UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

FAKULTÄT FÜR
ELEKTROTECHNIK,
INFORMATIK UND
MATHEMATIK

Model-Based Mutation Testing for Test Generation and Adequacy Analysis

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

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Dipl.-Inform. Axel Hollmann

Erster Gutachter:	Prof. Dr.-Ing. Fevzi Belli
Zweiter Gutachter:	Prof. Dr.-Ing. Hans-Dieter Kochs

Tag der mündlichen Prüfung: 07.06.2011

Paderborn 2011

Diss. EIM-E/277

To Beate

Acknowledgements

First of all, I would like to thank my supervisor Prof. Fevzi Belli for enabling me to write this thesis and his patience throughout these years. I also thank Prof. Hans-Dieter Kochs for his advices. Furthermore, I thank my parents supporting me during the years. For having fruitful discussions that gave me deeper insights I also thank Prof. Eric Wong (University of Texas at Dallas), Dr.-Ing. Christof Budnik, and Michael Linschulte. Finally, my thanks goes to Sonja Pohlmeier and Jörg Reker for their encouragements that helped me to conclude my thesis successfully.

Contents

Symbols and Notation

Set Theory

$A = \{a_1, ..., a_n\}$	Set A with n elements $a_1, ..., a_n$
$\emptyset$	Empty set
$a \in A$	a is an element of A
$a \notin A$	a is not an element of A
$A \subseteq B$	A is subset of B
$A \subset B$	$A \subseteq B$ and $A \neq B$
$A \setminus B$	Set subtraction
$A \cup B$	Union of sets A and B
$A \cap B$	Intersection of sets A and B
$A \times B$	Cartesian product of sets A and B
$\|A\|$	Cardinality of set A
$\mathcal{P}(A)$	Power set of A

Basic Sets

A	Set of arcs
E	Set of events
N	Set of nodes
S	Set of states
S_Γ	Set of final states
S_Ξ	Set of initial states
T	Test set
T^*	Test set generated based on a mutant
TR	Set of transitions
TR_{faulty}	Set of faulty transitions
$\mathcal{X}$	Set of mutation operators
Γ	Set of exit events
Ξ	Set of entry events

Miscellaneous

f	Transition labeling function
g	State labeling function
H	Hierarchy relation
M	A model
M^*	Mutant of model M

P	A program
P^*	Mutant of program P
$Spec$	A specification
$Spec^*$	Mutant of specification $Spec$
SC_f	Flattened statechart
$\widehat{SC_f}$	Completed, flattened statechart
SUC^*	Mutant of SUC
Φ	Test generation algorithm
σ	Error state

Mutation Operators

acI, acO, acC	Action insertion, -omission, -corruption
aI, aO, aC	Arc insertion, -omission, -corruption
cI, cO, cC	Condition insertion, -omission, -corruption
eI, eO, eC	Event insertion, -omission, -corruption
lI, lO, lC	Literal insertion, -omission, -corruption
nI, nO, nC	Node insertion, -omission, -corruption
rI, rO, rC	Rule insertion, -omission, -corruption
sI, sO, sC	Sequence insertion, -omission, -corruption
stI, stO, stC	State insertion, -omission, -corruption
tI, tO, tC	Transition insertion, -omission, -corruption

Abbreviations

ACC	Adaptive Cruise Control
CCS	Complete Communication Sequence
CES	Complete Event Sequence
CFTS	Complete Faulty Transition Sequence
CS	Communication Sequence
CSG	Communication Sequence Graph
CTS	Complete Transition Sequence
DE	Data Event
DG	Directed Graph
DT	Decision Table
ECU	Electronic Control Unit
ES	Event Sequence
ESG	Event Sequence Graph
FCES	Faulty Complete Event Sequence
FEP	Faulty Event Pair
FES	Faulty Event Sequence
FSM	Finite-State Machine
FT	Faulty Transition
FTS	Faulty Transition Sequence
GUI	Graphical User Interface

MBMT	Model-Based Mutation Testing
MBT	Model-Based Testing
RJB	RealJukebox
RSM	Randstreifenmäher (in English: Marginal Strip Mower)
OOP	Object-Oriented Programming
SC	Statechart
SUC	System Under Consideration
TP	Transition Pair
TS	Transition Sequence
UML	Unified Modeling Language

Chapter 1

Introduction

Testing is one of the traditional and most commonly used techniques for assuring quality of software. It is carried out by *test cases* that are ordered pairs of *test inputs* and expected *test outputs*. A *test execution* represents the execution of the system under consideration (SUC) against the test cases. If the results of the test execution differ from the expected output, then the SUC *fails*; otherwise it *succeeds* the execution. *Static testing* evaluates a system based on its form, structure, or documentation. In contrast, *dynamic testing* is the process of evaluating a system based on its behavior during execution. It can be divided into two concepts. *Black-box testing* (also termed *functional testing*) ignores the internal behavior of the SUC. It concentrates on the outputs that can be traced back to selected inputs. Hence, test cases can be generated without the implementation. *White-box testing* (or *structural testing*) exploits the knowledge of the implementation to generate control flow-based and/or data flow-based test cases. *Test adequacy criteria* are used as a stopping rule to determine when to stop the process of testing and to provide a measure of test quality [128]. Most of the adequacy criteria are *coverage-oriented*, that is, they rate the portion of the system specification or implementation that is covered by the given test set against the specification/implementation.

The initial step of the software development is usually the requirements elicitation, and its outcome is the specification of the system's behavior. It is important to define appropriate test processes and generate test cases in this early stage, in compliance with the user's expectancy of how the system should behave. Some methods

visualize and represent the relevant features of the SUC in its environmental context, leading to a *model* of SUC. Based on this representation, *model-based testing* (MBT) techniques generate test cases systematically. In the last decade MBT has grown in importance ([36]). Models are specified to represent the relevant, desirable features of the SUC. These models are used as a basis for (automatically) generating test cases to be applied to SUC.

1.1 Model-Based Mutation Testing

Another fundamental problem of testing stems from the huge variety of possible software faults to be considered. For this purpose, *mutation analysis* techniques generate faulty versions of a software system–termed *mutants*–by introducing simple but representative faults in the implementation using mutation operators. These operators are also known as mutant operators in the literature. A given test set is executed against these mutants to check how many of these faults are revealed. The ratio of detected to undetected faults is finally used to assess the fault detection capability of the test set. A characteristic of implementation-based mutation analysis is that it does not detect new faults in SUC. At best, it indicates potential faults. Furthermore, it is a white-box testing technique necessitating the implementation of SUC, which is not always available.

The primary objective of this thesis is to conduct mutation analysis in the context of model-based testing, leading to *model-based mutation testing* (MBMT). For MBMT, a model of SUC that is assumed to be correct is first mutated to generate mutants that can be viewed as *fault models*. Test cases are then generated for each mutant (that is, each mutated model) using a test generation algorithm from MBT. Finally, SUC is executed *against* these test cases to decide which mutants can be revealed. In contrast to implementation-based mutation analysis, MBMT has more different, possible alternative outcomes (also refer to Figure 2.6 in Chapter 2). A test that reveals a model mutant of an SUC means that either the injected fault(s) in the mutant and/or fault(s) in SUC have been detected that were not found during MBT. Thus, the present approach enables the evaluation of the fault detection capability of test sets and, at the same time, the revelation of non-injected, latent faults in SUC, if any.

1.2 Modeling Techniques

A broad variety of formal or informal models exist for modeling software as recommended in standards such as the Unified Modeling Language (UML) [100] or the Testing and Test Control Notation (TTCN-3) [68]. Depending on user needs, those models describe SUC at different levels of granularity and preciseness. Graph-based models consist of nodes and arcs that connect the nodes. The semantics associated with these nodes and arcs determines the level of granularity of SUC description. For algebraic modeling, such levels primarily depend on the number of operations and operands used and their semantics. The issue of which model to use is not determined by solely identifying which model might be best for modeling the SUC, but also for performing the test, including fault modeling, test generation, and determining the test effectiveness.

Ideally, a model should be able to be converted between graphic and algebraic representations. For example, event sequence graphs and finite-state machines can be converted to regular expressions and vice versa by using algorithms from automata theory and formal languages ([11], [19], [54], [67], [111], [118]). Similarly, statecharts can be converted to extended regular expressions [16]. This implies that mutants with respect to these models can be generated by using either graph or algebraic manipulation operators and algorithms. This thesis prefers formal graph-based models; the following ones were selected and used in order of increasing expressive power.

- Directed Graph (DG): At this initial level, models include no semantics, that is, nodes and arcs have no interpretation. This level is used for introducing notations, especially mutation operators, syntactically.

- Event Sequence Graph (ESG): Nodes are interpreted as events; arcs define sequences of events ([10], [18]).

- Finite-State Machine (FSM): In addition to events, which are attributes of arcs of FSM, nodes are interpreted as states, whereas arcs symbolize state transitions. Outputs can be considered as additional attributes of arcs (*Mealy* machines) or states (*Moore* machines) ([103], [106], [115], [124]).

- Basic Statechart (SC): In addition to states and state transitions of FSM, concurrency and hierarchy aspects are considered ([22], [100]).

The above hierarchy is used to exemplify the approach; it is not meant to be complete. These models enable application of graph theory, discrete mathematics, formal logic, etc. In this thesis, one-sort nodes and arcs are considered and not two-sort ones (such as Petri nets) or multiple-sort ones (such as some UML diagrams). Of particular interest are those that formally specify functional behavior of SUC.

1.3 Mutation Operators

Different strategies have been proposed for mutation analysis. Some are applied to source code of the SUC while others are applied to models derived from specifications. Mutants generated can be used at the unit testing [2], [83], as well as at the integration testing level [50] (further details on related work are given in Chapter 8). A common problem of these studies is that they use a large number of mutation operators to generate mutants because they empirically determine the mutation operators based on selected fault models. This makes the exact number of operators that should be developed for a given fault model a matter of debate.

A different approach is used in this thesis by introducing two *basic* mutation operators: *insertion* and *omission*. Starting with a DG as the underlying model, the basic operators are applied to other models, such as ESG, FSM, and SC in the same manner. This is a major difference between this thesis and other studies, such as [34], [45], [50], [55], [57], [58], [84], [112], [117], [125]. Many mutation operators known from literature can be reduced to these two basic operators introduced, and their combination and iteration. For example, the "sdl" operator in the Mothra tool set [44] deletes a statement that is equivalent to an "omission" operator, and the "svr" operator does a scalar variable replacement that is an "omission" followed by an "insertion." In fact, mutants generated by the mutation operators in the aforementioned literature could be viewed as special cases of the fault model developed in this thesis. However, mutants generated by the two basic operators (with multiple iterations) and their combinations presented in this thesis might not be generated by the operators in previous publications.

1.4 Case Studies

For demonstration and validation of the proposed MBMT approach incorporating the graph-based models and basic operators, three case studies on industrial and commercial applications were conducted. They are the music management system RealJukebox (RJB) developed by RealNetworks, the adaptive cruise control system (ACC) developed by Hella Corp., and the control desk of a marginal strip mower (RSM13) mounted on the Unimog truck manufactured by Mercedes-Benz. The experiments on the broad area of these real-life applications covering interactive, reactive, and proactive systems delivered valuable data for the comparison and evaluation of the fault detection capability of test sets generated based on different kinds of graph models (ESG, FSM, and SC).

The results of these experiments suggest that test sets generated by the proposed approach can be effective in detecting faults in SUC. For example, the application used in the third study (RSM13) was previously (and independently) tested by a technical control board in Germany and released for public use. Test cases generated by the approach described here could detect faults in the released version that were not previously found.

1.5 Main Contributions and Novelties

To sum up, the main contributions and novelties of this thesis are:

- a rigorous introduction to the concept of MBMT that combines advantages of MBT and mutation analysis and positioning this concept in the landscape of mutation analysis while extending this landscape;
- systematizing and superseding the broad variety of existing mutation operators by using two basic mutation operators (*insertion* and *omission*);
- analyzing the mutants generated by the basic operators to avoid multiple occurrences of the same mutants;
- demonstrating the use of MBMT as an alternative to implementation-based

mutation techniques for analyzing the adequacy of test sets and extending the approach to a new testing strategy; and

- enabling the evaluation of the fault detection capability of test sets and revelation of residual (non-injected) faults in SUC by combining notations of mutation adequacy from implementation-based mutation analysis with test generation based on model mutants to detect faults in SUC.

1.6 Outline

The thesis is organized as follows. Chapter 2 introduces MBMT and explains the difference between implementation-based mutation analysis and this new approach. Chapter 3 defines a hierarchy of selected graph models along with the basic mutation operators, *insertion* and *omission*. It discusses their relationship and how to generate a broad class of mutants by applying these two operators multiple times in separate or combined ways. Chapter 4 presents model-based test processes for graph-based models discussed in the previous chapter. It also includes a discussion on the tool support. Chapter 5 reports case studies on three real-life industrial applications to demonstrate the applicability of the approach for mutant generation. The fault detection capability of test cases generated based on these mutants is also evaluated. Moreover, limitations are pointed out. As a result of the case studies a further coverage criterion is defined in Chapter 6 that is used to extend the model-based test processes described in Chapter 4. Chapter 7 presents different applications of basic operators and mutation testing in the context of robustness testing, model checking, and integration testing. Chapter 8 gives related work on mutation analysis based on programs, specifications, and models. Note that throughout the thesis related work is referred to whenever it is significant. Conclusions and perspectives are summarized in Chapter 9.

Chapter 2

Model-Based Mutation Testing (MBMT)

Mutation analysis, as introduced in the late seventies of the last century, is an implementation and fault injection-based white-box testing technique for software ([39], [51], [79]). Mutation analysis relies on two hypotheses: the *competent programmer hypothesis* ([1], [37]) and the *coupling effect* ([101]). The competent programmer hypothesis suggests that experienced programmers tend to write programs that are "close" to being correct. That is, although a program written by a competent programmer may be incorrect, it will differ from the correct version by only relatively simple faults in terms of their syntax/semantics. The task of testing is therefore reduced to validating a program that is probably not correct but is very close to the correct one. The coupling effect assumes that a test set that detects all simple faults in a program is sensitive enough to detect also more complex faults [101]. As Morell [94] points out, the distinction between "simple" and "complex" faults is not always clear. This thesis follows Offutt's definition [101]: a *simple fault* is a fault that can be fixed by making a *single* change to a source statement, and a *complex fault* is one that cannot be fixed by making a single change to a source statement.

Mutants generated by introducing only a single change to a program under test are known as *first-order* mutants. *Second-order* mutants are generated by making two simple changes, *third-order* by making three simple changes, and so on. Mutants of other than first-order are also known as *higher-order* mutants. To put it

simply, first-order mutants model simple faults, and higher-order mutants model complex faults. According to the coupling effect, higher-order mutants are likely to be detected by test cases that detect first-order mutants. In general, therefore, only first-order mutants are generated and used in implementation-based mutation analysis. However, according to Jia and Harman ([80]), there exist higher-order mutants constructed from first-order mutants that are harder to detect and thus more interesting than first-order mutants.

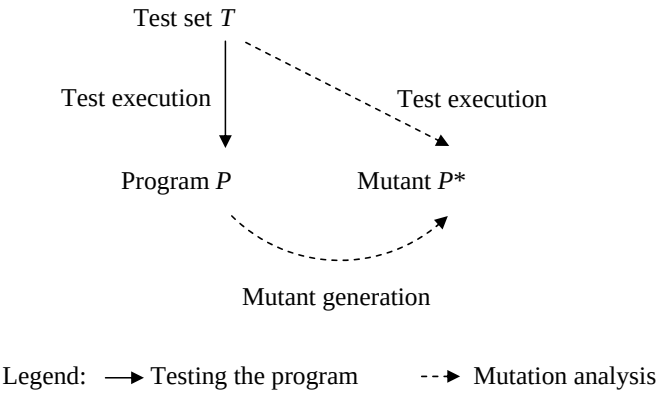


Figure 2.1: Implementation-based mutation analysis. T is *mutation adequate* if it kills all P^*. T can be used for testing the program P (see, e.g., [83]).

For explanatory purposes, assume that a program P and a test set T are given. A set of mutation operators $\mathcal{X}$, which is subject to be defined, is then applied to P to generate mutants by introducing one or more syntactical changes into P. Based on the coupling effect, the focus is on first-order mutants. The *fault detection capability* of T can be measured in terms of the number of mutants that can be distinguished ("killed") from the original program by test cases in T. More precisely, a mutant is *killed* by a test case $t \in T$ if the observed behavior of P and the mutant P^* differ from each other when executed against t. A mutant is *equivalent* to P if it always behaves the same as P for every case (or possible input). The test set T is said to be *mutation adequate* ([128]) with respect to P and $\mathcal{X}$ if every non-equivalent mutant of P generated by applying the mutation operators in $\mathcal{X}$ can be killed by at least one test case in T (see Figure 2.1). Mutation analysis has also been extended to validate a specification $Spec$ (see Figure 2.2). To do this, a set of mutation operators is

applied to $Spec$ and the fault detection capability of T is measured by how many mutated specifications $Spec^*$ can be killed by test cases in T.

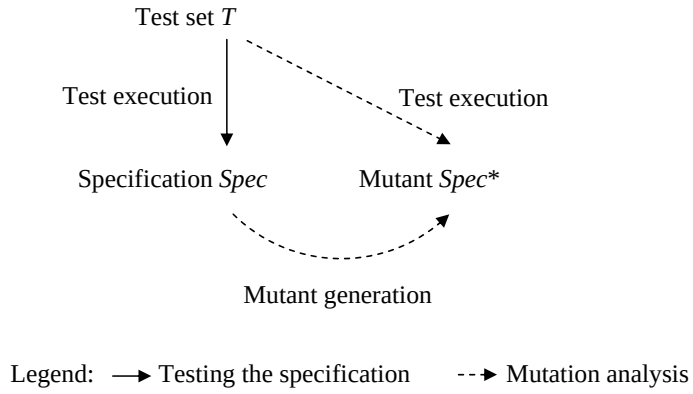


Figure 2.2: Specification-based mutation analysis. T is *mutation adequate* if it kills all $Spec^*$. T can be used for testing the specification or the implementation based on the specification (see e.g., [84] or [117], respectively).

2.1 Mutation Analysis for Evaluating Model-Based Test Generation

In model-based testing (MBT), SUC is described by means of a *model* M that is built in accordance with the specification (note that specification-based testing does not necessarily require a model). It is assumed that M is correct. A test generation algorithm Φ using a test selection criterion (for example, covering every node if M is graph-based) is applied to M such that $\Phi(M)$ generates a test set $T = (t_1, ..., t_i, ..., t_n)$ with each t_i ($i \in \{1, ..., n\}$) being a test case as an ordered pair

$$t_i = (\textit{input to SUC, expected output from SUC}) \in T.$$

After executing SUC against all the test cases in T, the question usually arises whether or not there are still more faults in SUC that have not been revealed. In

other words, the fault detection capability of T needs to be assessed. This can be carried out by

- either mutating the source code of SUC to construct code mutants of the SUC (Figure 2.3), or
- mutating the model M to construct model mutants of the SUC (Figure 2.4).

While the first view leads to *implementation-based* mutation analysis, the second view is used to define *model-based* mutation testing. Both approaches assume that (i) prior to the mutation activities, the conformance of model M and the corresponding SUC has been validated by MBT, and (ii) there may be still latent faults in SUC that MBT could not reveal. Solid lines in Figure 2.3 and 2.4 denote this. In these figures, the dashed lines describe different, follow-on mutation activities. The mutants are denoted by $SUC^*_{i,j}$ and $M^*_{i,j}$, respectively, where index i corresponds to i^{th} mutation operator and index j corresponds to the j^{th} mutant generated by the i^{th} operator.

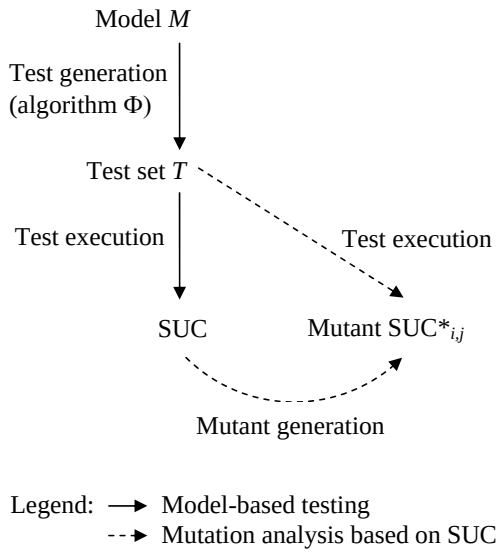


Figure 2.3: Mutation analysis for evaluating model-based test generation based on the implementation of SUC

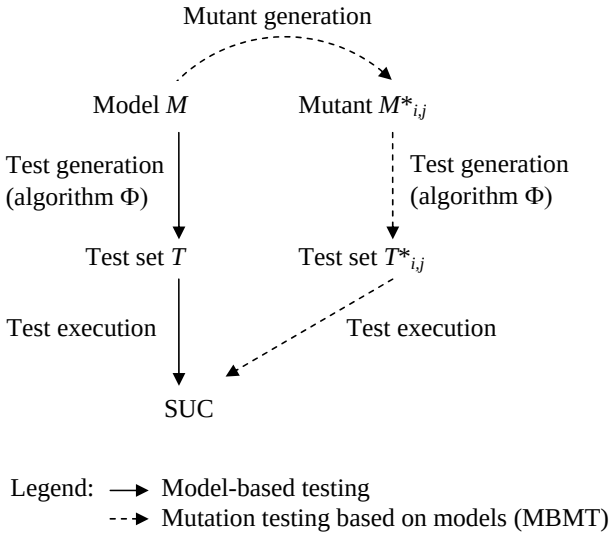


Figure 2.4: Mutation testing for evaluating model-based test generation based on model M of SUC

Both approaches enable an evaluation of test generation algorithm Φ by measuring the *mutation adequacy* in terms of how many mutants ($SUC^*_{i,j}$ or $M^*_{i,j}$) are killed. The implementation-based view (Figure 2.3) generates mutants directly with respect to the corresponding SUC. Note, however, that this approach is not always feasible, for example, if SUC is on a firmware or an embedded system that is difficult to mutate or if the source code is not available. Furthermore, an implementation-based approach does not necessarily reveal remaining faults in SUC.

This thesis focuses on the model-based approach that is depicted in Figure 2.4. As Section 2.2 will explain more in detail, a set of mutation operators is defined and each operator is applied to model M, yielding a set of mutants (with each mutant being a mutated model). With respect to each mutant $M^*_{i,j}$, a test set $T^*_{i,j}$ is generated using (model-based) test generation algorithm Φ. Then, SUC is executed against every test case in each $T^*_{i,j}$. If additional faults in SUC are detected by these $T^*_{i,j}$ that have not been revealed by test cases in T during MBT, then the test set T is not *mutation-adequate* with respect to the mutants. Therefore, apart from the possibility of detecting remaining faults in SUC, the model-based approach enables an assessment of the algorithm Φ and the test set T by checking the mutation ad-

equacy. However, MBMT leads to different interpretations of the predicates *killed* and *live*, as will be explained in the following section.

The competent programmer hypothesis can apply to MBMT, as well. It is assumed that programs that are implemented based on models only slightly differ from the correct ones. Thus, SUC deviates slightly from the original model that is presumed correct. Similarly, the coupling hypothesis is used to generate only first-order mutants with respect to the underlying model. Some other studies, for example, [3], [6], and [33], also generate test cases from model mutants and execute SUC against these test cases. Figure 2.5 depicts the procedure used by them. For each mutant $M^*_{i,j}$, a specific test set is generated to kill that mutant using an algorithm $\Phi_{i,j}$. The approach in Figure 2.4 differs from Figure 2.5 in that the same test generation algorithm Φ is used for all mutants in Figure 2.4 to generate test cases and the conformance testing based on the algorithm Φ has been carried out beforehand.

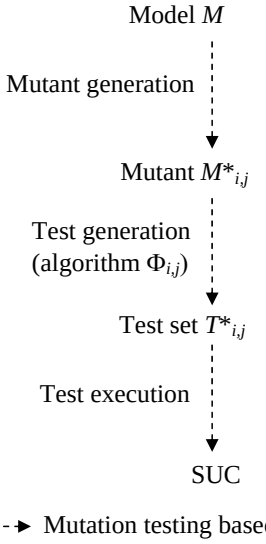


Figure 2.5: Mutation testing based on model M of SUC

In contrast to Figure 2.3, the processes in Figure 2.4 and Figure 2.5 generate mutants and test cases for mutation testing with respect to models. Figure 2.4 describes the MBMT-based test generation and test adequacy analysis, which is the focus of

this thesis. Throughout the rest of the thesis, wherever used, the term MBMT addresses this concept. The following section presents this approach in detail.

2.2 MBMT for Test Generation and Test Adequacy Analysis

The *set of mutation operators* is defined as $\mathcal{X} = (\chi_1, ..., \chi_i, ..., \chi_k)$. Each operator defines a fault model. Applying an operator χ_i to M (denoted by $\chi_i(M)$) generates a *set of mutated models* $M_i^* = (M_{i,1}^*, M_{i,2}^*, ..., M_{i,j}^*, ..., M_{i,p}^*)$, where $p \geq 1$. These mutants are called *model mutants* of M. For the rest of the thesis, they are referred to as *mutants*. The notation M^* is used as a generic representation for a mutant with respect to M. From each $M_{i,j}^*$, a test set $T_{i,j}^*$ is generated based on test generation algorithm Φ, that is, $\Phi(M_{i,j}^*) = T_{i,j}^* = (t_{i,j_1}^*, t_{i,j_2}^*, ..., t_{i,j_n}^*)$. The notation T^* will be used as a generic representation for a test set with respect to $T_{i,j}^*$. SUC will be executed against every test case in each $T_{i,j}^*$.

In the following, predicates are defined to determine whether M^* is a valid or an invalid mutant of M, and whether M^* has been killed or is still live with respect to test cases in T^*, or whether or not M^* is equivalent to M.

Definition 2.1 (Valid Mutant) A mutant M^* is a valid mutant of M if and only if M^* satisfies the syntactic and semantic requirements that model M has to fulfill. Otherwise, M^* is an invalid mutant, that is,

$valid(M^*) := \{\ M^* | M^*$ satisfies the syntactic and semantic requirements imposed by the model type of $M\ \}$.

As an example, if M is an FSM, then M^* (a mutated model) must also be a valid FSM. A syntactical requirement for an FSM is that all transitions must be associated with states. Semantics might require that an FSM must be deterministic or that all states are reachable from an initial state. More examples will be given in Chapter 3.

Definition 2.2 (Killed Mutant) If there is at least one test case t^* in T^* such that SUC generates an output that differs from the one expected from t^* when executed against t^*, then the mutant M^* is killed by t^*, that is,

$$killed(M^*, \Phi, SUC) := valid(M^*) \wedge \exists t^* \in \Phi(M^*) : ActualOutput(SUC, t^*) \neq ExpectedOutput(t^*)$$

Note that the output of SUC is also the output of M as the conformance between SUC and M has been validated with respect to $T = \Phi(M)$ as shown in Figure 2.4.

Definition 2.3 (Live Mutant) M^* is live with respect to $\Phi(M^*) = T^*$ if and only if no test case t^* in T^* can kill M^*; that is,

$$live(M^*, \Phi, SUC) := valid(M^*) \wedge \forall t^* \in \Phi(M^*) : ActualOutput(SUC, t^*) = ExpectedOutput(t^*)$$

Definition 2.4 (Equivalent Mutant) A mutant M^* is equivalent to M if and only if they produce the same outputs on every possible input in the input domain of SUC; that is,

$$equivalent(M, M^*) := \forall t \text{ in the input domain of SUC: } Output(M, t) = Output(M^*, t).$$

2.2.1 Mutant Classification

In mutation analysis based on the implementation/specification, it is important to classify mutants into *killed* and *live* in order to assess the adequacy of a test set T in terms of its capability for detecting the faults that are injected into SUC. However, analyzing mutants only with respect to the predicates *killed* and *live* in MBMT is not sufficient. This is because mutants and SUC must be further analyzed to find out why the expected and the actual outputs differ or are the same. The challenge is that, in addition to faults in the mutants (which have been injected on purpose), faults may also exist in SUC which may or may not be the same as the injected faults in the mutants. Note that this is also subject of discussion in data mutation [114]. Thus, in spite of *killing* all the mutants, there might still be remaining faults in SUC; moreover, SUC might not contain any faults represented by the mutants under consideration. The same argument also applies to mutants that are *live*. As a consequence of the discussion on predicates *killed* and *live*, Figure 2.6 illustrates six different types of mutants as follows.

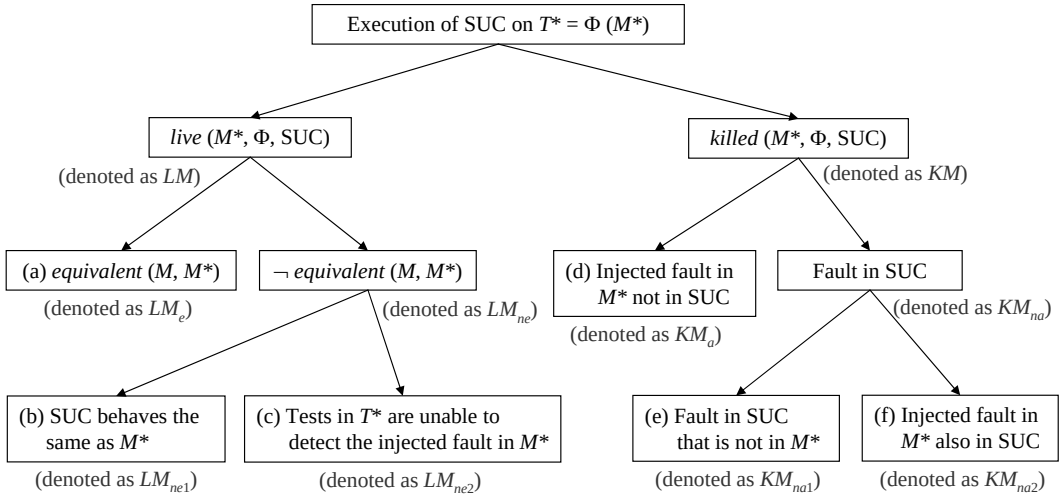


Figure 2.6: Mutant classification

- *Live mutants* (LM): set of all mutants that are live
  - LM_e: subset of LM consisting of all mutants equivalent to M
  - LM_{ne}: subset of LM consisting of all mutants that are non-equivalent to M
    - LM_{ne_1}: subset of LM_{ne} where SUC behaves the same as M^*
    - LM_{ne_2}: subset of LM_{ne} where tests in T^* are unable to detect the injected fault in M^*
- *Killed mutants* (KM): set of all mutants that have been killed
  - KM_a: subset of KM such that the injected fault in M^* is not in SUC
  - KM_{na}: subset of KM such that T^* reveals a fault in SUC
    - KM_{na_1}: subset of KM_{na} such that there is a fault in SUC that is not in M^*
    - KM_{na_2}: subset of KM_{na} such that the injected fault in M^* is also in SUC

A mutant M^* can be live for one of the following reasons. First, it is possible that M^* is equivalent to M (see Box (a) in Figure 2.6). The second possibility (see Box (b)) is that SUC behaves the same as M^* (which is not equivalent to M) with respect to test cases in T^*. This indicates that the implementation of SUC deviates from model M. As a result, SUC may contain "code" to introduce certain "unexpected" behavior different from that specified by M. This helps testers to reveal faults in SUC that have not been detected. Finally (see Box (c)), T^* is unable to detect the fault(s) in M^*. There are two possible reasons. Test cases in T^* do not execute the mutated part of M^*, or they cannot reveal the difference between M and M^*. Figure 2.7 gives a visual interpretation of Boxes (b) and (c).

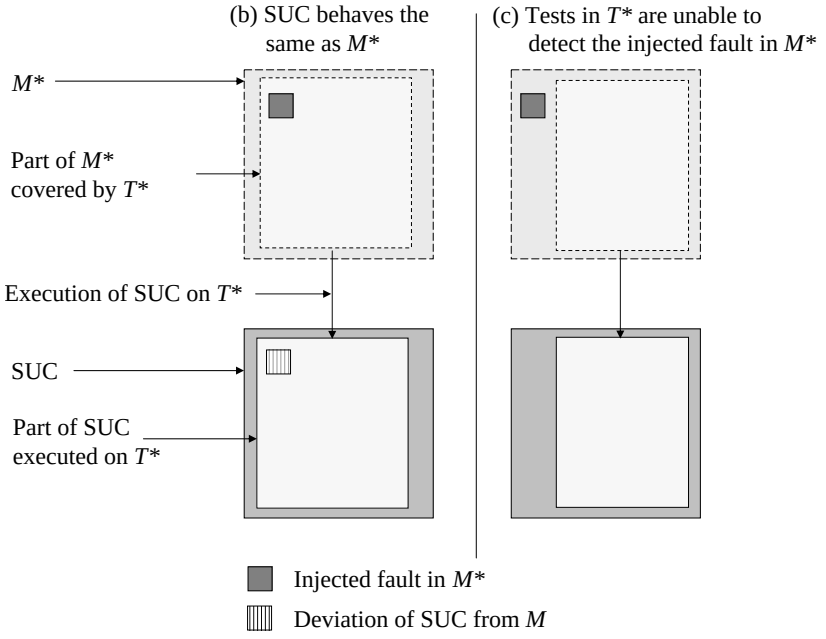


Figure 2.7: Interpretation of a live, non-equivalent mutant M^* in Figure 2.6, where $\forall t^* \in T^* : ActualOutput(SUC, t^*) = ExpectedOutput(t^*)$

When a mutant M^* is killed by T^*, then SUC behaves differently from M^* on at least one test case $t^* \in T^*$. For each of such t^*, this can also be further divided into three cases. First (see Box (d)), M^* has a fault that is not in SUC. As a result, when SUC is executed against t^*, it does not show certain faulty behavior of M^*.

Second (see Box (e)), SUC has a fault that is not in M^*. For example, there is a fault in SUC which causes its execution against test case t^* to fail, but M^* does not contain this fault. Due to the changes made in the mutant, the test set T^* generated using the algorithm Φ contains a test case t^* that reveals the fault in SUC that does not correspond to any fault in M^*. Third (see Box (f)), the fault in M^* (covered by a $t^* \in T^*$) reveals a fault in SUC that makes the SUC fail; for example, crash or inoperable. Note that the fault in SUC and M^* are the same faults with respect to model M but with different effects when executing the $t^* \in T^*$ against SUC and mutant M^*. Figure 2.8 provides a visual interpretation of these three cases.

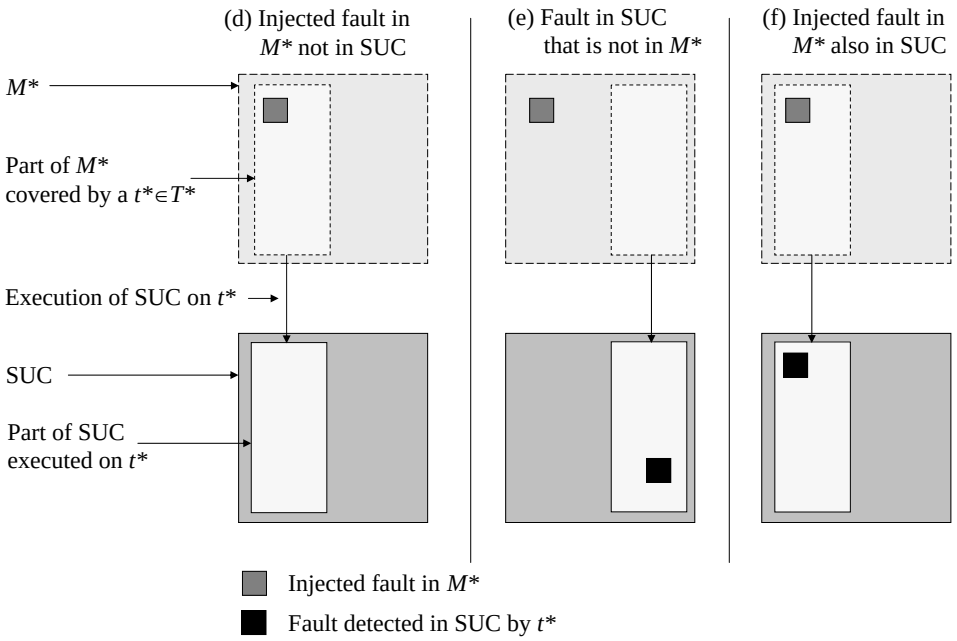


Figure 2.8: Interpretation of a mutant M^* killed by a test case $t^* \in T^*$ in Figure 2.6, that is, where $ActualOutput(SUC, t^*) \neq ExpectedOutput(t^*)$

Note that Boxes (a), (c), and (d) can be viewed as the traditional interpretation of predicates *killed* and *live* with respect to M^* and T^*; and Boxes (b), (e), and (f) in Figure 2.6 denote cases where faults in SUC are revealed by MBMT. In other words, if there are any mutants in these categories, then SUC has some faults that are not in M and cannot be detected by the test set T generated by applying test

generation algorithm Φ to the model M. Hence, test generation algorithm Φ (which also implies the generated test sets T and T^*) should be improved if any of the following conditions occur.

- LM_{ne_1} is not empty; SUC deviates from M and such deviation(s) have not been detected by the test set T during MBT.
- LM_{ne_2} is not empty; that is, some non-equivalent mutants are still live. These mutants do not correspond to faults in SUC but may be killed if $\Phi(M^*) = T^*$ is improved with additional test cases.
- KM_{na_1} or KM_{na_2} are not empty, that is, faults have been detected in SUC but were not detected during MBT by test set T.

Thus, the total number of faults in SUC (from the MBMT point of view) is $|LM_{ne_1}| + |KM_{na_1}| + |KM_{na_2}|$. Based on this, the following score is defined.

Definition 2.5 (Mutation Fault Detection Score) Given a model M, a test generation algorithm Φ, SUC, and a set of mutation operators $\mathcal{X}$, the *mutation fault detection score* is defined as

$$MFDS(M, \Phi, SUC, \mathcal{X}) := \frac{|LM_{ne_1}| + |KM_{na_1}| + |KM_{na_2}|}{|M^*| - |LM_e|},$$

where $|M^*|$ denotes the number of all mutants generated.

Definition 2.5 relates the number of mutants that revealed remaining faults in SUC to all mutants generated, except for equivalent mutants. If MBMT does not detect any remaining faults in SUC, the score is 0.

Definition 2.6 (Test Generation Mutation Adequacy Score) Given a model M, a test generation algorithm Φ, SUC, and a set of mutation operators $\mathcal{X}$, the *test generation mutation adequacy score* is defined as

$$TGMAS(M, \Phi, SUC, \mathcal{X}) := \frac{|KM_a|}{|M^*| - |LM_e|},$$

where $|M^*|$ denotes the number of all mutants generated.

The score given by Definition 2.6 corresponds to the traditional mutation score, that is, comparing the mutants that were killed (but did not reveal any faults in

SUC) to all mutants, except for equivalent ones. The score is 1, if no faults in SUC have been detected and there are no live mutants as the test cases generated based on the mutants were capable to distinguish all the mutants from SUC. A low score indicates that test cases in T generated based on $\Phi(M)$ were not sufficient to detect all (potential) faults in SUC.

In the following, an algorithm for mutant generation, execution of SUC, and classification of mutants into *killed* or *live* is presented. It is a model-independent algorithm that can be applied to any of the models introduced in Chapter 3 (DG, ESG, FSM, and SC) or other graph-based and algebraic models.

2.2.2 Generation and Execution of Mutants

Algorithm 2.1 describes how model mutants are generated and executed. The input consists of a model M that describes SUC, a set of mutation operators $\mathcal{X}$, and a test generation algorithm Φ. The output is a list of mutants generated by the mutation operators in $\mathcal{X}$ that are killed by test cases generated by Φ and a list of mutants that are still live. Note that this algorithm is not necessarily restricted to first-order mutants. Each mutation operator $\chi_i \in \mathcal{X}$ is applied to M to generate a set of mutants denoted as M_i^*. Invalid mutants are discarded. For each valid mutant $M_{i,j}^* \in M_i^*$, SUC is executed against the test cases in a corresponding test set $T_{i,j}^*$ generated by using the algorithm Φ. If at least one test case $t^* \in T_{i,j}^*$ exists in which the expected output predicted by t^* (namely, $ExpectedOutput(t^*)$) differs from the output of executing SUC against t^* (namely, $ActualOutput(SUC, t^*)$), then the mutant $M_{i,j}^*$ is killed. If no test case in $T_{i,j}^*$ can kill $M_{i,j}^*$, it is live. Note that a (manual) sub-classification of killed and live mutants according to the scheme in Figure 2.6 is still necessary to analyze the correctness of SUC.

If M^* is live, then the possible equivalence between model M and mutant M^* must be checked. Depending on the kind of model used, this could be done by using model checking. If M^* is non-equivalent, M^* and SUC have to be compared to identify where the fault(s) are injected in order to determine whether or not M^* belongs to the set LM_{ne_1} or LM_{ne_2}. If M^* is killed, it has to be checked whether or not the behavior of SUC is faulty. If it does not imply a fault in SUC, the mutant is classified as KM_a. Otherwise, the mutant and SUC have to be compared. If the

fault in SUC does not correspond to fault(s) in the mutant, it is classified as KM_{na_1}, otherwise as KM_{na_2}.

Algorithm 2.1: Mutant generation and execution

Input: Model M that describes SUC
Set of mutation operators $\mathcal{X} = (\chi_1, ..., \chi_i, ..., \chi_k)$
Test generation algorithm Φ

Output: A list of the mutants that are killed and the ones that are live

```
1  foreach χ_i ∈ 𝒳 do
2      M*_i := χ_i(M);
3      foreach M*_{i,j} ∈ M*_i do
4          if valid(M*_{i,j}) then
5              T*_{i,j} := Φ(M*_{i,j});
6              killed := false;
7              foreach t* ∈ T*_{i,j} do
8                  Execute SUC against t*;
9                  if ActualOutput(SUC, t*) ≠ ExpectedOutput(t*) then
10                     Add M*_{i,j} to the list of killed mutants;
11                     killed := true;
12             if killed = false then
13                 Add M*_{i,j} to the list of live mutants;
```

The runtime complexity of Algorithm 2.1 depends on the runtime complexity of (1) mutant generation, that is, the mutation of model M, (2) test generation, that is, algorithm Φ to generate a test set T^* for each mutant M^*; and (3) test execution, that is, the execution of SUC against the corresponding test sets T^*. Table 2.1 gives the runtime complexity for each step. The total number of mutants that can be generated depends on the number of mutation operators ($|\mathcal{X}|$) and the number of mutants ($|M_i^*|$) that are generated by each operator.

Table 2.1: Runtime complexities for generation and execution of mutants

Step	**Runtime complexity for a mutant** M^*	**Number of mutants**
(1) Mutant generation	$O(\|M^*\|)$ ($\|M^*\|$ denotes the size of the mutated model)	$\sum_{i=1}^{\|\mathcal{X}\|} \|M_i^*\|$
(2) Test generation	$O(\Phi)$	
(3) Test execution	$O(\|\Phi(M^*)\|) = O(\|T^*\|)$ ($\|T^*\|$ denotes the size of the test set)	

2.3 Summary

Mutation analysis is usually applied to programs or specifications to assess the ability of a test set T to reveal the mutants. Model-based mutation testing as proposed in this thesis integrates the concept of mutation analysis and model-based testing. Faults are injected in the model to systematically construct fault models that represent faulty behavior of SUC. Then, a test generation algorithm from MBT is applied to every mutated model to generate a test set T^*, which are then used for testing SUC. In contrast to implementation-based mutation analysis, each of the predicates *killed* and *live* has three possible interpretations that need to be further analyzed. A significant advantage is that the proposed MBMT allows mutation analysis to be applied to software (not just the model), where the source code is not available. This enables to detect faults also in SUC and at the same time to assess algorithm Φ and the test set T used in MBT by checking the mutation adequacy.

Chapter 3

Basic Mutation Operators Applied to Different Models

As explained in Chapter 1, selected graph-based models exemplify the approach introduced in this thesis. The models selected are directed graphs (DG), event sequence graphs (ESG), finite-state machines (FSM), and basic statecharts (SC). First, these models establish a hierarchy concerning their expressive power. ESG is more powerful than DG since it includes the notations of "event" and "sequence." FSM has additional notation "state," and SC extends the models with concurrency and hierarchy. Second, these models are well-known, widely accepted in practice, and are partly included in de-facto standards such as UML. Finally, these graph models consist of one-sort nodes and arcs and their formal view allows the application and, wherever necessary, adaptation of mathematical notations and techniques from graph theory and automata theory, which leads to algorithms to generate and select test cases efficiently.

The following sections present these graph-based models in such a way that the same elements are used for describing different kinds of graphs: ESG based on DG, FSM on DG, and SC on FSM. Doing so can easily demonstrate the increasing expressive power (representation capability) of these models. Two basic mutation operators–*insertion* and *omission*–are also introduced and it is discussed how to apply these operators to the models leading to model-specific mutation operators.

3.1 Initial Level: Mutation Operators for Directed Graphs (DG)

Directed graphs (DG) establish the basis for the graph-based models. The mutation operators are syntactically introduced and their basic features are discussed in this elementary level. This helps to understand the proposed operators easily.

Definition 3.1 (Directed Graph) A directed graph $DG = (N, A)$ is an ordered pair where

- N is a finite set of nodes, and
- $A \subseteq N \times N$ is a finite set of arcs.

In Definition 3.1, the nodes and arcs do not have any semantics. It is assumed that a DG has to be strongly connected; that is, a model given by an unconnected graph is invalid.

Definition 3.2 (Subgraph) A *subgraph* $DG' = (N', A')$ of a directed graph $DG = (N, A)$ is defined as follows:

$$DG' \subset DG := ((N' \subset N \wedge A' \subseteq A) \vee (N' \subseteq N \wedge A' \subset A))$$

In case that two directed graphs DG and DG' are not subgraphs of each other, this is denoted by $DG \bowtie DG' := \neg(DG \subset DG') \wedge \neg(DG' \subset DG)$

Two types of mutants (node and arc mutants) can be generated for each DG. For each type, two basic mutation operators are defined, namely *insertion* and *omission*. To generate a first-order mutant, a mutation operator χ is applied once to a DG denoted by $\chi(DG) = DG^*_\chi = (N^*_\chi, A^*_\chi)$. The notation $DG^* = (N^*, A^*)$ will also be used as a generic representation for a mutant with respect to DG^*_χ. Note that even in this initial step, the mutated graph may easily become invalid (e.g., unconnected). Only in the case that an operator would always generate invalid mutants are a minimum set of additional operations (see Figure 3.1 and Figure 3.2 as two examples) also applied to make the mutant valid (that is, strongly connected). Higher-order mutants can be generated by applying the basic operators or their combinations, multiple times.

Using the notations of Definition 3.2 three types of graph mutants DG^* can be classified, which can arise by applying the basic mutation operators to a directed graph DG.

Definition 3.3 (I-, D-, C-Mutant)

1. DG^* is an *increscent mutant* (*I-mutant*) of DG if $DG \subset DG^*$.
2. DG^* is a *decrescent mutant* (*D-mutant*) of DG if $DG^* \subset DG$.
3. DG^* is a *cross mutant* (*C-mutant*) if $DG \bowtie DG^*$.

3.1.1 Node Mutants of DG

Node mutants of a directed graph are defined as follows.

Definition 3.4 (nI-Operator) Let $\alpha \notin N$ be a node as defined in Definition 3.1. The *node insertion* operator (nI-operator) inserts α into a given DG. To keep the resulting graph valid (strongly connected), it also inserts an incoming arc to and an outgoing arc from α:

$$nI(DG, \alpha) \rightarrow DG^* = (N \cup \{\alpha\}, A \cup \{(\beta, \alpha) \cup (\alpha, \gamma)\}),$$

where β and γ are arbitrarily chosen from N ($\beta \neq \alpha$ and $\alpha \neq \gamma$).

An example is given in Figure 3.1 where a new node α is inserted and two arcs in dashed lines are also inserted to make the mutant valid.

Definition 3.5 (nO-Operator) Let $\alpha \in N$ be a node as defined in Definition 3.1. The *node omission* operator (nO-operator) removes node α from a given DG. It keeps the graph valid by also removing the arcs that have α as either the starting node or the ending node:

$$nO(DG, \alpha) \rightarrow DG^* = (N \setminus \{\alpha\}, A \setminus \{(\beta, \alpha) | (\beta, \alpha) \in A \wedge \beta \in N\} \setminus \{(\alpha, \gamma) | (\alpha, \gamma) \in A \wedge \gamma \in N\}).$$

If the nO-operator violates the validity of DG, that is, if the mutant becomes invalid (unconnected), then it is *discarded*. An example of applying the nO-operator to a DG is given in Figure 3.2.

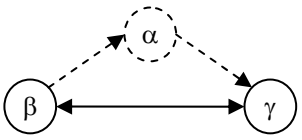


Figure 3.1: Adding necessary arcs to make a mutant valid after inserting a node α

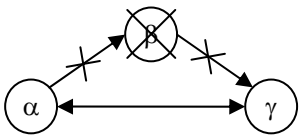


Figure 3.2: Deleting arcs to make a mutant valid after deleting node β

The nI- and nO-operators can be repeatedly applied to a DG m times, represented as nI^m and nO^m with $m \in \mathbb{N}$, respectively. They can also be combined with each other arbitrarily as illustrated in the following examples.

Example 3.1 $(nI^2 + nO^1)$ represents two nodes inserted, one node deleted or both. The notation "+" is the "or."

Example 3.2 $(nI^2 \bullet nO^1)$ represents two nodes inserted and one node is deleted. The notation "•" is the "and."

3.1.2 Arc Mutants of DG

Arc mutants of a directed graph are defined as follows.

Definition 3.6 (aI-Operator) Let $(\alpha, \beta) \notin A$ be an arc where α and $\beta \in N$ as defined in Definition 3.1. The *arc insertion* operator (aI-operator) inserts the arc (α, β) into a given DG:

$$aI(DG, (\alpha, \beta)) \rightarrow DG^* = (N, A \cup \{(\alpha, \beta)\}).$$

An example of this is shown in Figure 3.3.

Definition 3.7 (aO-Operator) Let $(\alpha, \beta) \in A$ be an arc as defined in Definition 3.1. The *arc omission* operator (aO-operator) removes the arc (α, β) from a given DG. If the resulting graph becomes invalid, the graph (that is, the mutant) is discarded:

$$aO(DG, (\alpha, \beta)) \rightarrow DG^* = (N, A \setminus \{(\alpha, \beta)\}).$$

Figure 3.4 depicts an example of omitting an arc. Similar to the nI- and nO-operators, the aI- and aO-operators can be repeatedly applied to a given DG. In addition, they can be arbitrarily combined with each other.

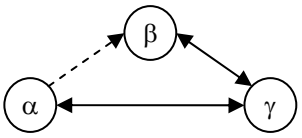


Figure 3.3: Inserting a new arc (α, β)

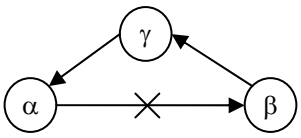


Figure 3.4: Omitting an arc (α, β)

Example 3.3 $(aI^3 + aO^2)$ represents three arcs inserted, two arcs deleted, or both.

Example 3.4 $(aI^3 \bullet aO^2)$ represents three arcs inserted and two arcs deleted.

The node mutation operators (nI and nO) can also be combined with the arc mutation operators (aI and aO).

Example 3.5 $(nI^2 \bullet aO^2)$ represents two nodes inserted and two arcs deleted.

3.1.3 Node Mutants versus Arc Mutants

A problem with mutant generation is that the application of different mutation operators can lead to the same mutants. As test cases are generated with respect to each mutant, multiple occurrences of any duplicate mutants cause inefficiency of test sets and unnecessary waste of testing resources.

Lemma 3.1 Using the notations of Definition 3.3, the following is observed.

- Mutants DG^* generated by the nI- and aI-operators are I-mutants.
- Mutants DG^* generated by the nO- and aO-operators are D-mutants.
- C-mutants can only be generated by applying an insertion operator (nI or aI) followed by an omission operator (nO or aO), or vice versa. Each operator can be applied multiple times, if necessary.

The following theorem checks whether duplicate mutants will occur while mutating a DG by the basic operators. The focus is only on the first-order mutants; that is, the mutants generated by applying nI, aI, nO, or aO exactly once to a given DG.

Theorem 3.1 First-order mutants of a DG generated by the nI-, aI-, nO-, and aO-operators are different from each other.
Proof: The proof is carried out in three steps.

- nI vs. nO, nI vs. aO, aI vs. nO, aI vs. aO:
  From Lemma 3.1 it follows that mutants are different.

- nI vs. aI:
  $DG^*_{nI} \bowtie DG^*_{aI}$ as $(A^*_{nI} \nsubseteq A^*_{aI}) \wedge (A^*_{aI} \nsubseteq A^*_{nI})$
  Hence, mutants generated by the nI-operator are different from those generated by the aI-operator.

- $nO(DG, \alpha)$ vs. $aO(DG, (\beta, \gamma))$:
  If $(\alpha = \beta \vee \alpha = \gamma)$ then
  $DG^*_{nO} \subset DG^*_{aO}$ as $(N^*_{nO} \subset N^*_{aO}) \wedge (A^*_{nO} \subset A^*_{aO})$
  else $DG^*_{nO} \bowtie DG^*_{aO}$ as $(A^*_{nO} \nsubseteq A^*_{aO}) \wedge (A^*_{aO} \nsubseteq A^*_{nO})$
  Hence, mutants generated by the nO-operator are different from those generated by the aO-operator. ■

3.1.4 Number of First-Order DG Mutants

Let $|N|$ and $|A|$ be the number of nodes and arcs of a DG. Each time an aO-operator is applied to a DG, it generates a mutant by deleting an existing arc from the graph. This implies that the maximum number of first-order mutants that can be generated by the aO-operator is the number of arcs in the graph. Similarly, each time the nO-operator is applied to a DG, it generates a mutant by deleting an existing node from the graph. Hence, the maximum number of first-order mutants that can be generated by the nO-operator is the number of nodes in the graph. The set of arcs that can be inserted into a DG without causing multiple arcs in the same direction between any two nodes is given by $(N \times N) \setminus A$. Hence, the maximum number of first-order mutants that can be generated by the aI-operator is $|N|^2 - |A|$. The number of first-order mutants generated by the nI-operator depends on the number m of nodes that can be inserted (which can, for example, be the value of $|N|$) and on the number of arcs that are connected with these nodes. Further, it is assumed that the mutated graph has to be a valid graph, which implies the necessity of having

at least one incoming arc to and one outgoing arc from each of the inserted nodes. Thus, there are at most $|N|^2$ first-order mutants for each inserted node. Table 3.1 summarizes the discussion.

Table 3.1: The maximum number of first-order mutants that can be generated with respect to a $DG = (N, A)$ by the nI-, nO-, aI-, and aO-operators

Mutation operator	aI	aO	nI	nO
Maximum number of first-order mutants generated	$\|N\|^2 - \|A\|$	$\|A\|$	$m \cdot \|N\|^2$	$\|N\|$

3.1.5 Corruption (Replacement) Operators

For convenience, a *node corruption* (nC) is defined as an operator which replaces an existing node of a given DG by another node. Note, however, that nC is not a basic operator because it can be presented as a combination of an nO-operator (to delete a node) and an nI-operator (to insert a new node, namely the node that is to replace the deleted node).

Similarly, an *arc corruption* (aC) is defined as an operator that changes the direction of an existing arc of a given DG. Again, aC is not a basic operator as it can be presented as a combination of an aO-operator (to delete an arc) and an aI-operator (to insert a new arc which is the opposite direction of the deleted arc).

Similar to the nI-, nO-, aI-, and aO-operators, the nC- and aC-operators can also be applied repeatedly to a DG n times. They can also be combined with each other and the nI-, nO-, aI-, and aO-operators, arbitrarily using "$+$" (or) and "$\bullet$" (and).

3.2 Second Level: Event-Based Interpretation of Directed Graphs–Event Sequence Graphs (ESG)

Generally, an *event* represents an externally observable phenomenon, such as an environmental or a user stimulus, or a system response punctuating different stages

of the system activity of an SUC. To take this concept into account, DG is enriched with semantic information to interpret its set of nodes as a set of events and arcs to form sequences of events leading to event sequence graphs ([10], [12], [18]). They are similar to the concept of event flow graphs [92] and are used for analysis and validation of user interface requirements prior to implementation and testing of the code. ESG can also be used to reveal structural features of graphical user interfaces (GUI) to expose further test opportunities ([40]). This thesis favors event sequence graph notation because it intensively uses formal, graph-theoretical notations, and algorithms [13], [14].

Definition 3.8 (Event Sequence Graph) An event sequence graph $ESG = (DG, \Xi, \Gamma)$ is a directed graph, where

- $DG = (E, A)$, as defined in Definition 3.1, with E being a finite set of events and $A \subseteq E \times E$ being a finite set of arcs;
- Ξ (entry events) $\subseteq E$ and Γ (exit events) $\subseteq E$ are finite sets of distinguishing events with $|\Xi| \geq 1$ and $|\Gamma| \geq 1$. $\forall e \in E$ and $e \notin \Xi$, $\exists k \in \mathbb{N}$ and $\exists \xi \in \Xi$, such that there is at least one sequence of events $(\xi, e_1, ..., e_k)$ from ξ to $e_k = e$ with $(e_i, e_{i+1}) \in A$, for $i \in \{1, ..., k-1\}$. Similarly, $\forall e \in E$ and $e \notin \Gamma$, $\exists m \in \mathbb{N}$ and $\exists \gamma \in \Gamma$, such that there exists at least one sequence of events $(e_1, ..., e_m, \gamma)$ from $e_1 = e$ to γ , with $(e_j, e_{j+1}) \in A$, for $j \in \{1, ..., m-1\}$.

The syntax of a *valid* ESG requires that every event has to be reached by an entry and that from every event an exit has to be reached, otherwise the ESG is *invalid*. To identify the entry and exit events of an ESG graphically, every $\xi \in \Xi$ is preceded by a pseudo event "[" $\notin E$ and every $\gamma \in \Gamma$ is followed by a pseudo event "]" $\notin E$.

The semantics of the arcs of an ESG are as follow. For two events, e and $e' \in E$, e' must be enabled after the execution of e if $(e, e') \in A$. Note that an ESG model focuses on events and their sequences and ignores states and state transitions. Such a representation disregards the details of internal system behavior. An ESG is a more abstract representation compared to an FSM. This will be further discussed in Section 3.3.

Example 3.6 For the ESG given in Figure 3.5, $E = \{a, b, c, d\}$, $A = \{(a, b), (b, c), (b, d), (c, a), (c, c), (c, d), (d, a), (d, c), (d, d)\}$, $\Xi = \{a\}$, and $\Gamma = \{b, c, d\}$. Note that arcs from pseudo event "[" and to pseudo event "]" are not included in A.

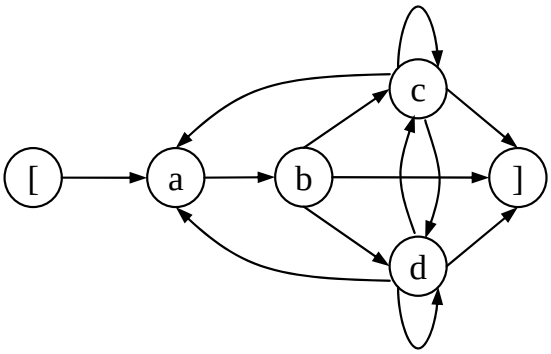


Figure 3.5: An ESG with pseudo nodes "[" and "]"

When using ESGs to model a system, it is often necessary to use the same event for similar operations in different contexts or states of the system. Such an example is the operation *copy* used to copy a picture, a file, text, etc. In such cases, the system usually carries out the corresponding action using the context information. Figure 3.6 (the left part) describes an ESG that has two occurrences of an event a. These different occurrences of a are to be indexed to distinguish between the event a that leads to b or c, and event a that can be reached via b and leads only to c. For example, a_1 represents the first a and a_2 the second a as shown in the right part of Figure 3.6.

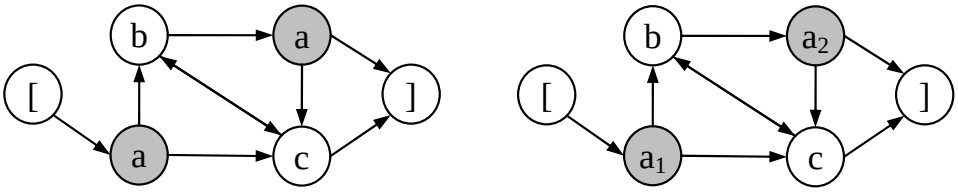


Figure 3.6: An ambiguity caused by two occurrences of event a and indexing of this event to avoid the ambiguity

It is assumed that an ESG correctly specifies the expected behavior of SUC. This ESG can then be used to generate mutants $ESG^* = (E^*, A^*, \Xi^*, \Gamma^*)$ to specify erroneous and/or undesirable behavior of SUC. These mutants help describe how the system is *not* supposed to behave. Event-based mutant generation is discussed below, followed by sequence-based mutant generation.

3.2.1 Event Mutants of ESG

Event mutants of an ESG can be generated by using

- an *event insertion* operator (eI-operator) to insert an extra event e into the ESG resulting in $E^* = E \cup \{e\}$ or
- an *event omission* operator (eO-operator) to delete an existing event e from the ESG resulting in $E^* = E \setminus \{e\}$.

Insertion of an *intrinsic* event e (that is, $e \in E$) generates an *intrinsic* mutant, whereas insertion of an *extrinsic* event e (that is, $e \notin E$) leads to an *extrinsic* mutant. Extrinsic events are events that are not included in the event set E. Their number generally depends on the nature of the SUC. If all events of the SUC are included in the model (this is typically the case for elements representing a GUI such as buttons, input fields, etc.), then it is not necessary to insert any extrinsic events which are to be considered if events are forgotten or neglected in the design phase. This thesis focuses on the intrinsic mutants.

The eO- and eI-operator are applied similarly to ESG as the corresponding nO- and nI-operator to DG (Section 3.1). For the nI-operator, as for the eI-operator, there is a need to insert an additional incoming/outgoing arc to and from the inserted event. Accordingly, for the eO-operator there is a need to remove the arcs connected with the deleted event.

3.2.2 Sequence Mutants of ESG

For a given ESG, its sequence mutants can be generated by using the *sequence insertion* operator (sI-operator) to insert an extra arc in any direction into the ESG and to make sure that no multiple arcs in the same direction between two events

are produced. Similarly, the *sequence omission* operator (sO-operator) is used to delete an existing arc from the ESG. Note that a sequence insertion extends the ESG, whereas a sequence omission reduces it. After applying the sO- or sI-operator to an ESG, the validity of the resulting ESG should be checked. If it is invalid (see Definition 3.8), the mutant is discarded.

Similar to the nI-, nO-, aI-, and aO-operators of DG, the eI-, eO-, sI-, and sO-operators can also be repeatedly applied to an ESG n times, represented as eI^n, eO^n, sI^n, and sO^n, respectively. They can also be combined with each other arbitrarily using "$+$" (or) and "$\bullet$" (and).

3.2.3 Event Mutants versus Sequence Mutants

Theorem 3.1 can be adapted to ESG as follows. Note that mutants of entry and exit events that lead to trivial constellations are not considered.

Theorem 3.2 First-order mutants of an $ESG = (DG, \Xi, \Gamma)$ with $DG = (E, A)$ generated by eI-, sI-, eO-, and sO-operators are different from each other.
Proof: The proof is carried out in three steps.

- eI vs. eO, eI vs. sO, sI vs. eO, sI vs. sO:
  From Lemma 3.1 it follows that mutants are different.

- eI vs. sI:
  $ESG^*_{eI} \bowtie ESG^*_{sI}$ as $(A^*_{eI} \not\subseteq A^*_{sI}) \wedge (A^*_{sI} \not\subseteq A^*_{eI})$
  Hence, mutants generated by the eI-operator are different from those generated by the sI-operator.

- $eO(ESG, e_1)$ vs. $sO(ESG, (e_2, e_3))$:
  If $(e_1 = e_2 \vee e_1 = e_3)$ then
  $ESG^*_{eO} \subset ESG^*_{sO}$ as $(E^*_{eO} \subset E^*_{sO}) \wedge (A^*_{eO} \subseteq A^*_{sO})$
  else $ESG^*_{eO} \bowtie ESG^*_{sO}$ as $(A^*_{eO} \not\subseteq A^*_{sO}) \wedge (A^*_{sO} \not\subseteq A^*_{eO})$
  Hence, mutants generated by the eO-operator are different from those generated by the sO-operator. ■

3.2.4 Number of First-Order ESG Mutants

Following a similar analysis for the mutants generated for a DG (Section 3.1), the maximum number of first-order mutants that can be generated by different operators for ESG is given in Table 3.2. Similar to the nI-operator, the eI-operator can generate up to $m \cdot |E|^2$ mutants, where m is the number of events that can be inserted. If m is set to $|E|$, there are at most $|E|^3$ first-order intrinsic mutants that can be generated.

Table 3.2: The maximum number of first-order mutants that can be generated with respect to an $ESG = (E, A, \Xi, \Gamma)$ by the eI-, eO-, sI-, and sO-operators

Mutation operator	sI	sO	eI	eO
Maximum number of first-order mutants generated	$\|E\|^2 - \|A\|$	$\|A\|$	$\|E\|^3$	$\|E\|$

3.2.5 Corruption (Replacement) Operators

Similar to the corruption operator of DG, an *event corruption* (eC) can be defined as an operator which replaces an existing event of a given ESG by another event. However, eC is not a basic operator since it can be viewed as a combination of an eO-operator (to delete an event) and an eI-operator (to insert the event that is to replace the deleted event). Similarly, a *sequence corruption* (sC) can be defined as an operator that changes the direction of an existing arc of a given ESG. Again, sC is not a basic operator since it can be viewed as a combination of an sO-operator (to delete an arc) and an sI-operator (to insert a new arc that is the opposite direction of the deleted arc). Iteration and combination of basic operators can be built for ESG the same way as for DG.

3.2.6 Entry and Exit Mutants of ESG

Mutants concerning the entry and exit events can be generated as follows. Note that these operators may lead to invalid mutants, such as missing entry or exit events or

missing arcs.

- An *entry omission* operator removes an event from Ξ.
- An *entry insertion* operator adds an event ($\in E$ but $\notin \Xi$) to Ξ.
- An *exit omission* operator removes an event from Γ.
- An *exit insertion* operator adds an event ($\in E$ but $\notin \Gamma$) to Γ.

Two additional corruption operators can also be defined as follows:

- An *entry corruption* operator as a combination of an *entry omission* operator followed by an *entry insertion* operator.
- An *exit corruption* operator as a combination of an *exit omission* operator followed by an *exit insertion* operator.

Neither of these two corruption operators is a basic operator for ESG. Similar to the eI-, eO-, sI-, and sO-operators, the *entry omission*, *entry insertion*, *exit omission*, *exit insertion*, *entry corruption*, and *exit corruption* operators can also be applied repeatedly to an ESG n times. These operators can also be combined with each other and the eI-, eO-, sI-, and sO-operators, arbitrarily using "$+$" (or) and "$\bullet$" (and).

3.3 Third Level: Taking States into Account–Mutation with Finite-State Machines (FSM)

An ESG consists of a finite set of events and unlabeled arcs. The Moore finite-state machines consist of states (represented as nodes) labeled by outputs, and state transitions (represented as arcs between states) labeled by inputs. As for the Mealy machines, their arcs are labeled by inputs and outputs ([67], [111], [118]). However, for each Mealy machine there is an equivalent Moore machine. Based on the DG and ESG notations (see Definition 3.1, Definition 3.8, Section 3.1 and Section 3.2), the finite-state machines used in this thesis are defined as follow.

Definition 3.9 (Finite-State Machine) A finite-state machine is defined as $FSM = (DG, E, f, S_\Xi, S_\Gamma)$, where

- $DG = (S, TR)$, as defined in Definition 3.1, where S is a finite set of states and $TR \subseteq S \times S$ is a finite set of transitions;
- E is a finite set of events;
- $f : TR \to E$ is a labeling function, mapping an event to each transition; and
- $S_\Xi, S_\Gamma \subseteq S$ are finite sets of initial and final states.

Besides the notation of "events," FSM includes the notation of "transition." Nodes represent states. Arcs, labeled by events, are transitions between states. Furthermore, the following is assumed: (1) only deterministic FSMs are considered. An FSM is valid only if all transitions are deterministic, that is, for each state $s \in S$ and each event $e \in E$, there is at most one transition that can be triggered by event e in state s. Note that even though there may be several possible transitions in a non-deterministic FSM for each pair of state and event, it is known that non-deterministic and deterministic FSMs are equivalent. One can construct an equivalent deterministic FSM for any given non-deterministic FSM and vice-versa. (2) There exists exactly one initial state ($|S_\Xi| = 1$), but there can be more than one final state ($|S_\Gamma| \geq 1$). (3) It is necessary that each state can be reached from the initial state and a final state is reachable from each state.

An FSM can be viewed as a DG with semantically distinguished states (nodes) and state transitions (arcs) that are labeled by events. Note that ESG and FSM are equivalent, both of them accepting type-3 languages [98] (regular languages), which is depicted by the next example.

Example 3.7 The FSM in Figure 3.7 is represented by $S = \{s_1, s_2, s_3\}$, $TR = \{(s_1, s_2), (s_2, s_3)\}$, $E = \{a, b\}$, $f = \{((s_1, s_2), a), ((s_2, s_3), b)\}$, $S_\Xi = \{s_1\}$, and $S_\Gamma = \{s_3\}$.

An arrow from nowhere to a node indicates the initial state, and arrows from a node to nowhere indicate final states. States and inputs of an FSM can be merged to derive the corresponding ESG. For example, Figure 3.8 gives the corresponding ESG for the FSM in Figure 3.7.

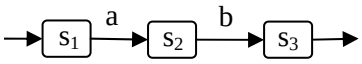


Figure 3.7: An FSM, which is equivalent to the ESG in Figure 3.8

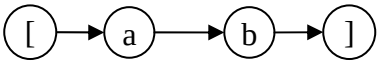


Figure 3.8: An ESG, which is equivalent to the FSM in Figure 3.7

3.3.1 State Mutants of FSM

State mutants of an FSM can be generated by using a *state insertion* operator (stI-operator) or a *state omission* operator (stO-operator). An stI-operator inserts an extra state into S. It also requires the insertion of an additional incoming/outgoing transition-event pair to/from the inserted state. An stO-operator deletes an existing state from S. All transitions connected with the deleted state should also be removed from the FSM. This is done in a similar way as the nO-operator for DGs and the eO-operator for ESGs. The function f also needs to be updated accordingly with respect to the stI- and stO-operators by adding or removing appropriate mappings between transitions and events.

Similar to the nI-, nO-, eI-, and eO-operators, the stI- and stO-operators can also be applied repeatedly to an FSM n times. This is represented as stI^n and stO^n with $n \in \mathbb{N}$, respectively. They can also be combined with each other an arbitrary number of times.

3.3.2 Transition Mutants of FSM

Transition mutants of an FSM can be generated as follows:

- Applying a *transition insertion* operator (tI-operator) to add a transition $tr = (s, s')$ into TR, where s and $s' \in S$. The transition has to be triggered by an intrinsic or extrinsic event e and moves the FSM from state s to state s'.
- Applying a *transition omission* operator (tO-operator) to delete an existing transition tr from TR. The function f also needs to be adapted accordingly.

The tI- and tO-operators can be applied repeatedly to an FSM n times. This is represented as tI^n and tO^n with $n \in \mathbb{N}$, respectively. They can also be combined with each other or with the stI- and stO-operators an arbitrary number of times.

3.3.3 State Mutants versus Transition Mutants

Theorem 3.3 First-order mutants of an $FSM = (DG, E, f, S_\Xi, S_\Gamma)$ with $DG = (S, TR)$ generated by the stI-, tI-, stO-, and tO-operators are different from each other.

Proof: The proof is carried out in three steps.

- stI vs. stO, stI vs. tO, tI vs. stO, tI vs. tO:
  From Lemma 3.1 it follows that mutants are different.

- stI vs. tI:
  $FSM^*_{stI} \bowtie FSM^*_{tI}$ as $(TR^*_{stI} \nsubseteq TR^*_{tI}) \wedge (TR^*_{tI} \nsubseteq TR^*_{stI})$
  Hence, mutants generated by the stI-operator are different from those generated by the tI-operator.

- $stO(FSM, s_1)$ vs. $tO(FSM, (s_2, s_3))$:
  If $(s_1 = s_2 \vee s_1 = s_3)$ then
  $FSM^*_{stO} \subset FSM^*_{tO}$ as $(S^*_{stO} \subset S^*_{tO}) \wedge (TR^*_{stO} \subseteq TR^*_{tO})$
  else $FSM^*_{stO} \bowtie FSM^*_{tO}$ as $(TR^*_{stO} \nsubseteq TR^*_{tO}) \wedge (TR^*_{tO} \nsubseteq TR^*_{stO})$
  Hence, mutants generated by the stO-operator are different from those generated by the tO-operator. ■

3.3.4 Number of First-Order FSM Mutants

The maximum number of first-order mutants that can be generated for an FSM is given in Table 3.3. The number of mutants generated by tO- and stO-operators is given by the cardinality of the corresponding sets. The determination for the tI-operator is more intricate because there are at least $|S|^2$ possibilities of inserting an additional arc. Each arc can be labeled by an event. Subtraction of all existing transitions results in a maximum of $|S|^2 \cdot |E| - |TR|$ mutants. Note that non-deterministic, and therefore invalid, mutants may be included. The number of stI-mutants is given by the number of transitions that can be inserted from an existing state to a new state ($\leq |S| \cdot |E|$) multiplied by the number of possible transitions from the new state to an existing one ($|S| \cdot |E|$).

Table 3.3: The maximum number of first-order mutants that can be generated with respect to an $FSM = (DG, E, f, S_\Xi, S_\Gamma)$ by the tI-, tO-, stI-, and stO-operators

Mutation operator	tI	tO	stI	stO
Maximum number of first-order mutants generated	$\|S\|^2 \cdot \|E\| - \|TR\|$	$\|TR\|$	$(\|S\| \cdot \|E\|)^2$	$\|S\|$

3.3.5 Corruption (Replacement) Operators

Corruption operators can also be defined for FSM. However, none of them is a basic mutation operator since each can be viewed as a combination of an insertion operator and an omission operator. A *state corruption* operator (stC-operator) replaces an existing state of a given FSM by another state. This is equivalent to the application of an stO-operator followed by an stI-operator. Differing from the arc corruption (aC) for DGs and the *sequence corruption* (sC) operator for ESGs, a *transition corruption* operator (tC-operator) for FSMs generates two types of corrupted mutants. First, it replaces the event e that triggers a transition by another event, $e' \in E$ or $e' \notin E$. Second, it reverses the direction of a transition (that is, swapping the source and target states of a transition). Each scenario can be viewed as the application of a tO-operator followed by a tI-operator. After applying the tO- or tC-operator, it is important to check the validity of the resulting FSM. Invalid mutants should be discarded. It is also important to update the labeling function f accordingly. The stC- and tC-operators can be combined with each other and the stI-, stO-, tI-, and tO-operators arbitrarily using "$+$" (or) and "$\bullet$" (and).

3.3.6 Entry (Initial State) and Exit (Final State) Mutants of FSM

Mutants of initial and final states of FSM are generated in analogy to ESG (see Section 3.2).

- An *entry omission* operator removes a state from S_Ξ.
- An *entry insertion* operator adds a state ($\in S$ but $\notin S_\Xi$) to S_Ξ.

- An *exit omission* operator removes a state from S_Γ.
- An *exit insertion* operator adds a state ($\in S$ but $\notin S_\Gamma$) to S_Γ.

Note that the *entry omission* and *entry insertion* operators generate invalid mutants. They are solely defined to be used for the *entry corruption* operator, which is a combination of an *entry omission* operator, followed by an *entry insertion* operator. Similarly, there is an *exit corruption* operator as a combination of an *exit omission* operator followed by an *exit insertion* operator.

Neither of these two corruption operators is a basic operator for FSM. All the operators for the initial and final states can be combined with each other and the stI-, stO-, tI-, and tO-operators arbitrarily using "$+$" (or) and "$\bullet$" (and).

3.4 Advanced Level: Considering Concurrency and Hierarchy–Mutation with Statecharts (SC)

At the last step, concurrency and hierarchy are also considered in addition to events, states, and state transitions. Statecharts (SC) include these additional aspects and are also popular in research and industrial practice [70]. This might explain the variety of statechart notations that differ slightly from each other in syntax but significantly in their execution semantics [122]. To avoid such a notational problem, and to be consistent with the representations starting with ESG based on DG and FSM on ESG, an alternative representation of statecharts [22] is given in Definition 3.10. It is based on FSM and includes common key features of most of the existing statechart representations. A more comprehensive statechart notation can be found in [81].

Definition 3.10 (Basic Statechart) A *basic statechart* is given by $SC = (FSM, H, g)$, where

- $FSM = (S, TR, E, f, S_\Xi, S_\Gamma)$ as defined in Definition 3.9,
- $H \subseteq S \times S$ is a hierarchy relation, and

- $g : S \rightarrow \{simple, and, xor\}$ is a labeling function. It defines the following sets of states:
  - $S_{\text{and}} := \{s | s \in S \wedge g(s) = and\}$
  - $S_{\text{xor}} := \{s | s \in S \wedge g(s) = xor\}$
  - $S_{\text{simple}} := \{s | s \in S \wedge g(s) = simple\}$
  - $S_{\text{comp}} := S_{\text{and}} \cup S_{\text{xor}}$

A set H is used to represent the hierarchy relation between nested states. Moreover, each state $s \in S$ is identified as either a simple state ($s \in S_{\text{simple}}$) or a composite state ($s \in S_{\text{comp}}$), which is further classified as either an AND-state ($s \in S_{\text{and}}$) or an XOR-state ($s \in S_{\text{xor}}$). Simple states, in contrast to composite states, may not have sub-states. Each XOR-state owns an immediate sub-state, which is marked as the *initial state*. If an XOR-state is active, then exactly one of its immediate sub-states must also be active. The immediate sub-states of AND-states represented by XOR-states are called *regions*. It means that the statechart resides simultaneously (concurrently) in each region of the AND-state; that is, in contrast to XOR-states each immediate sub-state (region) of an AND-state is an initial state.

The sets of initial and final states are denoted by S_Ξ and S_Γ, whereas $|S_\Xi| \geq 1$ and $|S_\Gamma| \geq 1$. The final states represent possible exits of the system. The set H defines a binary relation on the set S that must form a tree and be consistent with g in order to be valid. Given two s and $s' \in S$, if $(s, s') \in H$, then s' is an immediate sub-state of s. Transitions must be deterministic and associated with an event. The source and target states of a transition may also consist of composite states. This corresponds to a (graphical) simplification since such transitions may be replaced by several transitions whose source and target consist of simple states.

Table 3.4 depicts functions used to formalize properties concerning the sets. The functions in and out deliver all transitions that lead to or from a simple state. They also comprise transitions that are located in different tiers of the hierarchy.

An example of a simple statechart SC is given in Figure 3.9 where $S = \{s_1, s_2, s_3\}, TR = \{tr_1 = (s_2, s_3), tr_2 = (s_3, s_2)\}, E = \{e_1, e_2\}, f = \{(tr_1, e_1), (tr_2, e_2)\}, S_\Xi = \{s_1, s_2\}, S_\Gamma = \{s_3\}, H = \{(s_1, s_2), (s_1, s_3)\}$, and $g = \{(s_1, xor), (s_2, simple), (s_3, simple)\}$. A more comprehensive example of a statechart with an AND-state

Table 3.4: Statechart functions

Function	Description
$root: \quad \rightarrow S$	delivers the root state $r \in S$
$initial : S_{\text{xor}} \rightarrow S$	delivers the initial state from the immediate substates of $s \in S_{\text{xor}}$
$in : S_{\text{simple}} \rightarrow \mathcal{P}(TR)$	delivers all transitions that lead to a simple state s
$out : S_{\text{simple}} \rightarrow \mathcal{P}(TR)$	delivers all transitions that lead from a simple state s

consisting of two regions ("Cutter unit" and "Automatic modus") is given by Figure 6.1 in Chapter 6. How the two basic mutation operators (insertion and omission) apply to statecharts is explained in the following.

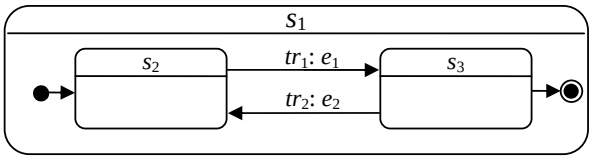


Figure 3.9: Example of a simple statechart

3.4.1 Mutants of SC

State mutants of SCs are generated the same way as those of FSMs. The stO-operator removes a state and associated transitions from an SC, whereas the stI-operator adds a new state and an incoming/outgoing transition to/from this state to an SC. Functions f and g, as well as the hierarchy relation H, also need to be updated accordingly. In most cases, the stI- and stO-operators only apply to simple states. Removing/inserting composite states from/into an SC typically results in invalid mutants. Removing a composite state would leave its sub-states and transitions "exposed" (that is, promoting them to their parents' level in the H graph). Adding an enclosing state (new level in the hierarchy) would lead to missing initial or final states. An example of applying the stI-operator to the statechart in Figure 3.9 is given in Figure 3.10. Note that the following sets have to be updated:

$S^* = S \cup \{s_4\}$, $TR^* = TR \cup \{tr_3 = (s_2, s_4), tr_4 = (s_4, s_3)\}$, $E^* = E \cup \{e_3\}$, $f^* = f \cup \{(tr_3, e_3), (tr_4, e_1)\}$, $H^* = H \cup \{(s_1, s_4)\}$, and $g^* = g \cup \{(s_4, simple)\}$.

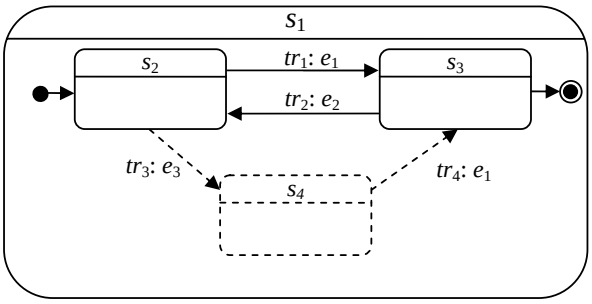


Figure 3.10: A sample stI-mutant of the SC in Figure 3.9

Transition, entry, exit and corruption mutants for SC are defined in the same way as for FSM. After applying these operators, the validity of mutated SCs has to be checked against Definition 3.9 and Definition 3.10. Invalid mutants are discarded. An example of a tI-mutant of the statechart in Figure 3.9 is given in Figure 3.11. Consequently, the following sets of the statechart also have to be updated: $TR^* = TR \cup \{tr_3 = (s_3, s_3)\}$ and $f^* = f \cup \{(tr_3, e_1)\}$.

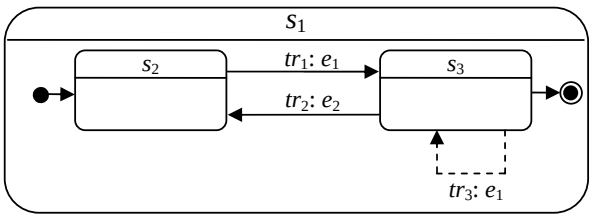


Figure 3.11: Example of tI-operator: A new transition tr_3 is inserted

The tI-, tO-, stI-, and stO-operators for an SC, as well as the corruption operators, can be combined and repeated arbitrarily using "$+$" (or) and "$\bullet$" (and).

3.4.2 Hierarchy and Concurrency Mutants of SC

The hierarchy and concurrency mutants of an SC can be generated by mutating the hierarchy relation H and the labeling function g as given in Definition 3.10. How-

ever, the two basic mutation operators (*insertion* and *omission*) discussed in this thesis cannot be applied to H and g without generating invalid mutants (because the mutated SCs do not satisfy the conditions set by Definition 3.10). For example, adding/removing an element (s, s') into/from H invalidates the underlying statechart because the binary relation defined by H on S will no longer form a tree. On the other hand, the function g can be mutated to convert a simple state into a composite state and vice versa. Also, the AND- and XOR-states may be exchanged. However, in all cases the resulting statecharts become invalid. To sum up, further research is necessary to define appropriate operators to generate concurrency and hierarchy mutants of SC. First approaches in that direction have recently been described by Trakhtenbrot ([119], [120]). He defined informal mutations to validate statechart features such as hierarchy, orthogonality and time expressions.

3.5 Comparison of Mutation Operators

Mutation analysis typically makes use of a cluster of different operators. In contrast, basic operators consist of a small set of operators. A benefit of using such a set is also justified by previous and current research work suggesting that even a small set of operators is sufficient ([96], [99], [102]). Table 3.5 summarizes the differences between implementation- and specification-based mutation operators from literature and basic mutation operators introduced here. Corruption operators correspond to higher-order mutants, having the capability to potentially reveal more faults. Typically, they represent first-order operators as used in implementation- and specification-based mutation analysis. First-order basic operators are not intended to be applied at the program level. Pure insertion or omission of elements based on programs commonly results in the creation of many invalid mutants due to incorrect syntax.

3.6 Summary

Two basic mutation operators (*omission* and *insertion*) supersede and systematize the broad variety of existing mutation operators. These two operators are generic

in the sense that they can apply to any model of SUC. Iteration and combination of these basic operators increases their expressive power. Applications of these operators to models in order of increasing expressive power have been defined for DG, followed by the enhancement of their semantics for ESG, FSM, and SC.

In addition, higher-order mutants can be generated by multiple iterations and/or combinations of the basic operators whenever necessary. Hence, it is clear that the two basic mutation operators and their combinations and iterations not only are sufficiently comprehensive but also subsume many operators reported in the literature.

Table 3.5: Implementation- and specification-based mutation operators versus basic mutation operators

	Implementation- and specification-based mutation operators	**Basic mutation operators**
Number of operators	Arbitrarily; typically a cluster of operators	Insertion and omission
Fault model	Operators are chosen to simulate typical faults a programmer/modeler might make	Fault models are not explicitly defined
Construction	By experience/empirical aspects	Systematically
Intended level to be applied to	Programs, Specifications & Models	Specifications & Models
Order of mutants typically used	First-order mutants	First-order mutants, second-order mutants (corruption operators)
Subsumption	These mutation operators can be derived from basic operators	–

Chapter 4

Test Generation and Tool Support

A test process is developed using the definitions and notations introduced in Chapter 2 and Chapter 3. As DG has no semantics, the focus is on ESG, FSM, and SC, which are used to model SUC. Section 4.3 describes tools for effectively performing the test process. The algorithms underlying these tools for test generation are briefly discussed and illustrated by examples in Section 4.1 and Section 4.2.

4.1 ESG-Based Test Generation

To generate test cases, a test generation algorithm Φ is applied to an ESG model. The following explains the process $\Phi(ESG)$.

Definition 4.1 (Event Sequence) Let E and A be the finite set of events and arcs of an ESG. Any sequence of events $(e_1, ..., e_k)$ with $e_i \in E$ for $i \in \{1, ..., k\}$ is called an *event sequence* (ES), if $(e_i, e_{i+1}) \in A$ for $i \in \{1, ..., k-1\}$.

Let α and ω be the functions to determine the first and the last events of an ES; for example, for $ES = (e_1, ..., e_k)$, the first event is $\alpha(ES) = e_1$ and the last event is $\omega(ES) = e_k$, respectively. Also, let the function l (symbolizing *length*) return the number of events of an ES.

Example 4.1 For the ESG in Figure 3.5, $bcdc$ is an ES of length 4 with b and c as the first and the last events.

Definition 4.2 (Complete Event Sequence) An ES is a *complete event sequence* (CES) if $\alpha(ES) \in \Xi$ and $\omega(ES) \in \Gamma$.

Example 4.2 Referring to Figure 3.5, the event sequence abc is a CES.

Each CES represents a walk from the entry of an ESG to its exit, realized by a chain of user inputs and system responses. As explained earlier, a test case is an ordered pair of an input and an expected output of the SUC. A test set can contain any number of test cases. A CES of an ESG can be used as a test case for the corresponding SUC; its execution is in general expected to be successful. The test cases are thereby formed by the events, particularly by the user activities (inputs) and the expected system responses (outputs).

Definition 4.3 (Fault Model) A test execution with respect to a CES *fails* if the CES cannot reach the corresponding exit event due to a failure in SUC or reaches an exit event but does not produce the expected outputs.

Definition 4.4 (k-Event Coverage) Generate complete event sequences that sequentially conduct all event sequences of length $k \in \mathbb{N}$.

Based on the above definitions, Algorithm 4.1 describes for a given ESG a coverage-based test generation process to produce a set of CES to cover all event sequences of a required length. To minimize the total length of a test set, solutions of the *Chinese Postman Problem* can be used; that is, finding the shortest path or circuit in a graph by visiting each arc. Algorithms supporting this test generation process have been published in previous work ([13], [14]). For the case studies in Chapter 5, test sets to cover event sequences of length $k = 2$ are generated as a minimal requirement to cover all events and arcs.

The ESG in Figure 4.1 is used as an example for ESG-based test generation: $ES_2 = \{\text{event sequences of length } 2\} = \{(a, b), (b, a), (b, d), (c, a), (c, d), (d, c)\}$. To cover these sequences, a test set $T_2 = \{(c, a, b, a, b, d, c, d)\}$ is generated containing one complete event sequence. T_2 has a total length of 8, which is the minimum among all the test sets that can cover the six event sequences in ES_2. Similarly, $ES_3 = \{\text{each sequence of length } 3\} = \{(a, b, a), (a, b, d), (b, a, b), (b, d, c), (c, a, b), (c, d, c), (d, c, a), (d, c, d)\}$. To cover ES_3, a test set $T_3 = \{(c, a, b, a, b, d, c, a, b, d),$

Algorithm 4.1: An ESG-based test generation process $\Phi(ESG)$

Input: An ESG
A number $n \in \mathbb{N}$ as sequence length

Output: A test set $T = \{t_1, t_2, ..., t_m\}$ to cover all event sequences of length n. Each test case in T is a CES.

1 Generate a set of CES to cover all event sequences of length n in the ESG;

2 Minimize the total length $\sum_{i=1}^{m} l(t_i)$ of test set T;

$(c, d, c, d)\}$ is generated with two complete event sequences. T_3 has a total length of $14(= 10 + 4)$, which is the minimum among all the test sets that can cover all event sequences in ES_3.

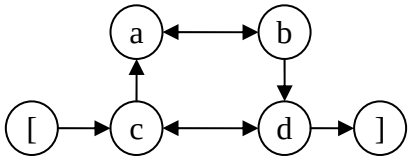


Figure 4.1: A sample ESG for the ESG-based test generation

4.2 FSM/SC-Based Test Generation

Test generation for FSMs and basic SCs is conducted similarly as for ESGs by covering their transitions and, thereby, also states [22]. Some notations required for the FSM/SC-based test generation process (Algorithm 4.2) are introduced in the following.

Definition 4.5 A *transition pair* (tr, tr') with $tr, tr' \in TR$ is a sequence of an incoming transition to an outgoing transition of a (simple) state so that $\exists\ s \in S_{\text{simple}} : tr \in in(s) \wedge tr' \in out(s)$.

Sequences of transitions can now be defined as follows.

Definition 4.6 (Transition Sequence) A sequence of k transitions $(tr_1, ..., tr_k)$ with $tr_i \in TR$ for $i\ in\{1, ..., k\}$, where TR is defined as in Definition 3.9 and where (tr_i, tr_{i+1}) denotes a valid transition pair for all $i \in \{1, ..., k-1\}$ is called a *transition sequence* (TS) of length k. The function l (standing for *length*) determines the number of transitions of a TS.

Definition 4.7 (Complete Transition Sequence) A sequence $(tr_1, ..., tr_k)$ is a *complete transition sequence* (CTS) if and only if tr_1 starts at the initial state of the statechart that is entered firstly and tr_k ends at a final state.

As a precondition for setting up a coverage criterion a fault model is used as introduced in the previous section for ESG. A test (of a complete transition sequence) is assumed to fail if a final state cannot be reached, or a final state is reached, but the expected result differs from the actual result. This assumes that only the final states of the statechart can be observed. Thus, the *oracle problem* of specifying the expected outcome for a specified input is solved in accordance with the fault model as follows. It is assumed that a test based on complete transition sequences will succeed. Based on Definitions 4.6 and 4.7 the following coverage criterion is defined.

Definition 4.8 (k-Transition Coverage) Generate complete transition sequences that sequentially conduct all transition sequences of length $k \in \mathbb{N}$.

Definition 4.8 guarantees that all possible transition sequences of length k will be tested. A test set consisting of all transition sequences of a fixed length k does not necessarily cover a set of all sequences of length $i \in \{1, ...k-1\}$ as there may exist sequences of length i that cannot be expanded to length k.

The process in Algorithm 4.2 describes a coverage-based test generation process for FSM and SC to generate a set of CTS to cover all transition sequences of a given length. Note that each $CTS = (tr_1, ..., tr_k)$ is mapped to a complete sequence of events $CES = (f(tr_1), ..., f(tr_k)) = (e_1, ..., e_k)$ so that it can be applied to the underlying SUC. For the case studies in Chapter 5, test sets to cover transition sequences of length $k = 1$ are generated as a minimal requirement to cover all states and transitions.

Algorithm 4.2: An FSM/SC-based test generation process $\Phi(FSM/SC)$

Input: An FSM/SC
A number $n \in \mathbb{N}$ as sequence length

Output: A test set $T = \{t_1, t_2, ..., t_m\}$ to cover all transition sequences of length n. Each test case in T is a CTS.

1 Generate a set of CTS to cover all transition sequences of length n in the FSM/SC;

2 Minimize the total length $\sum_{i=1}^{m} l(t_i)$ of test set T;

3 Map all $CTS = (tr_1, ..., tr_k)$ to $CES = (f(tr_1), ..., f(tr_k)) = (e_1, ..., e_k)$;

4.3 Tool Support

The approach described in the previous sections is straightforward with a simple structure. The large amount of data to be gathered and analyzed is, however, hardly feasible and, equally critical, error-prone when processed manually. Therefore, a chain of tools, available under a uniform GUI, was developed to cope with the scalability problem in large projects. They have also been used to conduct the case studies in Chapter 5.

Some of the tools are add-ons to commercial ones, while others implement the test algorithms described in the previous sections. An ESG-based *mutant and test set generator* (MTSG) forms the heart of the tool chain. A snapshot in Figure 4.2 represents its entry window. For mutant generation, MTSG computes first-order mutants with respect to each operator. For example, the configuration in Figure 4.2 indicates that 50% of the sI-mutants, 150 of the sO-mutants, 0% of the eI-mutants, and 100% of the eO-mutants are generated. For ESG-based test generation, MTSG checks the validity of the model analyzed as the first step before a test set is generated to cover all event sequences of length n. Each test case in this set is a CES (a complete event sequence) of the ESG. To reduce the execution cost, another unit of MTSG generates a test set in such a way that the total length of all the CES contained is minimal. This is done by using the algorithm for solving the Chinese Postman Problem (see Section 4.1; [13], [14]).

Commercial capture/playback tools (as for *WinRunner* HP, formerly Mercury

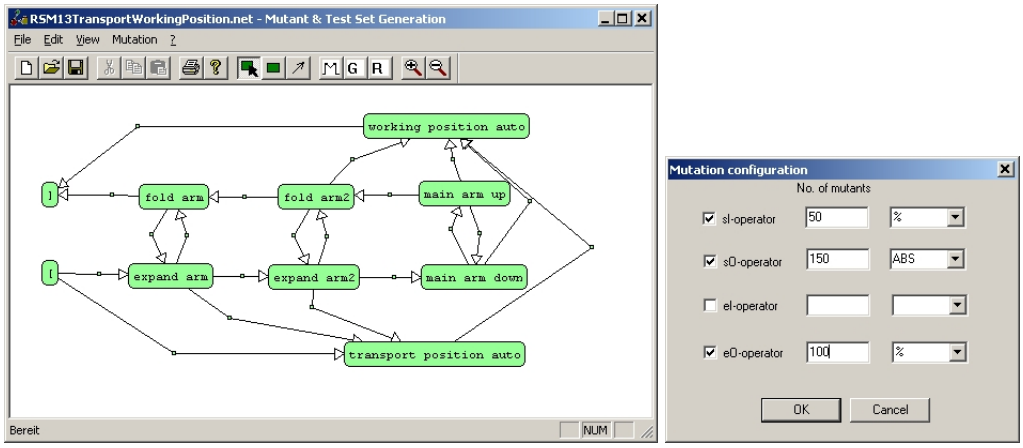


Figure 4.2: The entry window of MTSG and window for specifying the number or percentage of mutants to be generated with respect to each operator

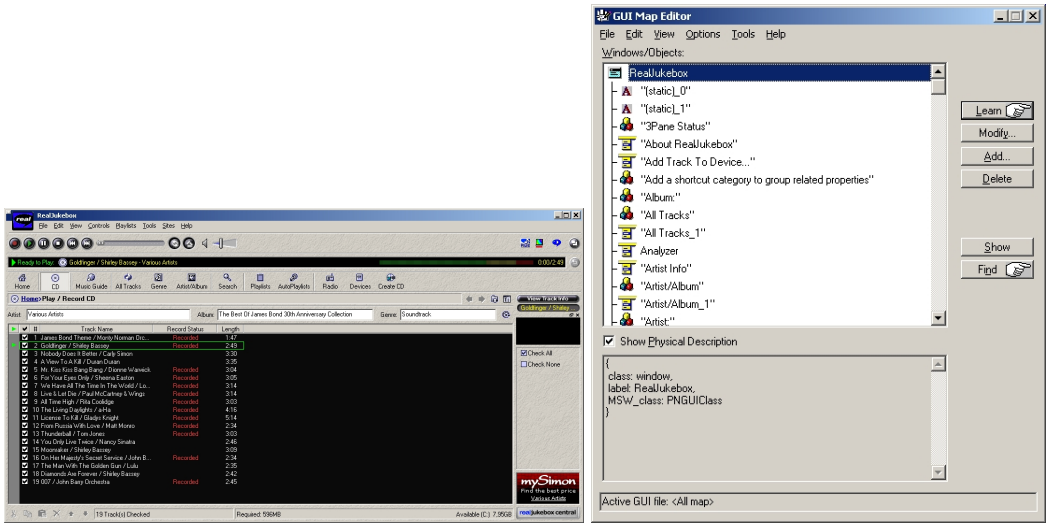


Figure 4.3: Part of the SUC of the first case study (RealJukebox) and properties of some captured GUI elements in WinRunner

Interactive) help automate the testing of GUIs. Modern GUIs consist of elements such as buttons, checklists and labels. These elements and their properties, such as position, size, type, and window they belong to, can automatically be captured. Figure 4.3 depicts a dialog containing different elements of the SUC from the first case study. A built-in test script language is provided for automating test execution. Test scripts can either be implemented manually or generated by recording the user's interactions with SUC. The MTSG tool can generate test sets consisting of sequences of events to be transformed into the WinRunner test scripts. Figure 4.4 shows the main window displaying part of a test script.

The tool BAT ("Bibliotheksbasierte Annotierung von Testmodellen") has been developed to generate executable test scripts directly from ESG and FSM ([127]). The main screen which is shown by Figure 4.5 offers a uniform GUI for ESG and FSM models using directed graphs as the underlying data structure. The user can set up specific code libraries. This is done by inserting different code fragments of a capture/playback tool such as WinRunner or Selenium ([113]) which is a popular plug-in for Firefox browser for automating the tests of web applications. During the course of modeling, the user annotates the events (in case of ESG) or transitions (in case of FSM) by these code fragments. Concluding, test generation is done according to Algorithm 4.1 for ESG and Algorithm 4.2 for FSM. As a result of the test generation process a test script is generated to be executed by the corresponding capture/playback tool on SUC.

4.4 Summary

A coverage-based test generation process for ESG and FSM/SC has been defined by covering all events/transitions of a fixed length $k \in \mathbb{N}$ by means of complete event/transition sequences. The tool MTSG implements such a coverage-based test process for ESG, minimizes the test sequence length and helps to generate different mutants. The tool BAT generates test scripts based on user-defined code libraries for capture/playback tools such as WinRunner or Selenium. BAT also enables the user to choose between ESG and FSM models.

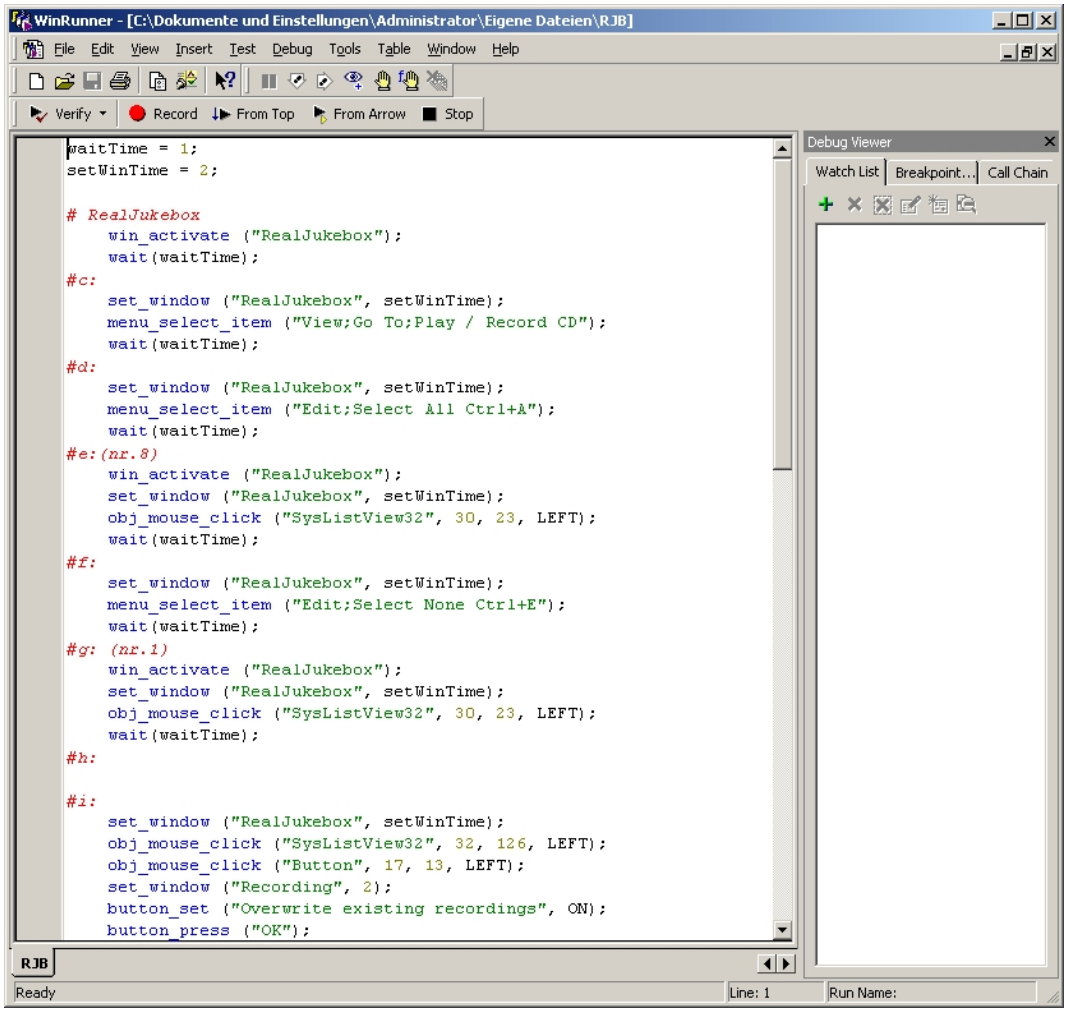


Figure 4.4: A sample test script to be executed against SUC

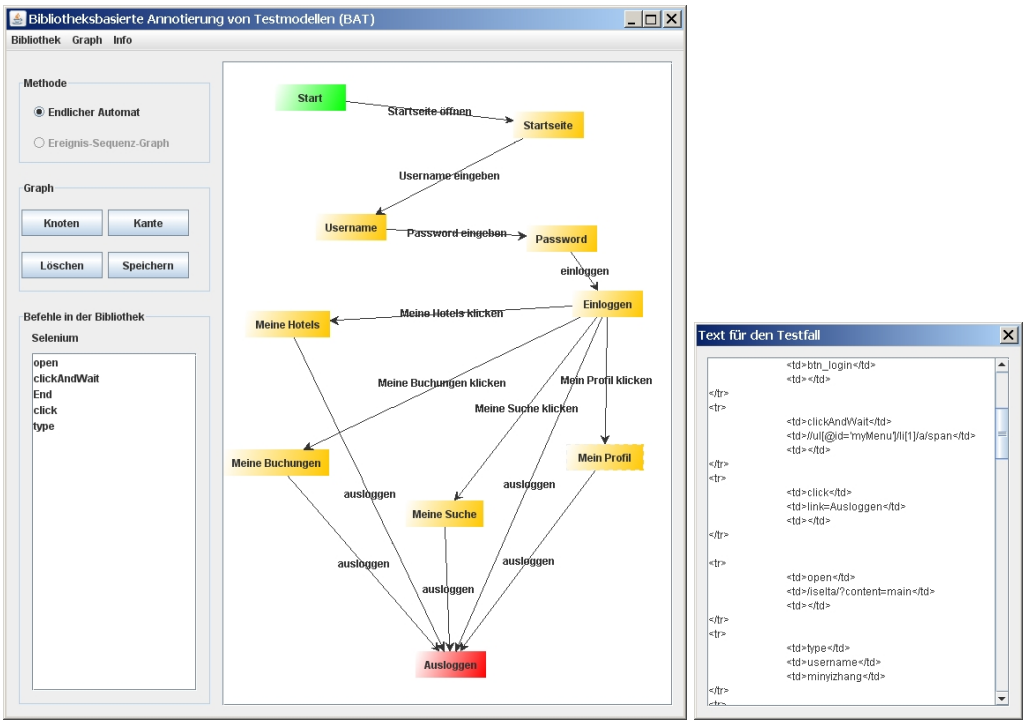


Figure 4.5: Entry window of the software BAT and test script generated

Chapter 5

Three Case Studies

This chapter validates the proposed MBMT and determines its characteristic features using three case studies on three different industrial and commercial real-life projects:

- An interactive commercial music management system (Section 5.1)
- A reactive/adaptive, real-time cruise control system (Section 5.2)
- A proactive control desk unit of a marginal strip mower mounted on a truck (Section 5.3)

The diversity of the case studies increases confidence that the results are representative. Furthermore, this diversity helps to demonstrate the wide range of applicability of the proposed approach. The first two studies use ESGs (Definition 3.8) for modeling the system behavior while the third study uses ESGs and SCs (Definition 3.10). FSMs have not explicitly been considered in the case studies since they are equivalent to ESG and form a subset of SCs. Objectives of the studies are to

- analyze the fault detection capability of test sets generated using the proposed MBMT approach which is done by
  - determining whether there are still remaining faults in SUC that can be detected by these test sets; and

  - measuring the score $MFDS$ (Definition 2.5)

- assess the mutation adequacy of test sets generated using the proposed MBMT approach to kill the mutants measured by the score $TGMAS$ (Definition 2.6);
- compare the fault detection capability of test sets generated by ESG-based mutation versus SC-based mutation, that is,
  - test sets generated based on sequence and event mutants of ESG; and
  - test sets generated based on transition and state mutants of SC.

Generation, execution, and classification of mutants was carried out as described by Algorithm 2.1 in Section 2.2. Test sets for ESG mutants were generated based on the process described in Algorithm 4.1 to cover all events and arcs (sequences), that is, all event sequences of length $k = 2$. Similarly, a test set was generated for each SC mutant based on the process in Algorithm 4.2 to cover all transition sequences of length $k = 1$, that is, to cover all states and transitions. Thus, there are as many test sets as mutants. The number and the size, that is, the number of events, of test cases per test set, is variable and is determined by algorithms to solve the Chinese Postman Problem for the specific ESG that models SUC. However, the test effort primarily depends on the size of the test cases and not their number. To evaluate the fault detection capability of test sets generated by the test generation processes described in Section 4.1 and 4.2 (Algorithm 4.1 and Algorithm 4.2) with respect to Definition 2.5 and Definition 2.6, first-order mutants were generated by applying the sI-, sO-, and eO-operators to ESGs and the tI-, tO-, and stO-operators to SCs. For reasons of costs and practicability, eI- and stI-mutants are not considered in the case studies. Compared to sI- and tI-mutants, they would have produced considerably more mutants. Furthermore, in cases with graphical user interfaces, these mutants represent a similar fault model as sI- and tI-mutants.

5.1 Case Study I: Music Management System–an Interactive System

RealJukebox (RJB) is an interactive personal music management software developed by RealNetworks [108] for PCs running the Microsoft Windows operating system.

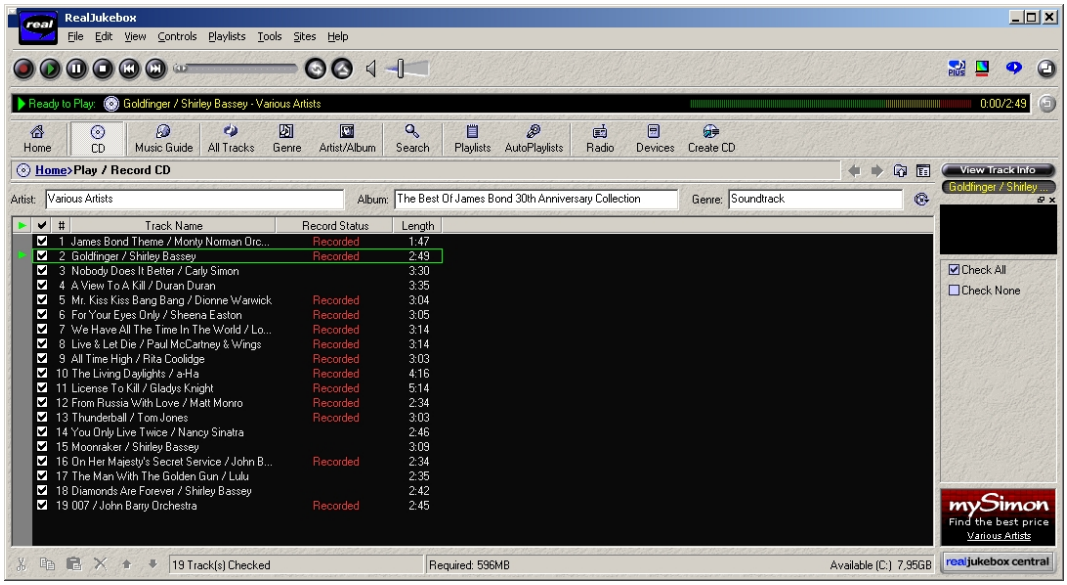


Figure 5.1: Main window of RealJukebox

5.1.1 System Under Consideration

The basic English version of RealJukebox 2 (build 1.0.2.340) of RealNetworks was used as the SUC. It is a music management system used to build, manage, and play digital music libraries on a personal computer. RJB supports different media types such as compact discs, audio files, and Internet services. The main GUI of RJB (see Figure 5.1), has several menus (“File,” “Edit,” etc.) that invoke other components. Each one has further sub-options. There are additional window components that duplicate the functionality of the menus and sub-menus, creating many possible

combinations and, accordingly, many applications. In this study, RJB was installed on a PC running Windows XP with Service Pack 2 and satisfying the recommended hardware requirements (300 MHz CPU, 64 MB RAM, full duplex sound card, 16 bit color video card) as described in the product's manual.

5.1.2 Modeling of SUC

Due to the lack of a manufacturer's system specification, that is, a functional description of RJB, the help facilities and handbook of RJB were used to produce the references for generating test cases based on ESGs. These functions describe the steps as to how to reach situations the user wants, that is, desirable events in terms of system functions (responsibilities). Seven functions of RJB were used in this study. Each function represents a complete interaction scenario for a well-defined task.

- Playing and recording a compact disc or a track
- Creating and playing a playlist
- Editing playlists and/or auto playlists
- Viewing lists and/or tracks
- Editing a track
- Changing the application views by using different skin layouts
- Configuring RJB

Each function is modeled by an ESG and several corresponding refinements as sub-ESGs. Figures 5.2, 5.3, and 5.4 depict sample mutants generated by the sI-, sO-, and eO-operators. In Figure 5.2, the sI-operator is applied to insert an arc from *Jump Beginning* to *Track position*, whereas in Figure 5.3 the arc from *Jump Beginning* to *Stop* is removed by the sO-operator. In Figure 5.4, the event *track position* is removed by an eO-operator. All arcs connected with this event are also removed.

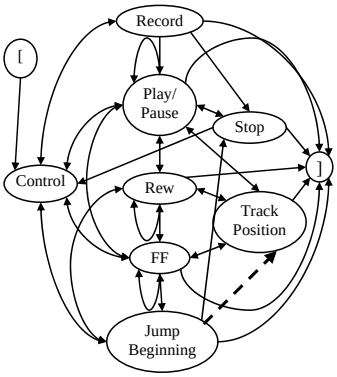


Figure 5.2: Application of sI-operator

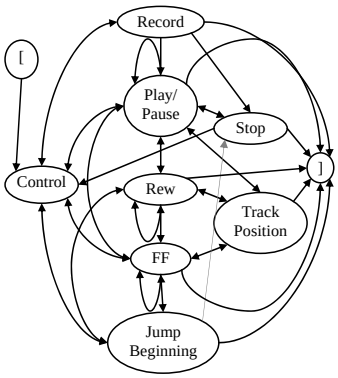


Figure 5.3: Application of sO-operator

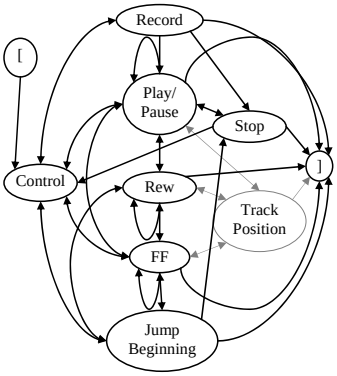


Figure 5.4: Application of eO-operator

5.1.3 Mutant Generation and Results

Table 5.1 lists the number of mutants that were generated with respect to each mutation operator and the number of faults detected in the deployed application (see Table 5.2 for a detailed description of these faults), which is the number of mutants given by $|LM_{ne_1}| + |KM_{na}|$ for each operator as discussed in Section 2.2. As previously explained, SUC and its model M may differ from each other; it is assumed that M is correct. Note that there are only 178 instead of 200 eO-mutants since it is the maximum number that could be generated.

Table 5.1: Number of mutants generated and number of faults detected in RJB

χ	**Number/Percentage of mutants generated**	**Number/Percentage of faults detected**
sI	200 (34.6%)	18 (81.8%)
sO	200 (34.6%)	3 (13.6%)
eO	178 (30.8%)	1 (4.5%)
total	578	22

From Table 5.1, it can be observed that inserting additional arcs into an ESG is the most effective operator helping to detect remaining faults in RJB. Omitting

arcs also helped to reveal some faults, but the omission of events only helped to reveal one fault. An explanation is that when sequences (arcs) are inserted into the underlying ESGs, they in general represent some illegal behavior of the SUC, such as a button that should be disabled in a certain context but becomes enabled due to this additional sequence (arc). Thus, these faults are more likely to be detected.

Table 5.2 lists all of the 22 faults detected in RJB and the mutation operators $\chi \in \mathcal{X}$ that helped to reveal them. A significant observation worth noting is that all the faults in Table 5.2 were detected using a version of RJB 2 that had been in use for years. An advantage of the proposed approach is that it reveals faults that could not be detected by test cases generated to cover the model M, that is, all event sequences of length 2.

5.2 Case Study II: A Driver Assistance System–an Embedded Reactive System

The second case study comes from the automotive industry ([8], [23], [25]). It is a driver assistance system within a car. The system was manufactured by Hella Corp. [73], one of the major German suppliers for electronic devices.

5.2.1 System Under Consideration

The focus is on the adaptive cruise control (ACC)–an electronic control unit (ECU) of the driver assistance system, which is an embedded reactive system (Figure 5.5). To automatically control the distance between vehicles, ACC is supplemented with a 24 GHz radar sensor and connected over CAN Bus (controller area network) to five other ECUs and thereby indirectly coupled to more than 7 sensors. Figure 5.6 demonstrates the physical dependencies. More illustrations and examples are given by Figures A.1, A.2, and A.3 in Appendix A. The core controller and scheduler unit of ACC combines the signals of the other ECUs and is the most complex part of the software to be tested.

Table 5.2: Description of the faults detected in RJB

ID	Description of the fault	χ
1	*Track Position* cannot be set anymore and the application freezes when the end of track is reached.	sI
2	GUI shows state *Paused* but the music is still playing.	sI
3	Pushing *Play* button plays a wrong track.	sI
4	*AutoPlay* does not start playing when a CD is inserted.	sI
5	*Pause* button has no impact despite the fact that the system is playing.	sI
6	If at least one track of a CD is selected, double-clicking with mouse on unselected starts playing a selected one from beginning.	sI
7	When double clicking a track that is not selected while in *Shuffle* mode, the application is closed without a warning (not repeatable).	sI
8	RealJukebox shows the wrong track when playing a CD in parallel in another application.	sI
9	Muting the sound in the taskbar is not shown in RealJukebox (it works vice versa).	sI
10	Deselecting and selecting a recorded track leads to a wrong percentage of work done shown in the title bar of the GUI (more than 100%).	sI
11	Pushing *Rew* while recording cancels recording.	sI
12	Pushing *FF* while recording cancels recording.	sI
13	While recording and playing a track in parallel, the track bar cannot be set.	sI
14	Setting the track bar during *Pause* mode, the track begins to play.	sO
15	Fast-Forwarding track during *Pause* mode, the track is being played.	sO
16	Rewinding a track during *Pause* mode, the track is being played.	sO
17	Track button cannot be set if track is stopped.	sI
18	If no tracks selected, buttons *Play*/*Pause* are still active.	sI
19	If no tracks selected, button *Record* is still active.	sI
20	Recording from *Playlist*: after recording, entries in the list are damaged.	sI
21	When creating an *Auto Playlist* with no genre/artist chosen, all tracks are listed in the new list.	eO
22	Active skin of the GUI can be deleted without warning.	sI

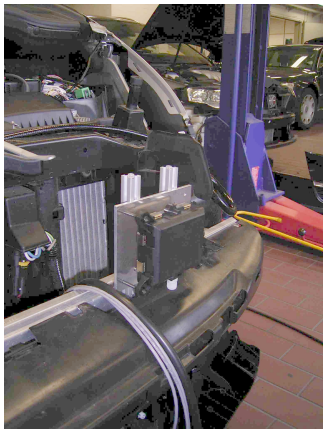


Figure 5.5: Adaptive Cruise Control (ACC)

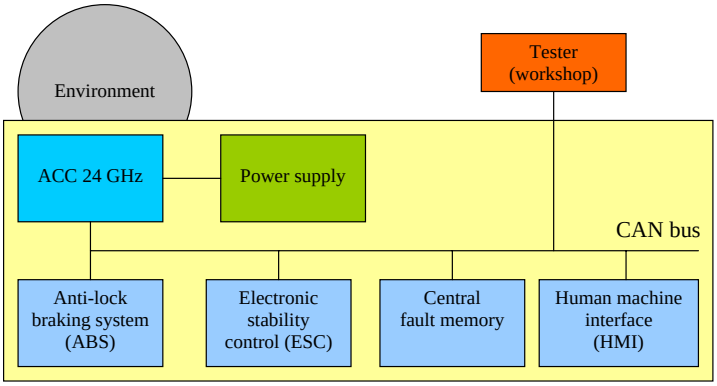


Figure 5.6: The system and test environment of ACC and its interfaces with other sub-systems

5.2.2 Modeling of SUC

This section reports on experiments on a relatively small sub-system of ACC, which, nevertheless, allowed for interesting observations and insights. Appropriate events were selected for modeling the SUC that are specific to the system and test environment, for example, short circuits. Arcs connect these events according to the system specification and logical relations. For an effective execution of the tests, a company-developed test environment was used that included a PC, SUC, a programmable logic controller (PLC), a programmable power supply, and, last but not least, the standard software CANoe [121] for

- simulation and monitoring of specific traffic scenarios,
- simulation of all other involved ECUs, and
- creation of test reports based on XML and HTML.

While generating the test sequences for the software, CANoe state-based information was augmented by conditions and variables. Apart from testing different street traffic situations, the test environment also allows testing of short circuits, different supply voltages on the bus and/or ECUs, etc.

5.2.3 Mutant Generation and Results

Based on the models constructed, mutants were generated using sI-, sO-, and eO-operators as follows:

- 82 mutants using the sI-operator,
- 62 mutants using the sO-operator, and
- 36 mutants using the eO-operator.

As a result of the mutation testing, five faults were detected. Table 5.3 lists these faults detected in ACC and the operator $\chi \in \mathcal{X}$ used to generate the corresponding mutant.

Table 5.3: Description of the faults detected in ACC

ID	Description of the fault	χ
1	The system resides in the error state and the speed of the vehicle drops to 5 km/h. If a simulation is now requested, the system remains in the error state (instead of switching to the simulation state as required).	sO
2	If an error occurs, the system does not switch in time to the error state.	sI
3	If the system resides in the error state and the error vanishes, the system does not switch to the requested active state in time.	sI
4	If the system resides in the error state and the error vanishes, the system does not switch to the requested inactive state in time.	sI
5	The system does not switch from initial state to active/inactive state in time.	sI

Fault 1 represents an undesirable situation in which the system remains in an error state instead of switching to the simulation state as required. The other four faults represent violations of time constraints, meaning that the system does not transfer to the requested state within the specified time. Thus, the detection and

correction of the faults listed in Table 5.3 contributed considerably to an increase in safety of SUC.

5.3 Case Study III: The Control Desk of a Marginal Strip Mower–a Proactive System

The third case study is on a proactive system to control a marginal strip mower (RSM13) mounted on the Unimog, a truck manufactured by Mercedes-Benz [93]. The study focuses on the control desk developed for RSM13.

Figure 5.7: Control desk for RSM13

Figure 5.8: Marginal strip mower (RSM13)

5.3.1 System Under Consideration

The control desk (Figure 5.7) of RSM13 is the SUC. Considering safety aspects, the most significant component of the strip mower are revolving knives (Figure 5.8) that are controlled by the SUC. Such systems are used within cities in Germany, even in presence of heavy traffic. The kinematics of the rotating point and the specially designed run of the mow head guide enables to reach large areas behind the

crash guides. The mow head is protected against stone chipping by a special cutting system. Operation is affected either by the front power take-off or by the power hydraulic. The buttons on the control desk simplify the operation and mitigate safety risks so that the mow head returns to the working position or transport position in potential emergencies, such as when a button is accidentally pressed. The position of the mow head can also be continuously varied. Besides the positioning, the support pressure and incline of the mowing unit can be controlled.

5.3.2 Modeling of SUC

The control desk has four different modes. The first mode (Actuator) is used to control the actuators; the second mode is used to switch between transport and working position (transport/working position). Additionally, there are two operation modes (Operation I and Operation II). These operation modes have been modeled by ESG and SC. An example of an ESG of mode *RSM13 transport/working position* is given in Figure 5.9. The same mode is represented by an SC in Figure 5.10. All the ESG and SC models used are depicted in Figure B.1 - Figure B.10 in Appendix B.

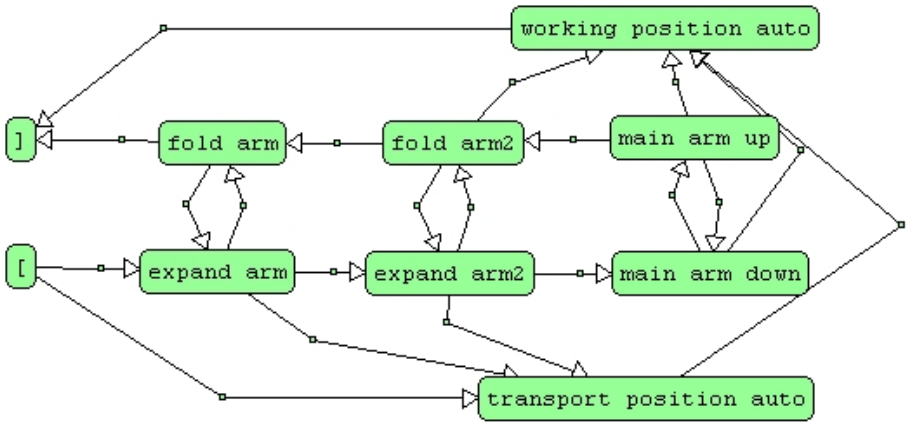


Figure 5.9: ESG of RSM13 transport/working position

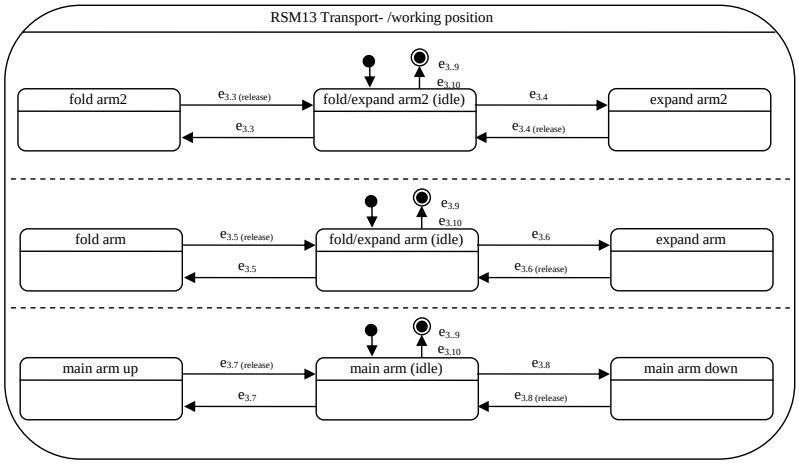


Figure 5.10: Statechart of RSM13 transport/working position

5.3.3 Mutant Generation and Results

Table 5.4 lists the number of all valid (and invalid) mutants that were generated for ESG and SC using mutation operators defined in Chapter 3. ESG-based mutants are more numerous than SC-based, which is easy to explain. In an ESG, orthogonal regions cannot explicitly be modeled. Therefore, more sequences (arcs) can be inserted, for example, by an sI-operator. In an SC, each region has been handled separately–transitions as inserted by a tI-operator do not cross orthogonal regions.

Table 5.4: Number of mutants generated and number of faults detected in RSM13

	χ	**Number of all valid (invalid) mutants generated**	**Number of faults detected in SUC**
ESG	sI	184 (0)	5
	sO	22 (19)	1
	eO	15 (18)	0
Statechart	tI	41 (0)	5
	tO	26 (18)	1
	stO	15 (20)	0

Both approaches detect the same number of faults in RSM13. It turns out that the operators sO and tO, and sI and tI help to detect the same faults. Table 5.5 lists the faults that have been detected in SUC. It is worth noting that this application was tested by a technical control board in Germany and released for public use. The approach in this study detects faults in the released version that were not previously found. Two of these faults are extremely safety-critical. Fault 1 indicates that the cutting unit can be activated without having any pressure on the bottom, which is very dangerous if pedestrians approach the working area. Another problem (Fault 6) is observed by keeping the button for shifting the mow head pressed and changing to another screen. The mower head with its cutting unit cannot immediately be stopped in an emergency. Furthermore, restarting the hydraulic gear while it is already running can cause serious damage.

Table 5.5: Description of the faults detected in RSM13

ID	**Description of the fault**	χ
1	When the function *bearingPressure* is deactivated, the function *cuttingUnit* can be activated.	sI for ESG or tI for SC
2	When the function *cuttingUnit* is activated, the function *bearingPressure* can be deactivated.	sI for ESG or tI for SC
3	When Engine I is activated, a change from the view *RSM_Actuator* to the view *Start* is possible.	sI for ESG or tI for SC
4	When Engine II is activated, a change from the view *RSM_Actuator* to the view *Start* is possible.	sI for ESG or tI for SC
5	When Axis Lock is activated, a change from the view *RSM_Actuator* to the view *Start* is possible.	sI for ESG or tI for SC
6	A change from the view *RSM_Operation_II* to the view *RSM_Operation_I* while function *ArmShiftLeft* is active is possible if the function *ArmShiftLeft* is activated.	sO for ESG or tO for SC

5.4 Overall Results

This section presents detailed results of the three case studies. The mutants generated for the case studies are classified according to the scheme introduced in Section 2.2 in Figure 2.6. Table 5.6, Table 5.7, and Table 5.8 classify the ESG mutants generated for RJB, ACC, and RSM13, respectively, by showing the number of *live* (LM_{ne_1}, LM_{ne_2}), *killed* (KM_a, KM_{na}) and *equivalent* (LM_e) mutants with respect to each mutation operator.

Table 5.6: Classification of ESG mutants generated for RJB

χ	LM_{ne_1}	LM_{ne_2}	LM_e	KM_a	KM_{na}	total	$MFDS$	$TGMAS$
sI	0	0	0	182	18	200	0.09	0.91
sO	0	142	0	55	3	200	0.02	0.28
eO	0	59	0	118	1	178	0.01	0.66
total	0	201	0	355	22	578	0.04	0.61

Table 5.7: Classification of ESG mutants generated for ACC

χ	LM_{ne_1}	LM_{ne_2}	LM_e	KM_a	KM_{na}	total	$MFDS$	$TGMAS$
sI	0	21	0	57	4	82	0.05	0.70
sO	0	46	0	15	1	62	0.02	0.24
eO	0	11	0	25	0	36	0	0.69
total	0	78	0	97	5	180	0.03	0.54

Although the three selected case studies are from different areas with large diversity, the overall results have strong similarities. In all cases, the sets LM_{ne_1} and LM_e are empty and equivalent mutants were not discovered. In absolute numbers, the sI-operator used for ESG helped to reveal most of the faults as denoted by the set KM_{na}. In most cases, the other sI-mutants, as denoted by the set KM_a, could also be killed successfully. A major reason for this, especially for GUIs of RJB and RSM13, is that such illegal behavior can be easily detected due to non-executable

Table 5.8: Classification of ESG mutants generated for RSM13

χ	LM_{ne_1}	LM_{ne_2}	LM_e	KM_a	KM_{na}	total	$MFDS$	$TGMAS$
sI	0	0	0	179	5	184	0.03	0.97
sO	0	5	0	16	1	22	0.05	0.73
eO	0	3	0	12	0	15	0	0.80
total	0	8	0	207	6	221	0.03	0.94

sequences of events, that is, impossible interactions with the GUIs. In contrast, many mutants generated by the sO-operator were still live as denoted by the set LM_{ne_2} and a low $TGMAS$ score. This can be explained by the fact that the predicted outputs contained in the test cases are not sufficient to reveal the difference. On the other hand, these mutants might have been killed using a test generation algorithm Φ that also covers non-existing arcs of the model. Nevertheless, this still does not help to detect mutants generated by eO-operator.

Table 5.9 summarizes the classification of the SC mutants generated for RSM13. Mutants generated by the tI-operator could be killed in all cases. Test sets generated based on these mutants revealed most of the faults in SUC. Similar to the case for ESG, killing all mutants generated by the tO- and stO-operators was not possible.

Table 5.9: Classification of SC mutants generated for RSM13

χ	LM_{ne_1}	LM_{ne_2}	LM_e	KM_a	KM_{na}	total	$MFDS$	$TGMAS$
tI	0	0	0	36	5	41	0.12	0.88
tO	0	8	0	17	1	26	0.04	0.65
stO	0	3	0	12	0	15	0	0.80
total	0	11	0	65	6	82	0.07	0.79

To sum up, the comparison of the fault detection capability of test sets between ESG-based mutation and SC-based mutation does not point out any significant tendency, but the data confirms the effectiveness of MBMT when applied using different models.

5.5 Analysis and Discussion of the Results

This section provides an overall cross-comparison of test results using ESG-based mutants generated for each case study. Table 5.10 lists the percentage (α) of the total generated mutants corresponding to each mutation operator, as well as the percentage (β) of the total faults detected by test sets generated based on the mutants of that operator. The last row presents the number of mutants generated and faults detected in SUC. For all three case studies, it turns out that the sI-operator is the most effective mutation operator that helps to reveal approximately 80% of the faults detected in each SUC. In conclusion, a test generation algorithm Φ used in MBT should also consider covering arcs that are not included within the model to test such faulty behavior of SUC.

Table 5.10: Comparison of ESG-based mutants (α: percentage of mutants generated; β: percentage of faults detected in SUC)

χ	RJB		ACC		RSM13	
	α	β	α	β	α	β
sI	34.6	81.8	45.5	80	83.2	83.3
sO	34.6	13.6	34.4	20	9.9	16.6
eO	30.8	4.5	20.0	0	6.8	0
total	578	22	180	5	221	6

The eO-operator applied to RJB and RSM helped to detect only one fault. A major problem concerning mutants generated by sO- and eO-operators is the oracle problem, which is to determine whether the expected outputs by the mutants correspond to the output of SUC. For graphical user interfaces in particular, omitting events might lead to testing only the correctness of a reduced part of the original model. Mutants generated by the sO-operator could not be killed in many cases because the test cases generated were not adequate to detect them. As a result, the test generation algorithm used in the case studies should also cover the arcs that are not contained within the model. Figure 5.11 gives a graphical representation of the fault detection capability of test sets generated based on ESG mutants.

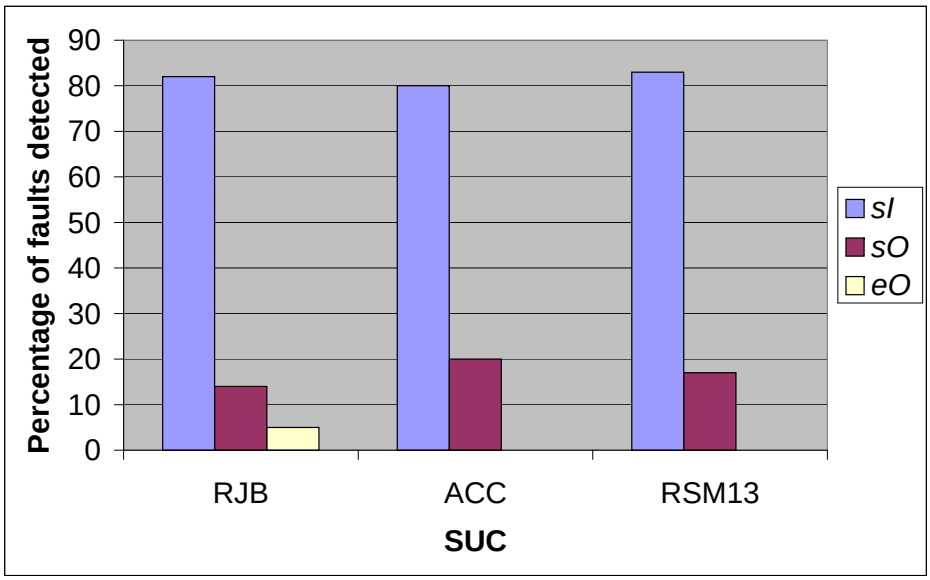


Figure 5.11: Fault detection capability of test sets generated based on ESG mutants

The classification of different mutants in the previous section facilitates a discussion of the adequacy of test sets generated by test generation algorithm Φ. Table 5.11 gives an overall classification of the ESG mutants of all three case studies.

Table 5.11: Overall classification of the ESG mutants

χ	LM_{ne_1}	LM_{ne_2}	LM_e	KM_a	KM_{na}	total	$MFDS$	$TGMAS$
sI	0	21	0	418	27	466	0.06	0.90
sO	0	193	0	86	5	284	0.02	0.30
eO	0	73	0	155	1	229	0	0.68
total	0	287	0	659	33	979	0.03	0.67

As discussed above, test sets generated with respect to the sI-mutants revealed most of the faults in SUC (as can be seen from the corresponding $MFDS$.) The sI-mutants that do not correspond to a fault in SUC are easily killed in most cases, as denoted by the value of $TGMAS$. For GUIs (RSM13 and RJB) in particular, all of these mutants have been killed.

Many mutants generated by the sO- and eO-operators could not be killed. Especially the value of $TGMAS$ of the sO-operator is only 30%, that is, 70% of the mutants could not be killed. A major reason is that these mutants represent valid subsets of the original model. As a result, they remain live. On the other hand–again–these mutants indicate the necessity for testing non-existing arcs. A test generation algorithm Φ that covers these arcs (representing potential faulty behavior) would have killed more of these mutants. The faults that were detected by test sets generated using the sO- and eO-mutants have been found due to enforcement of different sequences.

5.6 Limitations and Threats to Validity

Although the case studies in Sections 5.1 through 5.3 give a good insight into the use of the proposed techniques in practice, they do not aim to draw general conclusions that need formally sound arguments in addition to the empirical ones gained by analyzing the case studies. Several reasons cause this uncertainty.

- Various type of real life applications are used in practice on a daily basis and these applications possess different behavioral properties. Thus, results obtained from one type of system may not be valid for another type.
- Both number of mutants and their order are limited for practical reasons and due to the nature of mutation analysis.
- These threats may affect the results of the case studies, such as the evaluation of the mutation operators, comparison of ESG with SC models, and effectiveness of MBMT. Thus, the models and operators used should not be interpreted as a generalization.
- First-order mutants were generated, assuming that higher-order mutants are likely to be killed due to the coupling effect. The benefit of using a combination of several operators (as proposed in Chapter 3) is therefore not clear.
- With respect to mutation analysis based on the implementation, the coupling effect argues that test cases that kill first-order mutants are also likely to kill

higher-order mutants. With respect to the proposed approach, mutants generated by any corruption operator (which is a combination of an omission operator followed by an insertion operator) are regarded as second-order mutants, while the same mutants generated by some implementation-based mutation operators are regarded as first-order mutants. Thus, in this context the coupling effect becomes arguable.

- Some recent studies ([71], [80]) emphasize that using higher-order mutation provides benefits. Among an exponentially large number of higher-order mutants, some of them are harder to detect than first-order mutants and thus are more interesting. Such mutants might be more appropriate to simulate real faults. Therefore, future work could put more focus on the benefit of using basic operators to generate higher-order mutants and the validity of the coupling effect for the proposed approach.

- There are also many other models that could not be considered, including UML, extended/timed automata, Petri nets, algebraic models such as (extended) regular expressions, and other generic ones (grammars or temporal logic). Applying the proposed approach to them could lead to different results.

- Only sequence-based coverage criteria are considered and particular algorithms are used to generate test sets achieving these criteria. It is still possible to make use of these structures with different algorithms and to achieve other coverage criteria.

In clarification of the last remark, note that it is usually possible to increase the number of arguments of this type in any experiment due to the nature of testing. This stems from the fact that some aspects may or may not be much more difficult to formalize at some points. Considering the coverage criteria, one cannot absolutely conclude whether a given test set is the most adequate or not when only satisfying some specified coverage criteria. This is because a coverage criterion is something empirical that is generally agreed upon rather than being proved mathematically to always work or yield the best (see also [128]).

5.7 Summary

Three case studies demonstrated the application of the proposed MBMT on industrial and commercial real-life applications. The experimental data suggest that MBMT can be effective in detecting remaining faults, including those that have not been revealed by other test techniques, e.g., a German technical control board previously tested the application used in the third study. Mutants generated by the approach introduced in this thesis helped to reveal safety-critical faults that were not found before.

Furthermore, the results of MBMT show that mutants that were generated by sI-/tI-operators could be killed in most cases and that test cases generated from these mutants revealed most of the faults in SUC. As a result, model-based test generation for ESG and FSM/SC is to be extended by additional test cases to reveal these faults as well.

Chapter 6

Mutation Adequate Test Generation

As a result of the three case studies the test process for ESG and FSM/SC described in Chapter 4 needs to be extended to achieve test sets that are also capable in revealing the faults that were detected in SUC by MBMT. Therefore, a fault model is given to model also the undesirable behavior represented by mutants. As a result, a new coverage criterion is defined and an extended methodology for generating and minimizing test sets is introduced ([15], [16], [17], [21], [74]).

6.1 Fault Modeling

The results of the case studies in Chapter 5 turned out that test sets generated from mutants created by sI- and tI-operators revealed most of the faults in SUC that could not be detected by MBT as described by Algorithm 4.1 and 4.2; that is by covering sequences of events or transitions of length $k \in \mathbb{N}$. As a consequence, the fault model of the corresponding mutation operators needs also to be considered for test generation in MBT.

ESG and FSM are not explicitly considered in the following. According to Definition 3.10, FSMs are a subset of SCs. Furthermore, an ESG can be transformed into an FSM in polynomial time as shown by Algorithm 6.1. The events of the graph ESG represented by its nodes E are taken over as events for the FSM. Then, an initial state in FSM is inserted. For each event of ESG a new state in FSM is created. For each arc of ESG, a transition in FSM is inserted. Obviously, this

conversion is correct as each event represented by a node is simply pre-drawn to its incoming arcs (transitions). The runtime of Algorithm 6.1 is given by $O(|A|)$.

Algorithm 6.1: Conversion of an ESG to FSM

Input: $ESG = (E, A, \Xi, \Gamma)$
Output: $FSM = (S, TR, E', f, S_\Xi, S_\Gamma)$

```
1  TR := f := S_Γ := ∅;
2  E' := E;
3  S := S_Ξ := {s_[};
4  foreach e ∈ E do
5      S := S ∪ {e};
6  foreach (e, e') ∈ A do
7      TR := TR ∪ (e, e');
8      f := f ∪ ((e, e'), e');
9  foreach ξ ∈ Ξ do
10     TR := TR ∪ (s_[, ξ);
11     f := f ∪ ((s_[, ξ), ξ);
12 foreach γ ∈ Γ do
13     S_Γ := S_Γ ∪ {γ};
```

Mutants generated by sI- and tI-operator represent malfunctions of the system that may potentially lead to failures. To consider these potential faults, they are to be modeled as undesirable events and transitions. For modeling the *faulty*, that is, undesirable events, a statechart SC is completed by an *error state* and *faulty transitions*. The notations *error state* and *faulty transition* are used for explicitly describing the potentially faulty behavior of the modeled system. Therefore, a statechart is augmented by a set of faulty transitions TR_{faulty} and an error state σ. For each simple state s and for each event that does not trigger a legal transition in the context of state s a faulty transition is added. This completion is only done for the purpose of generating test cases from that model.

The completion process as described does not work for orthogonal regions. Given the active state s of an orthogonal region, an event e is undesirable if and

only if in all other regions no active state exists from where this event may be triggered legally. Additionally, the effect of an undesirable event may vary depending on the active states of the other orthogonal regions. Therefore, as a precondition for completing a statechart by undesirable, faulty events an explicit-making (*flattening*) of all possible state combinations over the regions needs to be conducted.

Each AND-state, along with its regions, has to be converted into equivalent XOR-states by taking the Cartesian product of the substates from each region. As a precondition for flattening a single AND-state it is necessary that all XOR-states have to be removed within the regions. All transitions that are connected with these XOR-states have to be converted into transitions that are solely connected with simple states. Due to the removal of transitions XOR-states within the regions become unnecessary and can be removed. Removing this hierarchy will result in fewer states but more transitions. The resulting statechart consists solely of simple and XOR-states. This "flattened" statechart denoted by SC_f, equivalent to the one given, is of course not intended to be legible to human readers. It is solely to be used as an input for the process of test generation. As an example for flattening a statechart the one in Figure 6.1 is used. The resulting statechart is given by Figure 6.2. Finally, the flattened statechart from Figure 6.2 is to be completed by inserting an error state and faulty transitions as is shown in Figure 6.3.

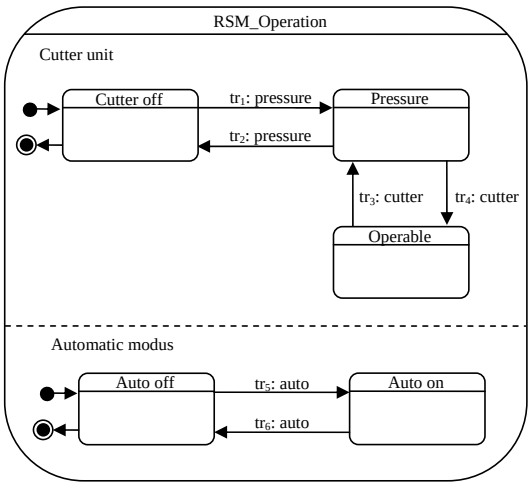


Figure 6.1: Example of a statechart SC

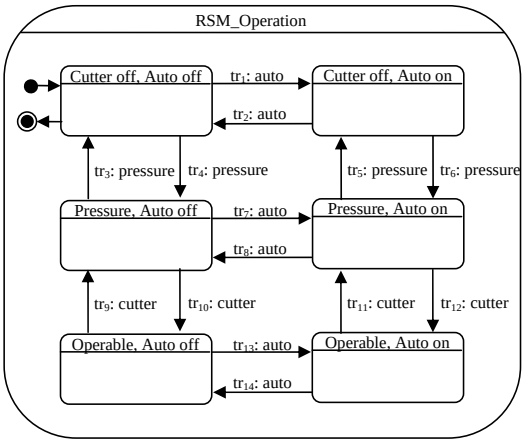


Figure 6.2: Flattened statechart SC_f of Figure 6.1

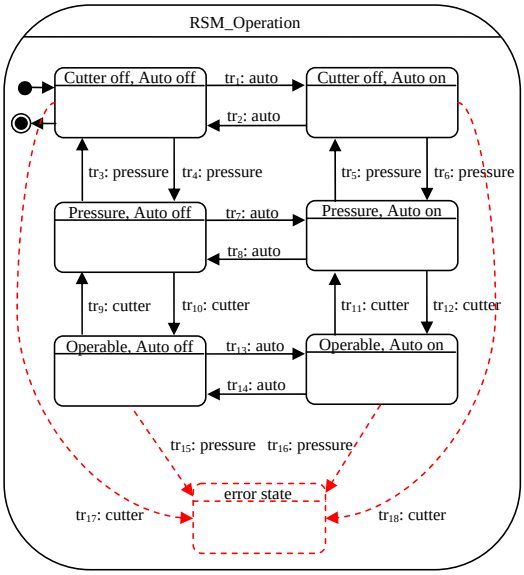


Figure 6.3: Completed statechart $\widehat{SC_f}$ of Figure 6.2

A general problem of flattening orthogonal regions is the growth in the number of states. If an AND-state compounds m regions $s_1, s_2, ..., s_m$ where $|s_i|$ denotes the number of simple states within region s_i, the corresponding XOR-state will contain $|s_1| \cdot |s_2| \cdot ... \cdot |s_m|$ simple states in the worst case. On the other hand, a system modeled by FSM would count approximately the same number of states.

6.2 A Further Coverage Criterion and Extended Test Process

A completed statechart $\widehat{SC_f}$ based on SC_f is formally defined as follows.

Definition 6.1 (Completed Statechart) A completed statechart is given by $\widehat{SC_f} = (SC_f, TR_{\text{faulty}})$ where

- $S = S \cup \{\sigma\}$ (where σ is an error state),
- $H = H \cup \{(root(), \sigma)\}$, and
- $TR_{\text{faulty}} \subseteq S \times E \times \{\sigma\}$ such that $TR_{\text{faulty}} = \{(s, e, \sigma) | s \in S \wedge e \in E \wedge \forall s' \in S : ((s, s'), e) \notin f\}$.

Definition 6.2 (Faulty Transition) A transition $(s, e, \sigma) \in TR_{\text{faulty}}$ is called a *faulty transition*.

Definition 6.3 (Faulty Transition Sequence) A *faulty transition sequence* $FTS = (tr_1, ..., tr_i, tr)$ consists of $i + 1$ transitions, forming a transition sequence of length $i + 1$ including a faulty transition $tr \in TR_{\text{faulty}}$. A faulty transition sequence is called *complete* if it starts at the initial state of the statechart, abbreviated as CFTS. The sequence $(tr_1, ..., tr_i)$ is called a *start sequence*.

Thus, a complete faulty transition sequence is expected to cause a failure. Based on Definition 6.3 the following coverage criterion is defined.

Definition 6.4 (Faulty Transition Coverage (FTC)) Generate for each faulty transition in a completed statechart $\widehat{SC_f}$ a complete faulty transition sequence.

Definition 6.4 guarantees that potential malfunctions given by faulty transitions will be tested. The extended process of generating test cases of a statechart SC including Definition 4.8 and Algorithm 4.2 is presented by Algorithm 6.2.

Algorithm 6.2: Mutation adequate test generation and execution process

Input: $SC = (S, TR, E, f, S_{\Xi}, S_{\Gamma}, H, g)$

1 $SC_f :=$ Flatten all orthogonal regions of SC;
2 $\widehat{SC_f} :=$ Add an error state and faulty transitions to SC_f;
3 $n :=$ required length of transition sequences to be covered;
4 Generate a set of CTS to cover all (legal) transition sequences (TS) of length $k \in \{1, .., n\}$;
5 Generate a set of CFTS to cover all faulty transitions;
6 Map the transition sequences of the test sets given by selected CTSs and CFTSs to sequences of events;
7 Apply the test sets to SUC;
8 Observe system output to determine whether the system response is in compliance with the expectation;

6.3 Optimizing Test Sets

A simple method to fulfill the k-transition coverage criterion (Definition 4.8) is to set up a test case for each transition sequence of length k and then to compute a start sequence for each one. Such a set of test cases will not be minimal due to multiple occurrences of the same transition sequences. A test set consisting of complete transition sequences fulfilling k-transition coverage can be minimized with respect to aspects such as the number of test cases, the total length of all test cases or a combination of the preceding two criteria. The first criterion may be suitable if for each complete transition sequence a costly or complex restart of the SUC is necessary.

It may be useful to weight all transitions by assigning each transition a numerical value. Such weights can represent costs (such as time) for executing a transition. However, this requires that costs for a single transition can be determined and that

they are repeatable. If transitions are not explicitly weighted, it is assumed that all transitions are weighted by the same value.

Definition 6.5 (Minimal Test Set) A test set is said to be *minimal* if the sum over the costs of its transition sequences is minimal. The costs of a transition sequence are given by the sum of the costs of each transition denoted by the function $costs : TR \rightarrow \mathbb{R}_{\geq 0}$. If that function is not specified, it is assumed that this function is defined for an arbitrary transition tr by $costs(tr) := val$ where val denotes a constant value.

6.3.1 *k*-Transition Coverage

As a precondition for setting up a minimized test set, a data structure termed a *transition graph* is needed.

Definition 6.6 A *transition graph* $TG = (V, A, c)$ represents a directed graph that consists of a set of nodes V, a set of directed arcs A, and a cost function $c : A \rightarrow \mathbb{R}_{\geq 0}$.

Algorithm 6.3 describes how a transition graph is constructed for a statechart $\widehat{SC_f}$. The set of nodes V consists of $|TR| + 2$ vertices representing the legal transitions of statechart $\widehat{SC_f}$. An explicit initial node denoted by $tr_{\lfloor}$ and a final node denoted by $tr_{\rceil}$ are also included. For each transition pair (tr, tr') of the statechart, a directed arc is created. Node $tr_{\lfloor}$ has to be connected with all transitions that may be triggered from the initial state. Transitions leading into a final state have to be connected with node $tr_{\rceil}$. The runtime of this algorithm is given in the worst case by $O(|TR|^2)$ for inserting the arcs into the graph.

Based on a transition graph, transition coverage is done by visiting all vertices of the graph at least once by starting in node $tr_{\lfloor}$ and ending in node $tr_{\rceil}$. The problem of computing a route for visiting all vertices of a graph by minimizing the length of the route is well known as the *traveling salesman problem* (TSP). If visiting vertices and traversing arcs more than once is allowed, it is called the *graphical traveling salesman problem* (GTSP) [109]. In general the traveling salesman problem belongs to the *NP* complexity class. But despite of this fact "there are very good heuristics yielding solutions which are only a few percent above optimality" [109].

Algorithm 6.3: Construction of a transition graph

Input: $\widehat{SC_f}$
$costs$ function
Output: A transition graph $TG = (V, A, c)$

```
1  V := {tr_[, tr_]};
2  A := ∅;
3  foreach tr ∈ TR do
4      V := V ∪ {tr};
5  foreach s ∈ S_simple do
6      foreach tr ∈ in(s) do
7          foreach tr' ∈ {out(s) \ TR_faulty} do
8              A := A ∪ (tr, tr');
9              c((tr, tr')) := costs(tr');
10 foreach tr ∈ {out(initial(root())) \ TR_faulty} do
11     A := A ∪ (tr_[, tr);
12     c((tr_[, tr)) := costs(tr);
13 foreach s ∈ S_Γ do
14     foreach tr ∈ in(s) do
15         A := A ∪ (tr, tr_]);
16         c((tr, tr_])) := 0;
```

A test set fulfilling k-transition coverage for $k > 1$ is computed by transforming the transition graph stepwise ($k - 1$ times) and then applying the graphical traveling salesman problem. By also computing all complete transition sequences whose length is smaller than k and that cannot be expanded to longer sequences, a minimal test set fulfilling k-transition coverage for all $k \in \{1, ..., n\}$ is achieved. This proceeding is described in Algorithm 6.4.

Algorithm 6.4: Computation of a test set T_{TC} for k-transition coverage for $k \in \{1, ..., n\}$

Input: A statechart $\widehat{SC_f}$
$costs$ function
$n \in \mathbb{N}$

Output: A test set T_{TC} fulfilling k-transition coverage for $k \in \{1, ..., n\}$

```
1  Generate the transition graph TG := (V, A, c) from SC_f^ by applying
   Algorithm 6.3 with (SC_f^, costs);
2  T_TC := ∅;
3  for k := 2 to n do
4      foreach v ∈ V do
5          if (tr_[, v) ∈ A ∧ (v, tr_]) ∈ A ∧ indeg(v) = 1 ∧ outdeg(v) = 1 then
6              T_TC := T_TC ∪ {v};
7      Transform the transition graph TG by applying Algorithm 6.5 with
       (TG, costs);
8  A := A ∪ (tr_], tr_[);
9  c((tr_], tr_[)) := 0;
10 Apply the graphical traveling salesman problem to the transition graph TG;
11 Split up resulting tour in CTSs. Add them to set T_TC;
```

First, a transition graph is constructed based on a statechart $\widehat{SC_f}$. If n equals 1 the graphical traveling salesman problem can be applied directly. If n is greater than 1 the transition graph has to be transformed $k - 1$ times. The resulting graph represents all possible sequences of transitions of length k. Additionally, all sequences of length $k - 1$ are computed that cannot be expanded to longer sequences. In line

4 each node v represents a transition sequence of length $k-1$. These sequences are characterized by the fact that the corresponding node representing that sequence is solely connected with vertices $tr_{\lfloor}$ and $tr_{\rceil}$. The functions $indeg(v) := |\{v'|(v',v) \in A\}|$ and $outdeg(v) := |\{v'|(v,v') \in A\}|$ are used to compute these vertices. As these sequences are already complete, they are added to the set T_{TC}.

Subsequently, the transition graph is transformed by the application of Algorithm 6.5. The resulting graph $TG_{\text{out}} = (V_{\text{out}}, A_{\text{out}}, c_{\text{out}})$ contains a node $(tr_1, ..., tr_k)$ for each transition sequence of length k. Two vertices $(tr_1, ..., tr_k)$ and $(tr'_1, ..., tr'_k)$ are connected by a directed arc if it holds that $(tr_2, ..., tr_k)$ equals $(tr'_1, ..., tr'_{k-1})$ for each component, that is, $tr_2 = tr'_1, ..., tr_k = tr'_{k-1}$. These two vertices thus represent the sequence $(tr_1, ..., tr_k, tr'_k)$.

Concluding, the final transition graph used as a source for the graphical traveling salesman problem is augmented by an additional arc $(tr_{\rceil}, tr_{\lfloor})$. This arc ensures that the resulting graph is strongly connected, and therefore, a solution exists. The weight of this arc can be increased to force a tour delivered by an algorithm solving the graphical traveling salesman problem to contain a minimized number of occurrences of this arc. This in turn means that the number of test cases is minimized. The resulting tour may still need to be split up into single complete transition sequences. The runtime of Algorithm 6.5 is given by $O(|A_{\text{in}}|^2)$. For each arc of the graph $TG_{\text{in}} = (V_{\text{in}}, A_{\text{in}}, c_{\text{in}})$ that is to be transformed, in line 5 a new node is created in the graph TG_{out}. Inserting the new arcs in the resulting graph TG_{out} in lines 6-10 is done in $O(|V_{\text{out}}|^2) = O(|A_{\text{in}}|^2)$ steps.

6.3.2 Faulty Transition Coverage

To fulfill the criterion of covering faulty transitions (Definition 6.4), a transition sequence for each faulty transition $tr \in TR_{\text{faulty}}$ has to be computed as denoted in Algorithm 6.6. Each faulty transition has to be completed by adding a shortest start sequence with respect to its costs. To compute the shortest paths, the corresponding transition graph may be used for computing all shortest paths by applying e.g. the Floyd-Warshall algorithm. The worst case runtime is given by $O(|TR|^3)$ under the assumption that the transition graph (consisting of $|TR| + 2$ vertices) is used for computing all shortest paths by applying the Floyd-Warshall algorithm.

Algorithm 6.5: Transformation of a transition graph

Input: $TG_{\text{in}} = (V_{\text{in}}, A_{\text{in}}, c_{\text{in}})$
$costs$ function
Output: $TG_{\text{out}} = (V_{\text{out}}, A_{\text{out}}, c_{\text{out}})$

1 $V_{\text{out}} := \emptyset$;
2 $A_{\text{out}} := \emptyset$;
3 **foreach** $((tr_1, ..., tr_k), (tr'_1, ..., tr'_k)) \in A_{in}$ **do**
4 **if** $(tr_1, ..., tr_k) \neq tr_{[} \wedge (tr'_1, ..., tr'_k) \neq tr_{]}$ **then**
5 $V_{\text{out}} := V_{\text{out}} \cup (tr_1, ..., tr_k, tr'_k)$;
6 **foreach** $(tr_1, ..., tr_k, tr_{k+1}) \in V_{out}$ **do**
7 **foreach** $(tr'_1, ..., tr'_k, tr'_{k+1}) \in V_{out}$ **do**
8 **if** $(tr_2, ..., tr_k, tr_{k+1}) = (tr'_1, ..., tr'_k)$ **then**
9 $tr_{new} := ((tr_1, ..., tr_k, tr_{k+1}), (tr'_1, ..., tr'_k, tr'_{k+1}))$;
10 $A_{\text{out}} := A_{\text{out}} \cup tr_{new}$;
11 $c_{\text{out}}(tr_{new}) := costs(tr'_{k+1})$;
12 $V_{\text{out}} := V_{\text{out}} \cup \{tr_{[}, tr_{]}\}$;
13 **foreach** $(tr_1, ..., tr_k, tr_{k+1}) \in \{V_{out} \setminus \{tr_{[}, tr_{]}\}\}$ **do**
14 $tr_{current} := (tr_1, ..., tr_k, tr_{k+1})$;
15 **if** $(tr_{[}, (tr_1, ..., tr_k)) \in A_{in}$ **then**
16 $A_{\text{out}} := A_{\text{out}} \cup (tr_{[}, tr_{current})$;
17 $c_{\text{out}}((tr_{[}, tr_{current})) := \sum_{i=1}^{k+1} costs(tr_i)$;
18 **if** $((tr_2, ..., tr_k, tr_{k+1}), tr_{]}) \in A_{in}$ **then**
19 $A_{\text{out}} := A_{\text{out}} \cup (tr_{current}, tr_{]})$;
20 $c_{\text{out}}((tr_{current}, tr_{]})) := 0$;

Algorithm 6.6: Computing a test set T_{FTC} fulfilling faulty transition coverage

Input: A statechart $\widehat{SC_f}$
Output: A test set T_{FTC}

```
1 T_FTC := ∅;
2 foreach tr ∈ TR_faulty do
3     (tr_1, ..., tr_i) := STARTSEQUENCE(tr);
4     T_FTC := T_FTC ∪ (tr_1, ...tr_i, tr);
```

6.4 Example

This section exemplifies how the coverage criteria k-transition coverage (Definition 4.8) and faulty transition coverage (Definition 6.4) are used to generate test cases. An example of a completed statechart is given by Figure 6.4 that will be used for that purpose.

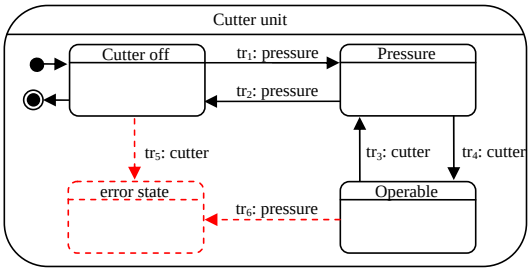


Figure 6.4: Example of completed statechart

Two minimized test sets T_{TC} and T_{FTC} fulfilling k-transition coverage for $k \in \{1, 2\}$ and faulty transition coverage are computed. Set T_{TC} is computed by applying Algorithm 6.4. The resulting two transition graphs for sequences of length 1 and 2 are shown in Figure 6.5. A shortest tour covering all vertices of the transition graph with sequences of length 2 (comprising all sequences of length 1) is given by:

$$((tr_{\lfloor}),\ (tr_1, tr_2),(tr_2, tr_1),\ (tr_1, tr_4),\ (tr_4, tr_3),\ (tr_3, tr_4),\ (tr_4, tr_3),\ (tr_3, tr_2),\ (tr_{\rceil}))$$

This results in one test case: $(tr_1,\ tr_2,\ tr_1,\ tr_4,\ tr_3,\ tr_4,\ tr_3,\ tr_2)$. Thus, the complete test set is given by

$$T_{TC} = \{(tr_1, tr_2, tr_1, tr_4, tr_3, tr_4, tr_3, tr_2)\}$$

To achieve faulty transition coverage, Algorithm 6.6 has to be applied. For the final test set, it is necessary to add a start sequence for the faulty transition tr_6.

$$T_{FTC} = \{(tr_5), (tr_1, tr_4, tr_6)\}$$

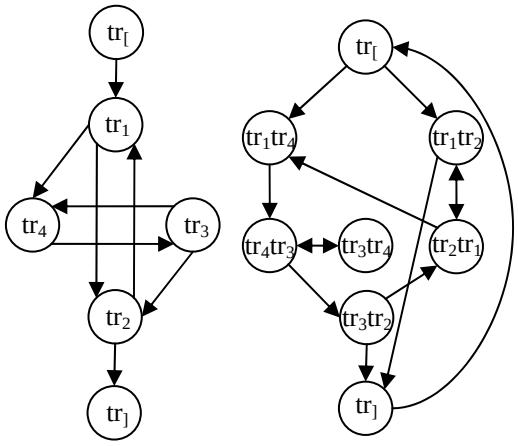


Figure 6.5: Transition graphs of length 1 and 2 for statechart of Figure 6.4

6.5 Summary

This chapter presented a coverage- and specification-oriented test approach based on FSM and SC. It extends modeling of the expected behavior as demonstrated in Chapter 4 by considering also the faulty behavior of sI-/tI-mutants. This is done by adding notations of faulty transitions and an error state. It represents a complementary view of the behavioral model. Based on this view, a new coverage criterion to cover faulty transitions was introduced. Furthermore, the test process aims to minimize the total length of the test sets generated.

Chapter 7

Further Applications of Basic Operators

This chapter presents further applications of basic operators and mutation testing based on different kinds of models such as ESG augmented by decision tables, model checker specifications, and communication sequence graphs.

7.1 Scalable Robustness Testing

This section proposes an approach to scalable testing the robustness of a software system using ESGs and decision tables (DT) ([29]). Basic operators are used to manipulate ESGs and DTs resulting in faulty models. Test cases generated from these faulty models are applied to SUC to check its robustness. Thus, the approach enables the quantification of robustness with respect to a universe of erroneous inputs.

Robustness is defined as the ability of a system to behave acceptably in the presence of unexpected inputs ([75]). The IEEE standard glossary defines robustness as the "degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" ([76]). Here, robustness is treated as the ability of a system to handle acceptably faulty inputs.

Different approaches have been proposed to evaluate the robustness of a system. Fernandez et al. ([59]) propose a model-based approach for testing the robustness of

a system. Test cases are generated from an operational specification and an abstract fault model. Mukherjee and Siewiorek ([97]) present a hierarchically structured approach to building robustness benchmarks. They evaluate various features that are desirable in a benchmark of system robustness and compare existing benchmarks according to these features. Fetzer and Xiao ([60]) have proposed an automated approach for increasing the robustness of C libraries. For that purpose robustness wrappers are generated that check the arguments of C functions before invoking them. Kropp et al. presented the Ballista methodology, an automated approach for testing the robustness of software ([85]). The goal is to identify failures by performing fault injection at software interfaces such as functions or procedures. Belli et al. ([26]) propose an event-based, graphical representation of the system and its environment for testing safety-critical systems. The events are user actions and system responses, and are ordered according to the threats posed by the resulting system states.

The approach proposed in this section differs from the ones cited above in that it allows modeling of incorrect behavior of software systems that can be traced back to a lack of robustness. Furthermore, an algorithmic approach is presented to generate test cases for testing a software-based system for robustness. Faulty models of an event sequence graph augmented by decision tables define a subset of faulty event sequences that represent faulty test inputs to test the exception handling ability of the SUC.

7.1.1 Modeling and Test Generation

Similar to the notation of *faulty transitions* and *(complete) faulty transition sequences* (Definition 6.2 and 6.3) this is defined accordingly for $ESG = (E, A, \Xi, \Gamma)$ ([18]).

Definition 7.1 (Faulty Event Pair) Any event pair $(e, e') \in A$ with $e, e' \in E$ is a *faulty event pair* (FEP) of an ESG.

Definition 7.2 (Faulty Event Sequence) Let $ES = (e_1, ..., e_i)$ be an event sequence of length i and $FEP = (e_i, e_{i+1})$ be a faulty event pair of an ESG. The concatenation of the ES and the FEP denotes a *faulty event sequence* $FES = (e_1, ..., e_i, e_{i+1})$.

Definition 7.3 (Faulty Complete Event Sequence) An FES is complete (or a *faulty complete event sequence* denoted as FCES) if $\alpha(FES) \in \Xi$.

An ESG can be augmented by decision tables (DT) ([9], [30]) (denoted by ESG^{DT}) to model different kinds of user inputs and control flow. In certain contexts it depends on the user input which event may follow after the current one. As an example, when logging into a system the combination of user name and password may be valid or invalid. Depending on the validity of the input the system will conclude or show the current login screen. Decision tables can be used to model such control flow. More formally, a decision table is defined as follows.

Definition 7.4 (Decision Table) A decision table is given by $DT = (AC, C, R)$, where

- $AC = \{ac_1, ..., ac_l\}$ is a set of actions,
- $C = \{c_1, ..., c_m\}$ is a set of conditions, and
- $R = \{r_1, ..., r_n\}$ is a set of rules.

When using decision tables in an ESG model, a decision table is represented by a distinguished node in the graph.

Definition 7.5 (Data Event [30]) An event $e \in E$ of an ESG is called a data event (DE) if e is represented by a decision table.

Test generation is done by a process as depicted by Algorithm 4.1. Additionally, for each DE in a CES multiple CES have to be generated by replacing each DE with data of the corresponding decision table. For each DE of an FCES multiple FCES have to be generated with data for every rule of the decision table ([30]). In the following, basic operators are defined for decision tables.

Definition 7.6 (Action Operators)

- An *action insertion* operator (acI) inserts a new action $ac \notin AC$:
$$acI(DT, ac) := (AC \cup \{ac\}, C, R)$$

- An *action omission* operator (acO) omits an action ac from set AC:
$$acO(DT, ac) := (AC \setminus \{ac\}, C, R)$$

- An *action corruption* operator (acC) removes an action ac' and replaces it with a new action ac'': $acC(DT, ac', ac'') := acO(DT, ac') \bullet acI(DT, ac'')$

Definition 7.7 (Condition Operators)

- A *condition insertion* operator (cI) inserts a new condition $c \notin C$:
$$cI(DT, c) := (AC, C \cup \{c\}, R)$$

- A *condition omission* operator (cO) omits a condition c from set C:
$$cO(DT, c) := (AC, C \setminus \{c\}, R)$$

- A *condition corruption* operator (cC) removes a condition c' and replaces it with a new condition c'': $cC(DT, c', c'') := cO(DT, c') \bullet cI(DT, c'')$

Definition 7.8 (Rule Operators)

- A *rule insertion* operator (rI) inserts a new rule $r \notin R$:
$$rI(DT, r) := (AC, C, R \cup \{r\})$$

- A *rule omission* operator (rO) omits a rule r from set R:
$$rO(DT, r) := (AC, C, R \setminus \{r\})$$

- A *rule corruption* operator (rC) removes a rule r' and replaces it with a new rule r'': $rC(DT, r', r'') := rO(DT, r') \bullet rI(DT, r'')$

Note that applying an action, condition or rule operator to a given DT is not always meaningful. An example is adding an extra rule to a DT that already contains rules for all combinations of conditions.

7.1.2 Robustness Testing Process

Based on ESG and DT and their basic operators a scalable robustness testing process is now defined. It is assumed that the legal behavior of SUC is tested based on given ESG^{DT} by CES to cover all ES of a desired length, that is, all ES of

length 2. Firstly, the set of operators $\mathcal{X} = \{\chi_1, \chi_2, ..., \chi_n\}$ has to be defined. This set may consist of the basic operators or a subset of them. An operator $\chi_i \in \mathcal{X}$ that is applied to ESG^{DT} is denoted by $\chi_i(ESG^{DT})$ and generates a set of faulty models $ESG^{DT^*}_{\chi_i} = \{ESG^{DT^*}_{\chi_i,1}, ..., ESG^{DT^*}_{\chi_i,k}\}$. The size of this set is defined as $k := |ESG^{DT^*}_{\chi_i}|$. Note that depending on the chosen operator χ_i the size k may vary: $1 \leq k \leq \max(\chi_i)$. For example, the eO-operator for ESG can generate a maximum of $|E|$ faulty models, that is, $\max(eO) := |E|$. This enables the definition of the following scalable robustness measure.

Definition 7.9 (*k*-Robustness) A system is k-robust if for every operator $\chi_i \in \mathcal{X}$ and FCESs generated based on k faulty models $\chi_i(ESG^{DT}) = \{ESG^{DT^*}_{\chi_i,1}, ..., ESG^{DT^*}_{\chi_i,k}\}$ the system handles acceptably the exceptional inputs.

Depending on the test resources available the number of faulty models (given by k) and/or the number of operators (given by n) can be adapted. Algorithm 7.1 depicts the overall robustness testing process. For each operator $\chi_i \in \mathcal{X}$ a set of k faulty models is generated. A test generation algorithm Φ is applied to a faulty model $ESG^{DT^*}_{\chi_i,j}$ yielding a test set consisting of FCES to be applied to SUC.

Algorithm 7.1: k-Robustness testing process

Input: ESG^{DT} that describes SUC
Set of basic operators $\mathcal{X}$
$k :=$ number of faulty models
Test generation algorithm Φ
Output: List of faults in SUC detected

```
1 foreach χ_i ∈ 𝒳 do
2     ESG^{DT*}_{χ_i} := χ_i(ESG^{DT});
3     for j := 1 to k do
4         T*_{i,j} := Φ(ESG^{DT*}_{χ_i,j});
5         foreach t* ∈ T*_{i,j} do
6             Execute SUC against t*;
7             Observe output of SUC;
```

7.1.3 Case Study

A case study checks the ability of different basic operators for revealing faults. The SUC is a commercial web portal (ISELTA–Isik's System for Enterprise-Level Web-Centric Tourist Applications [77]) for marketing touristic services. Figure C.1 (in Appendix C) shows the entry page. ISELTA enables hotels and travel agencies to create individual search masks. These masks are embedded in existing homepages of hotels as an interface between customers and the system. Visitors of the website can then use these masks to select and book services, e.g., hotel rooms or special offers. An example of a search mask for booking hotel rooms is given by Figure 7.1. A second example of a search mask is given by Figure C.2 (in Appendix C) showing a mask to book special offers. This part of the system has been used within the case study. To set up a special offer a dedicated website as shown by Figure C.3 (in Appendix C) is used. The user has to provide the following data for a new offer: Arrival/departure, kind of room, number of available special offers that can be booked by customers, price, photo, description, and a name.

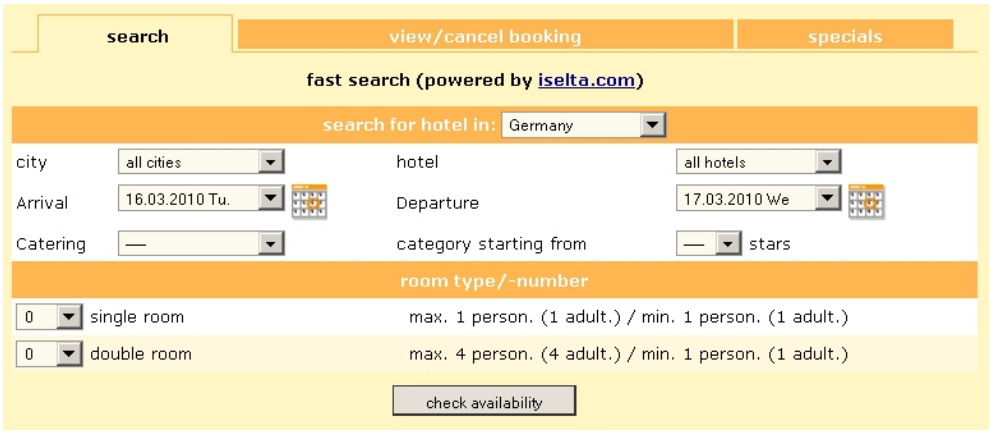


Figure 7.1: Search mask for hotel rooms in ISELTA

For the case study two tools have been used. ESG models were created by using the tool MTSG. An example of an ESG is given by Figure C.4 (in Appendix C). Decision tables were modeled with the ETES tool (decision tables for event sequences [129]). A snapshot of the program is given by Figure C.5 (in Appendix

C). The following set of operators has been chosen: $\mathcal{X} = \{sC, acC, cC, rC\}$. Test cases representing faulty complete event sequences were generated based on the mutants modeled by these operators. In total, 520 test cases were generated and applied to SUC, whereby $k = 130$. As a result 18 faults were detected. Table 7.1 shows the operators used and the number of faults detected by each one. It turns out that the cC-operator is the most effective one in revealing faults. Table 7.2 lists the faults revealed and the corresponding operator that was used to create the faulty model and thereby corresponding test case. However, the validity of the results is limited. Firstly, SUC is a web-based system. In contrast to other kinds of systems it runs on different browsers and operating systems featuring a server/client architecture. Furthermore, the results only hold for the operators and models chosen. Thus, other operators or other models could lead to different results.

Table 7.1: Number of faults detected in SUC

Mutation operator χ	**Number of faults detected**
cC	9
sC	4
rC	3
acC	2

Table 7.2: List of faults revealed

ID	Fault description	χ
1	Files with size greater than 1 MB can be chosen.	cC
2	Files with size greater than 1 MB can be appended.	cC
3	All kind of files can be uploaded as photos.	cC
4	If departure date is set firstly to the past, only the arrival date is set to the past.	sC
5	It is possible to set only a departure date. A PHP warning appears.	acC
6	Inserting "\" into price field leads to an error.	cC
7	The number of special offers available can be set to zero.	acC
8	Missing warning for wrong file types.	rC
9	Wrong input in other fields deletes the file path of the photo.	sC
10	No warning in case of a file with 0 bytes. The browser freezes.	cC
11	Arrival and departure field can remain empty.	cC
12	Processing faulty input data doubles "\" in different fields.	cC
13	In existing offers the arrival date can be set to dates into the past.	rC
14	In existing offers the departure date can be set to dates into the past.	rC
15	If a departure date in the past is set to another day in the past, the arrival is set to the current date.	cC
16	Missing warning for incorrect prices and number of special offers.	sC
17	Missing warnings for empty dates.	sC
18	The number of specials can be higher than rooms available.	cC

7.2 Mutant-Based Testing with Model Checkers

In model checking ([47]) a model checker is applied to a state-based model $\mathcal{M}$ of SUC that represents the behavior of the system. Temporal logic is used to specify the desired properties which the system must fulfill. For a given property φ, a model checker visits all reachable states of the model to verify whether the property is satisfied over each possible path (denoted by $\mathcal{M} \models \varphi$). In case that the property does not hold ($\mathcal{M} \not\models \varphi$), the model checker generates a counterexample in form of a trace as a sequence of states.

Thereby, model checking techniques enable generation of test cases for a variety of test adequacy criteria [128] that are formalized as properties in temporal logic. Test sequences are generated by using counterexamples that are produced from a model checker ([53], [66]). In [65], a technique has been presented to generate test cases from abstract state machine specifications to detect specific fault classes. The idea of combining model checking and mutation testing is not new, e.g., for generating test cases through comparison of the mutated specification with the original one [6], or for generating test cases to measure test coverage [4]. In order to reduce test costs, model checkers are used to detect equivalent mutants that result in redundant test cases. In [64], a new method of mutant minimization is proposed for reducing the number of mutants and thereby the size of the test suites. In general, mutants result in different, unexpected system behavior, contrary to the one as described by the original model of SUC.

There is a close relationship between mutation testing and fault class analysis because mutation operators are often designed to model common faults [86]. Some testing strategies are proposed to detect certain fault classes. Weyuker et al. introduced meaningful impact strategies to target specifically the variable negation fault that occurs when a Boolean variable is erroneously substituted by its negation [123]. Chen and Lau proposed two more powerful testing strategies capable to detect more fault classes [42]. If these test cases pass, it is guaranteed that no such faults are contained in the system, which is an important advantage over other testing methods.

In the following an approach to mutant-based testing with model checkers is presented to generate test cases from the original model to check the expected be-

havior (*positive testing*), and from well-defined mutants to check unexpected behavior (*negative testing*) of SUC ([24], [43]). Mutation operators are applied to the original model and can increase, or decrease its size, that is, the number of state transitions. Based on the original model and its mutants, a model checker is then used to generate counterexamples and thereby test cases that are finally applied to the SUC.

7.2.1 Model Checking as a Basis for Mutation Testing

Model checking typically depends on a finite state space, which is generated based on a set of initial states and a transition relation. The state space is usually represented by a graph structure. It is common to use Kripke models ([47]).

Definition 7.10 (Kripke Model) A Kripke model (or model) is a 3-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{R}, \mathcal{I})$, where:

1. $\mathcal{S}$ is a finite set of reachable states,
2. $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation, that must be total, that is, for every state $s \in \mathcal{S}$ there is a state $s' \in \mathcal{S}$ such that $(s, s') \in \mathcal{R}$, and
3. $\mathcal{I}$ is a set of initial states.

A mutant is a model $\mathcal{M}^* = (\mathcal{S}^*, \mathcal{R}^*, \mathcal{I}^*)$ which is similar to the original one $\mathcal{M} = (\mathcal{S}, \mathcal{R}, \mathcal{I})$, but differs from $\mathcal{M}$. It is assumed that $\mathcal{I} = \mathcal{I}^*$. $\mathcal{S} \cup \mathcal{S}^*$ is considered as the set of states both in $\mathcal{M}$ and $\mathcal{M}^*$, in which the states are not required to be reachable. Without losing generality, it is assumed that $\mathcal{S} = \mathcal{S}^*$. Hence, the difference of model $\mathcal{M}$ and mutant $\mathcal{M}^*$ is determined by $\mathcal{R}$ and $\mathcal{R}^*$. That is, $\mathcal{M} = \mathcal{M}^*$ if and only if $\mathcal{R} = \mathcal{R}^*$. Checking of extra or missing states can be reduced to checking extra or missing transitions. This view leads to the following formal definition of the notion *mutant* of a Kripke model.

Definition 7.11 (Mutant of Kripke Model) Given a model $\mathcal{M} = (\mathcal{S}, \mathcal{R}, \mathcal{I})$, a Kripke model $\mathcal{M}^* = (\mathcal{S}, \mathcal{R}^*, \mathcal{I})$ is a *mutant* of $\mathcal{M}$. $\mathcal{M}^*$ is equivalent to $\mathcal{M}$ if $\mathcal{R} = \mathcal{R}^*$.

Concerning their structure, mutants are classified in three types:

Definition 7.12 ($\mathcal{I}$-, $\mathcal{D}$-, $\mathcal{C}$-Mutant of Kripke Model)

1. $\mathcal{M}^*$ is called an *increscent mutant* ($\mathcal{I}$*-mutant*) of $\mathcal{M}$ if $\mathcal{R} \subseteq \mathcal{R}^*$.
2. $\mathcal{M}^*$ is called a *decrescent mutant* ($\mathcal{D}$*-mutant*) if $\mathcal{R}^* \subseteq \mathcal{R}$.
3. $\mathcal{M}^*$ is called a *cross mutant* ($\mathcal{C}$*-mutant*) if $\mathcal{M}^*$ is neither an $\mathcal{I}$-mutant nor a $\mathcal{D}$-mutant.

Definition 7.13 (Path) In a Kripke model $\mathcal{M}$, a *path* is an infinite sequence of states $\pi = s_1 s_2 \cdots$, such that $(s_i, s_{i+1}) \in \mathcal{R}$ for every $i \geq 1$. A prefix $s_1 \cdots s_m$ of path π is denoted by π_m.

A prefix π_m could also be considered as a set of transitions, that is, $\{(s_i, s_{i+1}) | 1 \leq i \leq m-1\}$. If each transition (s_i, s_{i+1}) in a prefix π_m is also in $\mathcal{R}$, then π_m is in $\mathcal{R}$, denoted by $\pi_m \subseteq \mathcal{R}$.

Definition 7.14 (Test Case) Let $\mathcal{M}^* = (\mathcal{S}, \mathcal{R}^*, \mathcal{I})$ be a mutant of $\mathcal{M} = (\mathcal{S}, \mathcal{R}, \mathcal{I})$. A test case is a finite prefix π_m such that $\pi_m \subseteq \mathcal{R} \cup \mathcal{R}^*$ and $\pi_m \not\subseteq \mathcal{R} \cap \mathcal{R}^*$.

Test cases are generated based on counterexamples produced by a model checker. A counterexample is interpreted as a prefix of a path showing the difference between model $\mathcal{M}$ and its mutant $\mathcal{M}^*$ that contains behavior different from the original one. It is assumed that there are no *hidden variables* in model $\mathcal{M}$ that represent mutants which cannot be observed by testing. Each counterexample is used to derive test cases to be applied to the SUC. Test cases are classified as two types:

Definition 7.15 (Positive Test Case) A test case π_m is *positive* if $\pi_m \subseteq \mathcal{R}$ and $\pi_m \not\subseteq \mathcal{R}^*$. If π_m fulfills this conditions it is denoted by $\pi_m^{(p)}$.

Definition 7.16 (Negative Test Case) A test case π_m is *negative* if $\pi_m \subseteq \mathcal{R}^*$ and $\pi_m \not\subseteq \mathcal{R}$. If π_m is a *negative test case*, it is denoted by $\pi_m^{(n)}$.

Linear temporal logic (LTL) is used to specify the desired properties which the system must satisfy. LTL consists of atomic propositions, Boolean operators and temporal operators [47]. Two temporal operators X and G will be used to specify the properties needed. X is the *next-state operator*, e.g., $X\varphi$ expresses that φ has to

be $true$ in the next state. G is the *always* operator, e.g., $G\varphi$ expresses that φ holds at all states of a path.

If $\mathcal{M}^*$ is a mutant of $\mathcal{M}$, then at least one of the following cases applies:

1. $\mathcal{M}$ contains behavior that is not in $\mathcal{M}^*$, that is, $(\mathcal{R} \setminus \mathcal{R}^*) \neq \emptyset$.

2. $\mathcal{M}^*$ contains behavior that is not in $\mathcal{M}$, that is, $(\mathcal{R}^* \setminus \mathcal{R}) \neq \emptyset$.

Therefore, the LTL formula $G\neg(s \wedge Xs')$ can be used to generate the counterexample with the desired transition $(s, s') \in (\mathcal{R} \setminus \mathcal{R}^*)$ or $(s, s') \in (\mathcal{R}^* \setminus \mathcal{R})$. A challenge is to extract the desired transitions from the model specification. The next section introduces the notions of positive and negative property for extracting these transitions using NuSMV model checker.

7.2.2 A Realization using NuSMV Model Checker

The model checker NuSMV [46] is used to demonstrate how mutation testing can be realized by model checking using positive and negative testing. Figure 7.2 shows a simple example of a NuSMV specification.

```
MODULE main
VAR
   request : boolean;
   state : {ready, busy};

ASSIGN
   init(state) := ready;
   next(state) := case
   state = ready & request = 1 : busy;
   1 : {ready, busy};
                  esac;
```

Figure 7.2: Example of NuSMV specification

In NuSMV the variables are defined in the `VAR` section. The transition system is defined in the `ASSIGN` section. The system's initial conditions are declared using `init()` statements. The transition relation is specified using a `case` expression, in which `next` statements assign the next values of variables. A similar method to specify the transition relation is realized using `TRANS` statements. Note that the conditions of `TRANS` expressions are ordering insensitive but the `case` statements are ordering sensitive. Each condition of the `TRANS` statement is formalized as $\alpha_i \wedge \texttt{next}(\beta_i)$. Each statement $\mathcal{T}_k$ interprets a set of transitions (s, s') such that s satisfies α_k and s' satisfies β_k.

Definition 7.17 (TRANS statement)

1. $\mathcal{T}_k := \alpha_k \wedge X(\beta_k)$,
2. $\mathcal{T} := \bigvee_{i=1}^{n} \mathcal{T}_i$, and
3. $\mathcal{T}/k := \bigvee_{i=1}^{k-1} \mathcal{T}_i \vee \bigvee_{i=k+1}^{n} \mathcal{T}_i$.

A model specification representing a Kripke model serves as input for a model checker. Instead of mutating the Kripke model, the NuSMV model specification is used. Mutation operators are defined to model single mutations, always assuming that the intended implementation closely matches the actual implementation (*competent programmer hypothesis*). An *atomic element* of a mutation of a Boolean specification is a literal, which is a variable or a negated variable. First-order, basic mutation operators are defined as follows.

Definition 7.18 (Literal Operators)

- A *literal omission operator* (lO-operator) omits a literal from a Boolean expression.
- A *literal insertion operator* (lI-operator) inserts another possible literal.
- A *literal corruption operator* (lC-operator) replaces a literal by another possible literal.

Example 7.1 Applying the lO-operator to the expression $a \wedge b \wedge \neg c$ yields $a \wedge b$ with omitting $\neg c$.

Example 7.2 Applying the lI-operator to the expression $a \wedge b \wedge \neg c$ could be implemented as $a \wedge b \wedge \neg c \wedge d$ by inserting d.

Example 7.3 Applying the lC-operator to the expression $a \wedge b \wedge \neg c$ could be implemented as $a \wedge b \wedge d$ with $\neg c$ replaced by d or a literal could be replaced by its negation, e.g., $a \wedge b \wedge \neg c$ is implemented as $a \wedge \neg b \wedge \neg c$ where b is replaced by $\neg b$.

For simplicity, only single mutations in one of the transition conditions are considered, denoted by $\mathcal{T}_k^*$, that is, either $\alpha_k \neq \alpha_k^*$ or $\beta_k \neq \beta_k^*$. The mutated transition conditions are classified in three types:

Definition 7.19 ($\mathcal{I}$-, $\mathcal{D}$-, $\mathcal{C}$-Transition)

1. $\mathcal{T}_k^*$ is called an *increscent transition* ($\mathcal{I}$*-transition*) if $\mathcal{T}_k \Rightarrow \mathcal{T}_k^*$, that is, $\mathcal{T}_k$ implies $\mathcal{T}_k^*$.

2. $\mathcal{T}_k^*$ is called a *decrescent transition* ($\mathcal{D}$*-transition*) if $\mathcal{T}_k^* \Rightarrow \mathcal{T}_k$, that is, $\mathcal{T}_k^*$ implies $\mathcal{T}_k$.

3. $\mathcal{T}_k^*$ is called a *cross transition* ($\mathcal{C}$*-transition*) if $\mathcal{T}_k^*$ is neither an $\mathcal{I}$-transition nor a $\mathcal{D}$-transition.

An equivalent mutated transition implies an equivalent mutant, but an inequivalent mutated transition does not imply an inequivalent mutant. An $\mathcal{I}$-transition will create an $\mathcal{I}$-mutant and a $\mathcal{D}$-transition will create a $\mathcal{D}$-mutant. However, a $\mathcal{C}$-transition also may create an $\mathcal{I}$-mutant or a $\mathcal{D}$-mutant, because mutated behavior may be masked by other transitions $\mathcal{T}_i$ $(i \neq k)$. Hence, mutated behavior will satisfy $(\mathcal{T}_k \wedge \neg\mathcal{T}_k^* \wedge \neg\mathcal{T}/k)$ or $(\mathcal{T}_k^* \wedge \neg\mathcal{T}_k \wedge \neg\mathcal{T}/k)$.

Definition 7.20 (Positive Property) Let $\mathcal{M}_k^*$ be the mutant of $\mathcal{M}$ w.r.t. $\mathcal{T}_k^*$. Then the positive property is defined as

$$\varphi_k = G\neg(\mathcal{T}_k \wedge \neg\mathcal{T}_k^* \wedge \neg\mathcal{T}/k).$$

Definition 7.21 (Negative Property) Let $\mathcal{M}_k^*$ be the mutant of $\mathcal{M}$ w.r.t. $\mathcal{T}_k^*$. Then the negative property is defined as

$$\varphi_k^* = G\neg(\mathcal{T}_k^* \wedge \neg\mathcal{T}_k \wedge \neg\mathcal{T}/k).$$

Let $\mathcal{M}_k^*$ be a mutant of $\mathcal{M}$ w.r.t. $\mathcal{T}_k^*$. The LTL formula φ_k is designed to find the expected behavior of $\mathcal{M}$, that is, transitions in $\mathcal{R} \setminus \mathcal{R}^*$. The LTL formula φ_k^* is designed to find the unexpected behavior of $\mathcal{M}_k^*$, that is, transitions in $\mathcal{R}^* \setminus \mathcal{R}$. Therefore, $\mathcal{M}^*$ is equivalent to $\mathcal{M}$ if and only if $\mathcal{M} \models \varphi_k$ and $\mathcal{M}^* \models \varphi_k^*$, for mutant $\mathcal{M}_k^*$ w.r.t. $\mathcal{T}_k^*$.

If $\mathcal{T}_k^*$ is an $\mathcal{I}$-mutant of $\mathcal{T}_k$ ($\mathcal{T}_k$ implies $\mathcal{T}_k^*$), then $\mathcal{T}_k \wedge \neg\mathcal{T}_k^* \equiv false$, thus $\varphi_k \equiv true$ and $\mathcal{M} \models \varphi_k$. Therefore, $\mathcal{M}$ is equivalent to $\mathcal{M}_k^*$ if and only if $\mathcal{M}_k^* \models \varphi_k^*$. Hence, a test case that detects an $\mathcal{I}$-mutant must be negative. Similarly, if $\mathcal{T}_k^*$ is a $\mathcal{D}$-mutant of $\mathcal{T}_k$, then $\mathcal{M}$ is equivalent to $\mathcal{M}_k^*$ if and only if $\mathcal{M} \models \varphi_k$. Hence, a test case that detects a $\mathcal{D}$-mutant must be positive. In Appendix D a thermostat system is used to illustrate the notions of $\mathcal{I}$-, $\mathcal{D}$- or $\mathcal{C}$-mutant.

The overall test process is given by Algorithm 7.2. SUC is to be modeled by a NuSMV model checker specification $\mathcal{M}$. For each mutation operator the number of mutants to be constructed is to be determined and assigned to variable n. Model checking is carried out using the positive and negative property. The final step translates the counterexamples into appropriate test cases which are then used to exercise SUC. The runtime of Algorithm 7.2 is essentially given by the number of mutants that are created. For each mutant model checking must be conducted, and concluding the resulting counterexample is to be applied as an appropriate input to SUC.

Algorithm 7.2: Mutant-based testing with NUSMV model checker

Input: NuSMV specification $\mathcal{M}$ for SUC
Set of mutation operators $\mathcal{X}$
Number of mutants n to be generated for each operator

```
1  foreach χ ∈ 𝒳 do
2      for i := 1 to n do
3          Use operator χ to generate a mutant M*_k w.r.t. to T*_k;
4          if M*_k ⊭ φ*_k then
5              Observe π^(n)_m;
6              Apply counterexample π^(n)_m to SUC;
7              Observe system behavior;
8          if M ⊭ φ_k then
9              Observe π^(p)_m;
10             Apply counterexample π^(p)_m to SUC;
11             Observe system behavior;
```

7.3 Model-Based Integration Testing

This section introduces an approach for model-based integration testing. After a summary of existing work communication sequence graphs (CSG) are introduced for representing the communication between software components on a meta-level ([27], [28]). Based on CSG, test coverage criteria and basic mutation operators are defined. These operators enable to evaluate the test cases generated by mutation analysis.

Several approaches to integration testing have been proposed. Binder ([31]) gives different examples of common techniques such as top-down and bottom-up integration testing. Hartmann et al. ([72]) use UML statecharts specialized for object-oriented programming. Delamaro et al. ([50]) introduced a communication-oriented approach that mutates the interfaces of software units. Saglietti et al. ([110]) introduced an interaction-oriented, higher-level approach and several test coverage criteria. In addition, different approaches for object-oriented software have been proposed. Buy et al. ([41]) defined method sequence trees for representing the call structure of methods. Daniels et al. ([48]) introduced different test coverage criteria for method sequences and Martena et al. ([91]) defined interclass testing.

In the following, communication sequence graphs (CSG) are introduced to represent source code at different levels of abstraction. Software systems with discrete behavior are considered. In contrast to existing, mostly state-based approaches described above, CSG-based models are stateless, that is, they do not concentrate on internal states of the software components, but rather focus on events. CSGs are directed graphs enriched by semantics to adapt them for integration testing. This enables the direct application of well-known algorithms from graph theory and automata theory for test generation and test minimization. The syntax of CSG is based on directed graphs (Definition 3.1). The goal is to enable a uniform modeling for unit and integration testing by using the same modeling techniques for both levels.

7.3.1 Fault Modeling

Depending on the applied programming language, a software component represents a set of functions including variables. Classes contain methods and variables in the

object-oriented paradigm. In case that no model exists, the first step is to model the components c_i of the SUC, represented as $C = \{c_1, ..., c_n\}$. Two components, $c_i, c_j \in C$ of a software system C communicate with each other by sending messages from c_i to c_j, that is, the communication is directed from c_i to c_j. Figure 7.3 shows the communication between a calling software component ($c_i \in C$) and an invoked component ($c_j \in C$).

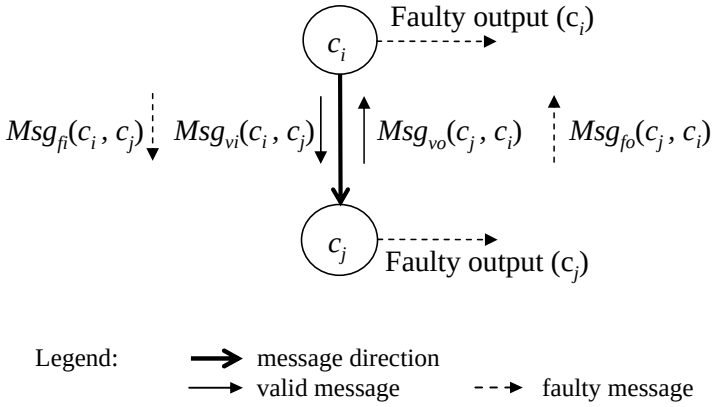


Figure 7.3: Message-oriented model of integration faults between two software components c_i and c_j

Messages to realize this communication are represented as tuples Msg of parameter values and global variables and can be transmitted correctly or faulty. This leads to the following combinations.

1. $Msg_{vi}(c_i, c_j)$: valid input from c_i to c_j,
2. $Msg_{vo}(c_j, c_i)$: valid output from c_j back to c_i,
3. $Msg_{fi}(c_i, c_j)$: faulty input from c_i to c_j, and
4. $Msg_{fo}(c_j, c_i)$: faulty output from c_j back to c_i.

It is assumed that either $Msg_{vi}(c_i, c_j)$ or $Msg_{fi}(c_i, c_j)$ is the initial invocation. As the result of this invocation, c_j sends its response back to c_i. The response is $Msg_{vo}(c_j, c_i)$ or $Msg_{fo}(c_j, c_i)$. It is further assumed that the tuples of faulty messages $Msg_{fi}(c_i, c_j)$ and $Msg_{fo}(c_j, c_i)$ cause faulty outputs of c_i and c_j as follows.

- component c_j produces faulty results based on
  - faulty parameters transmitted from c_i in $Msg_{fi}(c_i, c_j)$, or
  - valid parameters transmitted from c_i in $Msg_{vi}(c_i, c_j)$, but perturbed during transmission resulting in a faulty message.
- component c_i produces faulty results based on
  - faulty parameters transmitted from c_i to c_j, causing c_j to send a faulty output back to c_i in $Msg_{fo}(c_j, c_i)$, or
  - valid, but perturbed parameters transmitted from c_i to c_j, causing c_j to send a faulty output back to c_i in $Msg_{vo}(c_j, c_i)$ resulting in a faulty message.

This fault model helps to consider potential system integration faults and thus, to generate tests to detect them.

7.3.2 Communication Sequence Graphs for Unit Testing

In the following the term *actor* is used to generalize notions that are specific to the great variety of programming languages, for example, functions, methods, procedures, basic blocks, etc. An elementary actor is the smallest, logically complete unit of a software component that can be activated by or activate other actors of the same or other components. A software component $c \in C$ can be represented by a CSG as follows.

Definition 7.22 (Communication Sequence Graph) A communication sequence graph $CSG = (DG, \Xi, \Gamma)$ for a software component $c \in C$ is a directed graph, where

- $DG = (\Theta, A)$, as defined in Definition 3.1, with Θ being a finite set of actors of component c; and $A \subseteq \Theta \times \Theta$ being a finite set of arcs describing all valid pairs of concluding invocations (calls) within the component; and
- Ξ and Γ representing initial/final invocations.

To identify the initial and final invocations of a CSG graphically, all $\theta \in \Xi$ are preceded by a pseudo vertex "[" $\notin \Theta$ and all $\theta \in \Gamma$ are followed by another pseudo vertex "]" $\notin \Theta$. In object-oriented programming (OOP), these nodes typically represent invocations of a constructor and destructor of a class.

CSG are similar to ESG differing in following features. In an ESG, a node represents an event that can be a user input or a system response, both of which lead interactively to a succession of user inputs and expected system outputs. In a CSG, a node represents an actor invoking another actor of the same or another software component. An arc represents a sequence of immediately neighboring events in an ESG. In a CSG, an arc $(\theta, \theta') \in A$ denotes that actor θ' is invoked after the invocation of actor θ representing an invocation (call) of the successor node by the preceding node. Furthermore, CSG differ in following aspects from call graphs. CSGs have explicit boundaries (entry and exit in form of initial/final nodes) that enable the representation of not only the activation structure, but also the functional structure of the components, such as initialization and destruction of a software unit (for example, call of the constructor and destructor method in OOP).

CSG are directed graphs that enable the application of rich notions and algorithms of graph theory. The latter are useful not only for generation of test cases based on criteria for graph coverage, but also for optimization of test sets. CSG can easily be extended to represent not only the control flow, but also to precisely consider data flow, for example, by using Boolean algebra to represent constraints.

Definition 7.23 (Communication Sequence) Let Θ and A be the finite set of nodes and arcs of a CSG. Any sequence of nodes $(\theta_1, ..., \theta_k)$ is called a communication sequence (CS) if $(\theta_i, \theta_{i+1}) \in A$, for $i \in \{1, ..., k-1\}$.

The function l (symbolizing *length*) determines the number of nodes of a CS. In particular, if $l(CS) = 1$, then it is a communication sequence of length 1, which denotes a single node of CSG. Let α and ω be the functions to determine the initial and final invocation of a CS. For example, given a sequence $CS = (\theta_1, ..., \theta_k)$, the initial and final invocation are $\alpha(CS) = \theta_1$ and $\omega(CS) = \theta_k$, respectively.

Definition 7.24 (Complete Communication Sequence) A CS is a complete communication sequence (CCS) if $\alpha(CS)$ is an initial and $\omega(CS)$ is a final invocation.

Based on Definitions 7.23 and 7.24 the following coverage criterion is enabled.

Definition 7.25 (k-Communication Sequence Coverage) Generate complete communication sequences that sequentially invoke all communication sequences of length $k \in \mathbb{N}$.

All communication sequences of a given length k of a CSG are to be covered by means of complete communication sequences that represent test cases. Thus, test generation is a derivation of Chinese Postman Problem, that is, finding the shortest path or circuit in a graph by visiting each arc. Polynomial algorithms supporting this test generation process have been published in [14]. Algorithm 7.3 sketches the test generation process for unit testing.

Algorithm 7.3: Test generation for unit testing

Input: CSG
$k :=$ max. length of communication sequences (CS) to be covered
Output: Test report of succeeded and failed test cases

```
1 for i := 1 to k do
2     Cover all CS of CSG by means of CCS;
3     Apply test cases to SUC and observe system outputs;
```

7.3.3 Communication Sequence Graphs for Integration Testing

For integration testing, the communication between software components has to be tested. The approach is based on the communication between pairs of components.

Definition 7.26 (Invocation Relation) Communication between actors of two different software components, $CSG_i = (\Theta_i, A_i, \Xi_i, \Gamma_i)$ and $CSG_j = (\Theta_j, A_j, \Xi_j, \Gamma_j)$ is defined as an invocation relation $IR(CSG_i, CSG_j) = \{(\theta, \theta') | \theta \in \Theta_i$ and $\theta' \in \Theta_j$, where θ activates $\theta' \}$.

An actor $\theta \in \Theta$ may invoke an additional actor $\theta' \in \Theta'$ of another component. Without losing generality, the notion is restricted to communication between two

units. If θ causes an invocation of a third unit, this can also be represented by a second invocation considering the third one.

Definition 7.27 (Composed CSG) Given a set of $CSG_1, ..., CSG_n$ describing n components of a system C, and a set of invocation relations $IR_1, ..., IR_m$, the composed CSG_C is defined as $CSG_C = (\{\Theta_1 \cup ... \cup \Theta_n\}, \{A_1 \cup ... \cup A_n \cup IR_1 \cup ... \cup IR_m\}, \{\Xi_1 \cup ... \cup \Xi_n\}, \{\Gamma_1 \cup ... \cup \Gamma_n\})$.

Figure 7.4 gives an example of a composed CSG that consists of CSG_1 and CSG_2. Invocation of actor θ_1' by actor θ_2 is denoted by a dashed line, that is $IR(CSG_1, CSG_2) = \{(\theta_2, \theta_1')\}$.

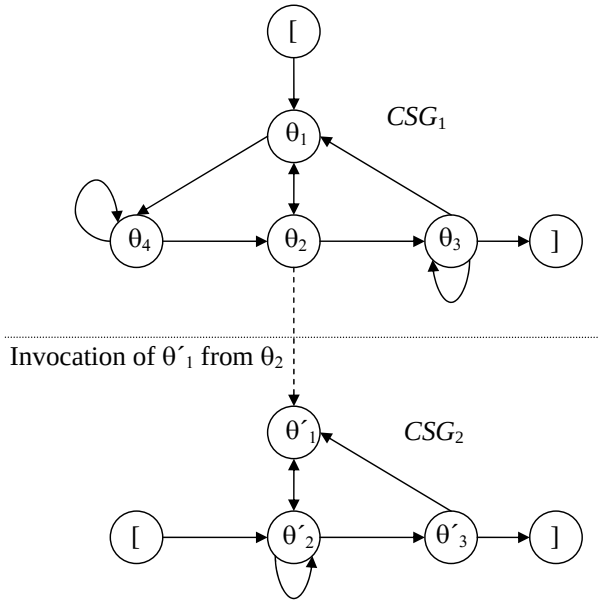


Figure 7.4: Composed CSG_C consisting of CSG_1 and CSG_2 and an invocation between them

Based on the k-communication sequence coverage criterion, Algorithm 7.4 represents a test generation procedure. For each software component $c_i \in C$, a communication sequence graph CSG_i and invocation relations IR serve as input. The composed CSG_C is to be constructed according to Definition 7.27. Concluding, Algorithm 7.3 is also applied for test generation.

Algorithm 7.4: Test generation for integration testing

Input: $CSG_1, ..., CSG_n$
$IR_1, ..., IR_m$
$k :=$ max. length of communication sequences (CS) to be covered

Output: Test report of succeeded and failed test cases

1 $CSG_C :=$
$(\{\Theta_1 \cup ... \cup \Theta_n\}, \{A_1 \cup ... \cup A_n \cup IR_1 \cup ... \cup IR_m\}, \{\Xi_1 \cup ... \cup \Xi_n\}, \{\Gamma_1 \cup ... \cup \Gamma_n\})$;

2 Use Algorithm 7.3 with (CSG_C, k) for test generation;

7.3.4 Mutation Analysis

In the following, an approach to mutation analysis is proposed to assess the adequacy of a test test with respect to its fault detection capability. The procedure of integration testing and mutation analysis for a system or program P based on CSG is illustrated in Figure 7.5. The solid lines describe test generation based on CSG as described by Algorithm 7.3 and Algorithm 7.4 and test execution based on test set T. The dashed lines describe the follow-on mutation analysis activities. Based on DG (Definition 3.1) and CSG, a set of basic operators $\mathcal{X}$ as defined in Table 7.3 realize insertion or omission of nodes and arcs of the CSG. The changes are accordingly applied to P to create the mutants. Finally, each mutant P^* is executed against test set T to compute the mutation score.

The nI-operator in Table 7.3 adds a new node to the CSG, generating a new communication sequence from a node via the new node to another node of the same software unit. That is, a new call is inserted in the source code of a component between two calls. The nO-operator omits a node from a CSG and connects the former in-going arcs to all successor nodes of the deleted node. This is done by removing an invocation from the source code of a software unit. The operator aI inserts a new arc from a node to another node of the same component that had no connection before applying the operator. Similarly, aO deletes an arc from a node to another node of the same software component.

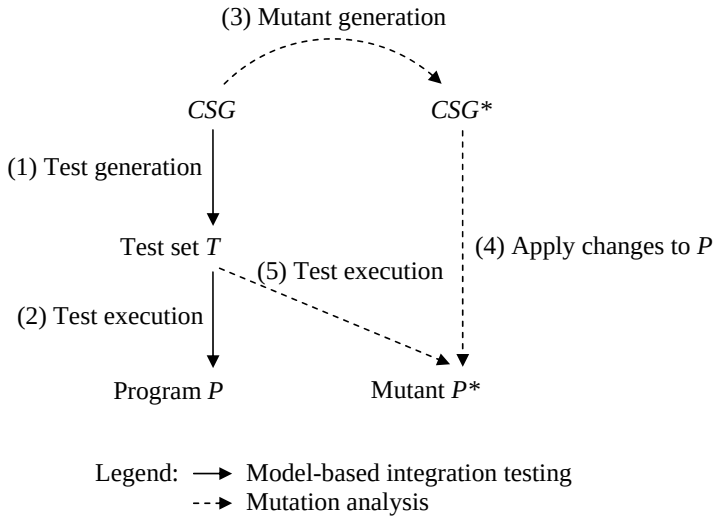


Figure 7.5: Model-based integration testing and mutation analysis with CSG

Table 7.3: Basic mutation operators for CSG

χ	**Description**
nI	Inserts a new node into the CSG
nO	Omits a node from the CSG
aI	Inserts a new arc into the CSG
aO	Omits an arc from the CSG
iI	Inserts an invocation between actor θ of component c to actor θ' of component c'
iO	Omits an invocation between actor θ of component c to actor θ' of component c'

7.4 Summary

This chapter presented different applications of basic operators and mutation testing using different kinds of models. Section 7.1 introduced a scalable robustness testing approach based on ESG and decision tables. Different operators are employed to create faulty models representing faulty behavior of SUC. Test cases are generated based on these models to reveal potential issues with respect to illegal inputs to SUC. A case study based on a commercial web portal shows the potential of the approach to reveal further critical faults that were not detected by previous tests with legal inputs. Furthermore, an approach to mutant-based testing with model checkers was presented in Section 7.2 to generate test cases using basic operators. Based on model-checker specifications mutants are generated. A model checker is then used to generate counterexamples and thereby test cases to be applied to SUC. Section 7.3 introduced communication sequence graphs and a mutation-oriented test process incorporating basic operators.

Chapter 8

Related Work

Related work discussed in this chapter focuses on mutation analysis. Related work on modeling, especially graph-based modeling, has been discussed in the previous chapters whenever it was significant in the context.

Mutation analysis was originally proposed as a white-box, fault injection-based technique for implementation-based software testing, especially at the unit-level. Recent works have also applied it to other testing levels such as integration-level, and other artifacts such as program specifications. An overview of variations of mutation analysis related to the one proposed in this thesis is presented focusing on the mutation operators used to generate these mutants.

Random testing compared to mutation analysis is also a commonly used technique for test data generation because of its simplicity, full automation, and direct implementation characteristics. However, its probability of selecting specific values of interest from the program input domain is small. According to Frankl et al. ([63]) random testing is a poor way to achieve mutation-adequate test sets. Thus, random testing is ineffective to generate test cases for achieving a high level of coverage.

8.1 Implementation-Based Mutation Analysis

In implementation-based mutation analysis, mutants are generated by applying mutation operators to the source code to introduce simple syntactic changes. These operators are language dependent. For example, there are different sets of operators

for Ada ([35], [104]), C ([2]), COBOL ([69]), Fortran 77 ([83]), Lisp ([38]), and Java ([82], [89], [90]). The operators can be further classified. *Traditional mutation operators* are used for testing a function or a unit of procedural programs, such as the 22 operators used by Mothra [52] for Fortran programs and the 77 operators used by Proteum [2], [49] for C programs. *Interface mutation operators* include operators such as those used by Proteum/IM [50] that apply mutation analysis at the integration-level. Test cases generated to kill these mutants are a good indicator of how well the interactions between different modules of an application have been tested. The focus is on the source code related to module interactions, including function calls, parameters, global variables, etc. *Object-oriented mutation operators* are a new category of mutation operators, such as the class and inter-class mutation operators ([82], [89], [90]). Theses operators are necessary to handle inheritance, polymorphism, dynamic binding, and other specific features of object-oriented languages.

8.2 Model-/Specification-Based Mutation Analysis

Mutation operators can also be applied to other artifacts such as program specifications or architecture and design models. For each method, it can be further divided by whether it is for testing the specification/model itself or for testing the corresponding implementation based on the specification/model. Another way to classify is to focus on whether a method follows the procedure described in Figure 2.2 or Figure 2.5. Studies such as [117] follow the procedure in Figure 2.2 to test the specification, whereas studies such as [33] and [84] follow the procedure in Figure 2.5 to test the implementation. However, the proposed method in this thesis is the only mutation testing method that follows Figure 2.4.

For mutation analysis methods that follow the procedure described in Figure 2.2, test cases are executed against the original specification/model and the mutated specifications/models (namely, the mutants) to see how many mutants they kill. This type of mutation analysis has some benefits over implementation-based mutation analysis discussed in Section 8.1. Implementation-based analysis requires the access to the source code of SUC which might not always be available. Additionally, mutation analysis based on a specification can be done independently of

the program development. According to [105], mutation analysis is actually easier at the specification level for some types of specifications. Furthermore, mutation analysis based on specifications can help to detect faults that are only hard to find by analyzing only the source code.

An overview of different studies that have proposed different sets of mutation operators for various specifications and models is presented in the following.

8.2.1 Estelle

Estelle is a formal description technique to model distributed systems, services, and protocol specifications. A hierarchy of extended FSMs (EFSM) communicate through bi-directional channels. Estelle has been developed by ISO (International Organization for Standardization) ([78]). Probert and Guo [107] defined a mutation analysis technique to validate the behavior of Estelle-based specifications. They called it E-MPT (Estelle-directed Mutation-based Protocol Testing). It focuses on validating the elementary structures of Estelle and the operations of the transitions. Later, Souza et al. [116] defined three categories of mutation operators: *module mutation*, *interface mutation*, and *structure mutation*. These operators consider the intrinsic characteristics of Estelle such as synchronous and asynchronous parallelism, asynchronous communication, and dynamic structures.

8.2.2 Finite-State Machines & Statecharts

Fabbri et al. [55] applied mutation analysis in the context of specifications based on FSMs. They defined a set of mutation operators that bases on the error classes defined by Chow [45] and on results of heuristic research to simulate typical errors made by a designer during modeling the FSM. To generate test sequences the W-Method and the TT-Method were used. The following nine operators were defined: *arc-missing*, *wrong-starting-state*, *event-missing*, *event-exchanged*, *event-extra*, *state-extra*, *output-exchanged*, *output-missing*, and *output-extra*. Later, a tool called Proteum/FSM was developed to support mutation analysis for FSM [56]. They also extended the approach to statecharts ([58]). They defined three different categories of operators: *FSM mutation operators* similar to the ones in [55],

EFSM mutation operators, and *Statecharts-feature-based mutation operators*. The tool PROTEUM/ST supports the use of mutation analysis in that context.

8.2.3 Model Checker Specifications

In model checking a system is described by a state machine that is checked if certain conditions defined by temporal logic constraints hold. In case that a certain property does not hold the model checker generates a counterexample in the form of a sequence of states. Another advantage of using model checkers is that equivalent mutants can be detected and therefore eliminated automatically. Ammann et al. combined mutation testing and model checking for automatically producing tests from model checker specifications [6] and measuring test coverage on the specification instead of the source code [5]. The mutation operators are either applied to the state machine or the temporal logic constraints. Counterexamples produced by the model checker are then converted into complete test cases and applied to the system. In case that the mutation operators have been applied to the state machine counterexamples represent *failing tests*, that is, the system is expected to diverge from the corresponding test cases. Mutants obtained from the temporal logic constraints result in *passing tests* where the system is supposed to pass the test cases. Black et al. ([32], [33]) refined the mutation operators defined by Ammann et al. [6] and defined new ones to be applied in the same manner to combine mutation testing and model checking.

8.2.4 Model-Driven Engineering

Testing the correctness of model transformations plays an important role in model-driven engineering (MDE) ([61], [62]). A model transformation process transforms an input model and returns an output model. In each case, these models must conform to an input and output meta-model as well. Mottu et al. [95] proposed an approach based on mutation analysis to evaluate the quality of test sets. Specific mutation operators are defined on the meta-model notion since the mutants must preserve the conformity with the meta-models.

8.2.5 Petri Nets

Fabbri et al. [57] applied mutation analysis to Petri nets. Based on the error classes of Chow [45] they defined the following set of mutation operators: *input-missing*, *input-extra*, *input-shifted*, *input-exchanged*, *output-missing*, *output-extra*, *output-shifted*, *output-exchange*, and *wrong-initial-marking*. The tool Proteum/PN supports the approach.

8.2.6 SDL

SDL (Specification and Description Language) is a language to describe the behavior of reactive and distributed systems such as telecommunication systems. It has been defined and standardized by ITU-T (International Telecommunications Union). Kovács et al. [84] applied mutation analysis to SDL to automate the process of conformance test generation and selection. They defined six types of mutation operators: *state modification operator*, *input modification operator*, *output modification operator*, *action modification operator*, *predicate modification operator*, and *save missing operator*. Two algorithms were defined for automatic test selection. The first algorithm derives test cases from an SDL specification by comparing the mutants of the specification with the original one. The second algorithm aims to optimize an existing test set by removing test cases that do not affect the mutation score. Although test cases generated/selected were used for conformance testing, their study still followed the procedure described in Figure 2.2, that is, generating/selecting test cases by comparing the execution results between the original SDL specification and the mutated SDL specifications. Sugeta et al. [117] proposed mutation analysis for SDL for testing the specification itself. They defined three groups of operators: *process mutant operators*, *interface mutant operators* and *structure mutant operators*.

8.2.7 XML & Web Applications

Lee and Offutt [87] presented an approach to use mutation analysis for testing the semantic correctness of XML-based interactions between web systems. The interactions are specified using an *interaction specification model* consisting of docu-

ment type definitions, message specifications and a set of constraints. Test cases and mutants are XML messages. Two initial mutation operator classes were proposed. Xu et al. [126] define *schema perturbation operators* that are used to modify XML schema and therefore to generate invalid messages. Li and Miller [88] proposed a technique for using mutation analysis to test the semantic correctness of W3C XML Schemas. Six mutation operators are designed to detect faults concerning name-spaces, user-defined types, and inheritance.

8.3 Comparison with Other Approaches

Many of the existing mutant generation techniques can be represented in a uniform way by using the basic operators introduced in this thesis. For discussion purposes, the 37 mutation operators reported by Fabbri et al. ([58]) are used to generate statechart-based mutants as an example. After a careful analysis, it can be noticed that each of them can be represented by an insertion operator, an omission operator, or a combination of these two, as follows:

- A missing arc, transition, event, state, input, history state, etc. can be represented by an omission operator.
- An extra arc, transition, event, state, input, history state, etc. can be represented by an insertion operator.
- A corrupted arc, transition, event, state, input, history state, etc. can be represented by an omission operator followed by an insertion operator.

Applying the basic operators leads to the following observations:

- omission of a state ("state-missing") is included in [58], but not in [55];
- states ("state-extra") can be inserted in [55], but not in [58].

Table 8.1 through Table 8.3 show how these 37 mutation operators can be generated by using an insertion (I) or omission (O) operator or a combination of these two. In each table, the first column contains an operator defined by Fabbri et al.

([58]). A cross "×" in the second and third column indicates insertion and/or omission of the artifact listed in the fourth column. Table 8.1 also includes another set of FSM operators defined by Fabbri et al. ([55]). From Table 8.1 through Table 8.3, it can be observed that the mutation operators introduced by Fabbri et al. can be structured and unified in a way that additional meaningful operators arise. For example, the two FSM operator sets defined by Fabbri et al. ([55] and [58]) in Table 8.1 can be represented by basic operators applied to *event*, *output*, *transition*, and *state*.

Table 8.1: FSM operator set and representation using the basic operators

FSM operator (left defined in [58] and right in [55])		**I**	**O**	**Artifact**
wrong-start-state	wrong-starting-state (default state)	×	×	start-state
arc-missing	arc-missing		×	arc
event-missing	event-missing		×	event
event-extra	event-extra	×		event
event-exchanged	event-exchanged	×	×	event
destination-exchanged	–	×	×	destination
output-missing	output-missing		×	output
output-exchanged	output-exchanged	×	×	output
–	output-extra	×		output
state-missing	–		×	state
–	state-extra	×		state

Table 8.2: EFSM operator set and representation using the basic operators

EFSM operator (defined in [58])	**I**	**O**	**Artifact**
expression deletion		×	expression
boolean expression negation	×		negation
term associativity shift	×	×	brackets
arithmetic operator by arithmetic operator	×	×	arithmetic operator
relational operator by relational operator	×	×	relational operator
logical operator by logical operator	×	×	logical operator
logical negation	×		logical negation
variable by variable replacement	×	×	variable
variable by constant replacement	×		constant
		×	variable
constant by required constant replacement	×		required constant
		×	constant
constant by scalar variable replacement	×		scalar variable
		×	constant

8.4 Summary

Model-based mutation testing (MBMT) as introduced in Section 2.2 has several advantages compared to existing approaches as cited here. It combines the concept of implementation-based mutation analysis with model-based testing and allows mutation analysis to be applied to software, where the source code is not available. The approach can also detect real (non-injected) latent faults in SUC. Furthermore, the iterations and combinations of the two basic mutation operators (*omission* and *insertion*) introduced in Chapter 3 can help to supersede and systematize the broad variety of existing mutation operators.

Table 8.3: Statecharts-feature-based operator set and representation using the basic operators

Statecharts-feature-based operator (defined in [58])	**I**	**O**	**Artifact**
transitions history deletion		×	transitions history
transition with history by transition replacement		×	transition with history
	×		transition
history-missing		×	history
h by h* replacement	×		*
h* by h replacement		×	*
h-extra	×		h
h*-extra	×		h*
in(s) condition-missing		×	in(s) condition
in(s) condition state replacement	×	×	in(s) condition state
not-yet(e) condition-missing		×	not-yet(e) condition
not-yet(e) condition event replacement	×	×	not-yet(e) condition event
exit(s) event-missing		×	exit(s) event
exit(s) event state replacement	×	×	exit(s) event state
entered(s) event-missing		×	entered(s) event
entered(s) event state replacement	×	×	entered(s) event state
broadcasting origin transition replacement	×	×	broadcasting origin transition
broadcasting destination transition replacement	×	×	broadcasting destination transition

Chapter 9

Conclusions and Perspectives

9.1 Conclusions

MBMT has been presented in this thesis as a new mutation testing technique that constructs mutants based on the *model* of SUC, thereby injecting faults into the model and not into the implementation. In contrast to mutation analysis based on the implementation, this technique enables not only the application of mutation analysis when source code of SUC is not available but also the evaluation of the quality of test generation algorithms and test sets used and, at the same time, the detection of non-injected, residual faults in SUC. Thus, the thesis extends the classification of mutants that are *live* and *killed* by additional sub-classes (see Figure 2.6).

Implementation- and specification-based mutation analysis techniques usually need a cluster of operators based on empirical analysis. The approach introduced in this thesis generates mutants using two basic operators: *insertion* and *omission*. Moreover, these basic operators with appropriate iterations and combinations can be used to systematically construct many existing mutation operators. The advantage of the basic operators is that they enable a rigorous systematic approach. Implementation- and specification-based mutation operators from literature are created on an as-needed basis rather than strictly mathematically. The proposed method mutates at the modeling level that focuses on relevant features of SUC.

The syntactical part of the notions of the approach are introduced by means of directed graphs, which are then semantically enriched and practically exemplified

using a collection of graph-based modeling tools, that is, event sequence graphs, finite-state machines, and statecharts. These modeling tools build a hierarchy in terms of their expressive power based on their formal features, which are uniformly represented. This enables a comparison of the mutant generation issues of the basic operators when systematically applied to nodes or arcs of the graphs.

Three case studies were conducted to demonstrate the method, to analyze its characteristics, and to compare the fault detection capability of test sets generated based on different kinds of mutants using several types of applications and corresponding tool support. Significant results of the case studies are:

1. Test sets generated based on mutants created by insertion operators turned out to be considerably more effective in revealing faults in SUC than test sets generated based on mutants constructed by omission operators.

2. Hence, it is strongly recommended that the test generation algorithm used should not only check the given model to validate the legal behavior of SUC under expected circumstances but also its non-existing, forgotten, or omitted elements, for example, arcs and transitions, to check faulty behavior under unexpected circumstances.

3. Mutants that can be easily constructed using the basic operators represent all the fault models, except hierarchy faults in SC. Moreover, one may consider extending the test oracle (predicted outputs) to kill more mutants created by omission operators.

4. The approach detected additional faults in systems that were already released. Even more, case study III contains an application that was tested by a German authority for testing and certification of technical systems and released for public use. Nevertheless, the proposed approach detected several faults in the released version of this system. In case study II and III, the manufacturers were not previously aware of the bugs detected by MBMT.

5. The method can effectively be used to improve test generation for an existing model-based testing process. It turned out that test sets generated based on ESG-mutants using the sI-operator revealed approximately 80% of the

remaining faults detected in each SUC. In conclusion, test generation algorithms for ESG should consider at least covering arcs that are not included within the model to test such illegal behavior of SUC.

9.2 Outlook and Perspectives

Despite the advantages mentioned in the previous section, further research is needed to increase the efficiency of the test algorithms used, especially for analyzing existing empirical approaches to represent their mutation operators by the basic ones introduced in this thesis. In addition, further empirical studies are necessary to consider practical aspects, such as the construction of mutants to represent typical faults in the selection of safety-critical systems. They could form a benchmark for the assessment of such systems. Starting point is given again by existing empirical studies; they need a uniform representation using the basic operators. Apart from these aspects, ongoing research work could include mutation optimization subject to their costs determined by their length, number, etc., and coverage capability of different elements of the models (node coverage, sequence coverage, etc.). Furthermore, an extension of the approach to higher-order mutation could be considered to compare the fault detection capability of test sets and analyze the coupling effect with respect to first-order mutants ([20]), especially using further graphical and algebraic modeling tools.

Bibliography

[1] A. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

[2] H. Agrawal, R. A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, W. Lafayette, Indiana, USA, Mar. 1989.

[3] B. K. Aichernig. *Systematic Black-Box Testing of Computer-Based Systems through Formal Abstraction Techniques*. PhD thesis, Technische Universität Graz, Austria, 2001.

[4] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering (HASE '99)*, pages 239–248. IEEE Computer Society Press, 1999.

[5] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. *International Journal of Reliability, Quality and Safety Engineering*, 8(4):275–299, Dec. 2001.

[6] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM '98)*, pages 46–54. IEEE Computer Society Press, 1998.

[7] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. *SIGSOFT Software Engineering Notes*, 21(3):280–292, 1996.

[8] F. Beidinger, A. Hollmann, M. Kleinselbeck, and W. Ritschel. Vergleich einer graphenbasierten Methode mit der Klassifikationsbaummethode für die Testfallermittlung anhand einer automotiven Fallstudie. In *Software Engineering (Workshops)*, volume 122 of *Lecture Notes in Informatics*, pages 367–374. Gesellschaft für Informatik e.V., 2008.

[9] B. Beizer. *Software Testing Techniques, Second Edition*. Van Nostrand Reinhold, New York, USA, Jun. 1990.

[10] F. Belli. Finite-state testing and analysis of graphical user interfaces. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 34–43. IEEE Computer Society Press, 2001.

[11] F. Belli and M. Beyazit. A formal framework for mutation testing. In *Proceedings of the 4th IEEE International Conference on Secure Software Integration and Reliability Improvement*, pages 121–130. IEEE Computer Society Press, Jun. 2010.

[12] F. Belli, M. Beyazit, A. Hollmann, M. Linschulte, and S. Padberg. Ereignisbasierter Test grafischer Benutzeroberflächen - ein Erfahrungsbericht. *GI-Softwaretechnik-Trends*, 30(2), 2010.

[13] F. Belli and C. J. Budnik. Minimal spanning set for coverage testing of interactive systems. In *Proceedings of the 1st International Colloquium Theoretical Aspects of Computing*, volume 3407 of *Lecture Notes in Computer Science*, pages 220–234. Springer, 2004.

[14] F. Belli and C. J. Budnik. Test minimization for human-computer interaction. *Journal of Applied Intelligence*, 26(2):161–174, 2007.

[15] F. Belli, C. J. Budnik, and A. Hollmann. A holistic approach to testing of interactive systems using statecharts. In *Proceedings of the 2nd South-East European Workshop on Formal Methods (SEEFM'05)*, pages 59–73. South-East European Research Centre (SEERC), 2005.

[16] F. Belli, C. J. Budnik, and A. Hollmann. Holistic testing of interactive systems using statecharts. *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, 1(3):54–64, 2005.

[17] F. Belli, C. J. Budnik, and A. Hollmann. Holistic testing of interactive systems using statecharts. In *Proceedings of the Conference on Sicherheit*, volume 77 of *Lecture Notes in Informatics*, pages 345–356, 2006.

[18] F. Belli, C. J. Budnik, and L. White. Event-based modelling, analysis and testing of user interactions: approach and case study. *Journal of Software Testing, Verification & Reliability*, 16(1):3–32, 2006.

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

[20] F. Belli, N. Güler, A. Hollmann, G. Suna, and E. Yildiz. Model-based higher-order mutation analysis. In *Proceedings of the International Conference on Advanced Software Engineering and Its Applications (ASEA 2010)*, pages 164–173. Springer, Dec. 2010.

[21] F. Belli and A. Hollmann. Holistic testing with basic statecharts. In *Software Engineering 2007 - Beiträge zu den Workshops*, volume 106 of *Lecture Notes in Informatics*, pages 91–100. Gesellschaft für Informatik e.V., 2007.

[22] F. Belli and A. Hollmann. Test generation and minimization with "basic" statecharts. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08)*, pages 718–723. ACM Press, 2008.

[23] F. Belli and A. Hollmann. Ereignisbasiertes Testen eingebetteter Systeme - Vergleich mit Klassifikationsbäumen anhand eines automotiven Beispiels. In *OBJEKTspektrum*. SIGS DATACOM, 2010.

[24] F. Belli, A. Hollmann, and Z. Chen. Mutant-based model-checking to ensure accessibility and safety aspects of human computer interfaces. In *Proceedings of the 2nd International Conference on Information and Communication Technology and Accessibility*, pages 65–74, May 2009.

[25] F. Belli, A. Hollmann, and M. Kleinselbeck. A graph-model-based testing method compared with the classification tree method for test case generation. In *Proceedings of the 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement*, pages 187–194. IEEE Computer Society Press, Jul. 2009.

[26] F. Belli, A. Hollmann, and N. Nissanke. Modeling, analysis and testing of safety issues - an event-based approach and case study. In *Proceedings of the 26th International Conference Computer Safety, Reliability, and Security (SAFECOMP)*, pages 276–282. Springer, 2007.

[27] F. Belli, A. Hollmann, and S. Padberg. Communication sequence graphs for mutation-oriented integration testing. In *Proceedings of the 1st IEEE workshop on Model-Based Verification & Validation*. IEEE Computer Society Press, Jul. 2009.

[28] F. Belli, A. Hollmann, and S. Padberg. *Model-Based Testing for Embedded Systems*, chapter Model-Based Integration Testing with Communication Sequence Graphs. Taylor & Francis, CRC Press, 2011 (to appear).

[29] F. Belli, A. Hollmann, and W. E. Wong. Towards scalable robustness testing. In *Proceedings of the 4th IEEE International Conference on Secure Software Integration and Reliability Improvement*, pages 208–216. IEEE Computer Society Press, Jun. 2010.

[30] F. Belli and M. Linschulte. On negative tests of web applications. *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, 1(5):44–56, 2007.

[31] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[32] P. E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 81–88. IEEE Computer Society Press, 2000.

[33] P. E. Black, V. Okun, and Y. Yesha. Mutation of model checker specifications for test generation and evaluation. *Mutation testing for the new century*, pages 14–20, 2001.

[34] G. V. Bochmann and A. Petrenko. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 109–124. ACM Press, 1994.

[35] J. H. Bowser. Reference manual for Ada mutant operators. Technical Report GIT-SERC-88/02, Georgia Institute of Technology, 1988.

[36] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[37] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[38] T. A. Budd and R. J. Lipton. Proving LISP programs using test data. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 374–403. IEEE Computer Society Press, 1978.

[39] T. A. Budd, R. J. Lipton, R. DeMillo, and F. G. Sayward. The design of a prototype mutation system for program testing. In *Proceedings of the National Computer Conference*, pages 623–627. AFIPS press, 1978.

[40] C. J. Budnik, F. Belli, and A. Hollmann. Structural feature extraction for GUI test enhancement. In *Proceedings of the 1st International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*. IEEE Computer Society Press, Apr. 2009.

[41] U. Buy, A. Orso, and M. Pezzè. Automated testing of classes. *SIGSOFT Software Engineering Notes*, 25(5):39–48, 2000.

[42] T. Y. Chen and M. F. Lau. Test case selection strategies based on boolean specifications. *Journal of Software Testing, Verification & Reliability*, 11(3):165–180, 2001.

[43] Z. Chen and A. Hollmann. Positive and negative testing with mutation-driven model checking. In *GI Jahrestagung (1)*, volume 133 of *Lecture Notes in Informatics*, pages 187–192. Gesellschaft für Informatik e.V., 2008.

[44] B. Choi, R. DeMillo, E. Krauser, R. Martin, A. Mathur, A. Offutt, H. Pan, and E. Spafford. The Mothra tool set. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume 2, pages 275–284. IEEE Computer Society Press, Jan. 1989.

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

[46] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[47] E. M. Clarke. *Model Checking*. MIT Press, Jan. 2000.

[48] F. J. Daniels and K. C. Tai. Measuring the effectiveness of method test sequences derived from sequencing constraints. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 74–83. IEEE Computer Society Press, 1999.

[49] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Proteum - a tool for the assessment of test adequacy for C programs: users guide. Technical Report SERC-TR-168-P, Software Engineering Research Center, Purdue University, W. Lafayette, Indiana, USA, Apr. 1996.

[50] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: an approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 2001.

[51] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[52] R. A. DeMillo and R. J. Martin. The Mothra software testing environment user's manual. Technical Report SERC-TR-4-P, Software Engineering Research Center, Purdue University, W. Lafayette, Indiana, USA, Sep. 1987.

[53] G. Devaraj, M. P. E. Heimdahl, and D. Liang. Coverage-directed test generation with model checkers: challenges and opportunities. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, pages 455–462. IEEE Computer Society Press, 2005.

[54] B. Eggers and F. Belli. Eine Theorie der Analyse und Konstruktion fehlertolerierender Systeme. In *Fehlertolerierende Rechensysteme*, volume 84 of *Informatik-Fachberichte*. Springer, 1984.

[55] S. C. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 220–229. IEEE Computer Society Press, 1994.

[56] S. C. P. F. Fabbri, J. C. Maldonado, and M. E. Delamaro. Proteum/FSM: a tool to support finite state machine validation based on mutation testing. In *Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC '99)*, pages 96–104. IEEE Computer Society Press, 1999.

[57] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. Wong. Mutation testing applied to validate specifications based on Petri nets. In *Proceedings of the IFIP TC6 8th International Conference on Formal Description Techniques*, pages 329–337. Chapman & Hall, Ltd., 1995.

[58] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 210–219. IEEE Computer Society Press, 1999.

[59] J.-C. Fernandez, L. Mounier, and C. Pachon. A model-based approach for robustness testing. In *Proceedings of the 17th IFIP TC6/WG 6.1 International*

Conference Testing of Communicating Systems, Lecture Notes in Computer Science, pages 333–348. Springer, 2005.

[60] C. Fetzer and Z. Xiao. An automated approach to increasing the robustness of C libraries. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 155–166. IEEE Computer Society Press, 2002.

[61] F. Fleurey, B. Baudry, P. A. Muller, and Y. L. Traon. Qualifying input test data for model transformations. *Journal of Software and Systems Modeling*, 8(2):185–203, 2009.

[62] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *Proceedings of 1st International Workshop on Model, Design and Validation*, pages 29–40. IEEE Computer Society Press, Nov. 2004.

[63] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[64] G. Fraser and F. Wotawa. Mutant minimization for model-checker based test-case generation. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 161–168. IEEE Computer Society Press, 2007.

[65] A. Gargantini. Using model checking to generate fault detecting tests. In *Proceedings of the 1st International Conference on Tests And Proofs (TAP)*, volume 4454 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2007.

[66] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Software Engineering Notes*, 24(6):146–162, 1999.

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

[68] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction to the testing and test control notation (TTCN-3). *The International Journal of Computer and Telecommunications Networking*, 42(3):375–403, Jun. 2003.

[69] J. M. Hanks. Testing COBOL programs by mutation. Technical Report GIT-ICS-80/04, Georgia Institute of Technology, Feb. 1980.

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

[71] M. Harman, Y. Jia, and W. B. Langdon. A manifesto for higher order mutation testing. In *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10)*. IEEE Computer Society Press, Apr. 2010.

[72] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 60–70. ACM Press, 2000.

[73] Hella KGaA Hueck & Co. http://www.hella.de.

[74] A. Hollmann, F. Belli, and C. J. Budnik. Test case generation and selection based on statecharts–extension of the holistic approach. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (Student Program Papers)*, 2006.

[75] M. N. Huhns and V. T. Holderfield. Robust software. *IEEE Internet Computing*, 6(2):80–82, 2002.

[76] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, Dec. 1990.

[77] Isik's System for Enterprise-Level Web-Centric Tourist Applications. http://www.iselta.de.

[78] ISO/IEC. Estelle - a formal description technique based on an extended state transition model. ISO 9074. 1989.

[79] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Technical Report TR-09-06, CREST centre, King's College London, 2009.

[80] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009.

[81] K. C. Kang and K.-I. Ko. Formalization and verification of safety properties of statechart specifications. In *Proceedings of the 3rd Asia-Pacific Software Engineering Conference (APSEC '96)*, pages 16–27. IEEE Computer Society Press, 1996.

[82] S.-W. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Journal of Software Testing, Verification & Reliability*, 11(3):207–225, 2001.

[83] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software–Practice and Experience*, 21(7):685–718, 1991.

[84] G. Kovács, Z. Pap, D. L. Viet, A. Wu-Hen-Chang, and G. Csopaki. Applying mutation analysis to SDL specifications. In *SDL Forum*, volume 2708 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2003.

[85] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, pages 230–239. IEEE Computer Society Press, Jun. 1998.

[86] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):247–276, 2005.

[87] S. C. Lee and J. Offutt. Generating test cases for XML-based web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200–209. IEEE Computer Society Press, Nov. 2001.

[88] J. B. Li and J. Miller. Testing the semantics of W3C XML schema. In *Proceedings of the 29th Annual International Computer Software and Applications Conference*, volume 1, pages 443–448. IEEE Computer Society Press, Jul. 2005.

[89] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 352–363. IEEE Computer Society Press, 2002.

[90] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: an automated class mutation system. *Journal of Software Testing, Verification & Reliability*, 15(2):97–133, 2005.

[91] V. Martena, A. Orso, and M. Pezzé. Interclass testing of object oriented software. In *Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, pages 135–144. IEEE Computer Society Press, 2002.

[92] A. M. Memon. An event-flow model of GUI-based applications for testing. *Journal of Software Testing, Verification & Reliability*, 17(3):137–157, 2007.

[93] Mercedes-Benz. http://www.mercedes-benz.de.

[94] L. Morell. Theoretical insights into fault-based testing. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, pages 45–62. IEEE Computer Society Press, Jul. 1988.

[95] J.-M. Mottu, B. Baudry, and Y. L. Traon. Mutation analysis testing for model transformations. In *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 376–390. Springer, 2006.

[96] E. S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: an empirical study. *Journal of Software Testing, Verification & Reliability*, 9(4):205–232, 1999.

[97] A. Mukherjee and D. P. Siewiorek. Measuring software dependability by robustness benchmarking. *IEEE Transactions on Software Engineering*, 23(6):366–378, Jun. 1997.

[98] J. Myhill. Finite automata and the representation of events. Technical Report TR 57-624, Wright Air Devel. Command, 1957.

[99] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 13th International Conference on Software Engineering*, pages 351–360. ACM Press, 2008.

[100] Object Management Group. Unified Modeling Language (UML). http://www.omg.org.

[101] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[102] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[103] A. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Journal of Software Testing, Verification & Reliability*, 13(1):25–53, 2003.

[104] A. J. Offutt, J. Payne, and J. M. Voas. Mutation operators for Ada. Technical Report ISSE-TR-96-09, Department of Information and Software Systems Engineering, George Mason University, Fairfax Virginia, USA, 1996.

[105] J. Offutt, P. Ammann, and L. L. Liu. Mutation testing implements grammar-based testing. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION '06)*. IEEE Computer Society Press, 2006.

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

[107] R. L. Probert and F. Guo. Mutation testing of protocols: principles and preliminary experimental results. In *Proceedings of the IFIP TC6 3rd International Workshop on Protocol Test Systems*, pages 57–76, 1991.

[108] RealNetworks, Inc. http://www.real.com.

[109] G. Reinelt. In *The Traveling Salesman: Computational Solutions for TSP Applications*, volume 840 of *LNCS*. Springer, 1994.

[110] F. Saglietti, N. Oster, and F. Pinte. Interface coverage criteria supporting model-based integration testing. In *Proceedings of the 20th International Conference on Architecture of Computing Systems (Workshop Proceedings)*, pages 85–93. VDE Verlag, 2007.

[111] A. Salomaa and I. N. Sneddon. *Theory of Automata*. Pergamon Press, 1969.

[112] B. Sarikaya. Conformance testing: architectures and test sequences. *Computer Networks and ISDN Systems*, 17(2):111–126, 1989.

[113] Selenium. http://www.seleniumhq.org/.

[114] L. Shan and H. Zhu. Generating structurally complex test cases by data mutation: a case study of testing an automated modelling tool. *The Computer Journal*, 52(5):571–588, 2009.

[115] R. Shehady and D. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing*, pages 80–88. IEEE Computer Society Press, Jun. 1997.

[116] S. D. R. S. D. Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. L. D. Souza. Mutation testing applied to Estelle specifications. *Software Quality Control*, 8(4):285–301, 1999.

[117] T. Sugeta, J. C. Maldonado, and W. E. Wong. Mutation testing applied to validate SDL specifications. In *Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004)*, volume 2978 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2004.

[118] R. D. Tennent. *Specifying Software*. Cambridge University Press, New York, NY, USA, 2001.

[119] M. Trakhtenbrot. New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 151–160. IEEE Computer Society Press, Sep. 2007.

[120] M. Trakhtenbrot. Implementation-oriented mutation testing of statechart models. In *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10)*, pages 120–125. IEEE Computer Society Press, Apr. 2010.

[121] Vector Informatik GmbH. http://www.vector-worldwide.com.

[122] M. von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, pages 128–148. Springer, 1994.

[123] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[124] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 110–121. IEEE Computer Society Press, 2000.

[125] W. E. Wong, A. Restrepo, Y. Qi, and B. Choi. An EFSM-based test generation for validation of SDL specifications. In *Proceedings of the 3rd International Workshop on Automation of Software Test (AST '08)*, pages 25–32. ACM Press, 2008.

[126] W. Xu, J. Offutt, and J. Luo. Testing web services by XML perturbation. In *Proceedings of the 16th International Symposium on Software Reliability Engineering*, pages 257–266. IEEE Computer Society Press, Nov. 2005.

[127] M. Zhang. Eine bibliotheksbasierte Annotierung von Testmodellen zur automatischen Testskriptgenerierung (diploma thesis in German). Technical Report 2010/6, University of Paderborn, Nov. 2010.

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

[129] R. Zirnsak. Eingabedatengenerierung für Positiv- und Negativ-Tests zur Testautomatisierung (diploma thesis in German). Technical Report 2010/2, University of Paderborn, Apr. 2010.

List of Figures

List of Tables

Appendix A

Adaptive Cruise Control (ACC)

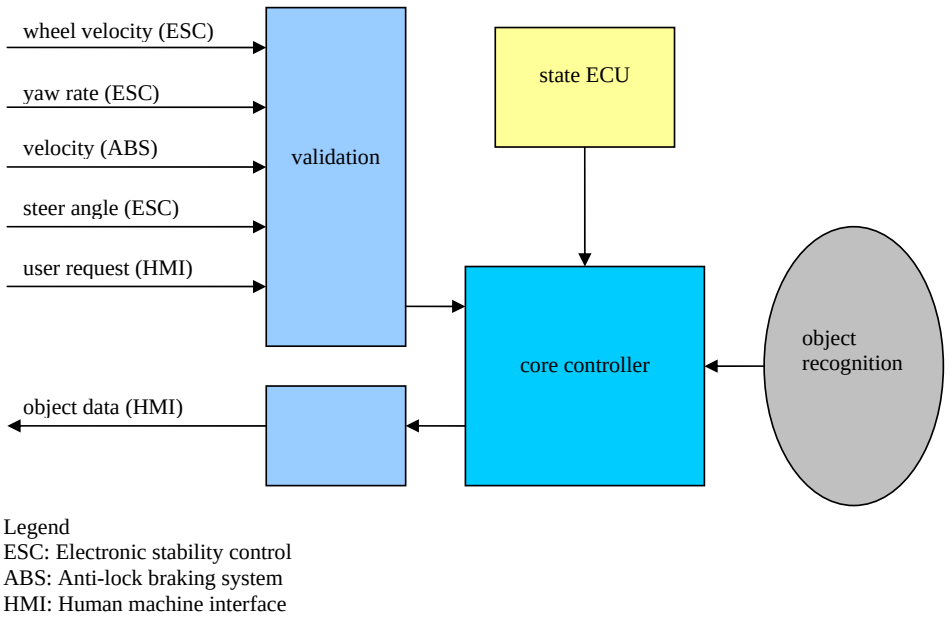


Figure A.1: Relevant signals for ACC

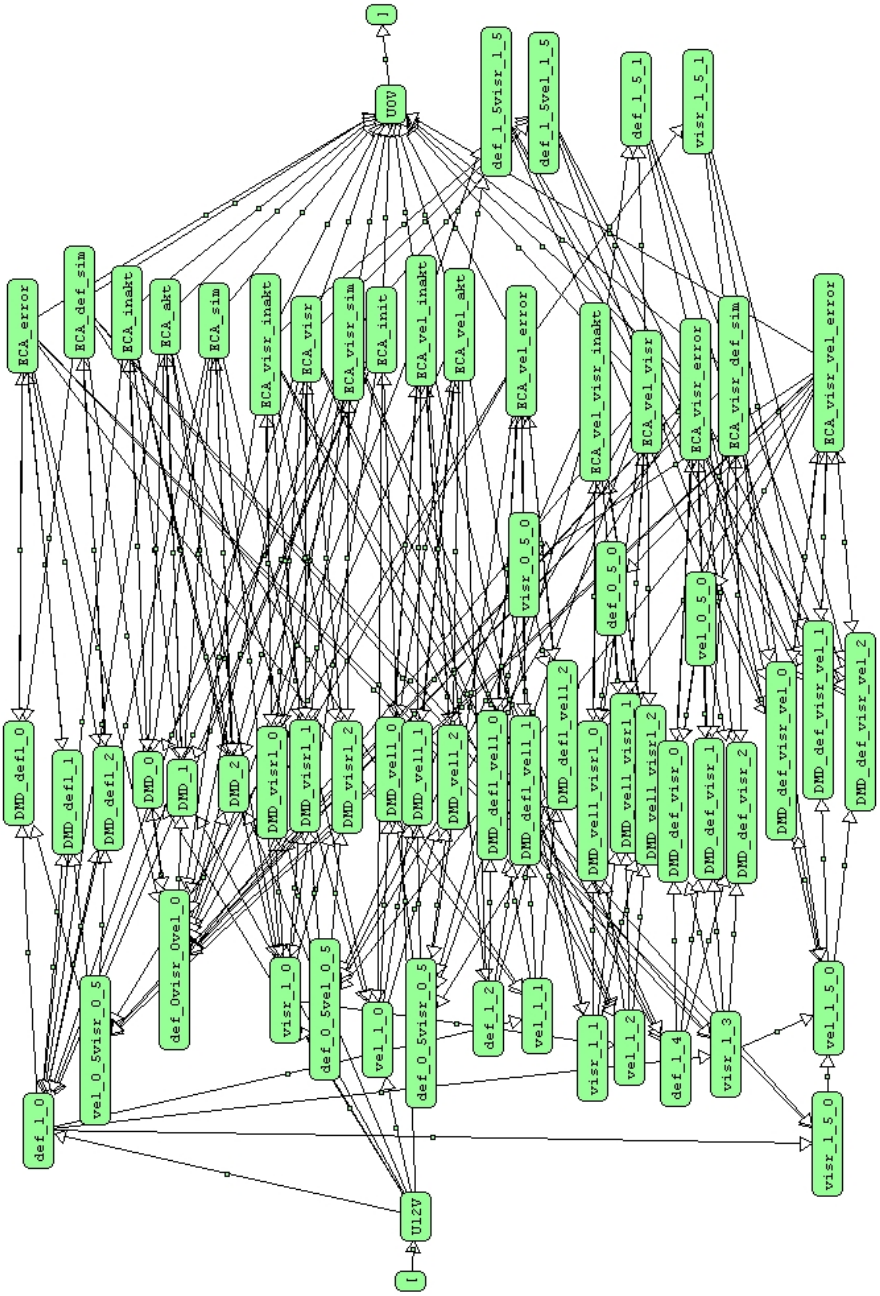
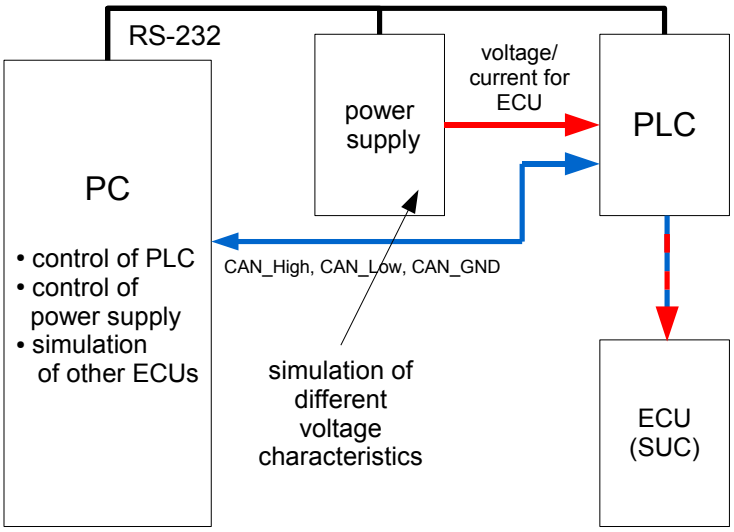


Figure A.2: An ESG model of an extracted ACC



Legend:
PLC: Programmable Logic Controller
PC: Personal Computer
ECU: Electronic Control Unit
RS-232: Serial Interface

Figure A.3: Block diagram of the integration test device

Appendix B

The Control Desk of a Marginal Strip Mower (RSM13)

B.1 Statecharts

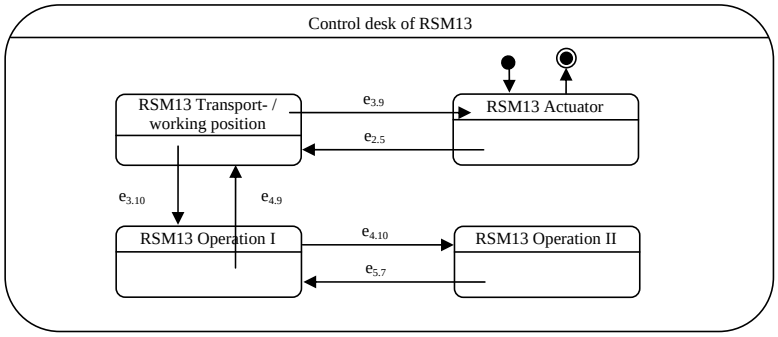


Figure B.1: SC of control desk of RSM13

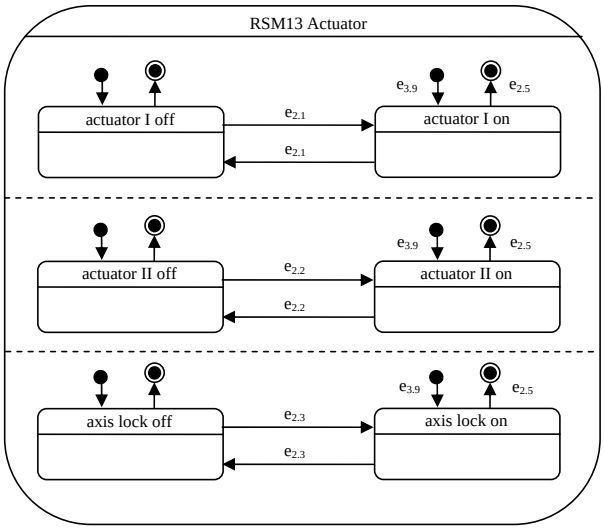


Figure B.2: SC of RSM13 actuator

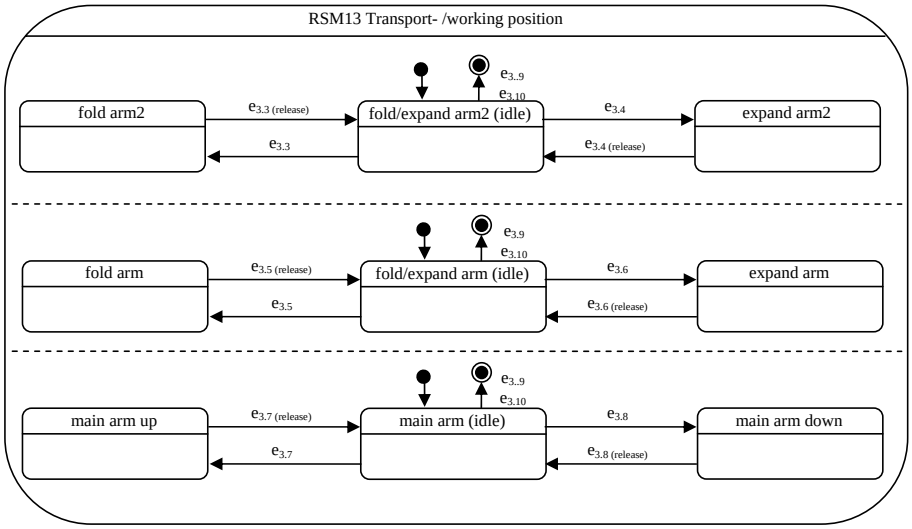


Figure B.3: SC of RSM13 transport-/working position

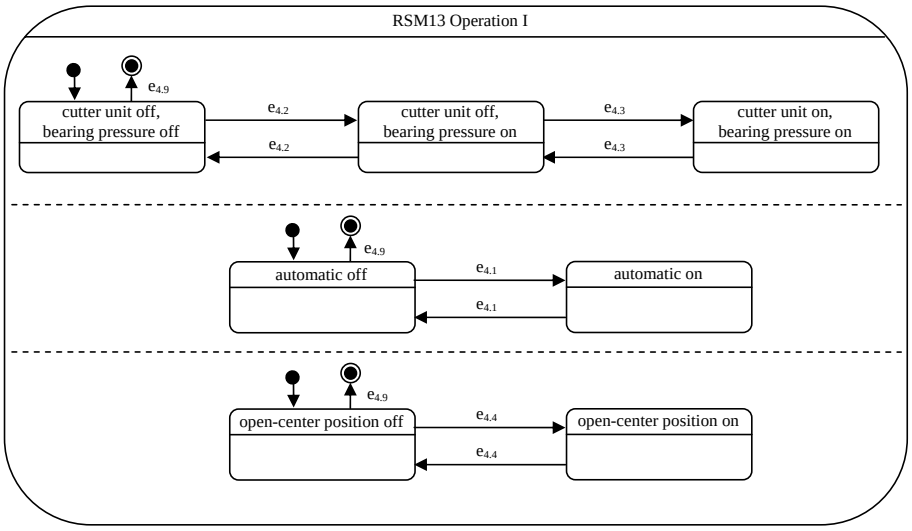


Figure B.4: SC of RSM13 operation I

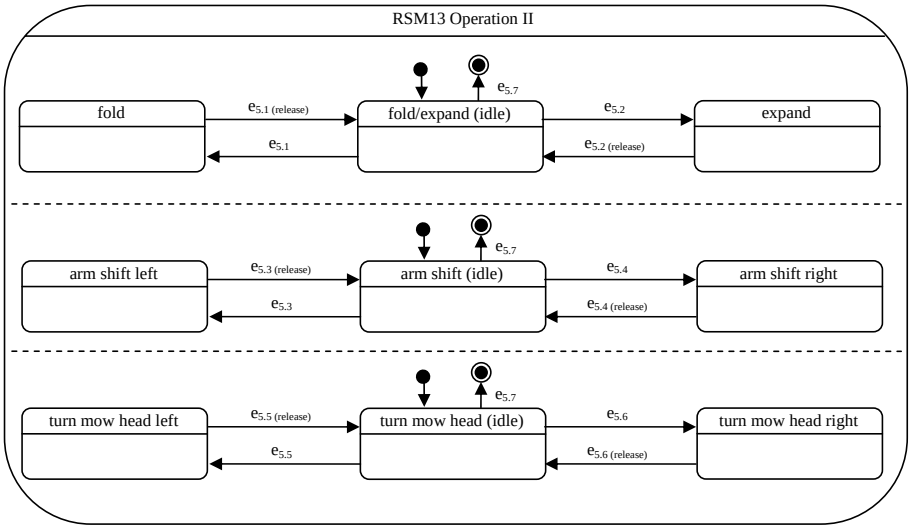


Figure B.5: SC of RSM13 operation II

B.2 ESGs

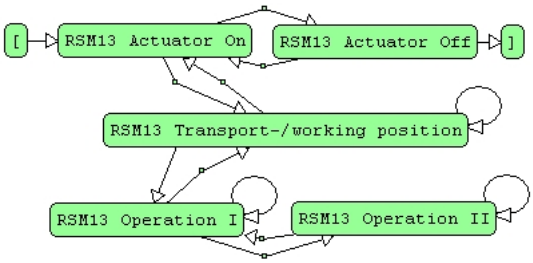


Figure B.6: ESG of control desk of RSM13

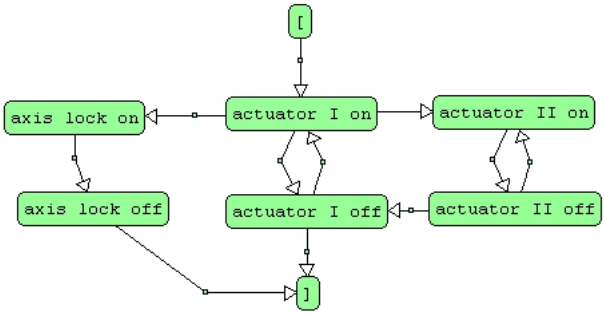


Figure B.7: ESG of RSM13 actuator

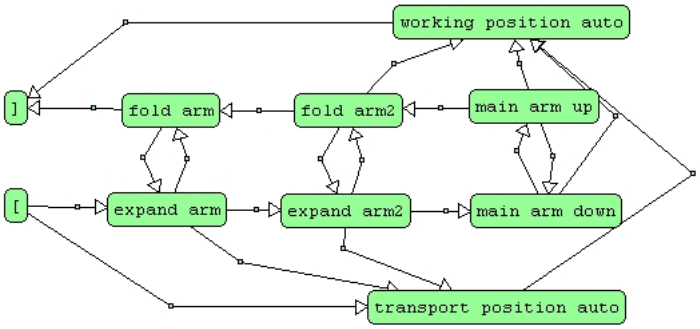


Figure B.8: ESG of RSM13 transport-/working position

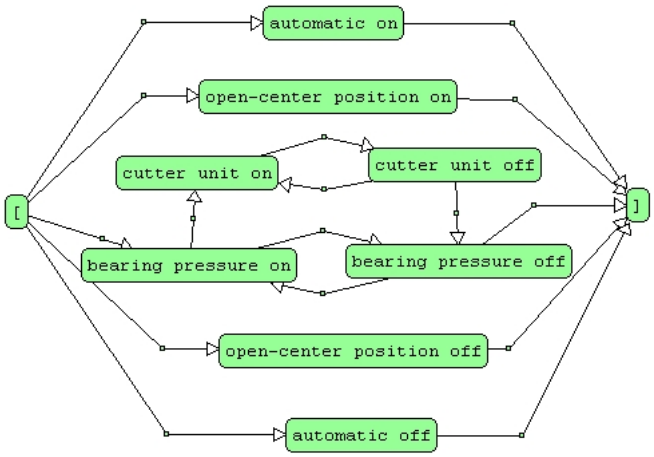


Figure B.9: ESG of RSM13 operation I

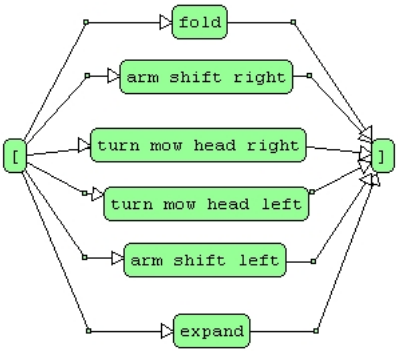


Figure B.10: ESG of RSM13 operation II

Appendix C

ISELTA

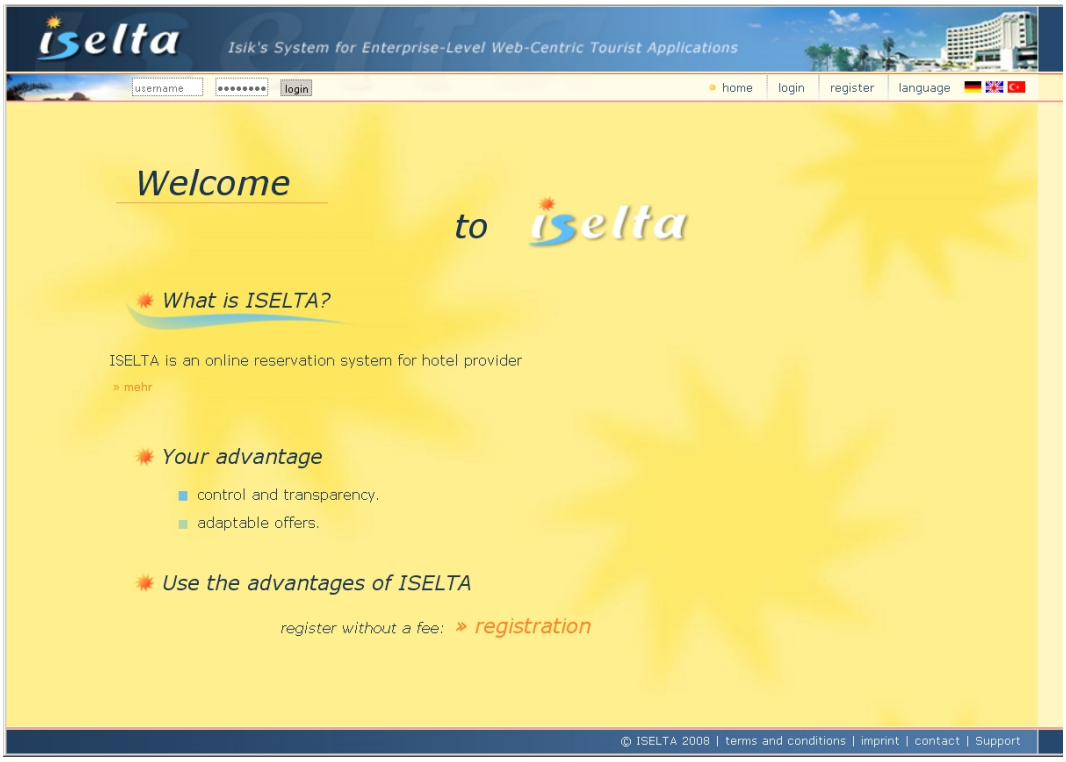


Figure C.1: Entry page of ISELTA ([77])

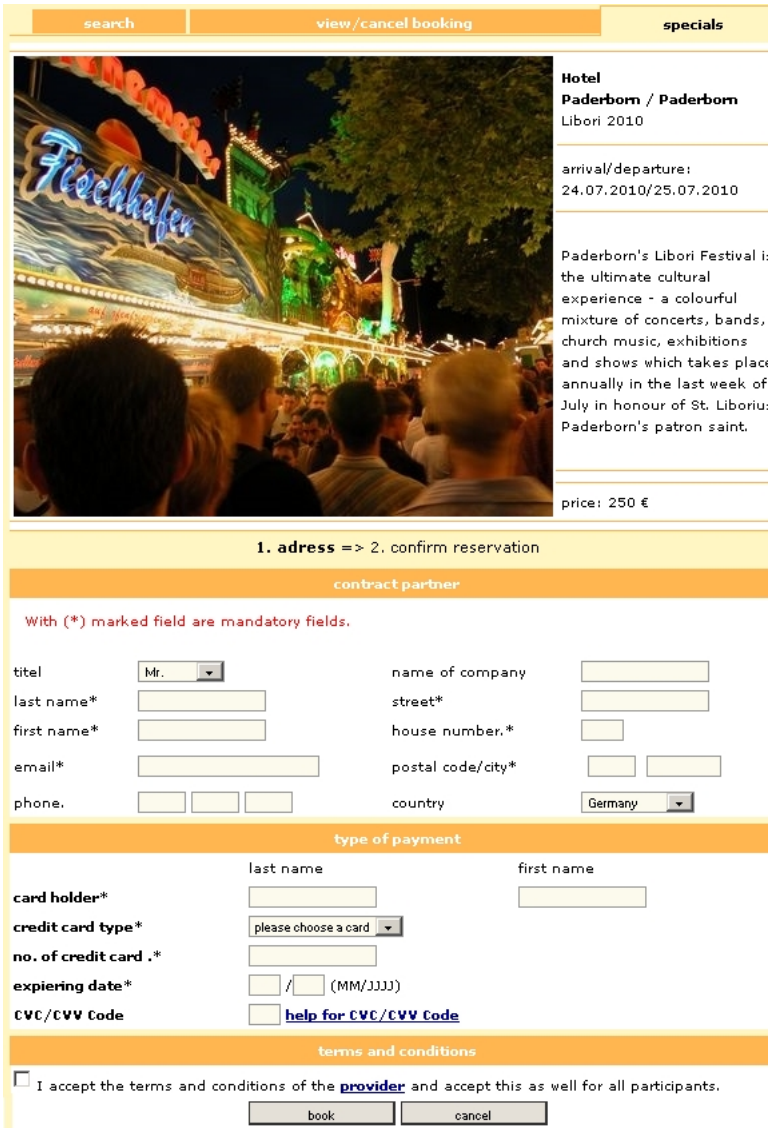


Figure C.2: Booking mask for special offers in ISELTA

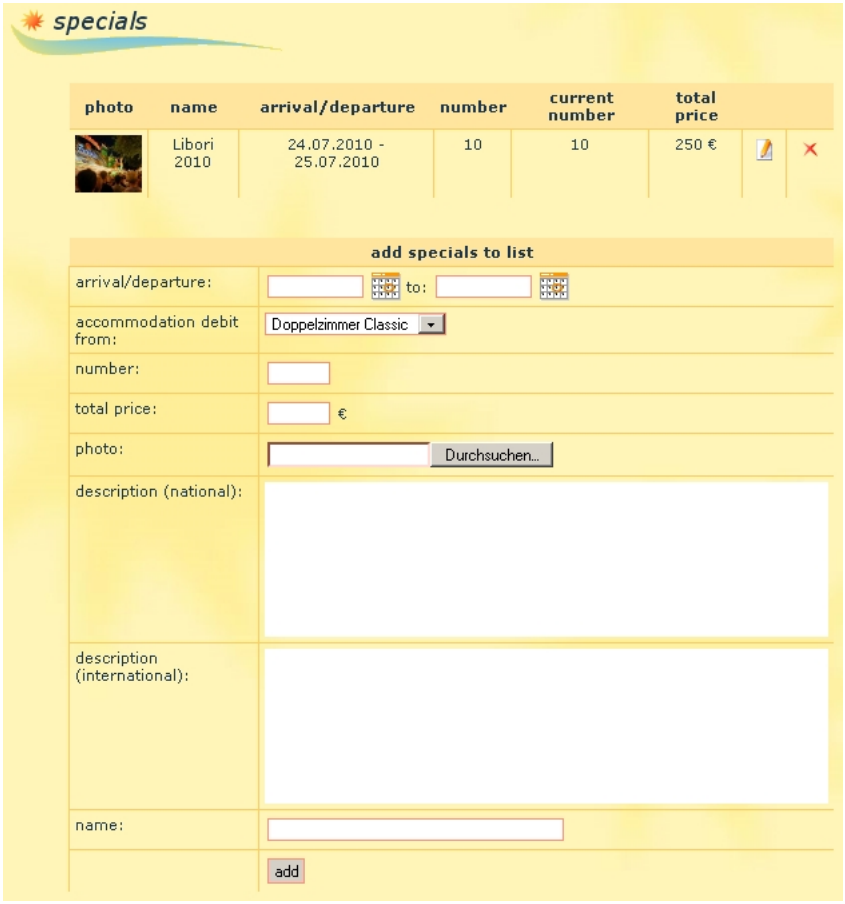


Figure C.3: Website to set up a special offer in ISELTA

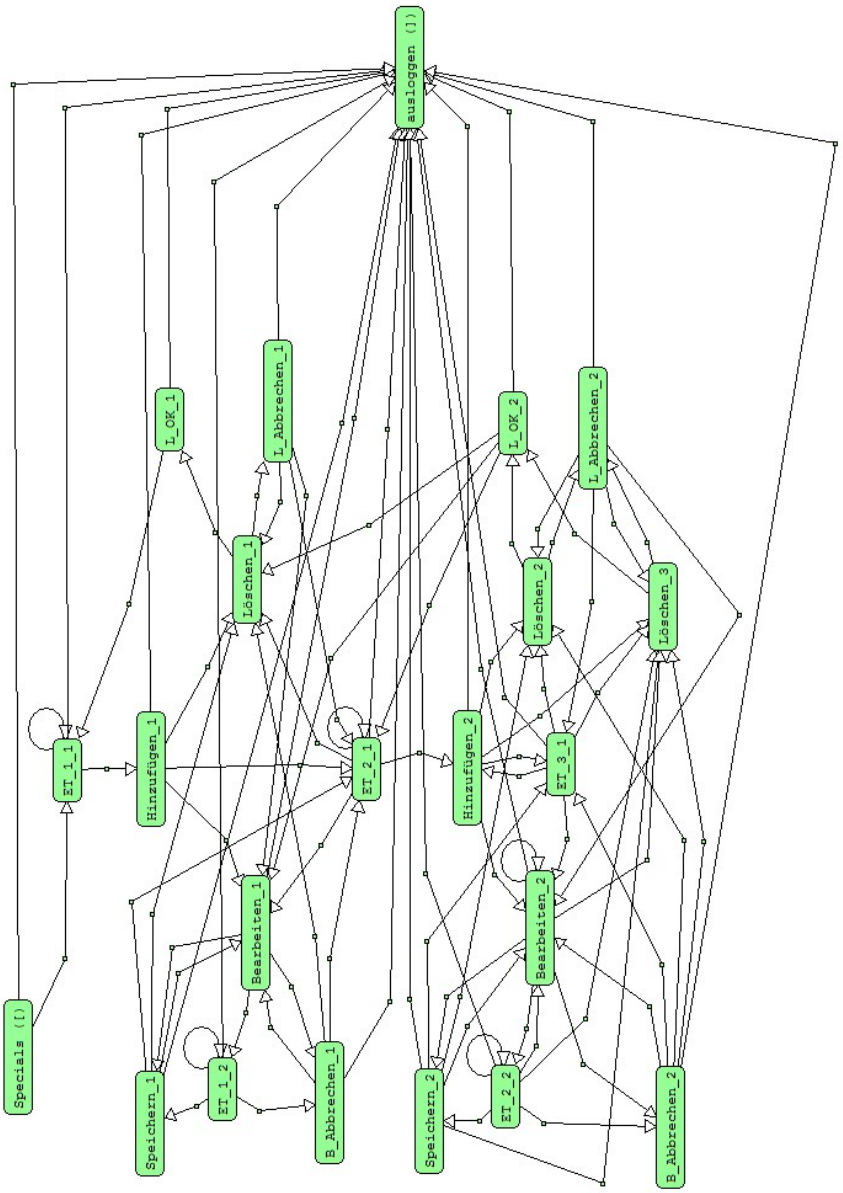


Figure C.4: ESG for setting up a special offer in ISELTA ([129])

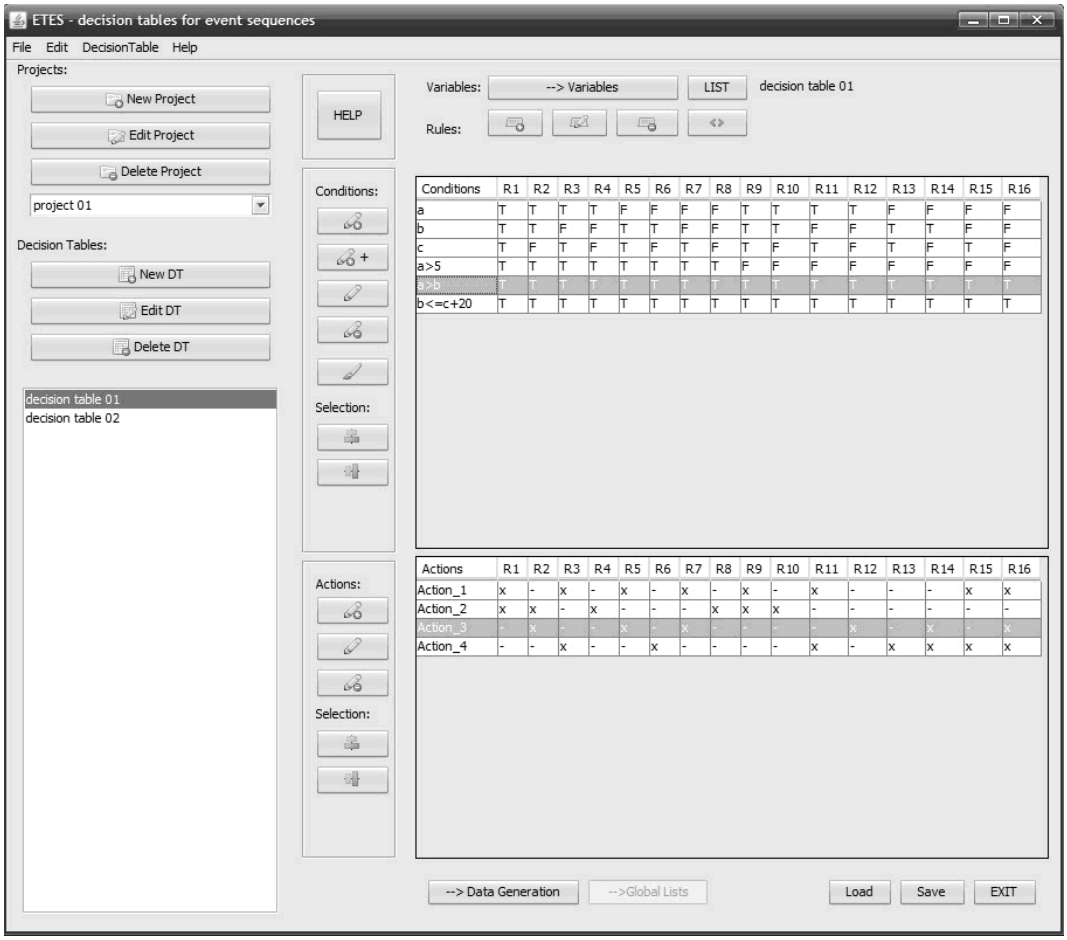


Figure C.5: ETES–decision tables for event sequences ([129])

Appendix D

Thermostat System

To illustrate the notions of $\mathcal{I}$-, $\mathcal{D}$- or $\mathcal{C}$-mutant (Definition 7.12) a thermostat system is used as an analytical case study. It is shown that depending on the type of the mutant the positive or negative property φ_k and φ_k^* (Definition 7.20 and 7.21) must be used to generate counterexamples.

There are some conditions that affect the thermostat's behavior: *SwitchIsOn*, *TooCold*, *TooHot*, and *TempOk*. System behavior is expressed as four modes of operation: *Off*, *Inactive* (the thermostat is regulating the room's air temperature, but the temperature does not need to be adjusted), *Heat* (the thermostat is trying to heat the air), and *AC* (the thermostat is trying to cool the air). Table D.1 shows the NuSMV specification model of the thermostat system. The original specification [7] has been simplified, without changing its functionality. The `case` statements have been translated into a `TRANS` expression.

Example D.1 ($\mathcal{I}$-mutant created by lO-operator) This example shows negative testing for an $\mathcal{I}$-mutant. The statement

$$\alpha_4 := \text{Thermostat} = \text{Inactive} \wedge \text{SwitchIsOn};$$
$$\text{is mutated to}$$
$$\alpha_4^* :=(\text{Thermostat} = \text{Inactive});$$

It is clear that α_4 implies α_4^*. Therefore, the positive property φ_4 is $true$ and $\mathcal{M} \models \varphi_4$. Hence, it requires a negative test case to detect the mutant w.r.t. α_4^*. The

Table D.1: NuSMV model specification of thermostat system

MODULE main

VAR
Thermostat: {Off, Inactive, Heat, AC};
Enuml: {TooCold, TempOk, TooHot};
SwitchIsOn : boolean;

ASSIGN
INIT(Thermostat=Off & !SwitchIsOn)

TRANS(
α_1 & next(β_1) | α_2 & next(β_2) | α_3 & next(β_3) | α_4 & next(β_4) |
α_5 & next(β_5) | α_6 & next(β_6) | α_7 & next(β_7) | α_8 & next(β_8) |
α_9 & next(β_9) | α_{10} & next(β_{10}) | deadlock & next(Thermostat)=Thermostat &
next(Enuml)=Enuml & next(SwitchIsOn)=SwitchIsOn)

DEFINE
α_1:= Thermostat=Off & !SwitchIsOn;
β_1:= SwitchIsOn & Enuml=TempOk & Thermostat=Inactive;
α_2:= Thermostat=Off & !SwitchIsOn;
β_2:= SwitchIsOn & Enuml=TooCold & Thermostat=Heat;
α_3:= Thermostat=Off & !SwitchIsOn;
β_3:= SwitchIsOn & Enuml=TooHot & Thermostat=AC;
α_4:= Thermostat=Inactive & SwitchIsOn;
β_4:= !SwitchIsOn & Thermostat=Off;
α_5:= Thermostat=Inactive & !(Enuml=TooCold);
β_5:= Enuml=TooCold & Thermostat=Heat;
α_6:= Thermostat=Inactive & !(Enuml=TooHot);
β_6:= Enuml=TooHot & Thermostat=AC;
α_7:= Thermostat=Heat & SwitchIsOn;
β_7:= !SwitchIsOn & Thermostat=Off;
α_8:= Thermostat=Heat & !(Enuml=TempOk);
β_8:= Enuml=TempOk & Thermostat=Inactive;
α_9:= Thermostat=AC & SwitchIsOn;
β_9:= !SwitchIsOn & Thermostat=Off;
α_{10}:= Thermostat=AC & !(Enuml=TempOk);
β_{10}:= Enuml=TempOk & Thermostat=Inactive;
deadlock:=!($\alpha_1|\alpha_2|\alpha_3|\alpha_4|\alpha_5|\alpha_6|\alpha_7|\alpha_8|\alpha_9|\alpha_{10}$);

negative property must be set to $\varphi_4^* := G\neg(\mathcal{T}_4^* \wedge \neg\mathcal{T}_4 \wedge \neg\mathcal{T}/4)$. Model checking yields the following counterexample (test case), that is, $\mathcal{M}_4^* \not\models \varphi_4^*$.
s_1: (Thermostat = Off, Enuml = TooHot, SwitchIsOn = 0)
s_2: (Thermostat = Inactive, Enuml = TempOk, SwitchIsOn = 1)
s_3: (Thermostat = Heat, Enuml = TooCold, SwitchIsOn = 0)
s_4: (Thermostat = Inactive, Enuml = TempOk)
s_5: (Thermostat = Off, Enuml = TooHot)
s_6: (Thermostat = Inactive, Enuml = TempOk, SwitchIsOn = 1)
s_7: (Thermostat = Off, Enuml = TooHot, SwitchIsOn = 0)

Example D.2 ($\mathcal{D}$-mutant created by lI-operator) This example shows positive testing for a $\mathcal{D}$-mutant. The statement

$$\alpha_5 := \text{Thermostat} = \text{Inactive} \wedge \neg(\text{Enuml} = \text{TooCold});$$
$$\text{is mutated to}$$
$$\alpha_5^* := (\text{Thermostat} = \text{Inactive} \wedge \neg\ (\text{Enuml} = \text{TooCold}) \wedge \text{SwitchIsOn});$$

Obviously, α_5^* implies α_5. Therefore, φ_5^* is $true$ and $\mathcal{M}_5^* \models \varphi_5^*$. Hence, it requires a positive test case to detect the mutant w.r.t. α_5^*. Therefore, the positive property is set to $\varphi_5 := G\neg(\mathcal{T}_5 \wedge \neg\mathcal{T}_5^* \wedge \neg\mathcal{T}/5)$. Model checking yields the following counterexample, that is, $\mathcal{M} \not\models \varphi_5$.
s_1: (Thermostat = Off, Enuml = TooHot, SwitchIsOn = 0)
s_2: (Thermostat = Inactive, Enuml = TempOk, SwitchIsOn = 1)
s_3: (Thermostat = Heat, Enuml = TooCold, SwitchIsOn = 0)
s_4: (Thermostat = Inactive, Enuml = TempOk)
s_5: (Thermostat = Heat, Enuml = TooCold)
s_6: (Thermostat = Inactive, Enuml = TempOk)
s_7: (Thermostat = AC, Enuml = TooHot)
s_8: (Thermostat = Inactive, Enuml = TempOk, SwitchIsOn = 1)
s_9: (Thermostat = Heat, Enuml = TooCold, SwitchIsOn = 0)

Example D.3 ($\mathcal{D}$-mutant created by lC-operator) This example shows testing for a $\mathcal{C}$-mutant. The statement

$$\alpha_9 := \text{Thermostat} = \text{AC} \wedge \text{SwitchIsOn};$$
is changed to
$$\alpha_9^* := (\text{Thermostat} = \text{Heat} \wedge \text{SwitchIsOn});$$

The positive property is set to $\varphi_9 := G\neg(\mathcal{T}_9 \wedge \neg\mathcal{T}_9^* \wedge \neg\mathcal{T}/9)$. Model checking results in the following counterexample ($\mathcal{M} \not\models \varphi_9$).
s_1: (Thermostat = Off, Enuml = TooHot, SwitchIsOn = 0)
s_2: (Thermostat = Inactive, Enuml = TempOk, SwitchIsOn = 1)
s_3: (Thermostat = AC, Enuml = TooHot)
s_4: (Thermostat = Off, SwitchIsOn = 0)
s_5: (Thermostat = Inactive, Enuml = TempOk, SwitchIsOn = 1)
s_6: (Thermostat = Off, Enuml = TooHot, SwitchIsOn = 0)

The negative property is set to $\varphi_9^* := G\neg(\mathcal{T}_9^* \wedge \neg\mathcal{T}_9 \wedge \neg\mathcal{T}/9)$. Model checking yields $\mathcal{M}_9^* \models \varphi_9^*$. The statement α_9^* is a $\mathcal{C}$-mutated transition, but it creates a $\mathcal{D}$-mutant, because its increscent mutated behavior is masked by other transitions. A weaker property of $\varphi_9^* := G\neg(\mathcal{T}_9^* \wedge \neg\mathcal{T}_9)$ would produce the following test case.
s_1: (Thermostat = Off, Enuml = TooHot, SwitchIsOn = 0)
s_2: (Thermostat = Heat, Enuml = TooCold, SwitchIsOn = 1)
s_3: (Thermostat = Off, Enuml = TooHot, SwitchIsOn = 0)
s_4: (Thermostat = Inactive, Enuml = TempOk, SwitchIsOn = 1)
s_5: (Thermostat = Off, Enuml = TooHot, SwitchIsOn = 0)

However, this is a wrong test case. It passes in the original model $\mathcal{M}$, because the transition (s_4, s_5) can be generated by $\mathcal{T}_4 = \alpha_4 \wedge X(\beta_4)$. Therefore, the condition $\neg\mathcal{T}/9$ is necessary for the positive and negative property (Definition 7.20 and 7.21).