort to detect a relatively small number of faults, especially compared to the previous experience [Bel01] 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.

5.10 Summary

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 FCES) and, thereby, helps manage the scalability of the test process.

6 Considering Safety Aspects

Additional to the GUI-focused case study of the previous chapter, the case study of this chapter uses an example based on a railway level crossing. The objective is to demonstrate the versatility of ESG-based approach for modeling and analysis of complex systems, considering also aspects of safety. This chapter bases of the work of Nimal Nissanke [ND02] and applies risk graphs to ESGs.

6.1 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* [BBN04].

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 it is not concerned with the actual quantification of risk, it is needed an ordering relation based on risk for the states of *M*. This offers a systematic selective approach to test generation.

Definition 6.1 (Risk Ordering): A risk ordering relation $\sqsubseteq$ is defined as $\sqsubseteq = \{(s_1, s_2) | s_1, s_2 \in R(M)$ and the risk level associated with state s_1 is less than that associated with state $s_2.\}$

The risk ordering function above is analogous to that used in [ND02]. 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 $\sqsubseteq$ can be found for other system attributes.

The risk ordering relation $\sqsubseteq$ is intended as a guide to determining an appropriate response to any faults 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.

Definition 6.2 (Defense Matrix): Given a set D of malfunctions, the defense matrix *DM*, *and the associated constraint,* are defined as $DM \in R(M) \times D \rightarrow R(M) : \forall s_1, s_2, d \bullet (s_1,d) \in \text{dom}\, DM \wedge DM(s_1,d) = s_2 \Rightarrow s_2 \sqsubseteq s_1$.

Note D (dysfunction) has been introduced in Section 3.2. 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 Definition 6.1, 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.

6.2 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 [HH02]. In this thesis it is preferred 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. Robustness is defined 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 [KKS98] is an elegant way to assess the robustness of a software system by the generation of special values and random inputs. Here an alternate ESG based approach is proposed to testing a software system for robustness. This approach allows the quantification of robustness with respect to a universe of erroneous inputs.

As mentioned in Chapter 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. How these faulty sequences are used to generate test inputs that test the exception handling ability of the SUT has been explained in Section 4.3.2. 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 and e the number of erroneous inputs, $e \leq n$, the robustness of the SUT is estimated to be the ratio $m / n \cdot n / e$.

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.

6.3 A Comprehensive Example: Railway Crossing

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.

6.3.1 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 (see also [Sto96, Wil91]). For this purpose a simple railway crossing is considered 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.

6.3.2 System Description and Model

An ESG model of such a crossing is shown in Figure 6.1. The set of input signals (or events *V*) are partitioned into the subsets V_{sys} and V_{env} with

- V_{sys} = { *R, G, C, O* } as *system signals* and
- V_{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 vulnerabilities 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 Figure 6.1, 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.

Figure 6.1 also indicates the relative risk levels brought about by the occurrence of the different events. In the ESG, the events posing greater vulnerabilities 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 V_{env} at the top in Figure 6.1. This is because they denote, in effect, human actions, including potentially faulty actions. Thus, risk level of R is less then the one of O ("Open gate") because "Turn red" implies halting the vehicle. In turn, the risk level of O is less than G ("Turn green") and C ("Close gate"), etc. Finally, the risk level of V ("Vehicle passes") and T ("Train passes") are maximums as they concern events which have most significant impacts on human life. 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.

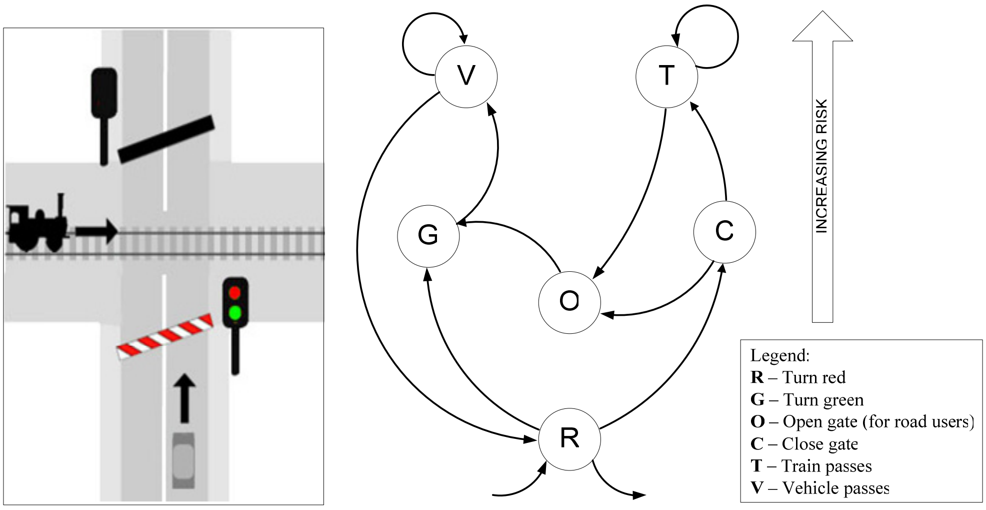


Figure 6.1: An ESG model of a railway level crossing

6.3.3 System Functions and Malfunctions

As is shown by directed arcs in Figure 6.1, the event pairs in this example are

$$\mathrm{EP} = \{ RG, GV, VV, VR, RC, CT, CO, TT, TO, OG \}$$

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

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

where λ denotes the empty event sequence. The faulty event pairs for the railway crossing are generated from the complete ESG shown in Figure 6.2. The FEPs are

$$\mathrm{FEP} =\{ RR, OO, CC, GG, RO, RT, RV, OR, OC, OT, OV, CR, CV, CG, TR, TC, TV, TG, VO, VC, VT, VG, GO, GC, GR, GT \}.$$

The FEPs are shown as dashed lines in the CESG in Figure 6.2 while the EPs are shown as solid lines. In the context of the framework introduced in Chapter 3, the expression RE above constitutes the *system function F*, while FEP represents the set of system malfunctions *D*, the elements of which are also called *vulnerabilities*.

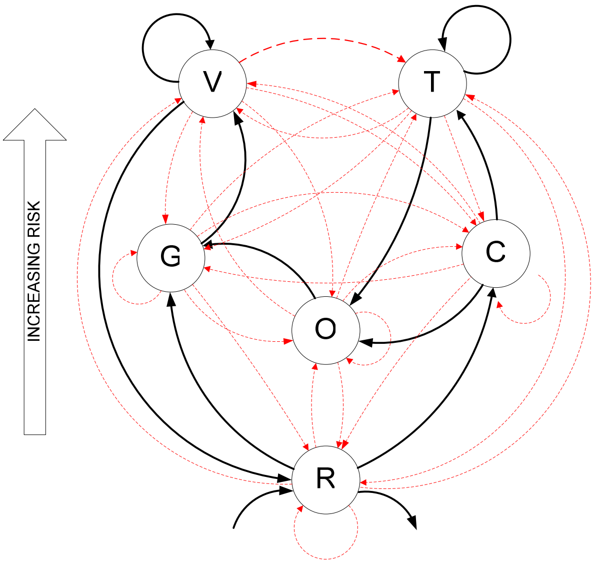


Figure 6.2: A CESG model of the railway level crossing with FEPs (dashed lines: FEPs)

Each FEP in FEP 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.,

$(RGV^{+})*RV$

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

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*. Table 6.1 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 Table 6.1.

Table 6.1: Level crossing vulnerabilities, the level of the faults 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	†

ES	FEP	Interpretation	Comment	Defense action
$(RGV^{+})^{*}RCT^{+}$	*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 accident	None
	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	†
	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	†
	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	None
	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.

6.3.4 Defense Mechanism and Risk Graph

To overcome the possible ambiguity in the descriptive nature of Table 6.1, a risk graph as in Figure 6.3 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 Definition 6.1, a directed edge from a state s_1 to s_2 in Figure 6.3 is equivalent to $s_1 \sqsubseteq s_2$, 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 $\sqsubseteq$ (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.

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 Figure 6.3 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 Table 6.1 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 Definition 6.2.

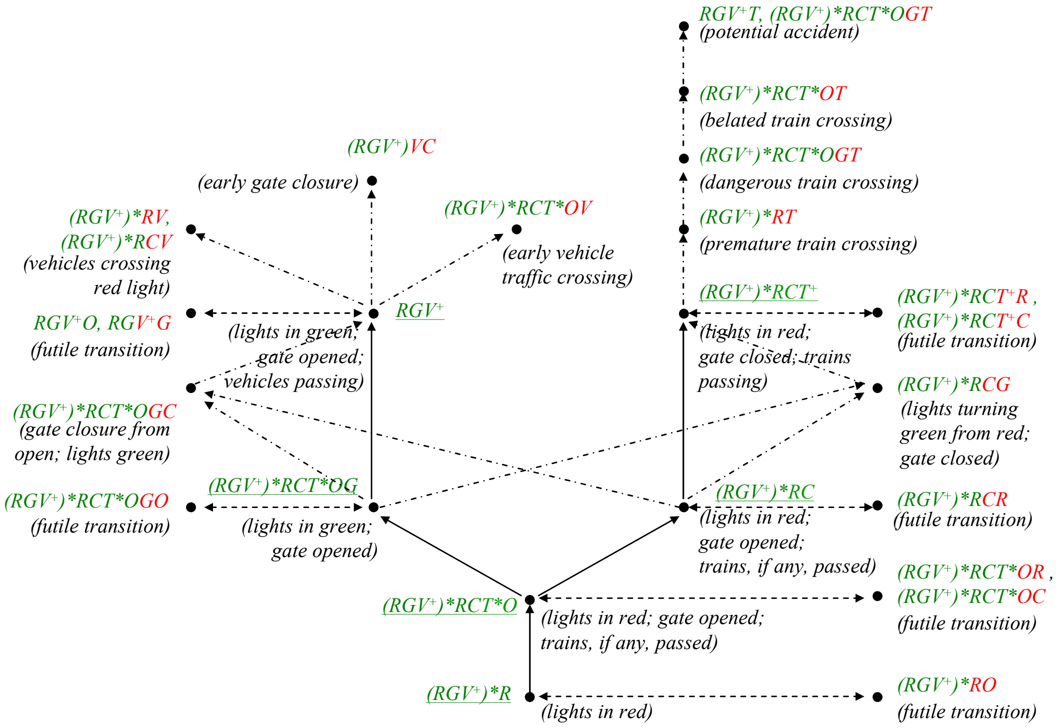


Figure 6.3: Risk graph of the railway crossing, covering both the system and vulnerability states

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 [IEC610, IEC615, Lev86].

6.3.5 Testing Issues

The test process can now be worked out analogous to that described in Section 4.2. Thus, the CES and FCES are systematically constructed and combined to cover the edges of the CESG as demonstrated in the introductory example in Chapter 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 $(RGV+)*RV$ 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 Figure 6.1.

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 Figure 6.2 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 Figure 6.2. It appears that following sets of event sequences, ESs, are of particular interest when dealing with system vulnerabilities

$$\mathrm{ES} = \{\ (RGV^{+})^{*}R,\ (RGV^{+})^{*}RC,\ (RGV^{+})^{*}RCT^{+},\ (RGV^{+})^{*}RCT^{*}O, \\ (RGV^{+})^{*}RCT^{*}OG,\ (RGV^{+})^{*}RCT^{*}OGV^{+},\ (RGV)^{*}RCT^{*}OGV^{+}R\ \}.$$

These ESs are possible prefixes, i.e., starters that can be constructed by analyzing the expression RE. 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 Table 6.1). 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 [RR97, Shn98], may be addressed in a similar manner.

6.4 Summary

This chapter illustrated the deployment of the risk ordering relation – an expression of relative levels of risks posed by hazardous states represents the degree of undesirability. The modeling and analysis of the example, railway crossing, illustrates the meaningful deployment of regular expressions, also for determination of defense actions, in an elegant, precise and concise way.

7 Case Study 2: Extending the Approach to Statecharts

Based on [Chr04/2] and [BBH05], this chapter applies the holistic approach described in the previous chapters to statecharts which extends conventional state-transition diagrams taking aspects of hierarchy, concurrency and communication into account [Har87].

7.1 Modeling Functions and Malfunctions

A statechart diagram compactly describes different constellations of states and transitions through which the system can proceed during its operation. Statecharts extend the conventional state-transition diagrams by adding notions of communication, hierarchy, concurrency, and history. They are de-facto-standardized in the *OMG Unified Modeling Language (UML) Specifications* [OMG03] which are also used in this work.

A statechart model of a system visualizes the system behavior in a clear and concise way. For an algebraic representation this work suggests extended regular expressions based on [Gar89, JS04, OAK03], the same way regular expressions adequately represent finite state automata.

Definition 7.1 (Extended Regular Expression): Let Σ be an alphabet that composes a set of symbols. The notion of regular expressions across Σ and the described sets of strings are extended to:

- When E and F are regular expressions, then $E \| F$ is a regular expression describing the *concurrency* of the languages $L(E)$ and $L(F)$, that is $L(E \| F) = L(E) \| L(F) = \{w \mid \exists e \in L(E) \; and \; \exists f \in L(F), \; w \in e \| f\}$ with $e \| \varepsilon = \varepsilon \| e = e, \forall e \in \Sigma$ and $a\,b \| c\,d = a(b \| c\,d) \cup c(a\,b \| d) \; \forall a, c \in \Sigma, b, d \in \Sigma^*$ with ε as the *empty string* denoting the set $\{\}$.
- When E is a regular expression and A a pseudo symbol representing the regular expression E, then the pseudo symbol A describes the language $L(E)$ that is $A=L(E)$ (A is handled in a regular expression as a symbol).
- A symbol s is a 3-tupel $s=(e,g,a)$ with *event e*, *guard g*, if existing, must be satisfied, and event *e* have been occurred, and *action a*, performed when *event e* occurs and *guard g* is satisfied.

Note that in the next sections the terms "regular expressions" and "extended regular expressions" are used interchangeably. Note also that the computational power of extended regular expressions is different than the one of regular expressions.

For transferring a statechart in an extended regular expression, each transition of this statechart will be denoted by a symbol s of the alphabet Σ. This regular expression, based on the alphabet Σ, is to be built by following rules.

(a) Sequential Transition

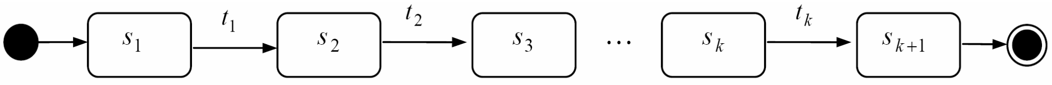


Figure 7.1: Sequential transitions

Figure 7.1 represents a *sequence* of state transitions in a statechart. In an extended regular expression, a sequence of transitions is denoted by the *concatenation* operator. For the statechart in Figure 7.1 the corresponding expression is

$$R = t_1 \, t_2 \ldots t_k$$

(b) Choice of Transition

A transition from a single state to a set of follow-on states forms a *choice* of transitions. In Figure 7.2, starting at the state s_1 enables a transition into one of the following states $s_2, \ldots, s_n$.

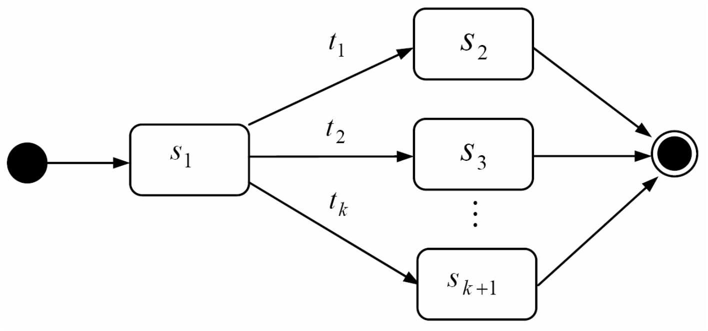


Figure 7.2: Choice of transitions

A choice of transitions is denoted by the *union* operator "+". The regular expression for the statechart in Figure 7.2 is given by

$$R = t_1 + t_2 + \ldots + t_k$$

(c) Transitions To and From States with Hierarchy and Concurrency

The transitions to and from the enclosing state form a sequence. In Figure 7.3, the transition t_k is followed by internal transitions; the sequence concludes with the transition t_l.

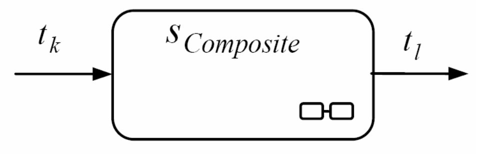


Figure 7.3: Transitions to and from composite states

Using the pseudo symbol $t_{Composite} = t_{Region\ 1} \,||\, \ldots \,||\, t_{Region\ n}$ the statechart in Figure 7.3 is given by

$$R = \ldots\ t_k\ t_{Composite}\ t_l$$

where $t_{Region\ i}$ with i=1,...,n denotes a regular expression that represents a sequence of internal transitions in region *i,* starting at the initial state and ending at any substate $s \in S_{Composite,\, Region\, i}$.

An enclosing state with one region describes a composite state with a single set of substates composed in a hierarchy. Thus, an enclosing state with more than one region describes a composite state of concurrent regions, each with a set of disjunctive substates.

(d) History State

A transition ending in a history state indicator 'H' can be represented by a set of *guarded* transitions to substates of the enclosing state. The guard has to be a variable that saves the last state the system was transferred into within the composite region. Therefore, all internal transitions of the history have to be extended by an action setting the variable on the source state of the internal transition (Figure 7.4). To resolve likely conflicts among multi-level transitions (t_4), the new transitions are indicated ($t_{41,}$ t_{42}).

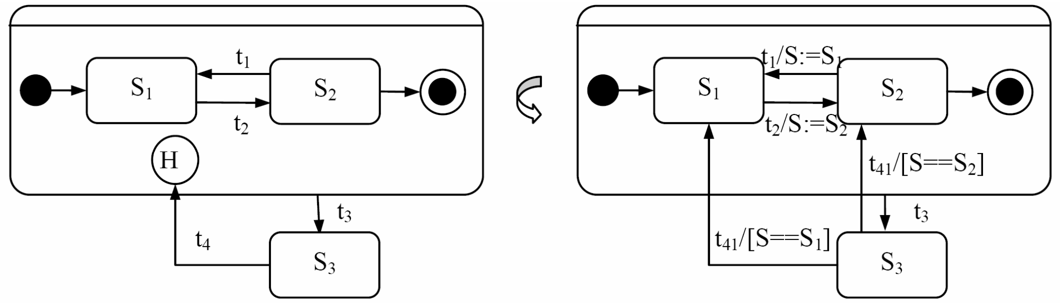


Figure 7.4: History state

Based on these four rules, any statechart can be converted into an extended regular expression by describing the sequences of transitions.

A statechart can be used for modeling interactive systems and analyzed to generate test cases. For modeling the *illegal*, i.e., undesirable user interactions the given statechart is to be complemented by *error states* and *faulty transitions* (Figure 7.5). The notations *error state* and *faulty transition* are used for explicitly describing the faulty behavior of the modeled system.

Faulty transitions run from each state of diagram to an error state caused by the events that trigger no (legal) transition in the context of this state. In Figure 7.5, only the (legal) transition t_3 can be triggered when the system is in state s_1. Therefore, the faulty transition from state s_1 to the *error state* is triggered by the faulty transition $t_{1,}$ $t_{2,}$ or t_4, if the transition set is given by $\{t_{1,} t_{2,} t_{3,} t_4\}$. The transitions represented by dashed lines are faulty ones.

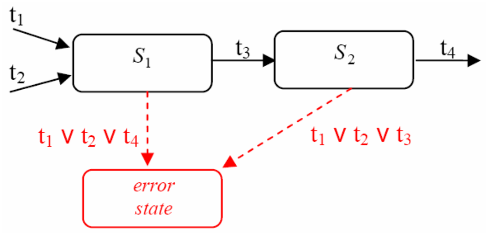


Figure 7.5: Fault model - error state and faulty transition

To generate the faulty guarded transitions the guards have to be negated, if existing. The test criteria introduced in Chapter 4 are based on the coverage of all n-tuples ($n \geq 2$) of legal and illegal events of user interactions. For statechart-based test case generation, other criteria are needed [OA99].

7.2 Test Criteria and Their Application to Statecharts

Based on the fault model introduced in the last section, legal and faulty transition pairs can be defined that are to be covered as a stopping rule of the test process.

Definition 7.2 (Transition Pair): A transition pair (TP) is a sequence of a legal incoming transition to a legal outgoing transition of a state.

Example 7.1: $t_1t_{3,}$ t_2t_3, t_3t_4 (Figure 7.5)

Definition 7.3 (Faulty Transition Pair): A faulty transition pair (FTP) is a sequence of a legal incoming transition to a faulty outgoing transition of a state.

Example 7.2: t_1t_1, t_1t_2, t_1t_4, t_2t_1, t_2t_2, t_2t_4, t_3t_1, t_3t_2, $t_3t_{3,}$ (Figure 7.5)

The notions of TP and FTP enable the definition of the following coverage criteria.

Definition 7.4 (Transition Pair Coverage): For any state of a statechart, generate test sequence(s) that sequentially conduct each TPs of any states.

Definition 7.5 (Faulty Transition Pair Coverage): For any error state of a statechart, generate test sequence(s) that sequentially conduct each FTPs of any state.

Definition 7.4 guarantees that all possible (legal) functions in each state of a system will be tested and the Definition 7.5 guarantees that all potential malfunctions, which can be derived from the given specification, will be tested. In order to generate tests, following rules realize the test criteria producing following application rules [Chr04/2].

Application Rule: Hierarchy.

(i) A transition to an enclosing state is equivalent to a transition into its initial substate.

(ii) A transition from an enclosing state is equivalent to the transitions from each of its substates.

(iii) The transition(s) that arose from (i) and (ii) must be taken into account when constructing legal and faulty transition pairs and test sequences according to Definition 7.4 and Definition 7.5.

Application Rule: Concurrency.

(i) Any transition within a region i of an enclosing state with concurrency have to be combined with any other transition in the regions j with $j \neq i$ to form (concurrent) transition pairs.

(ii) The transition(s) that arose from (i) must be taken into account when constructing legal and faulty transition pairs and test sequences according to Definition 7.4 and Definition 7.5.

Application Rule: History.

(i) A transition to a history state is equivalent to a guarded transition to any substate of the enclosing state. This guard enables the last state to be the enclosing state the system was within and to resume from.

(ii) The transition(s) that arose from (i) (and negative values of guards for faulty transitions) must be taken into account when constructing legal and faulty transition pairs and test sequences according to Definition 7.4 and Definition 7.5.

7.3 Test Case Generation

In the following, some definitions are informally introduced that are sufficient to describe the test generation algorithm.

A sequence of *n* consecutive (legal) states that represents the sequence of *n*+1 transitions is called a *transition sequence (TS) of the length n*+1, e.g., a *TP* (*transition pair*) is a TS of the length 2. A TS is *complete* if it starts at the initial state of the statechart diagram and ends at a final state; in this case it is called a *complete TS* (*CTS*).

A *faulty transition sequence* (*FTS*) *of the length n* consists of *n*-1 subsequent transitions that form a (legal) TS of the length *n*-2 plus a concluding, subsequent FTP (faulty TP). An FTS is *complete* if it starts at the initial state of the statechart diagram; in this case it is called *faulty complete TS*, abbreviated as *FCTS*.

The test criteria and the application rules introduced in Chapter 4 for identifying all potential incoming and outgoing transitions of a state enable the application of the approach. Generation and selection of test cases are carried out using the statechart of SUT or its equivalent extended regular expression.

It is assumed that an extended regular expression R over the alphabet Σ is given that describes a statechart. The symbols of Σ represent the set of transitions in the statechart diagram; the language $L(R)$ describes all (complete) correct sequences of transitions, i.e., *complete transition sequences* (*CTS*) in the statechart that are legal complete sequences of user interactions (*complete event sequences, CES*). Based on this set of transition sequences, all legal transition pairs (TP) can be identified by extracting all possible pairs of transitions given by the CTS. The remaining pairs of transitions given by the alphabet Σ form the set of faulty transition pairs (FTP). A FCTS is given by the beginning of a CTS and a concluding, subsequent FTP.

Based on the terminology introduced, Algorithm 7.1 below describes the test process.

Algorithm 7.1 (Test Generation and Execution):

Input: A statechart with
length:= required length of the transition sequences to be covered
Output: A test report of failed and succeeded test cases

```
Generate the completed statechart and apply the
rules for hierarchy, concurrency and history
FOR k:=2 TO length DO
BEGIN
  Cover all TSs of length k by means of CTSs
  subject to minimizing the sum of lengths of
  the CTSs
END
Cover all FTSs by means of FCTSs subject to
minimizing the sum of lengths of the FCTSs
Apply the test set given by the selected CTSs
and FCTSs to the SUT.
Observe the system output to determine whether
the system response is in compliance with the
expectation.
```

7.4 Testing the Marginal Strip Mower – RSM13

The objective of the case study is on the one hand to demonstrate that the introduced rules are sufficient to generate test cases and on the other hand to compare the effectiveness of test generation from ESG and statecharts.

7.4.1 System and Fault Model

The SUT used in the case study in this chapter is a control terminal which controls a marginal strip mower ("RSM 13", Figure 7.6), a sophisticated vehicle that takes optimum advantage of mowing around guide poles, road signs and trees, etc.

Due to the unique kinematics of the rotating point and due to the specially designed run of the mow head guide, even large areas behind the crash guides are reached. The shifting enables an exact adjustment of the mowing unit, even in very narrow areas. The mow head is protected against stoning due to its cutting system and the optimally arranged cutting units.

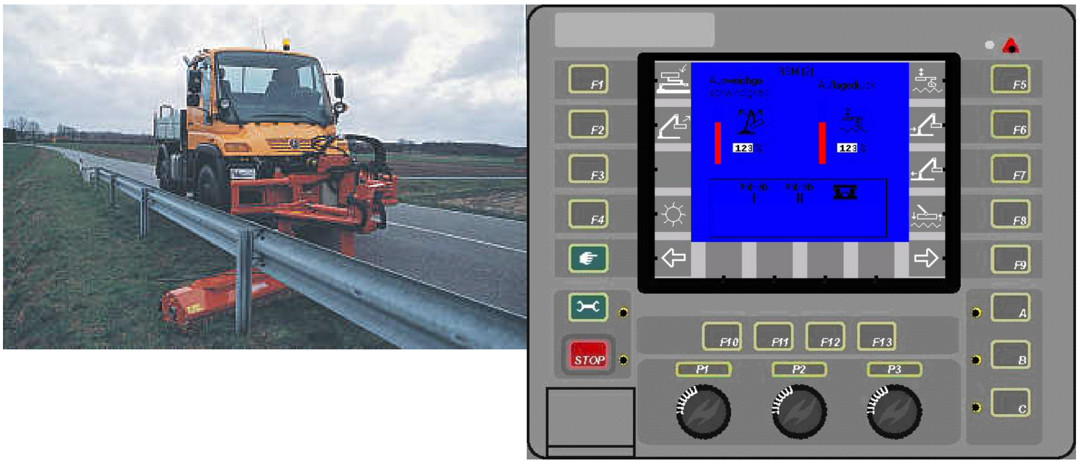


Figure 7.6: Marginal strip mower ("RSM 13") and its control desk

Operation is effected either by the power hydraulic of the light truck or by the front power take-off. The buttons on the control desk [Figure 7.6] simplify the operation, so that, e.g., the mow head returns to working position or to transport position when a button is pressed. The position of the mow head can also be infinitely varied. An alteration from working on the left to working on the right side is also possible. Beside the positioning the incline and support pressure of the mowing unit can be controlled.

7.4.2 Test Generation

The interactions between user and system can be modeled by the statechart given in Figure 7.7. Therein the substates are hidden and displayed as bars. The transitions to and/or from substates are indeed represented by so-called *stubbed transitions* [Har87]. The content of the hidden substates is illustrated in a separate statechart given in Appendix B. The dashed arcs in Figure 7.7 represent the pseudo symbol introduced in Definition 7.1 that denotes a sequence of internal transitions. Test cases are generated applying the rules introduced in Section 7.2.

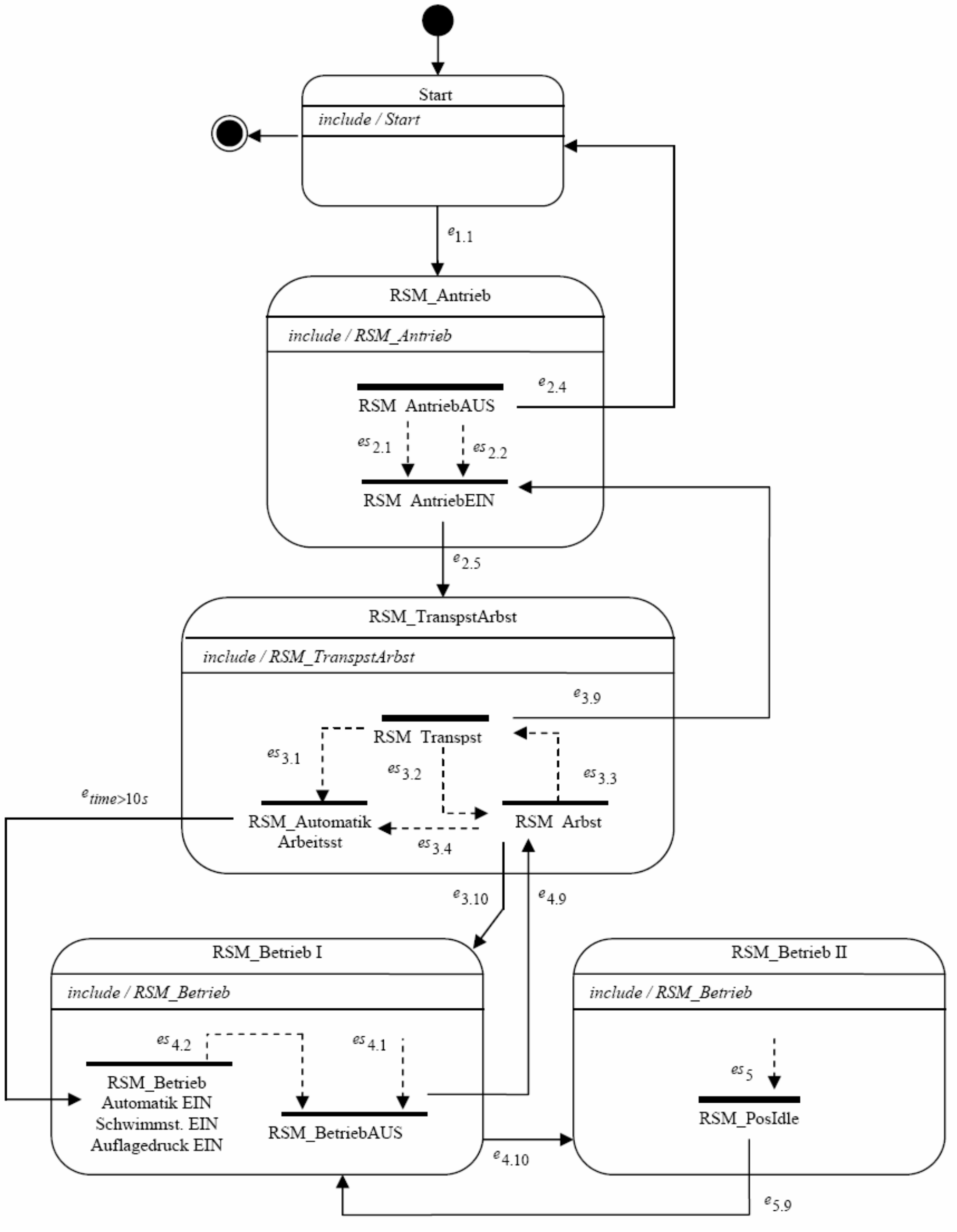


Figure 7.7: Statechart diagram of the control unit

The statechart diagram given in Figure 7.7 can be converted into an equivalent extended regular expression given as

$$\begin{aligned} R = (&e_{1.1}(es_{2.1}(e_{2.5}(es_{3.1}\ e_{time>10s}\ (e_{4.10}\, es_5\, e_{5.9})^*es_{4.2}(e_{4.10}\, es_5\, e_{5.9})^*e_{4.9} + es_{3.2}(e_{3.10}\ (e_{4.10}\, es_5 \\ &e_{5.9})^*(\lambda + es_{4.1})(e_{4.10}\, es_5\, e_{5.9})^*e_{4.9} + es_{3.4}\ e_{time>10s}\ (e_{4.10}\, es_5\, e_{5.9})^*es_{4.2}\ (e_{4.10}\, es_5\, e_{5.9})^* \\ &e_{4.9})^*es_{3.3})^*e_{3.9})^*es_{2.2})^*e_{2.4})^* \end{aligned}$$

with the corresponding extended regular expressions of the substates as

$$es_{2.1} = es_{2.2} = es_{Antrieb\ \ I} \parallel es_{Antrieb\ \ II} \parallel es_{Achsverr.}$$
$$es_{Antrieb\, I} = (e_{2.1}\, e_{2.1})^*e_{2.1}$$
$$es_{Antrieb\, II} = (e_{2.2}\ e_{2.2})^*e_{2.2}$$
$$es_{Achsverr.} = (e_{2.3}\, e_{2.3})^*e_{2.3}$$

$$es_{3.1} = es_{3.4} = (e_{3.1}\ e_{3.1,Release} + e_{3.2}\ e_{3.2,Release} + e_{ManPos})^*e_{3.2}$$
$$es_{3.2} = (e_{3.1}\ e_{3.1,Release} + e_{3.2}\ e_{3.2,Release} + e_{ManPos})^+e_{Arbst\_OK}$$
$$\begin{aligned} es_{3.3} = &(e_{3.1}\ e_{3.1,Release} + e_{3.2}\ e_{3.2,Release} + e_{ManPos})^*e_{3.1}\ e_{time>10s} \\ &+ (e_{3.1}\ e_{3.1,Release} + e_{3.2}\ e_{3.2,Release} + e_{ManPos})^+e_{Transpst\_OK} \end{aligned}$$
$$es_{ManPos} = es_{Arm2\ Ein-/\ Ausklappen} \parallel es_{Ein-/\ Ausklappen} \parallel es_{Hauptarm\ \ auf\,/\,ab}$$
$$es_{Arm2\ Ein-/\ Ausklappen} = (e_{3.3}\ e_{3.3,Release} + e_{3.4}\ e_{3.4,Release})^*$$
$$es_{Ein-/\ Ausklappen} = (e_{3.5}\ e_{3.5,Release} + e_{3.6}\ e_{3.6,Release})^*$$
$$es_{Hauptarm\ \ auf\,/\,ab} = (e_{3.7}\ e_{3.7,Release} + e_{3.8}\ e_{3.8,Release})^*$$

$$es_{4.1} = es_{Automatik(1)} \parallel es_{Auflagedruck,Messerwelle(1)} \parallel es_{Schwimmstellung(1)}$$
$$es_{Automatik(1)} = (e_{4.1}\ e_{4.1})^*$$
$$es_{Auflagedruck,Messerwelle(1)} = (e_{4.2}\ (e_{4.3}\ e_{4.3})^*e_{4.2})^*$$
$$es_{Schwimmstellung(1)} = (e_{4.4}\ e_{4.4})^*$$

$$es_{4.2} = es_{Automatik(2)} \parallel es_{Auflagedruck,Messerwelle(2)} \parallel es_{Schwimmstellung(2)}$$
$$es_{Automatik(2)} = (e_{4.1}\ e_{4.1})^*e_{4.1}$$
$$es_{Auflagedruck,Messerwelle(2)} = (e_{4.3}\ e_{4.3})^*e_{4.2}\ (e_{4.2}\ (e_{4.3}\ e_{4.3})^*e_{4.2})^*$$
$$es_{Schwimmstellung(2)} = (e_{4.4}\ e_{4.4})^*e_{4.4}$$

$$es_5 = es_{Ein-/\,Ausklappen} \,||\, es_{ArmVerschiebung} \,||\, es_{Mähkopf\ drehen}$$
$$es_{Ein-/\,Ausklappen} = (e_{5.1}\ e_{5.1,Release} + e_{5.2}\ e_{5.2,Release})*$$
$$es_{ArmVerschiebung} = (e_{5.3}\ e_{5.3,Release} + e_{5.4}\ e_{5.4,Release})*$$
$$es_{Mähkopf\ drehen} = (e_{5.5}\ e_{5.5,Release} + e_{5.6}\ e_{5.6,Release})*.$$

Table 7.1 lists the legal incoming and legal outgoing transition of each state. The analysis of the system by the statechart (Figure 7.7 and Table 7.1) necessitates some abstractions, entailing further refinements of the substates.

Table 7.1: Incoming and outgoing transitions of the statechart diagram in Figure 7.7

State	(Legal) Incoming transitions	(Legal) Outgoing transitions
Start	$e_{2.4}$	$e_{1.1}$
RSM_AntriebAUS	$e_{1.1}, es_{2.2}$	$e_{2.4}, es_{2.1}$
RSM_AntriebEIN	$es_{2.1}, e_{3.9}, e_{1.1}$	$e_{2.5}, es_{2.2}$
RSM_Transpst	$e_{2.5}, es_{3.3}$	$e_{3.9}, es_{3.1}, es_{3.2}$
RSM_Autom_Arbst	$es_{3.1}, es_{3.4}, e_{2.5}$	$e_{time>10s}$
RSM_Arbst	$e_{4.9}, es_{3.2}, e_{2.5}$	$es_{3.3}, es_{3.4}, e_{3.10}$
RSM_BetriebAUS	$e_{3.10}, e_{time>10s}, e_{5.7}$	$e_{4.9}, e_{4.10}$
RSM_Betrieb (AutomatikEIN Schwimmst.EIN AuflagedruckEIN)	$e_{3.10}, es_{4.2}, es_{4.1}, e_{5.7}$	$es_{4.2}, e_{4.10}$
RSM_PosIdle	$e_{4.10}, es_5$	$e_{5.7}$

Based on the statechart diagram given in Figure 7.7 all legal TPs can be identified for each state of the system. Table 7.1 lists the pairs of legal incoming and legal outgoing transition for the states of the substates. The set of TPs is generated by the cross product of incoming and outgoing transitions for each state to fulfill the transition pair coverage test criterion in Definition 7.2. For the state *RSM_Antrieb* in Figure 7.7 the resulting set of TPs is

$$\begin{aligned}\mathrm{TP}_{\mathrm{RSM\_Antrieb}} = \{&e_{1.1}e_{2.4}, e_{1.1}es_{2.1}, es_{2.2}e_{2.4}, es_{2.2}es_{2.1}, es_{2.1}e_{2.5,} es_{2.1}es_{2.2}, e_{3.9}e_{2.5}, e_{3.9}es_{2.2}, \\ &e_{1.1}e_{2.5}, e_{1.1}es_{2.2}\}\end{aligned}.$$

CTS can be constructed using the hierarchy application rules introduced in Section 7.2. As the same TPs can be covered by more than one CTS, a certain redundancy is likely. As an example, the TP $e_{1.1}es_{2.1}$ is itself already a CTS but also included another time in the CTS $e_{1.1}es_{2.1}e_{2.4}$. Test cases for a test specification in which those redundancies are eliminated for the state *RSM_Antrieb* are

$$\begin{aligned}\mathrm{CTS}_{\mathrm{RSM\_Antrieb}} = \{&e_{1.1}e_{2.4}, e_{1.1}es_{2.2}e_{2.4}, e_{1.1}es_{2.2}es_{2.1}, e_{1.1}es_{2.1}es_{2.2}, e_{1.1}e_{2.5}e_{3.9}e_{2.5}, e_{1.1}e_{2.5} \\ &e_{3.9}es_{2.2}\}\end{aligned}.$$

Accordingly, FTPs are generated by constructing all possible pairs if incoming and faulty outgoing transitions of each state of the statechart (Table 7.2).

Table 7.2: Incoming and faulty outgoing transitions of the statechart diagram in Figure 7.7

State	(Legal) Incoming transitions	Faulty outgoing transitions
Start	$e_{2.4}$	$e_{2.4}, e_{2.5}, e_{3.9}, e_{3.10}, e_{4.9}, e_{4.10}, e_{5.7}, es_{2.1}, es_{2.2}, es_{3.1}, es_{3.2}, es_{3.3}, es_{3.4}, es_{4.1}, es_{4.2}, es_{5}, e_{time>10s}$
RSM_AntriebAUS	$e_{1.1}, es_{2.2}$	$e_{1.1}, e_{2.5}, e_{3.9}, e_{3.10}, e_{4.9}, e_{4.10}, e_{5.7}, es_{2.2}, es_{3.1}, es_{3.2}, es_{3.3}, es_{3.4}, es_{4.1}, es_{4.2}, es_{5}, e_{time>10s}$
RSM_AntriebEIN	$es_{2.1}, e_{3.9}, e_{1.1}$	$e_{1.1}, e_{2.4}, e_{3.9}, e_{3.10}, e_{4.9}, e_{4.10}, e_{5.7}, es_{2.1}, es_{3.1}, es_{3.2}, es_{3.3}, es_{3.4}, es_{4.1}, es_{4.2}, es_{5}, e_{time>10s}$
RSM_Transpst	$e_{2.5}, es_{3.3}$	$e_{1.1}, e_{2.4}, e_{2.5}, e_{3.10}, e_{4.9}, e_{4.10}, e_{5.7}, es_{2.1}, es_{2.2}, es_{3.3}, es_{3.4}, es_{4.1}, es_{4.2}, es_{5}, e_{time>10s}$
RSM_Autom_Arbst	$es_{3.1}, es_{3.4}, e_{2.5}$	$e_{1.1}, e_{2.4}, e_{2.5}, e_{3.9}, e_{3.10}, e_{4.9}, e_{4.10}, es_{5.7}, es_{2.1}, es_{2.2}, es_{3.1}, es_{3.2}, es_{3.3}, es_{3.4}, es_{4.1}, es_{4.2}, es_{5}$

State	(Legal) Incoming transitions	Faulty outgoing transitions
RSM_Arbst	$e_{4.9}, es_{3.2}, e_{2.5}$	$e_{1.1}, e_{2.4}, e_{2.5}, e_{3.9}, e_{4.9}, e_{4.10}, es_{5.7}, es_{2.1}, es_{2.2}, es_{3.1}, es_{3.2}, es_{4.1}, es_{4.2}, es_{5}, e_{time>10}$
RSM_BetriebAUS	$e_{3.10}, e_{time>10s}, e_{5.7}$	$e_{1.1}, e_{2.4}, e_{2.5}, e_{3.9}, e_{3.10}, e_{5.9}, es_{2.1}, es_{2.2}, es_{3.1}, es_{3.2}, es_{3.3}, es_{3.4}, es_{4.2}, es_{5}, e_{time>10}$
RSM_Betrieb (AutomatikEIN Schwimmst.EIN AuflagedruckEIN)	$e_{3.10}, es_{4.2}, es_{4.1}, e_{5.7}$	$e_{1.1}, e_{2.4}, e_{2.5}, e_{3.9}, e_{3.10}, e_{4.9}, es_{5.7}, es_{2.1}, es_{2.2}, es_{3.1}, es_{3.2}, es_{3.3}, es_{3.4}, es_{4.1}, es_{5}, e_{time>10}$
RSM_PosIdle	$e_{4.10}, es_{5}$	$e_{1.1}, e_{2.4}, e_{2.5}, e_{3.9}, e_{3.10}, e_{4.9}, es_{4.10}, es_{2.1}, es_{2.2}, es_{3.1}, es_{3.2}, es_{3.3}, es_{3.4}, es_{4.1}, es_{4.2}, e_{time>10}$

A meaningful coverage criterion is given by the requirement that each of the FTPs given in Table 7.2 be executed by means of appropriate FCTSs (see "Faulty Transition Pair Test Criterion" in Section 7.2).

To execute a FTP, a *starter* (see Definition 3.9) is necessary that is a legal TS and starts at the initial state and ends at the state from where the faulty transition can be triggered. As an example the transition $e_{1.1}$ is a starter to reach the legal incoming transition of the state *Start*.

Thus, the sets of CTS and FCTS enable the coverage of the specified system functions and the malfunctions that can be derived by this specification.

7.5 Discussion: ESG vs. Statecharts

The case study was performed in two different ways to compare the fault detection capability of ESG vs. statechart modeling as introduced in this chapter. These different versions are called Case Study #1 and Case Study #2 that was carried out by one tester and two testers, respectively.

For the Case Study #1 the same tester created the ESGs and statecharts assuring that the models describe the same functionality of the SUT.

To take "exercising effects" into account, Case Study #1was performed twofold. First the tester started with the construction of statecharts and then the ESGs were constructed (Stage “A” in Table 7.3). Accordingly, Stage “B” was the other way around: first ESGs were created and then statecharts.

In the Case Study #2 different testers carried out the modeling job concurrently, constructing the ESGs and statecharts separately from each other, i.e., each tester created the model independently from each other.

Table 7.3 summarizes the results of both strategies. Note that about 50% of the faults have been detected by means of FCTS, i.e., complementary analysis [Chr04/2].

Table 7.3: Comparison of the fault detection capability of ESGs vs. statecharts

The sequence of the construction of ESGs and statecharts constructed		Faults detected only by ESG	Faults commonly detected by ESG and statecharts	Faults detected only by statecharts
#1	Stage A	-	48	-
	Stage B	2	46	-
#2		12	21	5

As summarized in Table 7.3, Case Study #1 detected 40+ faults (see Appendix B for the list of faults detected), no matter which model was constructed first.

Unexpectedly, constructing the statecharts and ESG separately by different testers (Case Study #2) lead to a smaller number of faults detected by statecharts than the number of faults detected by ESG. This can be explained easily: ESGs are simpler to be handled, and thus, the tester could work more efficiently, i.e., produce more and better detailed ESGs than statecharts, and accordingly, a better analysis and testing job could be performed.

Note that in the Case Study #2 the ESG model and the statechart model describe different functionalities of the SUT to avoid any biases in handling of the models.

To sum up, the comparison of the fault detecting capability of ESGs vs. statecharts could not point out any significant tendency but confirmed the effectiveness of the holistic approach when applied to different modeling methods.

This result is very important for the practice and apparently cannot be stressed strong enough: if the holistic approach is properly applied (no matter to ESGs or to statecharts), it reveals considerably more faults than an analysis that neglects the complementary view.

7.6 Summary

This chapter is based on [Chr04/2] and [BBH05] and applies the holistic approach to statecharts for modeling user-system interaction. For modeling the faulty system behavior, statechart given is complemented by an *error state*. In any non-erroneous, i.e., correct, state any other event than the legal transition transfers to the error state and forms a *faulty transition*. The test criteria for coverage of legal transition pairs and faulty transition pairs have been introduced and applied to a non-trivial control unit of a cutting machine.

Compared to statecharts, ESGs have limitations, primarily for representing complex notions of communication, hierarchy, concurrency, and history function. Nevertheless, different case studies have proven that they have about the same fault detection capability.

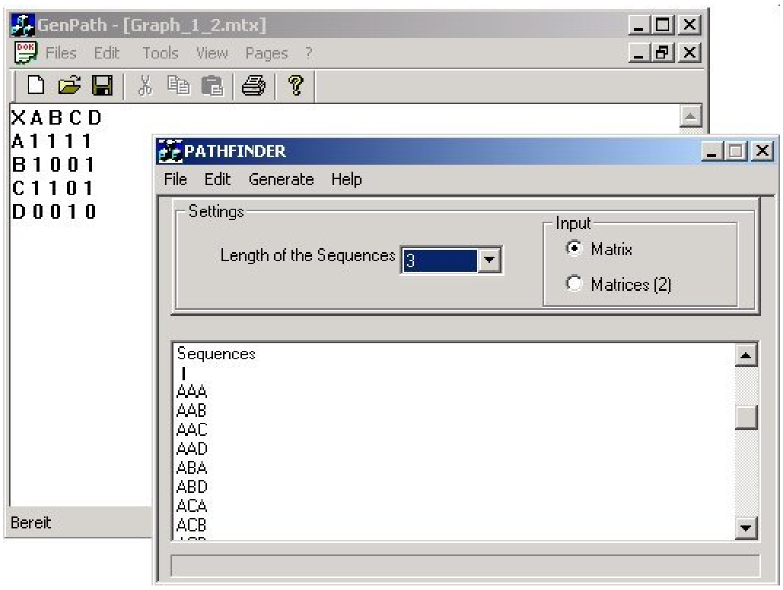
8 Tool Support

Generation of test cases from ESG and CESG and determination of the MSCESs and MSFCESs can be cumbersome and time consuming if done manually. This chapter describes a toolkit that helps with those tasks.

8.1 Test Case Generation

GenPath (GeneratePath) is a tool which is developed for the generation of test cases. The tool consists of two subsystems each of which can be opened in its own frame. One for the generation of ESs and FESs and the other one for the generation of MSCESs to achieve a given *n*-tuple event coverage. The input of the tool takes the ESG in the form of an adjacency matrix, or as a regular expression stored in a file. The main frame is mainly used to load the input or calculated output files. Several ESGs, which 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. Figure 8.1 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 one of the subsystems.

The other subsystem shown on the right-hand-side of Figure 8.2 generates MSCESs for an *n*-tuple coverage requirement [Hol04]. The main frame of GenPath depicts the result for the illustrated ESG. For this purpose the ESG given can be extended (Section 4.3.3) for generating sequences with length > 2. In addition, it displays the ESG under consideration and marks its EPs (Figure 8.2).



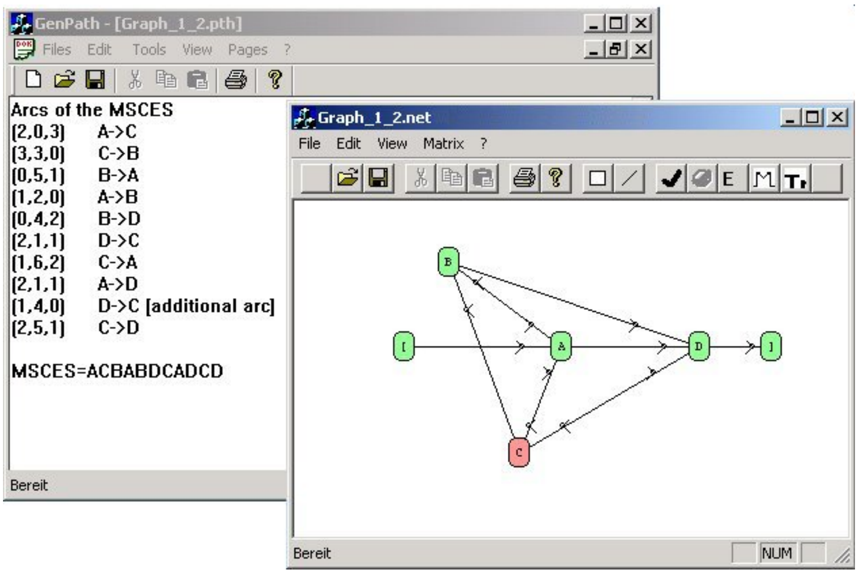
```
XABCD
A1111
B1001
C1101
D0010
```

Length of the Sequences: 3

Input: ● Matrix ○ Matrices (2)

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

Figure 8.1: Tool GenPath; mode to generate test cases



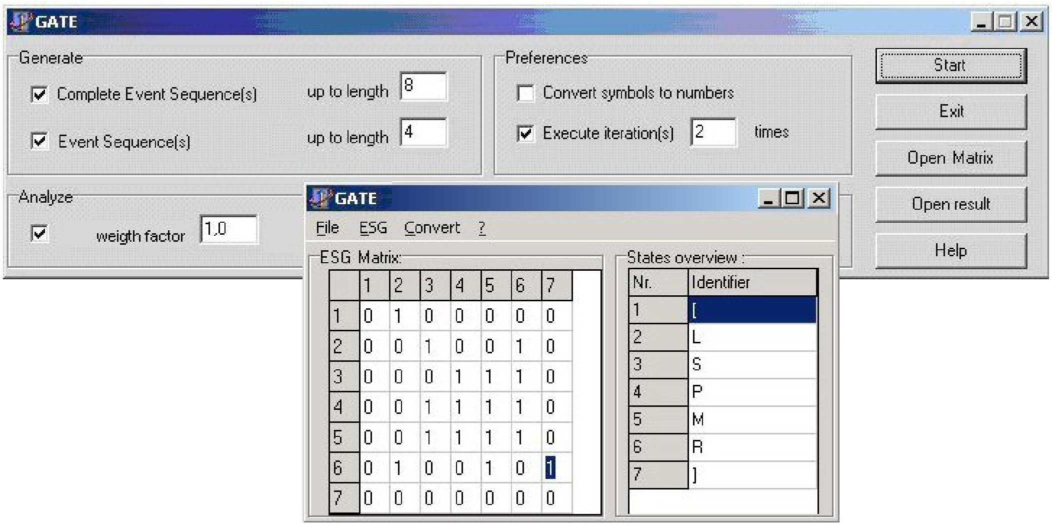
```
Arcs of the MSCES
(2,0,3)    A->C
(3,3,0)    C->B
(0,5,1)    B->A
(1,2,0)    A->B
(0,4,2)    B->D
(2,1,1)    D->C
(1,6,2)    C->A
(2,1,1)    A->D
(1,4,0)    D->C [additional arc]
(2,5,1)    C->D

MSCES=ACBABDCADCD
```

Figure 8.2: GenPath to generate MSCES

8.2 Test Case Analysis

To avoid tedious and error-prone manual work, the tool *GATE* (Generation and Analysis of Test Event sequences) was built [Chr04]. The tool accepts the adjacency matrix of the ESG and automatically generates test inputs of required length for a given ESG, i.e., ES, FES, CES, and FCES. As a first step, all ESs of a given length are produced. The tool can generate also CES of a given length which cover all ESs. Furthermore, GATE determines the effectiveness of a given test case in terms of covered ES. If no length is explicitly given, the tool constructs a CES of minimal length that covers all ESs.



Generate

☑ Complete Event Sequence(s) up to length 8

☑ Event Sequence(s) up to length 4

Preferences

□ Convert symbols to numbers

☑ Execute iteration(s) 2 times

Analyze

☑ weigth factor 1.0

ESG Matrix:

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	1	0	0	1	0
3	0	0	0	1	1	1	0
4	0	0	1	1	1	1	0
5	0	0	1	1	1	1	0
6	0	1	0	0	1	0	1
7	0	0	0	0	0	0	0

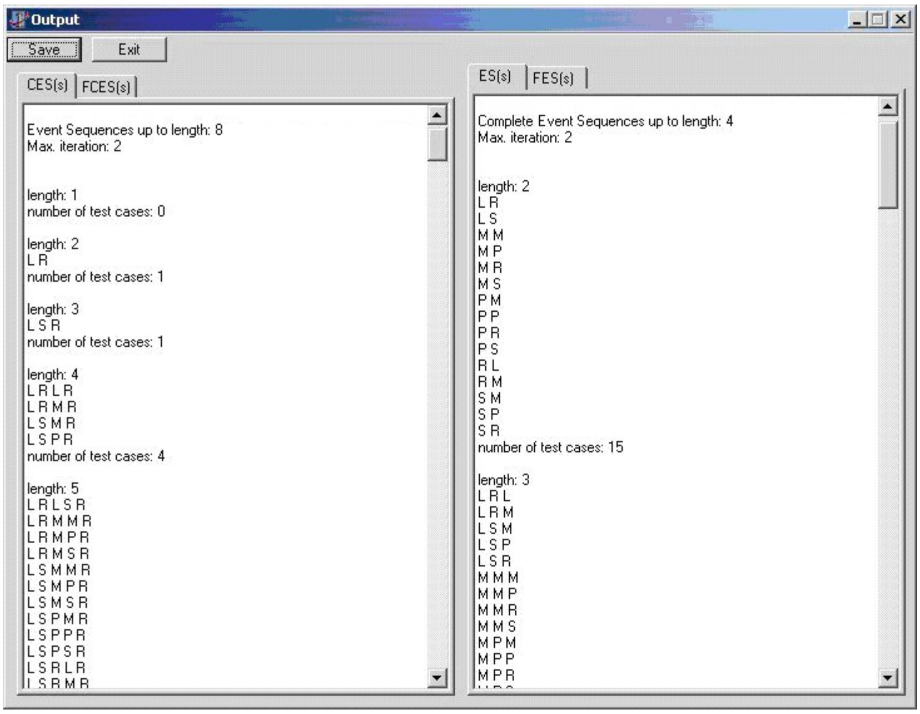
States overview :

Nr.	Identifier
1	[
2	L
3	S
4	P
5	M
6	R
7	]

Figure 8.3: Test tool GATE for test input generation of an ESG

Figure 8.3 depicts the main frame window for test generation. For the example ESG given by its adjacency matrix in Figure 8.3, GATE generates CESs of up to length 8 and ESs of up to length 4. The tester requires in this example that loops be run twice. Furthermore, the weight factor β is set to 1.0 (see Definition 4.3).

Figure 8.4 depicts the output of a test case set which is analyzed in Figure 8.5. GATE generates ESs (or FESs) and CESs (or FCESs) of different length, depicted on the right half and left half of the Figure 8.4, respectively.



```
Event Sequences up to length: 8
Max. iteration: 2


length: 1
number of test cases: 0

length: 2
L R
number of test cases: 1

length: 3
L S R
number of test cases: 1

length: 4
L R L R
L R M R
L S M R
L S P R
number of test cases: 4

length: 5
L R L S R
L R M M R
L R M P R
L R M S R
L S M M R
L S M P R
L S M S R
L S P M R
L S P P R
L S P S R
L S R L R
L S R M R
```

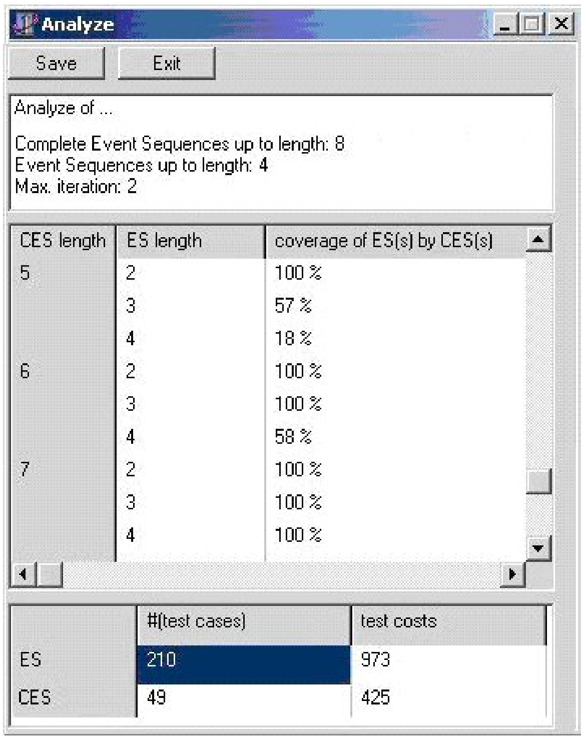
```
Complete Event Sequences up to length: 4
Max. iteration: 2


length: 2
L R
L S
M M
M P
M R
M S
P M
P P
P R
P S
R L
R M
S M
S P
S R
number of test cases: 15

length: 3
L R L
L R M
L S M
L S P
L S R
M M M
M M P
M M R
M M S
M P M
M P P
M P R
```

Figure 8.4: Test cases generated for ES/FES and CES/FCES

The results of the analysis of the ESG given in Figure 8.3 are summarized in Figure 8.5. The goal of the analysis is the determination of the coverage ratio of ESs (column 2) by appropriate CESs (column 1) in increasing order of their length. As an example, the first line (middle part of the Figure 8.5) indicates that 100% of EPs (as ESs of length 2) are covered by CESs of length 5. These CESs cover, however, only 57% of the event triple (as ESs of length 3). CESs of length 7 cover 100% of ESs of length up to 4. The bottom part of Figure 8.5 calculates for ESs and CESs the number of test cases in accordance with Definition 4.3. These ESs and CESs have been previously determined by the tool GATE as demonstrated in Figure 8.4. Finally, the test costs are given in the right column.



Analyze of ...

Complete Event Sequences up to length: 8
Event Sequences up to length: 4
Max. iteration: 2

CES length	ES length	coverage of ES(s) by CES(s)
5	2	100 %
	3	57 %
	4	18 %
6	2	100 %
	3	100 %
	4	58 %
7	2	100 %
	3	100 %
	4	100 %

	#(test cases)	test costs
ES	210	973
CES	49	425

Figure 8.5: Analysis of the generated test cases

8.3 More Automation – Towards Self-Testing

For testing of the software systems powerful commercial capture/replay tools are available which can significantly support the testing process, including regression testing. Nevertheless, numerous, time consuming and error-prone manual steps are still necessary to complete the test process, i.e., construction of adequate, user-oriented test scripts based on special test cases.

8.3.1 The Principle

Figure 8.6 identifies typical activities to be carried out while testing an interactive system that is supported by a test tool as follows:

- Capture all manual user-interactions, i.e., include all necessary object properties of each selected GUI object resulting in a “fully” recorded test script.

- Replay the recorded test scripts and analyze them to detect anomalies.

For automatically testing a SUT, the typical test process, as represented in Figure 8.6, does not consider the fact that the tester usually has a system model (specification) as a reference which he, or she, has to check against the recorded behavioral model of the SUT.

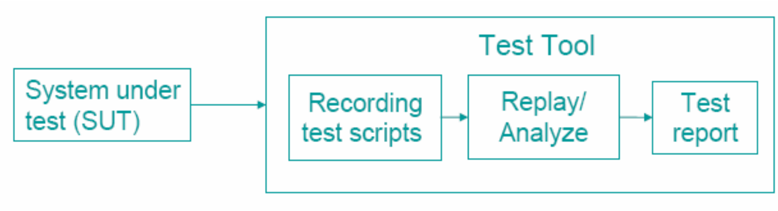


Figure 8.6: The common way of test tool-supported GUI testing

Figure 8.7 identifies activities that are necessary to be carried out for this comparison in following steps. The identification of the GUI objects of the SUT as well as the system model are necessary inputs that result in an executable test script as output.

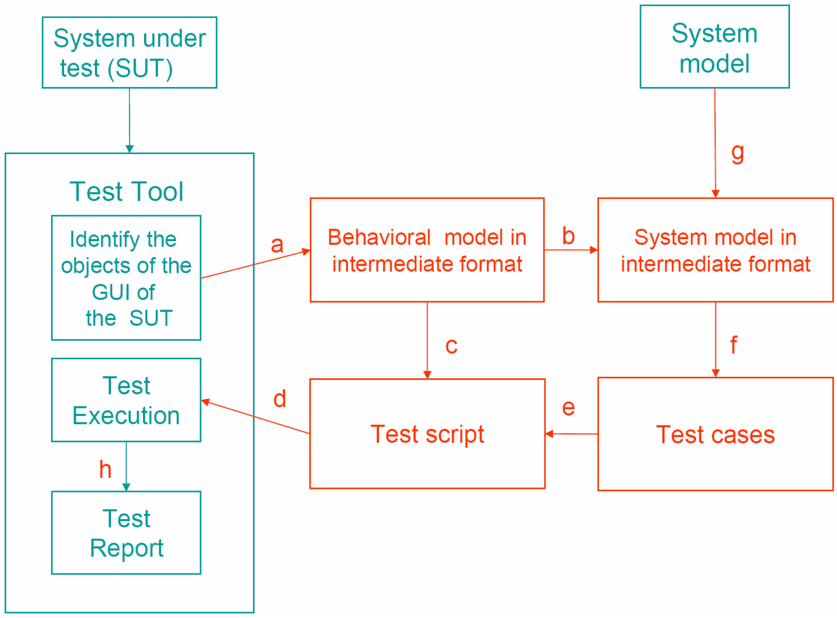

System under test (SUT)
System model
g
Test Tool
Identify the objects of the GUI of the SUT
a
Behavioral model in intermediate format
b
System model in intermediate format
c
f
Test Execution
d
Test script
e
Test cases
h
Test Report

Figure 8.7: Identification of the steps to be automatically carried out

The *GUI objects* are the instruments for the input, e.g., screens, windows, icons, menus, pointers, commands, function keys, alphanumerical keys, etc.

After recording of the properties of the GUI objects of the SUT through the test tool, the relevant information on the structure of the SUT is stored in a GUI file. Apart from the identification of the GUI objects, the step *a* (Figure 8.7) converts the GUI file, which incorporates the *behavioral model* of the SUT, in an intermediate, uniform format for further processing.

As a next step, the two inputs *g* and *b* (Figure 8.7) are used to convert the *system model* into the same intermediate format as the behavioral model has been captured by step *a*. This format unambiguously represents the GUI objects and the model of the SUT by mapping the (by the step *a*) identified objects to the objects of the model.

Step *f* generates test cases by analyzing the system model given in the intermediate format. Moreover, the test cases and the behavioral model (in the intermediate format) are necessary as input *c* and *e* to subsequently generate a test script in the appropriate format, which can be executed by the test tool.

Finally, the step *d* loads the test script into the test tool that executes the test and generates a test report (step *h*).

There is a good deal of research work done for modeling of interactive systems and generation of test cases from this model (steps *g* and *f* in Figure 8.6) [BL02, Hor99, PFV03]. The model can be state-based, e.g., as UML statecharts [OMG03], or event-based, e.g., event sequence graphs [BB04, Bel01]. The intermediate format can be XML [XML10]. Thus, the steps *g*, *f*, and *d* (Figure 8.7) can effectively be performed, based on sound formal methods.

The steps *b*, *c*, and *e*, however need additional effort for automation which is explained in following sections. As the test method selected is black-box-oriented, the source code (implementation) of the SUT is not needed for testing. Thus, the testing of the SUT solely needs domain knowledge, i.e., the information represented in the system model which is the output of the step *g*.

8.3.2 Test Script Generation

To enable the comparison for the identification of the objects, both the behavioral model and the system model are to be transformed into an intermediate format, e.g., XML. The approach is not applicable if the objects do not completely match and thus an unambiguous identification of the objects is not possible. However, it has been working on heuristics to determine a notion of "similarity degree" to enable the identification of a common subset of the objects of both models that could enable an automatic testing of the software system covering the corresponding objects.

The objects of an application can be identified by a *spy* feature that is available in most commercial test tools. Actually, this feature delivers the information that is gained during the recording of a system-based application, in a specific context that includes specific properties at the specific point of recording time. This means, the information that is available for recognizing the objects is *static*, i.e., it has been gained under specific circumstances. However, objects have *dynamic* properties that change, e.g., a button that is disabled can be enabled in another context – e.g., after the recording stops. The capturing*,* that has recorded this button in a disabled state, can than recognize the button if and only if it is disabled.

Generally, such a button as an object has more than only one property, e.g., window class, label, etc., whereas for this approach only active properties are of interest. Therefore, a list of objects of an application is necessary, and all instances of these objects, e.g., of a specific button, are to be included in this list. The recorded objects are usually hierarchically structured by the windows they belong to. Figure 8.8 depicts the captured objects of an application by the test tool WinRunner [WR70]. Consider, however, that the tool features used in the examples are included in most commercial capture/replay tools.

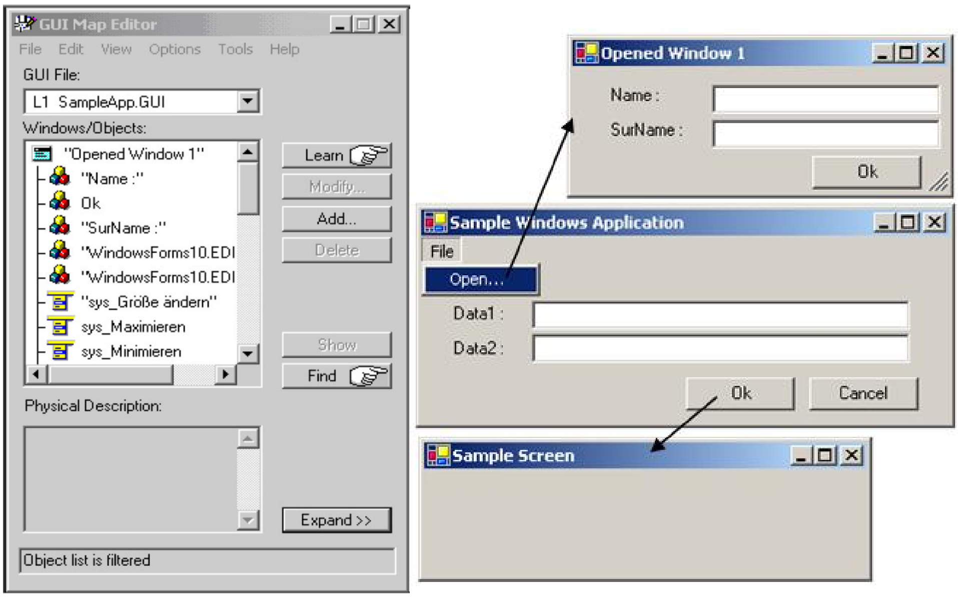
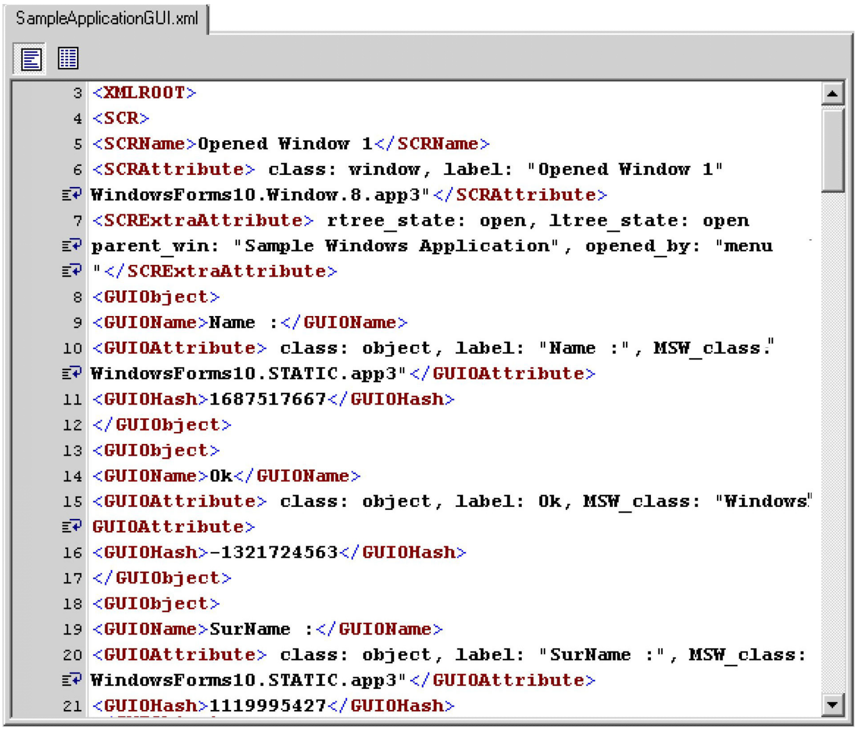


Figure 8.8: A sample application and their captured objects by a commercial test tool

The GUI Map Editor of WinRunner includes a feature for saving of a recorded window and its objects as a textual GUI file. This file contains the different instances each object can possess. For identification, a name tag can be assigned to each of these files, e.g., determined by the hashing method. To conclude the step *a* of (Figure 8.7), the GUI file is converted into a common format, as to XML, which many tools are compatible with. Figure 8.9 depicts the extended GUI file of the object of the "Sample Windows Application" given in Figure 8.8 which semantically corresponds to all following figures.

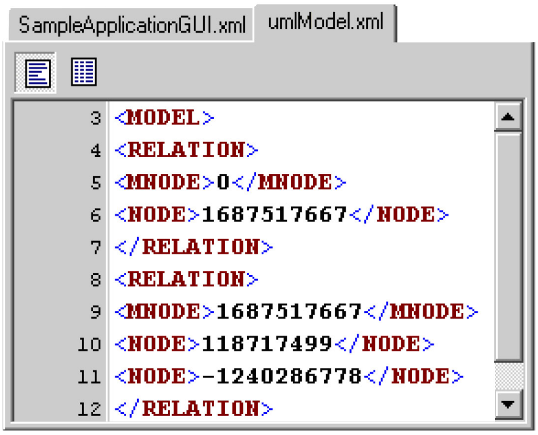


```
<XMLROOT>
<SCR>
<SCRName>Opened Window 1</SCRName>
<SCRAttribute> class: window, label: "Opened Window 1"
WindowsForms10.Window.8.app3"</SCRAttribute>
<SCRExtraAttribute> rtree_state: open, ltree_state: open
parent_win: "Sample Windows Application", opened_by: "menu
"</SCRExtraAttribute>
<GUIObject>
<GUIOName>Name :</GUIOName>
<GUIOAttribute> class: object, label: "Name :", MSW_class:"
WindowsForms10.STATIC.app3"</GUIOAttribute>
<GUIOHash>1687517667</GUIOHash>
</GUIObject>
<GUIObject>
<GUIOName>Ok</GUIOName>
<GUIOAttribute> class: object, label: Ok, MSW_class: "Windows"
GUIOAttribute>
<GUIOHash>-1321724563</GUIOHash>
</GUIObject>
<GUIObject>
<GUIOName>SurName :</GUIOName>
<GUIOAttribute> class: object, label: "SurName :", MSW_class:
WindowsForms10.STATIC.app3"</GUIOAttribute>
<GUIOHash>1119995427</GUIOHash>
```

Figure 8.9: Extended GUI file in XML format

For generating test cases automatically, a model of the system is necessary, e.g., a statechart or an ESG that describes which event can be reached from which other event. This model is stored also in an intermediate format.

Figure 8.10 depicts the corresponding ESG as a XML file which is generated by a tool developed by our group as a WinRunner "Add-on".



```
<MODEL>
<RELATION>
<MNODE>0</MNODE>
<NODE>1687517667</NODE>
</RELATION>
<RELATION>
<MNODE>1687517667</MNODE>
<NODE>118717499</NODE>
<NODE>-1240286778</NODE>
</RELATION>
```

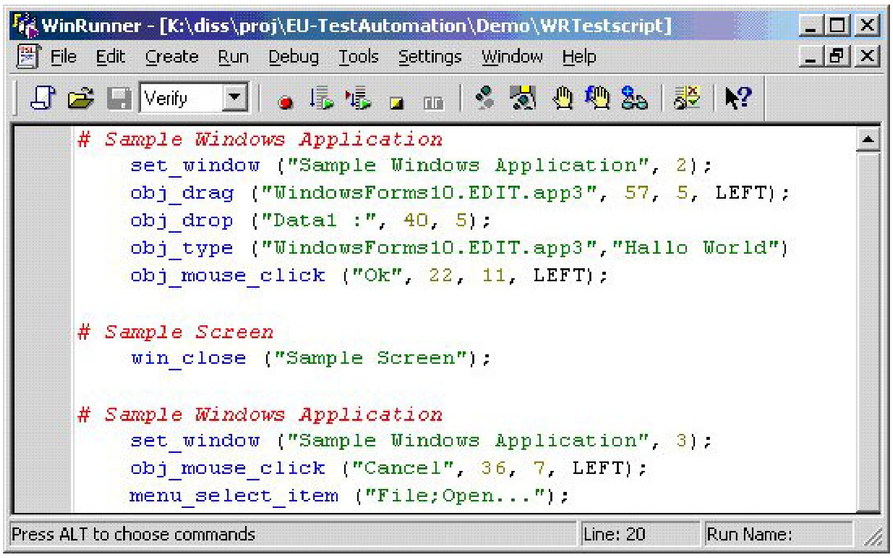
Figure 8.10: XML representation of the model given as an ESG using the "hashed" IDs

Once the model is available in the intermediate format, test cases can easily be generated, e.g., using the technique introduced in Chapter 4. For the XML file given in Figure 8.10, the following example excerpts some of the test cases that are automatically generated:

Example 8.1: Hash coded identifiers of the test cases that are to be executed in a test run sequentially.
[-1321724563,-1240296778,118717499,1687517667]
[-1321724563,-1240296778,-1687517667]

The test sequences that are automatically generated must be converted into the test script language of the selected test tool to enable the execution and analysis in its replay mode. The conversion is a straight-forward translation, needing only little amount of additional information that is not included in the generated test sequence: information about the hierarchical structure of the windows, i.e., whether the objects are to be sequentially executed (i.e., they belong to the same window), or another window is to be activated (opened) before the next object is to be executed.

In WinRunner, this information can be added by means of the command "set_window()" to activate another window so that the following object can be reached. At last the information input parameters for editable objects must be set by the command "edit_set()" as "textfield".


WinRunner - [K:\diss\proj\EU-TestAutomation\Demo\WRTestscript]
File Edit Create Run Debug Tools Settings Window Help
Verify
Press ALT to choose commands
Line: 20
Run Name:

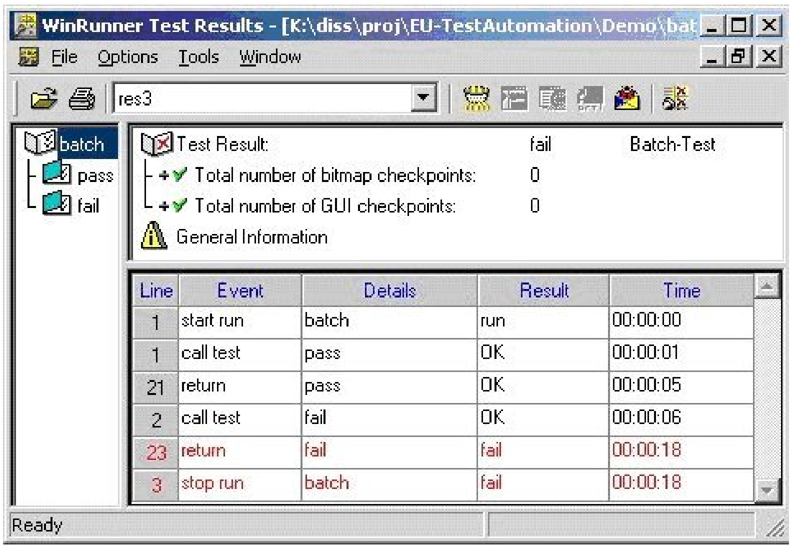
```
# Sample Windows Application
    set_window ("Sample Windows Application", 2);
    obj_drag ("WindowsForms10.EDIT.app3", 57, 5, LEFT);
    obj_drop ("Data1 :", 40, 5);
    obj_type ("WindowsForms10.EDIT.app3","Hallo World")
    obj_mouse_click ("Ok", 22, 11, LEFT);

# Sample Screen
    win_close ("Sample Screen");

# Sample Windows Application
    set_window ("Sample Windows Application", 3);
    obj_mouse_click ("Cancel", 36, 7, LEFT);
    menu_select_item ("File;Open...");
```

Figure 8.11: Executable Test Script

Figure 8.11 depicts the generated test script which can be replayed in WinRunner, whereas Figure 8.12 indicates the corresponding test report on failed and succeeded test cases.


WinRunner Test Results - [K:\diss\proj\EU-TestAutomation\Demo\bat
File Options Tools Window
res3
batch
pass
fail
Ready

Test Result: fail Batch-Test
Total number of bitmap checkpoints: 0
Total number of GUI checkpoints: 0
General Information

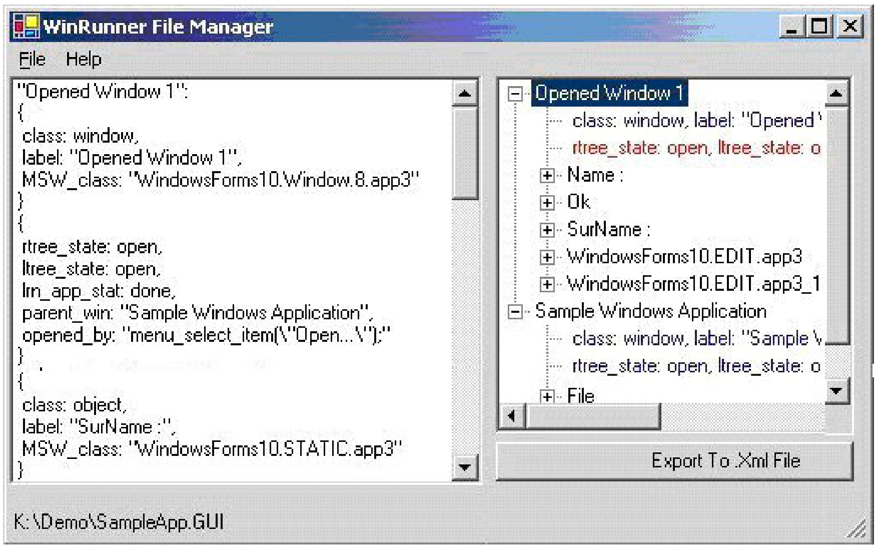
Line	Event	Details	Result	Time
1	start run	batch	run	00:00:00
1	call test	pass	OK	00:00:01
21	return	pass	OK	00:00:05
2	call test	fail	OK	00:00:06
23	return	fail	fail	00:00:18
3	stop run	batch	fail	00:00:18

Figure 8.12: Test report generated for the test script of Figure 8.11

8.3.3 A Prototype of the Environment

The implemented test tools aim to automate the manual steps of the deployment of commercial test tools during testing the SUT. Following, self developed test tool components are briefly introduced.

The *GUI File Parser* summarizes the information on the GUI structure. The original GUI File of WinRunner is on the left side of Figure 8.13; the extended GUI file on the right side lists all objects as a tree. This file is converted to an XML file upon pressing the Button “Export to XML File”.



```
"Opened Window 1":
{
 class: window,
 label: "Opened Window 1",
 MSW_class: "WindowsForms10.Window.8.app3"
}
{
 rtree_state: open,
 ltree_state: open,
 lrn_app_stat: done,
 parent_win: "Sample Windows Application",
 opened_by: "menu_select_item(\"Open...\");"
}
{
 class: object,
 label: "SurName :",
 MSW_class: "WindowsForms10.STATIC.app3"
}
```

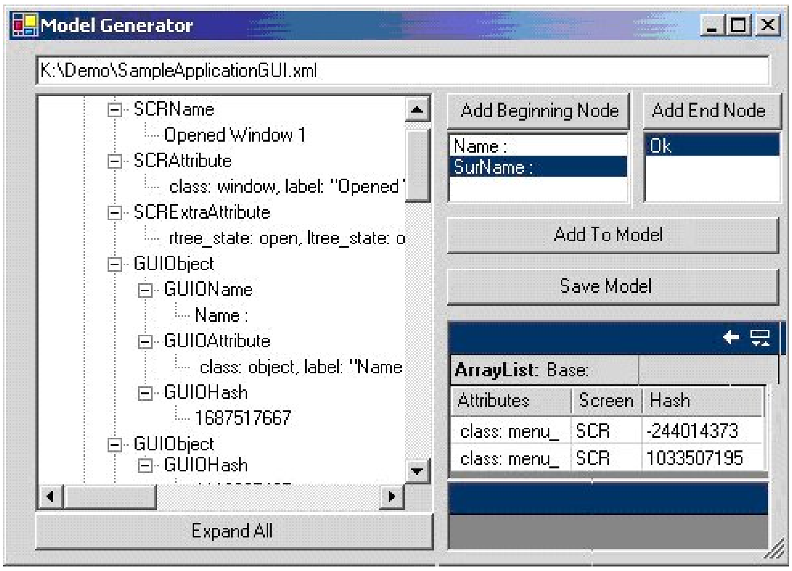
```
⊟ Opened Window 1
    class: window, label: "Opened '
    rtree_state: open, ltree_state: o
  ⊞ Name :
  ⊞ Ok
  ⊞ SurName :
  ⊞ WindowsForms10.EDIT.app3
  ⊞ WindowsForms10.EDIT.app3_1
⊟ Sample Windows Application
    class: window, label: "Sample \
    rtree_state: open, ltree_state: o
  ⊞ File
```

K:\Demo\SampleApp.GUI

Figure 8.13: The GUI File Parser

The *Model Generator* opens a previously generated XML file of a GUI for mapping of its objects to a model (represented also as a XML file). From the left side of Figure 8.14, the objects can be “dragged and dropped” and inserted as an end or starting node. Beginning with the end node, the starting node column has to be filled by those objects from which the end node can be reached. By pressing the “Add To Model” button the selected component is added to the model.

Once the model of the SUT is generated by the Model Generator, the GUI file containing the structure information is used to decompose all modal windows. Thus, each modal window is represented by its own ESG, and can separately be tested by generating the corresponding MSFCES.



K:\Demo\SampleApplicationGUI.xml

```
⊟ SCRName
    Opened Window 1
⊟ SCRAttribute
    class: window, label: "Opened '
⊟ SCRExtraAttribute
    rtree_state: open, ltree_state: o
⊟ GUIObject
  ⊟ GUIOName
      Name :
  ⊟ GUIOAttribute
      class: object, label: "Name
  ⊟ GUIOHash
      1687517667
⊟ GUIObject
  ⊟ GUIOHash
```

Name :
SurName :

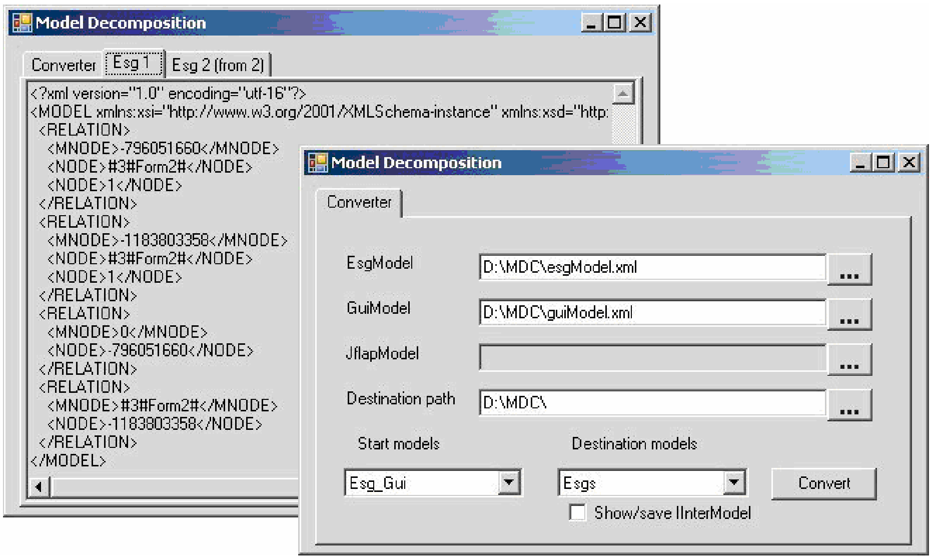
Ok

ArrayList: Base:

Attributes	Screen	Hash
class: menu_	SCR	-244014373
class: menu_	SCR	1033507195

Figure 8.14: The Model Generator

As shown in Figure 8.15, one needs only to push “Convert-Button” to start the *Model Decomposer* [Sto05] that generates an ESG for each modal window.

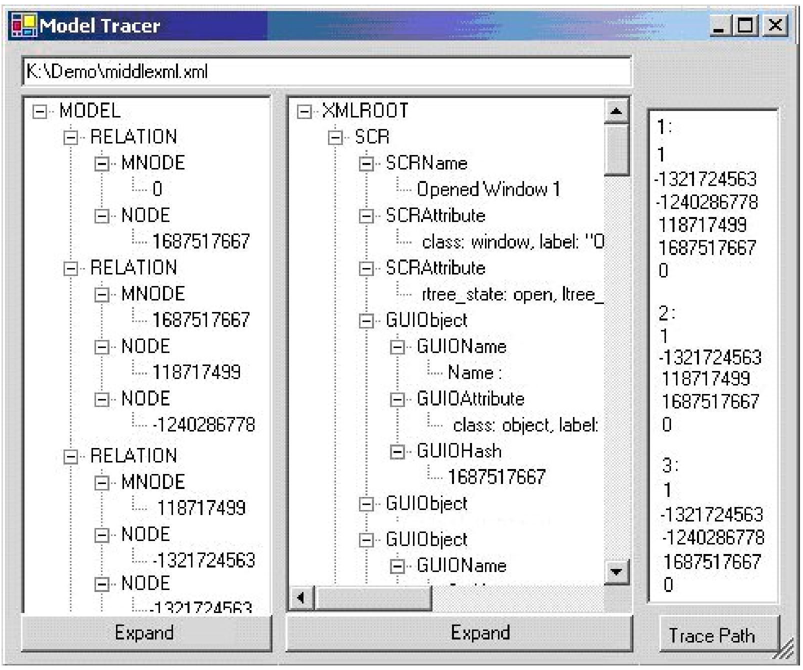


```
<?xml version="1.0" encoding="utf-16"?>
<MODEL xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http:
 <RELATION>
  <MNODE>-796051660</MNODE>
  <NODE>#3#Form2#</NODE>
  <NODE>1</NODE>
 </RELATION>
 <RELATION>
  <MNODE>-1183803358</MNODE>
  <NODE>#3#Form2#</NODE>
  <NODE>1</NODE>
 </RELATION>
 <RELATION>
  <MNODE>0</MNODE>
  <NODE>-796051660</NODE>
 </RELATION>
 <RELATION>
  <MNODE>#3#Form2#</MNODE>
  <NODE>-1183803358</NODE>
 </RELATION>
</MODEL>
```

EsgModel D:\MDC\esgModel.xml
GuiModel D:\MDC\guiModel.xml
JflapModel
Destination path D:\MDC\
Start models: Esg_Gui
Destination models: Esgs
□ Show/save IInterModel

Figure 8.15: The add-on tool *Model Decomposer*

The end node column can only contain a single node as an end node. When all objects are mapped, the model can be saved. The generated model can be checked in a frame (down, right side) where each object is represented by its hash coded ID.



K:\Demo\middlexml.xml

```
⊟ MODEL
   ⊟ RELATION
      ⊟ MNODE
          0
      ⊟ NODE
          1687517667
   ⊟ RELATION
      ⊟ MNODE
          1687517667
      ⊟ NODE
          118717499
      ⊟ NODE
          -1240286778
   ⊟ RELATION
      ⊟ MNODE
          118717499
      ⊟ NODE
          -1321724563
      ⊟ NODE
          -1321724563
```

```
⊟ XMLROOT
   ⊟ SCR
      ⊟ SCRName
          Opened Window 1
      ⊟ SCRAttribute
          class: window, label: "O
      ⊟ SCRAttribute
          rtree_state: open, ltree_
      ⊟ GUIObject
         ⊟ GUIOName
             Name :
         ⊟ GUIOAttribute
             class: object, label:
         ⊟ GUIOHash
             1687517667
      ⊟ GUIObject
      ⊟ GUIObject
         ⊟ GUIOName
```

```
1:
1
-1321724563
-1240286778
118717499
1687517667
0

2:
1
-1321724563
118717499
1687517667
0

3:
1
-1321724563
-1240286778
1687517667
0
```

Figure 8.16: The Model Tracer

The *Model Tracer* generates test sequences from the XML file of a GUI and XML file of the corresponding model as described in Section 8.3.2. At the same time when the test cases are generated, which is shown on the right column in Figure 8.16 (cf. to the example in Section 8.3.2), the executable test script is saved in a separate directory.

8.4 Remarks for Further Research and Development

If the SUT is modified, e.g., to produce a new release, the test frame, i.e., the test cases, and consequently, the test script might become obsolete. To avoid this obsolescence, the underlying system model is to be automatically updated which defines an important area of further research.

A key factor of the approach is the unambiguous identification of the objects of the SUT – both in the system model and the behavioral model. The approach introduced here requires both models be transformed into an intermediate format, e.g., XML, to enable to identify the objects. This aspect is unfortunately not supported to the same extent by all commercial test tools – a fact that might endanger an automatic identification of the objects. Therefore, it is worthwhile to consider different components of the approach, e.g., test script construction, for standardization of the industrial test process and test tools for interactive systems to enable a broad compatibility and interoperability of the great variety of the existing test techniques and commercial test tools [GHRS+03].

8.5 Summary

This chapter presents three dedicated tools that were developed to support the test case generation using ESGs. The tool GenPath generates ESs and FESs of different lengths and establishes the MSCESs of the corresponding ESG and its extension for length > 2. For the analysis of the test cost of the generated test cases, the GATE tool was applied.

Furthermore, the manual activities during the test were identified and analyzed to carry them out automatically. For demonstrating the practicability and benefits of the approach, a commercial test tool (WinRunner) is augmented by test facilities that were developed (as add-ons). However, any other adequate commercial test tool can be used instead of this one. For modeling the system, any state-based or event-based method can be used.

The benefits of the add-ons are: Firstly, the approach does not need a sight into the code of the SUT. Secondly, once the test script has been constructed, the system can be automatically tested.

9 Discussion and Conclusion

This work introduced an integrated approach to the modeling, analysis, and test generation for embedded and interactive systems. Modeling is carried out using event sequence graphs (ESGs). These graphs and their complements $\overline{ESGs}$ assist the verification of the expected system behavior in the presence of expected inputs as well as the analysis of robustness and risks associated with system behavior in unexpected situations, thus delivering a holistic view. For modeling complex systems, the model supports a hierarchical decomposition which in turn breaks down the complexity of test case generation.

The test process based on the fault model generates test cases as sequences of events 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, acting as an oracle of a high level of trustworthiness. Furthermore, criteria are developed to determine the completeness of the test process, enabling a scalability of the test costs and 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.

The case studies presented indicate how ESGs can be used to model and analyze the behavior of a system. It was also shown how the ESG-based model is used for test generation. The effectiveness of the tests so generated is reported.

The degree of undesirability is represented in the form of a risk ordering relation – an expression of relative levels of risks of event sequences. This allows targeting the design of tests at specific system attributes. Further on, the approach is extended to statecharts as introduced by D. Harel. For this purpose, the test case generation and selection criteria have been applied to statecharts and their translation into extended regular expressions and are augmented with faulty transitions.

Finally, the test set generation is supported by tools which were developed, based on the algorithms introduced in this work. These tools have been integrated (as add-ons) into the commercial test environment WinRunner, which can be replaced by any other adequate commercial test environment to enable an automatic test generation.

9.1 Advantages and Disadvantages of Modeling with ESG

The event-driven concept of ESG as opposed to a state-driven approach such as FSAs 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. 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 [Bin00], such as decision tables and Karnaugh-Veith diagrams, might be helpful for combining them with the ESG for constructing test cases. As in any problem solving activity, there may not be a “silver bullet” type single test that can cope with every kind of fault.

9.2 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. [Bel01]. 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” [Goo75] 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 [WN04].

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.

9.3 Conclusion and Perspectives for Future Work

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.

If a more sophisticated fault classification model, e.g., Orthogonal Defect Classification [CBCH+92], is required, the fault model must be extended accordingly, differing across states, inputs and outputs. Following the guidelines in Section 3.3, 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 [Sto96].

A first step in this direction has been reported [Gut03] by applying the approach introduced in this work to Statecharts [HN96]. Further work is planned to consider UML – an approach already exploited for generating test cases [BL02, KHBC99, OLAA03]. However, further research [Hol06] is needed to extending the notions and algorithms introduced and summarized in this work, particularly in relation to state explosion caused by additional nodes while completing the ESG and to account for concurrency in system behavior [RS94, Sch90].

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 [Pel01], 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 [EVK02].

Bibliography

[ADLU91] Aho, A.V., Dahbura, A.T., Lee, D., Uyar, M.Ü.: An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *IEEE Transaction on Communication*, vol. 39(11), 1991, pp. 1604-1615.

[AHU77] Aho, A.V., Hopcroft, J.E., Ullmann, J.D.: *Principles of Compiler Design*. Addison-Wesley, 1977.

[ALR04] Avizienis, A., Laprie, J.C., Randell, B.: Dependability and its Threats: A Taxonomy. *In Proceedings of the IFIP 18th World Computer Congress*, Kluwer Academic Publishers, 2004, pp.91-120.

[Azu93] Azulay, A.: Automated testing for X applications. *The X Journal*, May-June, 1993, pp. 67-70.

[BB04] Belli, F., Budnik, C.J.: Minimal spanning set for coverage testing of interactive systems. *Proceedings of the First International Colloquium on Theoretical Aspects and Computing (ICTAC)*, Lecture Notes of Computer Science (LNCS), vol. 3407, Springer, 2004, pp. 220-234.

[BBH05] Belli, F., Budnik, C.J., Hollmann, A.: Holistic testing of interactive systems using statecharts. *Journal of Mathematics, Computing & Teleinformatics*, vol. 1(3), 2005, pp. 54-64.

[BBL76] Boehm, B.W., Brown, R.R., Lipow, M.: Quantitative evaluation of software quality. *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, 1976, pp. 592-605.

[BBN04] Belli, F., Budnik, C.J., Nissanke, N.: Finite state modeling, analysis and testing of system vulnerabilities. *Proceedings of 17th International Conference on Architecture of Computing Systems (ARCS)*, Lecture Notes in Informatics (LNI), vol. 41, 2004, pp. 19-33.

[BBW06] Belli, F., Budnik, C.J., White, L.: Event-based modeling, analysis and testing of user interactions: Approach and case study. *Journal of Software Testing, Verification and Reliability (STVR)*, vol. 16(1), John Wiley & Sons, Ltd., 2006, pp. 3-32.

[BD97] Belli, F., Dreyer, J.: Program segmentation for controlling test coverage. *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society Press, 1997, pp. 72-83.

[Bei90] Beizer, B.: *Software Testing Techniques*. 2nd ed., New York, USA, Van Nostrand Reinhold Co., 1990.

[Bel01] Belli, F.: Finite-state testing and analysis of graphical user interfaces. *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society Press, 2001, pp. 34-43.

[BG91] Belli, F., Grosspietsch, K.-E.: Specification of fault-tolerant system issues by predicate/transition nets and regular expressions: Approach and case study. *IEEE Transactions on Software Engineering*, vol. 17(6), 1991, pp. 513-526.

[Bin00] Binder, R.V.: *Testing Object-Oriented Systems*. Addison-Wesley, 2000.

[BL02] Briand, L., Labiche, Y.: A UML-based approach to system testing. *Software and System Modeling,* vol. 1(1), 2002, pp. 10-42.

[BP94] Bochmann G.v., Petrenko A.: Protocol testing: Review of methods and relevance for software testing. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ACM Press, 1994, pp. 109-124.

[BSH86] Basili, V.R., Selby, R.W., Hutchens, D.H.: Experimentation in software engineering. *IEEE Transactions on Software Engineering*, vol. 12(7), 1986, pp. 733-743.

[CBCH+92] Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.-Y.: Orthogonal defect classification – Concept for in-process measurements. *IEEE Transactions on Software Engineering*, vol. 18(11), 1992, pp. 943-956.

[CDFP97] Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, vol. 23(7), 1997, pp. 437-444.

[CH00] Chhikara, R.S, Heydorn, R.P.: Event sequence diagrams for dynamic probabilistic risk analysis. *Annual Report of the Institute for Space Systems Operations*, University of Houston, 2000.

[Cho78] Chow, T.S.: Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, vol. 4(3), 1978, pp. 178-187.

[Chr04] Christoph, J.: Automatic generation of suboptimal test cases and their analysis for fault detection and classification of graphical user interfaces (in German). *Technical Report* 2004/2*, Study Thesis*, Universität Paderborn, Angewandte Datentechnik, 2004.

[Chr04/2] Christoph, J.: Requirements definition, design and validation of the interface of a marginal mower equipment mounted on a truck (in German). *Technical Report* 2004/6*, Master Thesis*, Universität Paderborn, Angewandte Datentechnik, 2004.

[Dij59] Dijkstra, E.W.: A note on two problems in connection with graphs. *Journal of Numeric Mathematics*, 1959, pp. 269-271.

[DLS78] DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, vol. 11(4), 1978, pp. 34-41.

[DMM01] Delamaro, M.E., Maldonado, J.C., Mathur, A.P.: Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, vol. 27(3), 2001, pp. 228-247.

[DO91] DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, vol. 17(9), 1991, pp. 900-910.

[Dre98] Dreyer, J.: *Program Segmentation for Controlling Software Testing and Analysis*. Shaker Verlag, 1998.

[EB84] Eggers, B., Belli, F.: A theory on analysis and construction of fault-tolerant systems (in German). *Proceedings, Informatik-Fachberichte*, 1984, Springer , pp. 139-149.

[EJ73] Edmonds, J., Johnson, E.L.: Matching, Euler tours, and the Chinese postman. *Mathematical Programming*, vol. 5, 1973, pp. 88-124.

[EVK02] Eckmann, S.T., Vigna, G., Kemmerer, R.: STATL: An attack language for state-based Intrusion Detection. *Journal of Computer Security*, vol. 10(1-2), 2002, 71-103.

[FBKA91] Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M.: Test selection based on finite state models. *IEEE Transactions on Software Engineering*, vol. 17(6), 199, pp. 1591-603.

[FMMD94] Fabbri, S.C.P.F., Maldonado, J.C., Masiero, P.C., Delamaro, M.E.: Mutation analysis testing for finite state machines. *Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society Press, 1994, pp. 220-229.

[FX02] Fetzer, X., Xiao, Z.: Increasing the robustness of C-libraries using robustness wrappers. *Proceedings of the International Conference on Dependable Systems & Networks*, IEEE Computer Society Press, 2002, pp. 155-166.

[Gar89] Garg, V.K.: Modeling of distributed systems by concurrent regular expressions. *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, 1989, pp. 313-327.

[Gau95] Gaudel, M.-C.: Testing can be formal, too. *Proceedings of Theory and Practice of Software Development (TAPSOFT)*, Lecture Notes in Computer Science (LNCS), vol. 915, 1995, Springer, pp. 82-96.

[GBBD05] Gossens, S., Belli, F., Beydeda, S., Dal Cin, M.: View graphs for analysis and testing of programs at different abstraction levels. *Proceedings of the High-Assurance Systems Engineering Symposium (HASE)*, IEEE Computer Society Press, 2005, pp. 201-208.

[GH99] Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. *Proceedings of the 7th Euro-*

pean Software Engineering Conference (ESEC-7) and the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7), Lecture Notes in Computer Science (LNCS), vol. 1687, 1999, Springer, pp. 146-162.

[GHRS+03] Grabowski, J., Hogrefe, D., Rethy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, vol. 42(3), 2003, pp. 375-403.

[Gil62] Gill. A.: *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, 1962.

[Glu63] Gluschkow, W.M.: Theorie der abstrakten Automaten, *VEB Verlag der Wissenschaft*, 1963.

[Goo75] Goodenough, J.B.: Exception handling – Issues and a proposed notation. *Communications of the ACM*, vol. 18(12), 1975, pp. 683-696.

[Gos02] Gossens, S.: Enhancing system validation with behavioral types. *Proceedings of the 7th International Symposium on High-Assurance Systems Engineering (HASE)*. IEEE Computer Society Press, 2002, pp. 201-208.

[Gut03] Gutzeit, Th.: Test case generation from statecharts to validate graphical user interfaces (in German). *Technical Report 2003/6, Master Thesis*, University of Paderborn, Angewandte Datentechnik, 2003.

[Ham94] Hamlet, D.: Foundation of software testing: Dependability theory. *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994, pp. 128-139.

[Ham96] Hamlet, D.: Predicting dependability by testing. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ACM Press, 1996, pp. 84-91.

[Har87] Harel, D.: Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, vol. 8(3), 1987, pp. 231-274.

[HH02] Huhns, M.N., Holderfield, V.T.: Robust software. *IEEE Internet Computing*, vol. 6(2), 2002, pp. 80-82.

[HI98] Holcombe, M., Ipate, F.: *Correct Systems: Building a Business Process Solution*. Springer, 1998.

[HN96] Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, vol. 5(4), 1996, pp. 293-333.

[Hol04] Hollmann, A.: Additional tool-support for generation of minimal spanning complete event sequences by GenPath (in German). *Technical Report* 2004/8, Universität Paderborn, Angewandte Datentechnik, 2004.

[Hol06] Hollmann, A.: Extension of a graph-based, holistic approach for generating and selecting test cases (in German). *Ongoing work, Master Thesis*, Universität Paderborn, Angewandte Datentechnik, 2006.

[Hor99] Horrocks, I.: *Constructing the User Interface with Statecharts*. Addison-Wesley, 1999.

[Ian60] Ianov, Y.I.: Logic Schemes of Algorithms. *Problems of Cybernetics*, vol. 1, 1960, pp. 82-140.

[IEC610] IEC 61025 Fault Tree Analysis.

[IEC615] IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, Part 3: Software Requirements.

[IEEE90] IEEE Std. 610.12, IEEE Standard Glossary of Software Engineering Terminology, 1990.

[JS04] Jansamak, S., Surarerks, A.: Formalization of UML statechart models using concurrent regular expression. *Proceedings of the 27th Conference on Australasian Computer Science (ACSC)*, vol. 26, Australian Computer Society Inc., 2004, pp. 83-88.

[JW74] Jensen, K., Wirth, N.: *Pascal, User Manual and Report*. Springer, 1974.

[Kep94] Kepple, L.R.: The black art of GUI testing. *Dr. Dobb's Journal of Software Tools*, vol. 19(2), 1994, p. 40.

[KHBC99] Kim, Y.G., Hong, H.S., Bae, D.H., Cha, S.D.: Test cases generation from UML state diagrams. *IEE Proceedings – Software*, vol. 146(4), 1999, pp. 187-192.

[KKS98] Kropp, N.P., Koopman, P.J., Siewiorek, D.P.: Automated robustness testing of off-the-shelf software components. *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, IEEE Computer Society Press, 1998, pp. 230–239.

[Kle56] Kleene, S.C.: Representation of events in nerve nets and finite automata. *Automata Studies*, Princeton University Press, 1956, pp. 3-41.

[Kor96] Korel, B.: Automated test data generation for programs with procedures. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ACM Press, 1996, pp. 209-215.

[KPY99] Koufareva, I., Petrenko, A., Yevtushenko, N.: Test generation driven by user-defined fault models. *Proceedings of the 12th IFIP International Workshop on Testing of Communicating Systems (IWTCS)*. Kluwer Academic, 1999, pp. 215-236.

[Kwa62] Kwan, M.-K.: Graphic programming using odd or even points. *Chinese Mathematics*, vol. 1, 1962, pp. 273-277.

[Lap92] Laprie, J.C.: *Basic Concepts and Terminology.* In English, French, German, Italian and Japanese, IFIP WG 10.4, Dependable Computing and Fault Tolerance, Springer, 1992.

[Lev86] Leveson, N.G.: Software safety: Why, what, and how. *ACM Computer Surveys*, vol. 18(2), 1986, pp. 125-163.

[LL95] Libeaut, L., Loiseau, J.J.: Admissible initial conditions and control of timed event graphs. *Proceedings of the 34th IEEE Conference on Decision and Control*, vol. 2, 1995, pp. 2011-2016.

[LN92] Lyu, M.R., Nikora, A.P.: CASRE – A computer-aided software reliability estimation tool. *Proceedings of the Fifth International Workshop on Computer Aided Software Engineering (CASE)*, 1992, pp. 264-275.

[Lyu96] Lyu, M.R.: *Handbook of Software Reliability Engineering*. McGraw-Hill, 1996.

[Mar97] Martin, J.C.: *Introduction to Languages and the Theory of Computation*. 2nd edn., McGraw-Hill, 1997.

[MB03] Marré, M., Bertolino, A.: Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, vol. 29(14), 2003, pp. 974-984.

[MBHN03] Memon, A.M., Banerjee, I., Hashin, N., Nagarajan, A.: DART: A framework for regression testing nightly/daily builds of GUI applications. *Proceedings of the International Conference on Software*

Maintenance (ICSM), IEEE Computer Society Press, 2003, pp. 410-419.

[MPS00] Memon, A.M., Pollack, M.E., Soffa, M.L.: Automated test oracles for GUIs. *Proceedings of the 8th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, ACM Press, 2000, pp. 30-39.

[MPS01] Memon, A.M., Pollack, M.E., Soffa, M.L.: Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, vol. 27(2), 2001, pp. 144-155.

[MPS01/2] Memon, A.M., Pollack, M.E., Soffa, M.L.: Coverage criteria for GUI testing. *Proceedings of the 8th European Software Engineering Conference (ESEC) and the 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, ACM Press, 2001, pp. 256-267.

[MS03] Memon, A.M., Soffa, M.L.: Regression testing of GUIs. *Proceedings of the 9th European software engineering conference (ESEC) held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, ACM Press, 2003, pp. 118-127.

[Myh57] Myhill, J.: Finite Automata and the Representation of Events. *Wright Air Development Command*, vol. 57(624), 1957, pp. 112-137.

[ND02] Nissanke, N., Dammag, H.: Design for safety in safecharts with risk ordering of states. *Safety Science*, vol. 40(9), 2002, pp. 753-763.

[NT81] Naito, S., Tsunoyama, M.: Fault detection for sequential machines by transition tours. *Proceedings of the IEEE Fault Tolerant Computing Conference (FTCS)*, 1981, pp. 238-243.

[OA99] Offutt, J., Abdurazik, A.: Generating tests from UML specifications. *Proceedings of the 2nd International Conference of The Unified Modeling Language (UML)*, vol. 1723, 1999, pp. 416-429.

[OAK03] Okazaki, M., Aoki, T., Katayama, T.: Formalizing sequence diagrams and state machines using concurrent regular expression. *Proceedings of the 2nd International Workshop on Scenarios and State Machines (SCESM*), 2003, pp. 74-79.

[OLAA03] Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, John Wiley & Sons, Ltd., vol. 13(1), 2003, pp. 25-53.

[OMG03] OMG Unified Modeling Language, Specification UML, version 1.5, 2003.

[Par69] Parnas, D.L.: On the use of transition diagrams in the design of a user interface for an interactive computer system. *Proceedings of the 24th ACM National Conference*, ACM Press, 1969, pp. 379-385.

[Pel01] Peled, D.A.: *Software Reliability Methods*, Springer, 2001.

[PFV03] Pavia, A.C.R., Faria, J.C.P., Vidal, R.F.A.M.: Specification-based testing of user interfaces. *Interactive Systems*, Lecture Notes of Computer Science (LNCS), 2003.

[Pra97] Prather, R.E.: Regular expressions for program computations. *The American Mathematical Monthly*, vol. 104(2), 1997, pp. 120-130.

[RLT78] Randell, B., Lee, P.A., Treleaven, P.C.: Reliability issues in computing system design. *ACM Computer Survey*, vol. 10(2), 1978, pp. 123-165.

[Rob00] Robinson, H.: Intelligent Test Automation. *Software Testing and Quality Engineering Magazine*, September 2000, pp.24-32.

[RR97] Redmill, F., Rajan, J.: *Human Factors in Safety-Critical Systems*. Butterworth-Heinemann, 1997.

[RS94] Raju, S.C.V., Shaw, A.C.: A prototyping environment for specifying, executing and checking communicating real-time state machines. *Software - Practice and Experience*, vol. 24(2), 1994, pp. 175-195.

[Sal69] Salomaa, A.: *Theory of Automata*. Pergamon Press, 1969.

[Sar89] Sarikaya, B.: Conformance testing: Architectures and test sequences. *Computer Networks and ISDN Systems*, vol. 17(2), 1989, pp. 111-126.

[Sch90] Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, vol. 22(4), 1990, pp. 299-319.

[Sch95] Schruben, L.W.: Building reusable simulators using hierarchical event graphs. *Winter Simulation Conference Proceedings*, 1995, pp. 472 -475.

[SD88] Sabnani, K., Dahbura, A.: A protocol test generation procedure. *Computer Networks and ISDN Systems*, vol. 15(4), 1988, pp. 285—297.

[Sha80] Shaw, A.C.: Software Specification Languages Based on Regular Expressions. Riddle, W.E., Fairley, R.E. (eds.), *Software Development Tools*, 1980, Springer, pp. 148-176.

[Shn98] Shneiderman, B.: *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 3rd. edn., Addison-Wesley, 1998.

[SLD92] Shen, Y.-N., Lombardi, F., Dahbura, A.T.: Protocol conformance testing using multiple UIO sequences. *IEEE Transactions on Communications*, vol. 40(8), 1992, pp. 1282-1287.

[SS97] Shehady, R.K., Siewiorek, D.P.:A method to automate user interface testing using finite state machines. *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society Press, 1997, pp. 80-88.

[Sto05] Stockmaier, C.: Structure-based improvement of the testability of interactive Systems (in German). *Technical Report* 2005/7*, Master Thesis*, Universität Paderborn, Angewandte Datentechnik, 2005.

[Sto96] Storey, N.: *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[TH83] Thomas, J.J., Hamlin, G.: Graphical input interaction techniques. *ACM Computer Graphics News*, vol. 17(1), 1983, pp. 5-30.

[Thi03] Thimbleby, H.: The directed Chinese postman problem. *Software - Practice and Experience*, vol. 33(11), 2003, pp. 1081-1096.

[TL02] Tai, K.-C., Lei, Y.: A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, vol. 28(1), 2002, pp. 109-111.

[WA00] White, L., Almezen, H.: Generating test cases for GUI responsibilities using complete interaction sequences. *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society Press, 2000, pp. 111-121.

[WAA01] White, L., Almezen, H., Alzeidi, N.: User-based testing of GUI sequences and their interactions. *Proceedings of the 12th International Symposium on Software Reliability (ISSRE)*, IEEE Computer Society Press, 2001, pp. 54-63.

[Wes96] West, D.B.: *Introduction to Graph Theory*. Prentice Hall, 1996.

[Whi96] White, L.J.: Regression testing of GUI event interactions. *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, 1996, pp. 350-358.

[Wil91] Williams, L.: Formal Methods in the Development of Safety Critical Software System. *Technical Report SERM-014-91*, Software Engineering Research, 1991.

[WN04] Westley, W., Necula, G.: Finding and preventing run-time error handling mistakes. *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM Press, 2004, pp. 419-431.

[WR70] WinRunner 7.0, Mercury Interactive, Version Oct. 2004, http://www.mercuryinteractive.com, 2004.

[WRHO+00]Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: *Experimentation in Software Engineering: An Introduction*. Kluwer Academic, 2000.

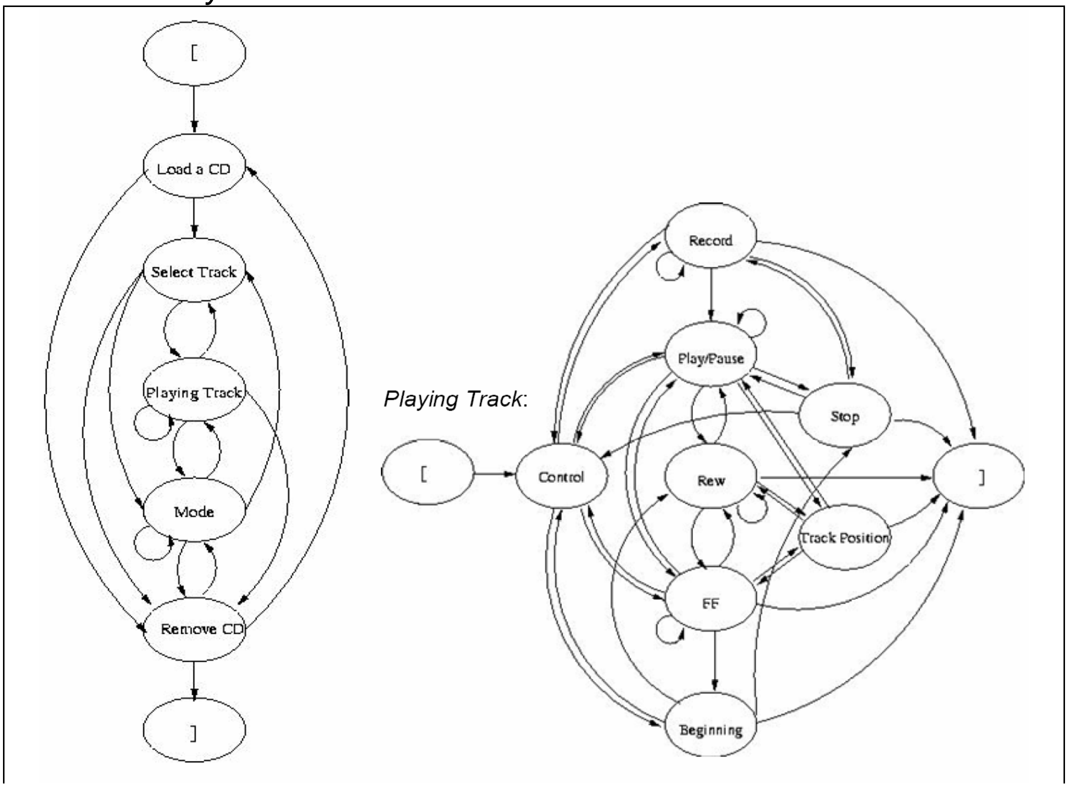
[XML10] Extensible Markup Language (XML), Bray, T. et.al. (eds.), version 1.0, Wide Web Consortium (W3C), http://www.w3.org/TR/REC-xml, 1998.

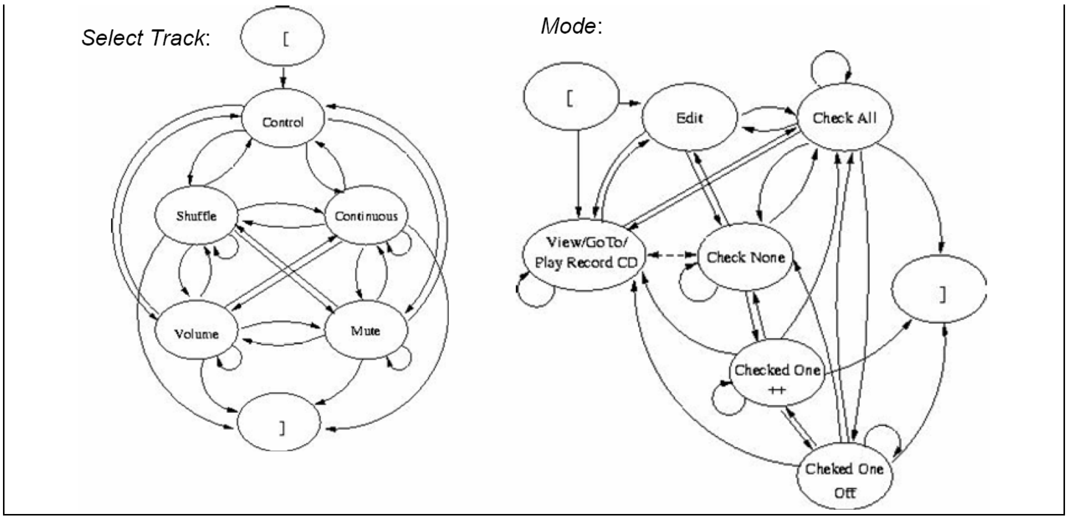
[ZHM97] Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy, *ACM Computing Surveys*, vol. 29(4), 1997, pp. 366-427.

A Case Study 1: The RealJukebox - RJB

A.1 ESGs of the Case Study 1

Function 1: *Play and Record a CD or Track*

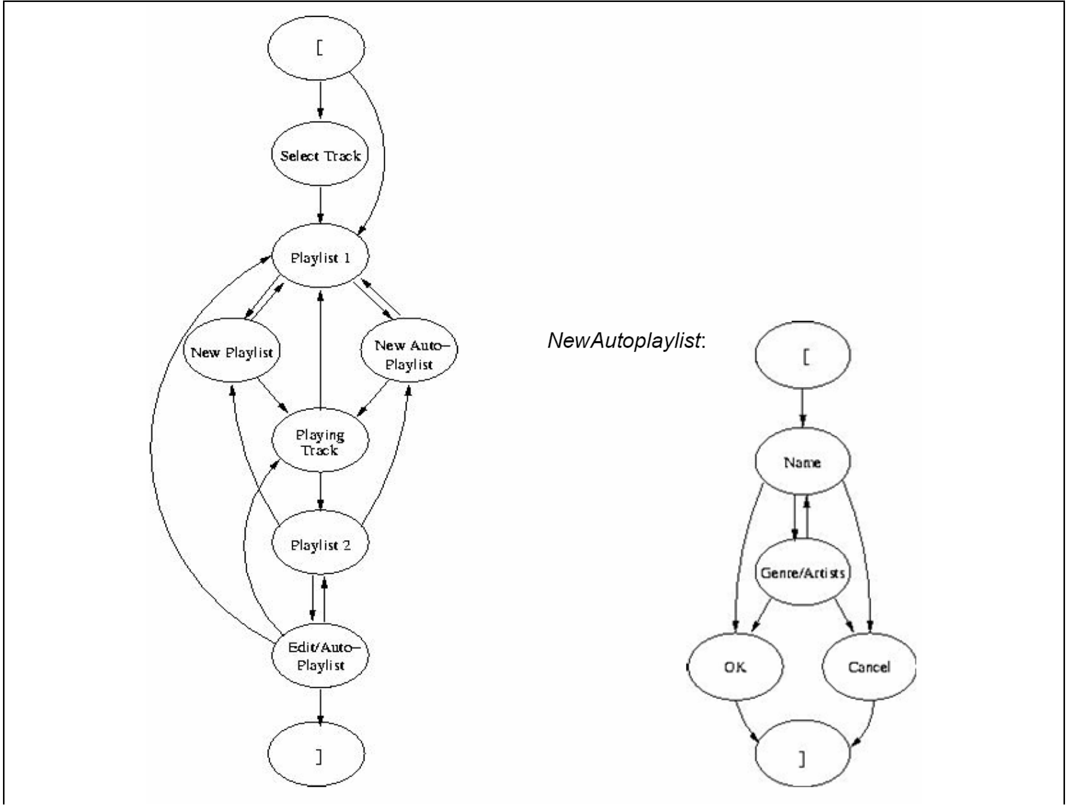




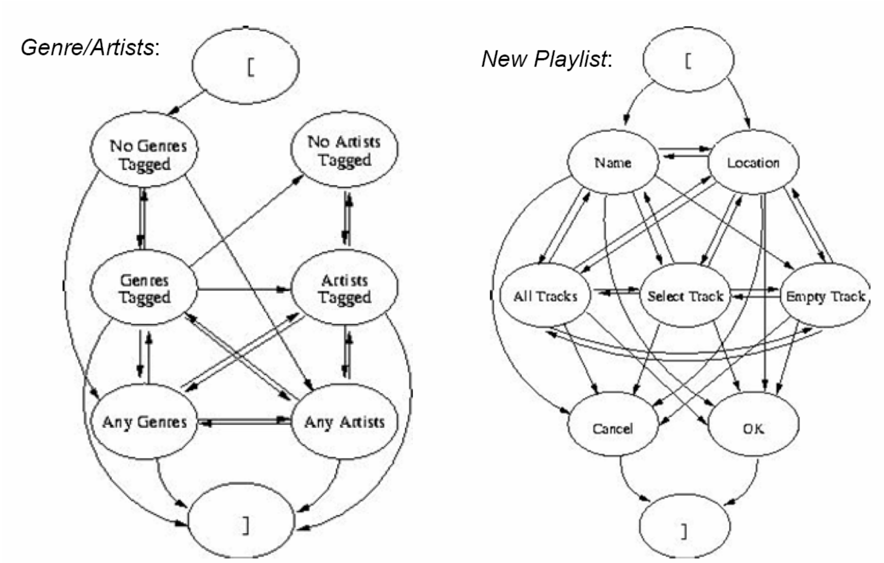
Select Track:

Mode:

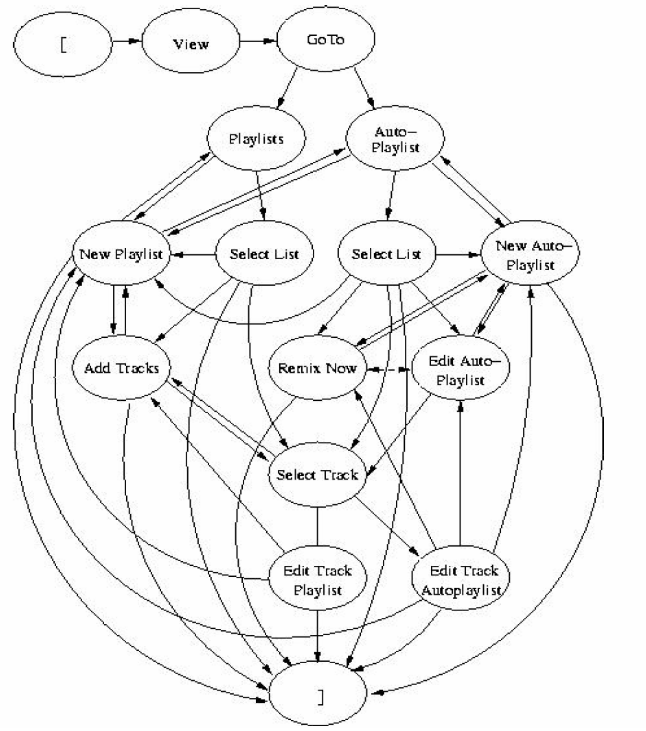
Function 2: *Create and Play a Playlist*

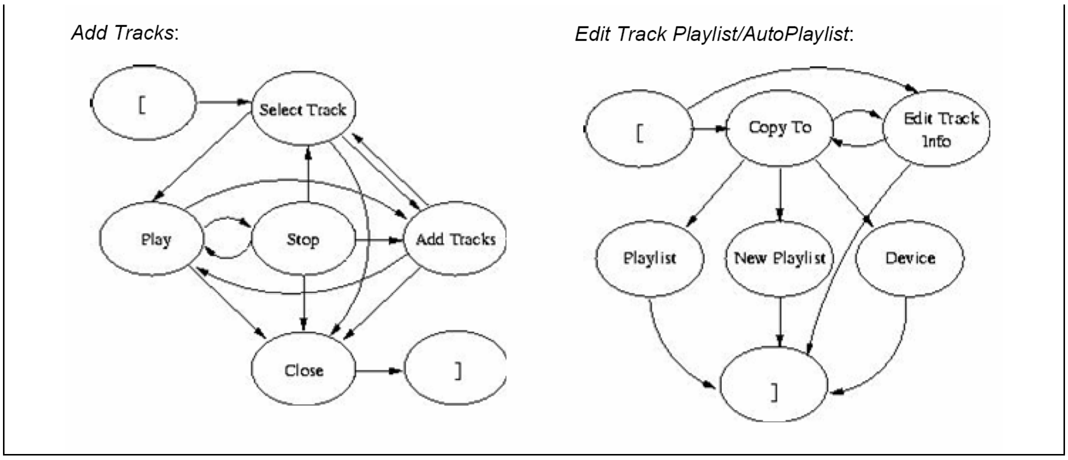


NewAutoplaylist:



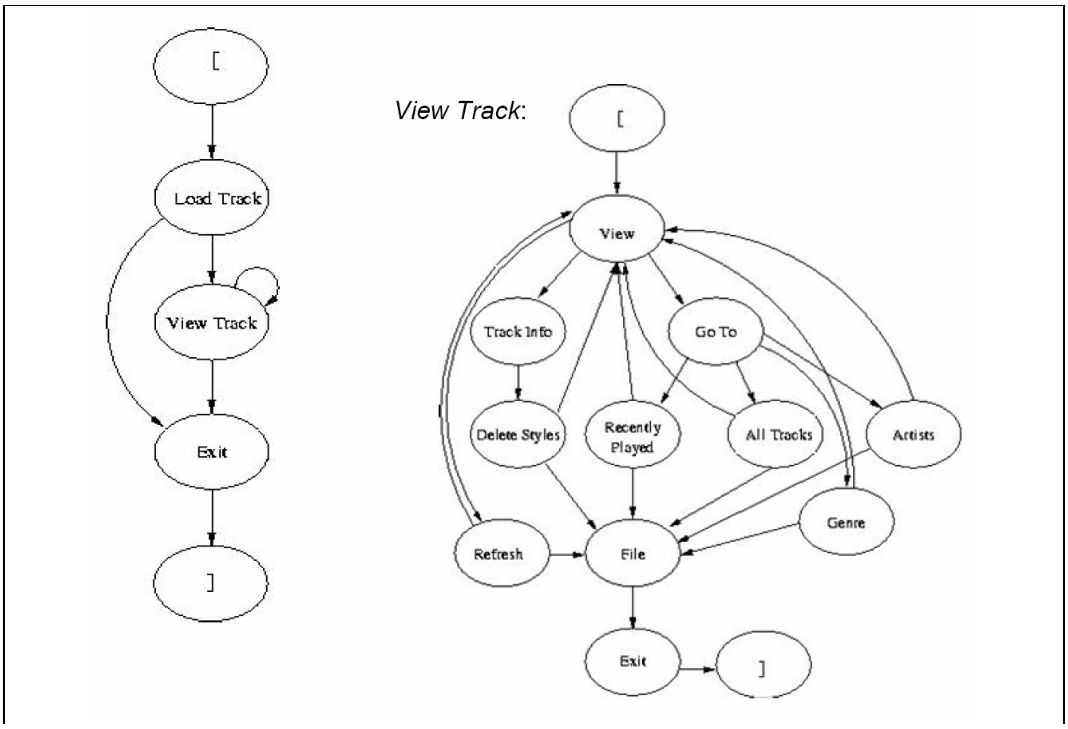
Function 3: *Edit Playlists and/or AutoPlaylists*



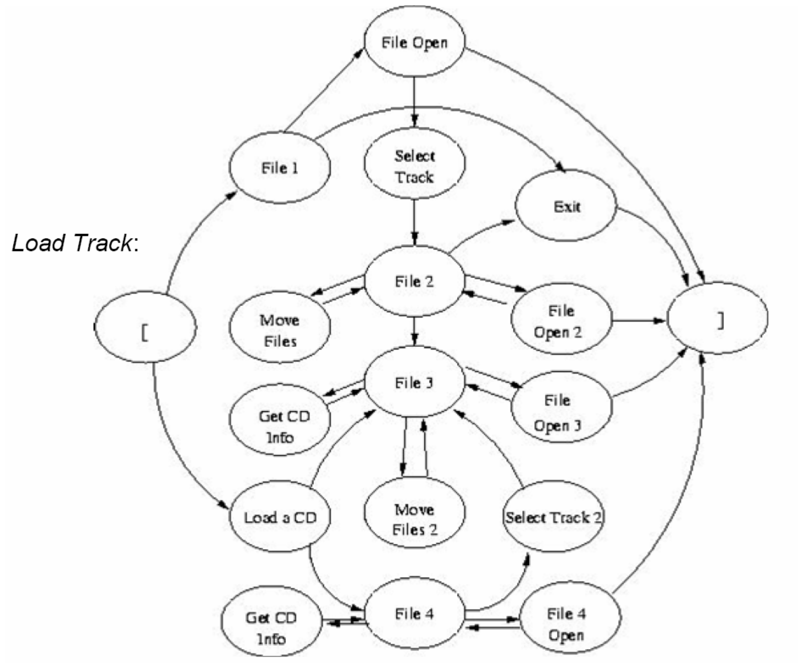

Add Tracks:

Edit Track Playlist/AutoPlaylist:

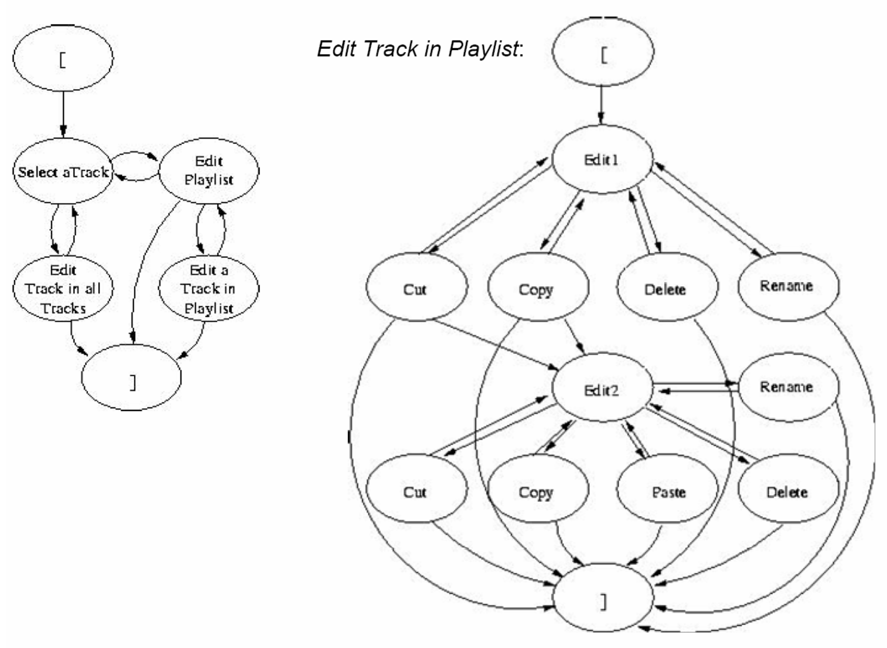
Function 4: *View Lists and/or Tracks*



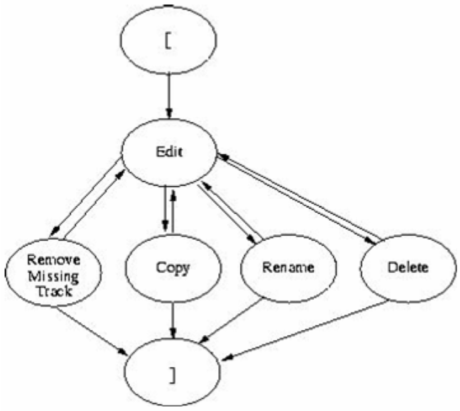
View Track:



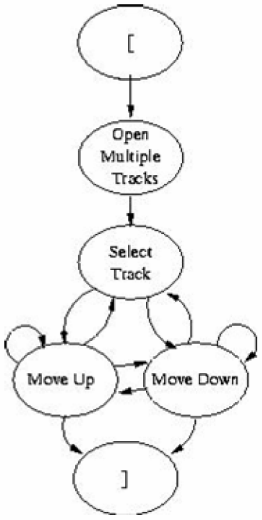
Function 5: *Edit a Track*



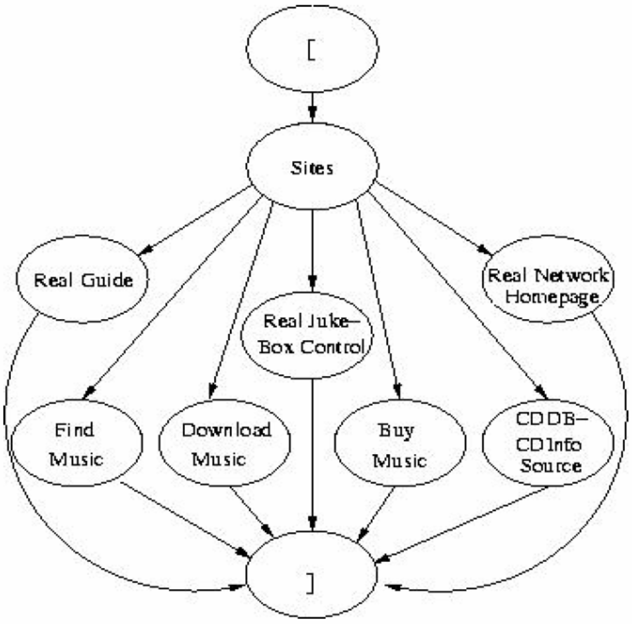
Edit Track in all Tracks:

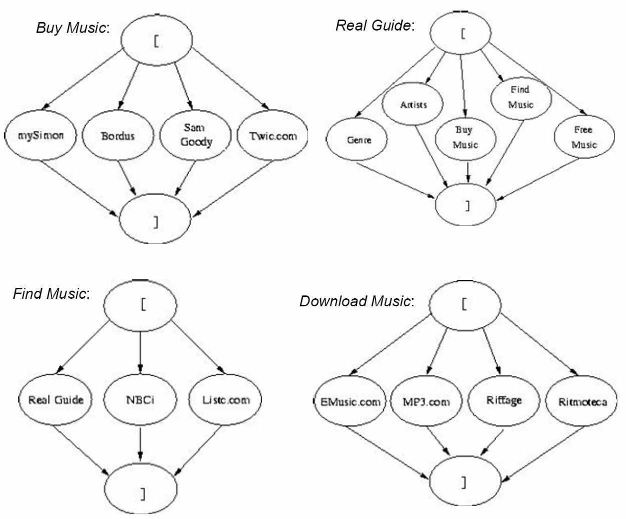


Move Track:

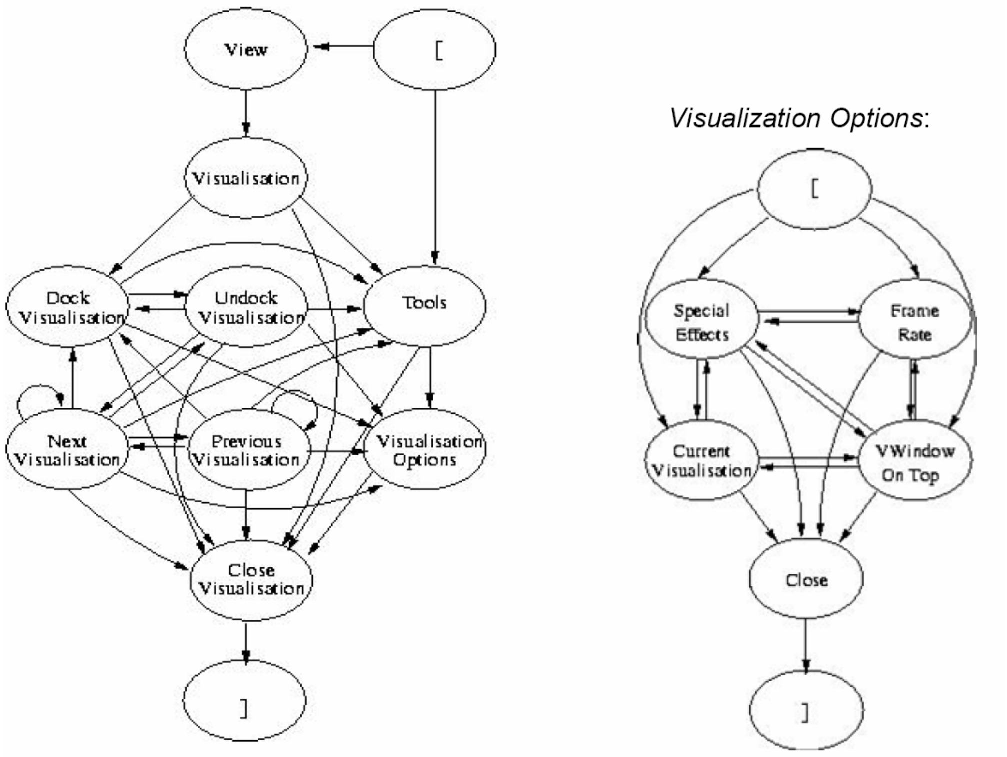


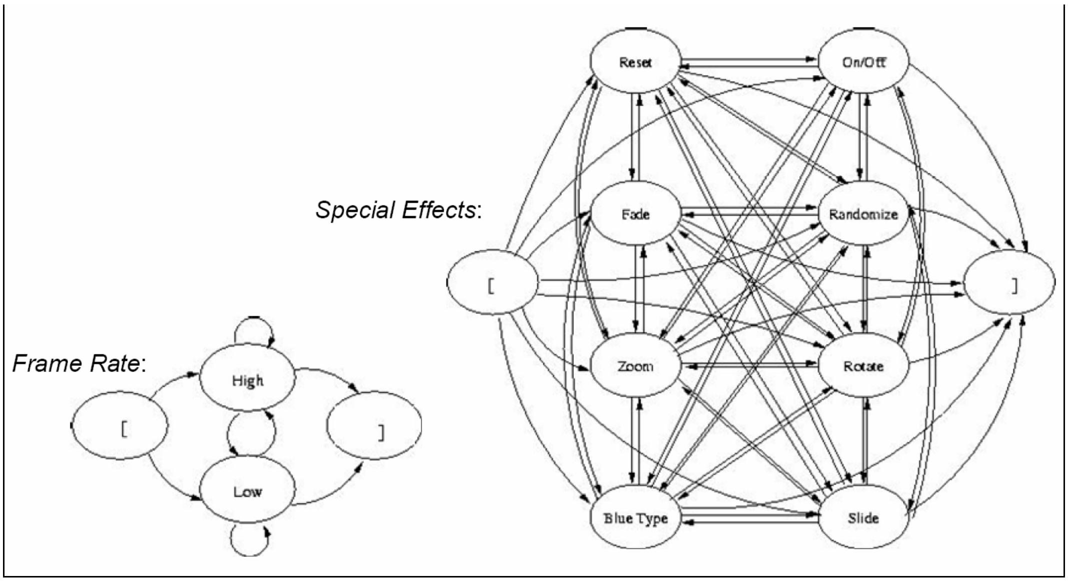
Function 6: *Visit the Sites*



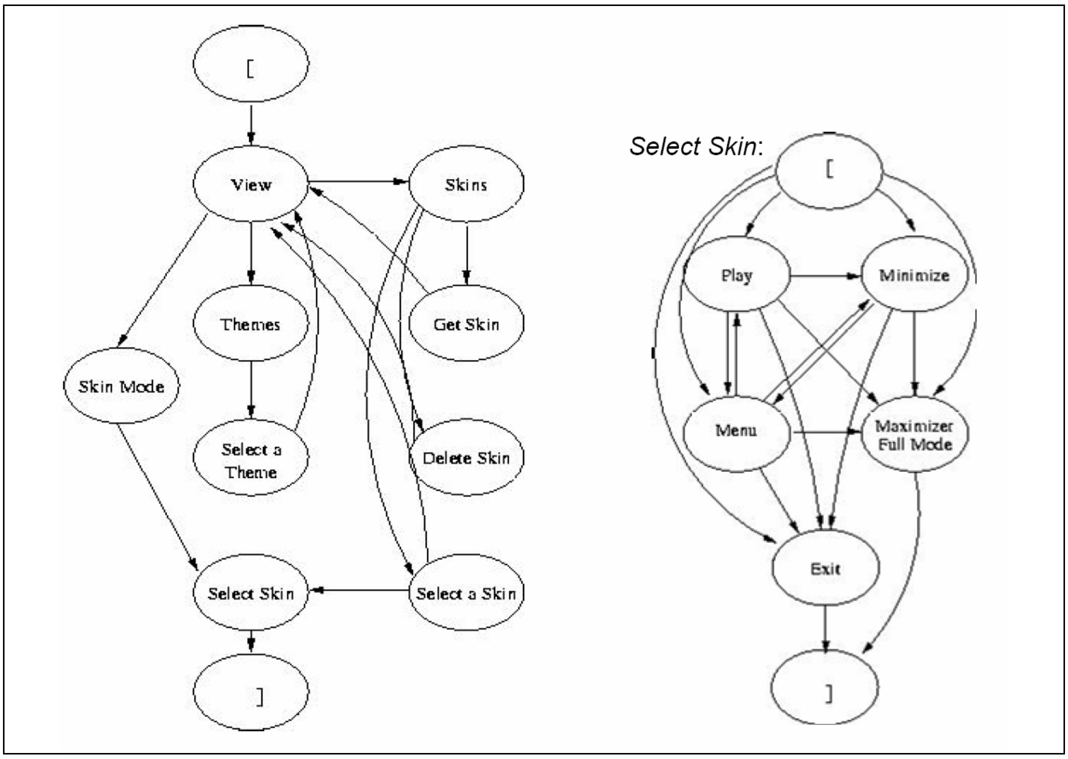


Function 7: *Visualization*

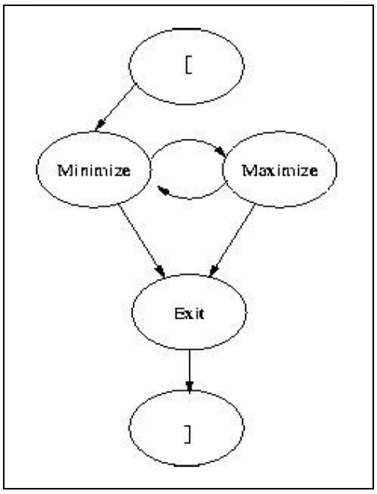




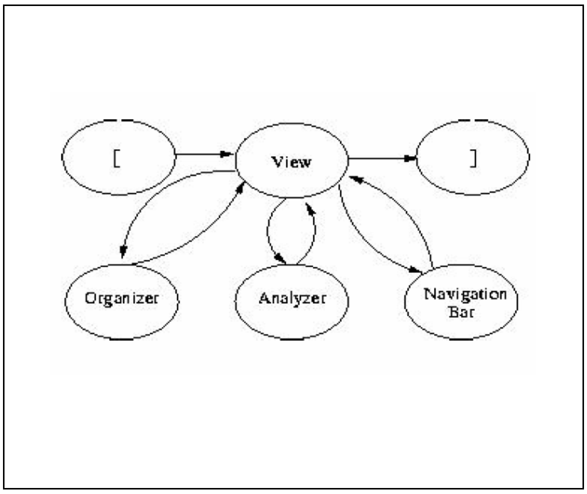
Function 8: *Skins*



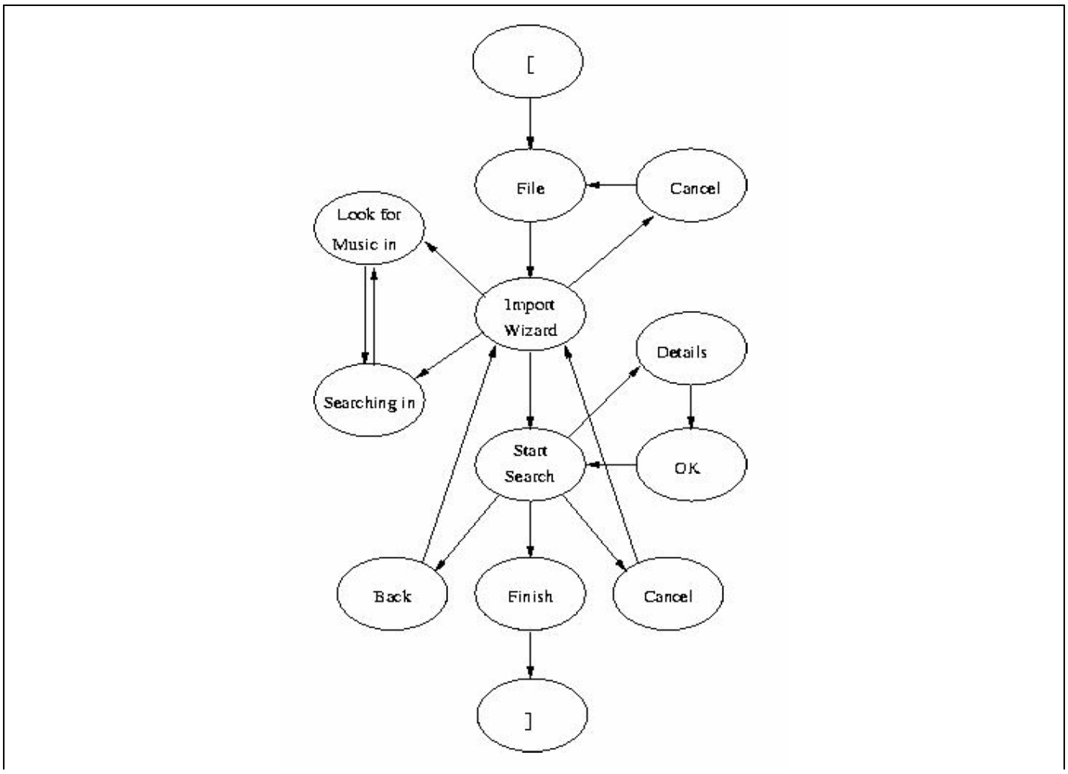
Function 9: *Sreen Size*

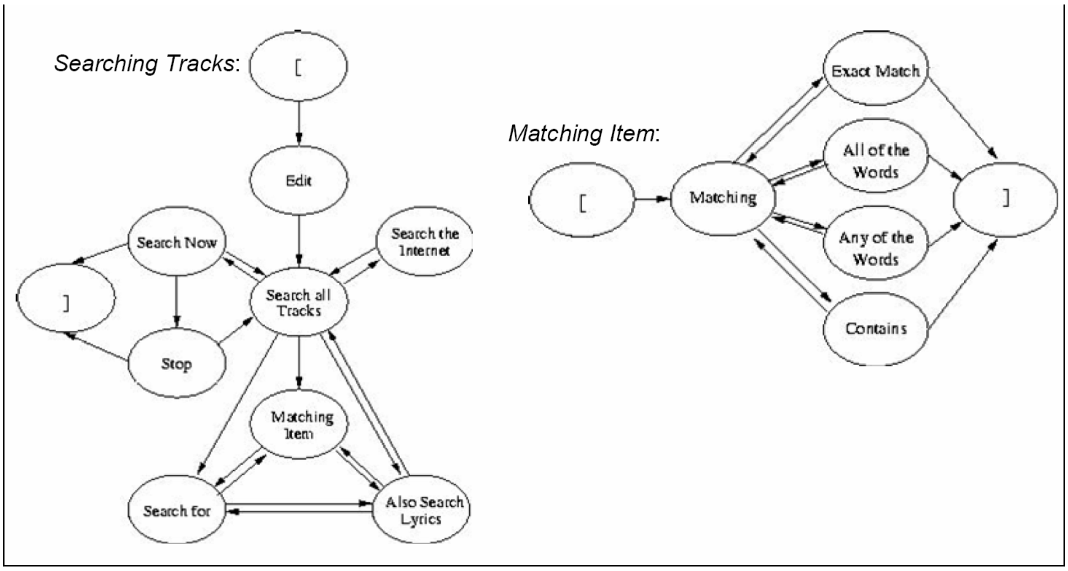


Function 10: *Different Views of Windows*

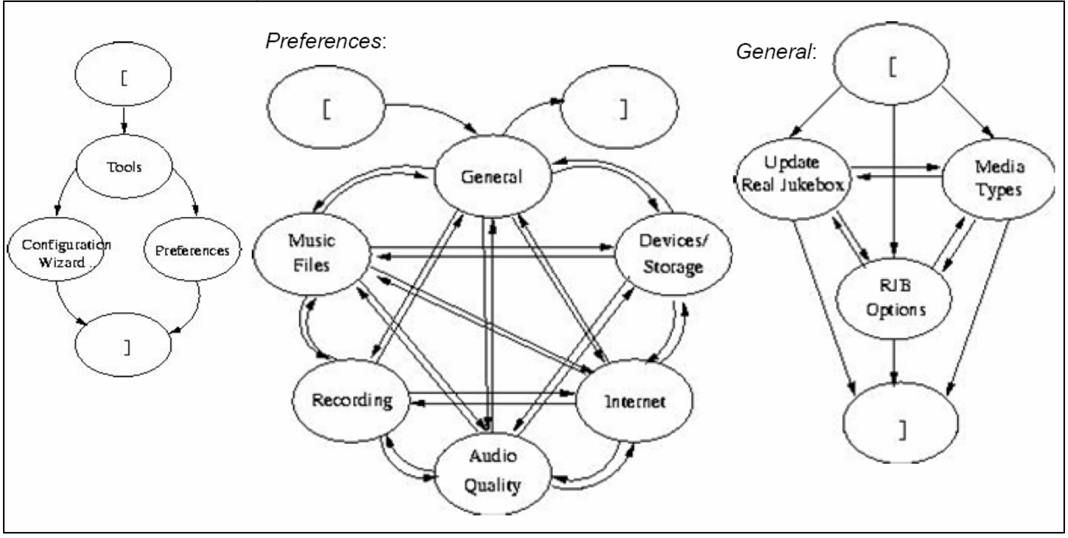


Function 11: *Find Music*





Function 12: *Configure RJB*



A.2 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/Paus*e is not the same as the control buttons displayed on the main window. Therefore, pushing start button on the control panel, while the track is playing, stops it.	functional	
		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	

Function	n-tuple to be covered	Faults detected by CES	Fault Type	Faults detected by FCES
1	2		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: "n 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 preceded sequentially.	sequencing	

Function	n-tuple to be covered	Faults detected by CES	Fault Type	Faults detected by FCES
1	2	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*>*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.

Function	n-tuple to be covered	Faults detected by CES	Fault Type	Faults detected by FCES
4	2	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 “age cannot be displayed”
			functional	Unmotivated, random error message: “Audio Instant Message Error in Program RealJukebox”
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 *Undock-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 *VIs.* windows causes switching to RJB.	sequencing	

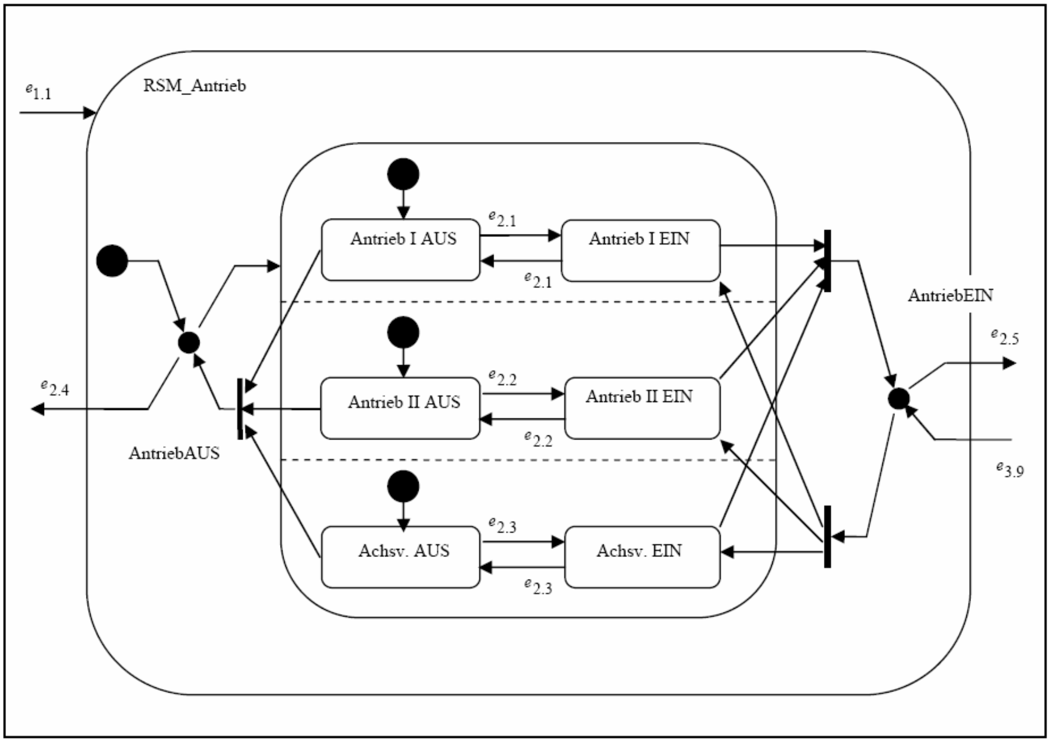
Function	n-tuple to be covered	Faults detected by CES	Fault Type	Faults detected by FCES
7	4		sequencing	Changing the *VIs.* in *Undock Mode* does not change into *Dock Mode* when closing the *Undock Mode*.
			functional	Special Effects should be disabled if they do not control any effect.
		When *VIs.* window in *Dock Mode* is clicked and moved several times during the window is settled, causes the settling to be abandoned.	functional	
		A newly installed *VIs.* 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.

Function	n-tuple to be covered	Faults detected by CES	Fault Type	Faults detected by FCES
11	2	*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.

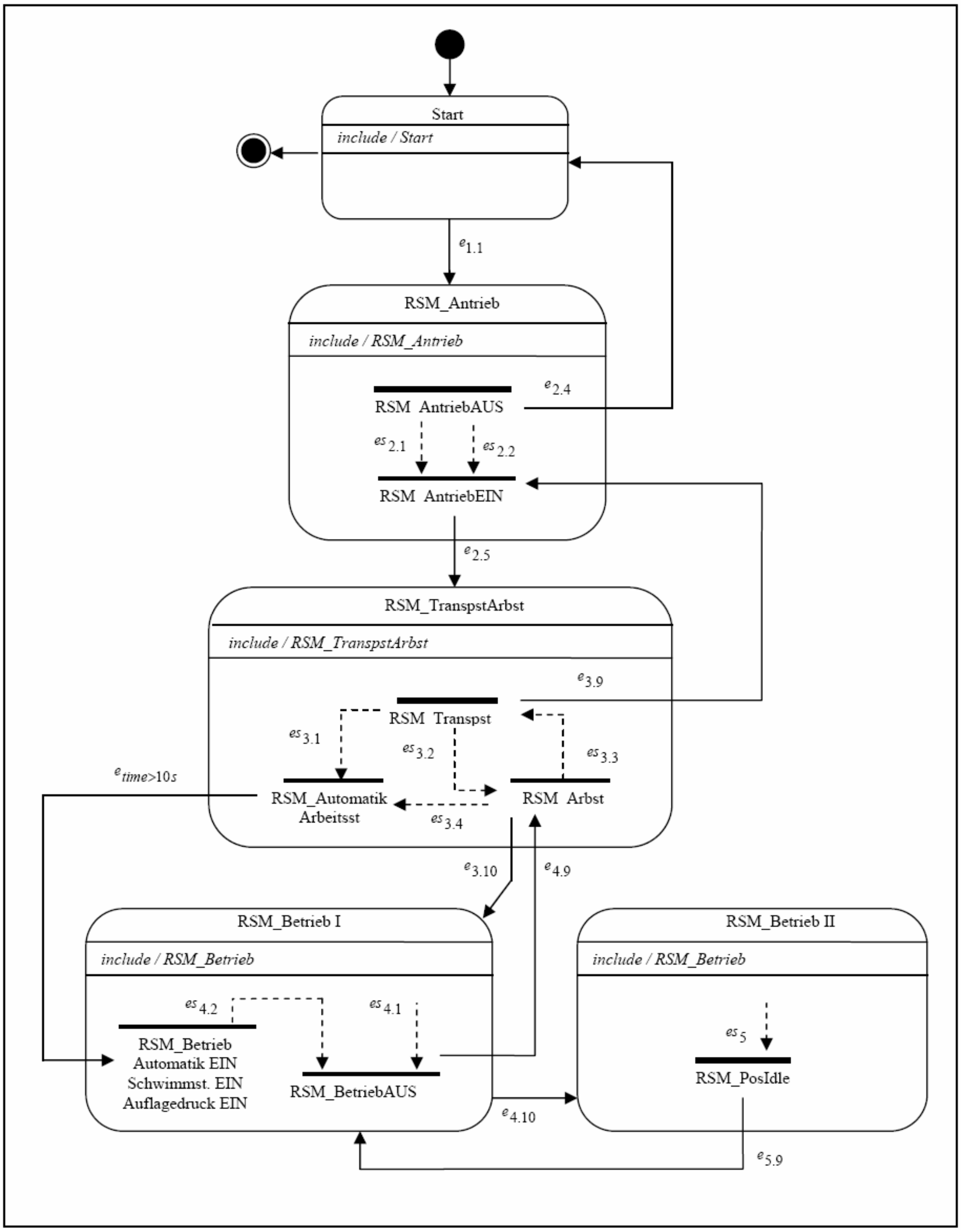
B Case Study 2: Marginal Strip Mower – RSM13

B.1 Statecharts of the Case Study 2

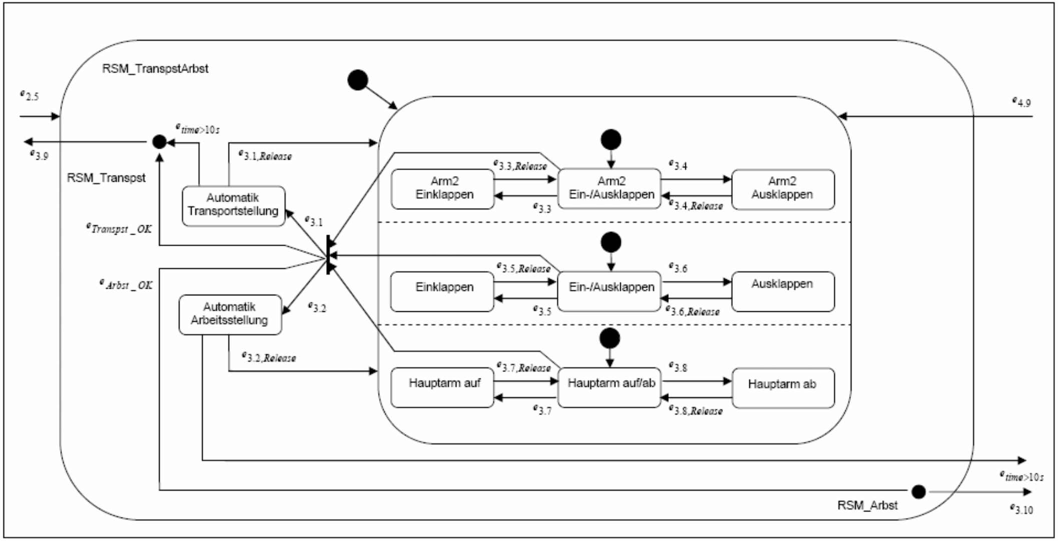
Function *RSM_Antrieb*:



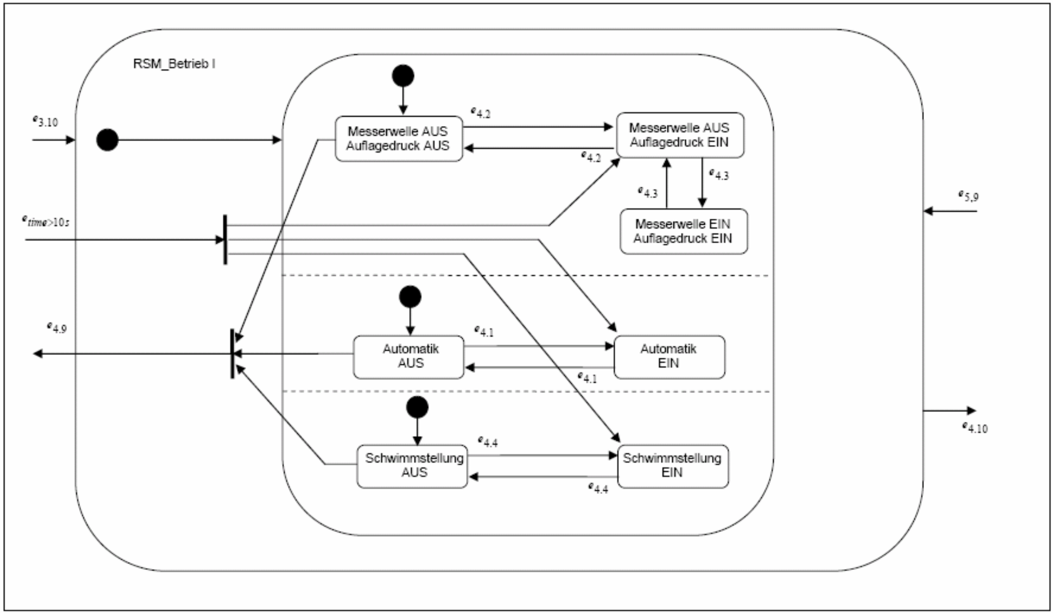
Function *Main:*



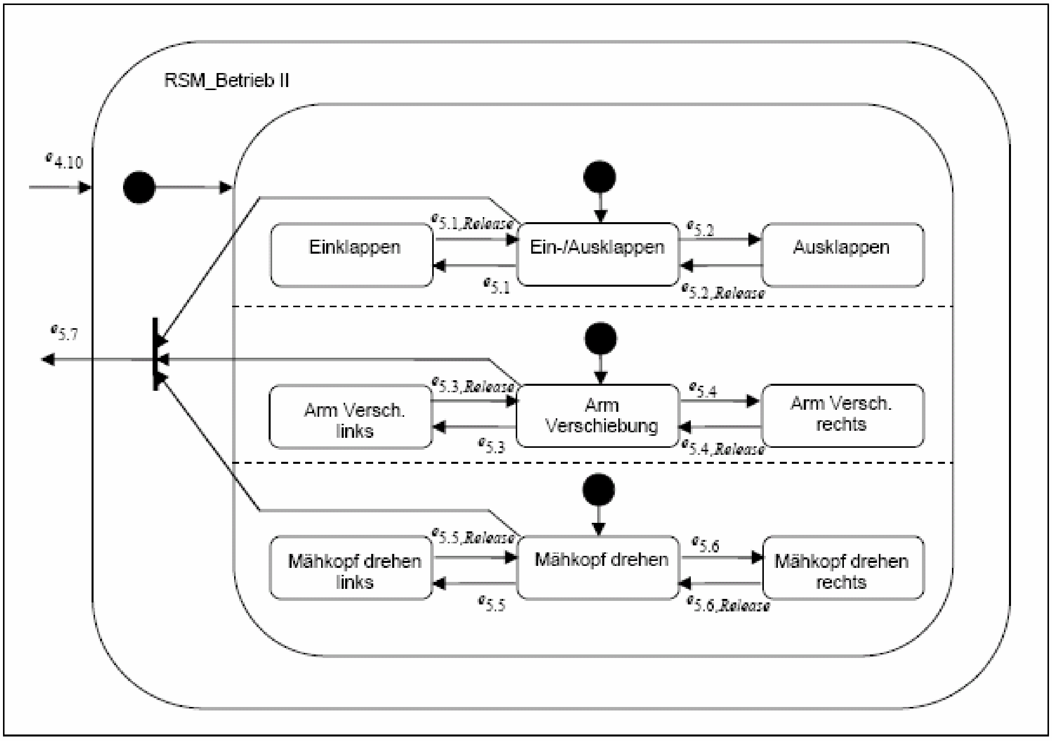
Function *RSM_TranspstArbst*:



Function *RSM_Betrieb I*:



Function *RSM_Betrieb II*:



B.2 List of Faults Revealed

Nb.	Function	Faults detected
1	*RSM_Antrieb*	When Engine I is deactivated, a change from the view *RSM_Antrieb* to the view *RSM_TranspstArbst* is only then possible if engine I is reactivated.
2		When Engine I is deactivated, a change from the view *RSM_Antrieb* to the view *Start* is only then possible if engine I is reactivated.
3		When Engine II is deactivated, a change from the view *RSM_Antrieb* to the view *RSM_TranspstArbst* is only then possible if engine II is reactivated.
4		When Engine II is deactivated, a change from the view *RSM_Antrieb* to the view *Start* is only then possible if engine II is reactivated.

Nb.	Function	Faults detected
5	*RSM_Antrieb*	When Axis Lock is deactivated, a change from the view *RSM_Antrieb* to the view *RSM_TranspstArbst* is only then possible if Axis Lock is reactivated.
6		When Axis Lock is deactivated, a change from the view *RSM_Antrieb* to the view *Start* is only then possible if Axis Lock is reactivated.
7	*RSM_Transpst Arbst*	A change from the view *RSM_TranspstArbst* to the view *RSM_Betrieb I* is only then possible if the RSM is in transport position.
8		When the function *Automatik Transportstellung* is activated the view can be changed neither to *RSM_Antrieb* or *RSM_Betrieb I*. Furthermore, the function *Automatik Arbeitsstellung* cannot be activated; neither is possible a manual positioning by the functions *Arm2 Ein-/Ausklappen*, *Ein-/Ausklappen* nor *Hauptarm auf/ab*.
9		When the function *Automatik Arbeitsstellung* is activated the view can be changed neither to *RSM_Antrieb* or *RSM_Betrieb I*. Furthermore, the function *Automatik Transportstellung* cannot be activated; neither is possible a manual positioning by the functions *Arm2 Ein-/Ausklappen*, *Ein-/Ausklappen* nor *Hauptarm auf/ab*.
10		When RSM is in a central position, a change from the view *RSM_TranspstArbst* to the view *RSM_Antrieb* is only then possible if RSM is positioned for transport.
11		When RSM is in a central position, a change from the view *RSM_TranspstArbst* to the view *RSM_Betrieb I* is only then possible if RSM is positioned for working.
12		When RSM is in a working position, a change from the view *RSM_TranspstArbst* to the view *RSM_Antrieb* is only then possible if RSM is positioned for transport.
13		When the function *Arm2 Einklappen* is activated, the function *Arm2 Ausklappen* can only then be activated if the function *Arm2 Einklappen* is deactivated.
14		When the function *Arm2 Einklappen* is activated, the function *Automatik Transportstellung* can only then be activated if the function *Arm2 Einklappen* is deactivated.
15		When the function *Arm2 Einklappen* is activated, the function *Automatik Arbeitsstellung* can only then be activated if the function *Arm2 Einklappen* is deactivated.
16		When the function *Arm2 Ausklappen* is activated, the function *Arm2 Einklappen* can only then be activated if the function *Arm2 Ausklappen* is deactivated.

Nb.	Function	Faults detected
17	*RSM_Transpst Arbst*	When the function *Arm2 Ausklappen* is activated, the function *Automatik Transportstellung* can only then be activated if the function *Arm2 Ausklappen* is deactivated.
18		When the function *Arm2 Ausklappen* is activated, the function *Automatik Arbeitsstellung* can only then be activated if the function *Arm2 Ausklappen* is deactivated.
19		When the function *Arm2 Einklappen* is activated, the function *Ausklappen* can only then be activated if the function *Einklappen* is deactivated.
20		In case function *Einklappen* is active an activation of function *Automatik Transportstellung* is only allowed after deactivating function *Einklappen*.
21		When the function *Einklappen* is activated, the function *Automatik Arbeitsstellung* can only then be activated if the function *Einklappen* is deactivated.
22		When the function *Ausklappen* is activated, the function *Einklappen* can only then be activated if the function *Ausklappen* is deactivated.
23		When the function *Ausklappen* is activated, the function *Automatik Transportstellung* can only then be activated if the function *Ausklappen* is deactivated.
24		When the function *Ausklappen* is activated, the function *Automatik Arbeitsstellung* can only then be activated if the function *Ausklappen* is deactivated.
25		When the function *Hauptarm auf* is activated, the function *Hauptarm ab* can only then be activated if the function *Hauptarm auf* is deactivated.
26		When the function *Hauptarm auf* is activated, the function *Automatik Transportstellung* can only then be activated if the function *Hauptarm auf* is deactivated.
27		When the function *Hauptarm auf* is activated, the function *Automatik Arbeitsstellung* can only then be activated if the function *Hauptarm auf* is deactivated.
28		When the function *Hauptarm ab* is activated, the function *Hauptarm auf* can only then be activated if the function *Hauptarm ab* is deactivated.
29		When the function *Hauptarm ab* is activated, the function *Automatik Transportstellung* can only then be activated if the function *Hauptarm ab* is deactivated.
30		When the function *Hauptarm ab* is activated, the function *Automatik Arbeitsstellung* can only then be activated if the function *Hauptarm ab* is deactivated.

Nb.	Function	Faults detected
31	*RSM_Betrieb I*	A change from the view *RSM_Betrieb_I* to the view *RSM_TranspstArbst* while function *Automatik* is active is only then possible if the function *Automatik* is deactivated.
32		When the function *Auflagedruck* is deactivated, the function *Messerwelle* can only then be activated if the function *Auflagedruck* is activated.
33		A change from the view *RSM_Betrieb_I* to the view *RSM_TranspstArbst* while function *Auflagedruck* is active is only then possible if the function *Auflagedruck* is deactivated.
34		A change from the view *RSM_Betrieb_I* to the view *RSM_TranspstArbst* while function *Auflagedruck* is active is only then possible if the function *Auflagedruck* is deactivated.
35		When the function *Messerwelle* is activated, the function *Auflagedruck* can only then be deactivated if the function *Messerwelle* is deactivated.
36		A change from the view *RSM_Betrieb_I* to the view *RSM_TranspstArbst* while function *Schwimmstellung* is active is only then possible if the function *Schwimmstellung* is deactivated.
37	*RSM_Betrieb II*	When the function *Einklappen* is activated, the function *Ausklappen* can only then be activated if the function *Einklappen* is deactivated.
38		A change from the view *RSM_Betrieb_II* to the view *RSM_Betrieb_I* while function *Einklappen* is active is only then possible if the function *Einklappen* is deactivated.
39		When the function *Ausklappen* is activated, the function *Einklappen* can only then be activated if the function *Ausklappen* is deactivated.
40		A change from the view *RSM_Betrieb_II* to the view *RSM_Betrieb_I* while function *Ausklappen* is active is only then possible if the function *Ausklappen* is deactivated.
41		When the function *Arm Verschiebung links* is activated, the function *Arm Verschiebung rechts* can only then be activated if the function *Arm Verschiebung links* is deactivated.
42		A change from the view *RSM_Betrieb_II* to the view *RSM_Betrieb_I* while function *Arm Verschiebung links* is active is only then possible if the function *Arm Verschiebung links* is deactivated.
43		When the function *Arm Verschiebung rechts* is activated, the function *Arm Verschiebung links* can only then be activated if the function *Arm Verschiebung rechts* is deactivated.

Nb.	Function	Faults detected
44	*RSM_Betrieb II*	A change from the view *RSM_Betrieb_II* to the view *RSM_Betrieb_I* while function *Arm Verschiebung rechts* is active is only then possible if the function *Arm Verschiebung rechts* is deactivated.
45		When the function *Mähkopf drehen links* is activated, the function *Mähkopf drehen rechts* can only then be activated if the function *Mähkopf drehen links* is deactivated.
46		A change from the view *RSM_Betrieb_II* to the view *RSM_Betrieb_I* while function *Mähkopf drehen links* is active is only then possible if the function *Mähkopf drehen links* is deactivated.
47		When the function *Mähkopf drehen rechts* is activated, the function *Mähkopf drehen links* can only then be activated if the function *Mähkopf drehen rechts* is deactivated.
48		A change from the view *RSM_Betrieb_II* to the view *RSM_Betrieb_I* while function *Mähkopf drehen rechts* is active is only then possible if the function *Mähkopf drehen rechts* is deactivated.