UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

FAKULTÄT FÜR
ELEKTROTECHNIK,
INFORMATIK UND
MATHEMATIK

On the Role of Test Sequence Length, Model Refinement, and Test Coverage for Reliability

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.-Wirt.-Inf. Michael Linschulte

Erster Gutachter:	Prof. Dr.-Ing. Fevzi Belli
Zweiter Gutachter:	Prof. Dr. rer. nat. Leena Suhl

Tag der mündlichen Prüfung: 15.04.2013

Paderborn 2013

Diss. EIM-E/289

To my family

Acknowledgements

First of all, I would like to thank Prof. Fevzi Belli for encouraging me to write this thesis and his support to start and conclude my thesis. I also thank Prof. Suhl for her advices. Furthermore, I thank my family for their moral support throughout these years. My thanks go also to Prof. Tugkan Tuglular (Izmir Institute of Technology), Prof. Harald Stieber (University of Applied Sciences Nuremberg), Prof. Hakki Toroslu (Middle East Technical University), and Prof. Bekir Taner Dincer (University of Mugla) for fruitful discussions and comments that helped to shape this thesis. Finally, I would like to offer my special thanks to Dr. Axel Hollmann, Dr. André Endo and Dr. Nevin Güler for their close collaboration.

Contents

Symbols and Notation

Set Theory

$\{x_1, x_2, \ldots, x_n\}$	set of elements $x_1, x_2, \ldots, x_n$
$\emptyset$	the empty set
$x \in M$	x is an element of set M
$x \notin M$	x is not an element of set M
$M \subseteq N$	M is a subset of N or even equals N
$M \subset N$	M is a subset of N
$M \cup N$	union of sets M and N
$M \cap N$	intersection of sets M and N
$M \setminus N$	set subtraction
$M \times N$	Cartesian product of sets M and N
$\|M\|$	cardinality of set M
$\mathcal{P}(M)$	power set of M

Basic Sets

C	a finite set of constraints
E	a finite set of edges (ESG) or a finite set of events (DT)
$N^-(v)$	set of predecessors of vertex v

$N^+(v)$	set of successors of vertex v
R	a finite set of rules
V	a finite set of labeled vertices
Ξ	set of entry nodes
Γ	set of exit nodes

Miscellaneous

$\overline{ESG}$	inversion of an ESG
$\widehat{ESG}$	completion of an ESG
$\digamma$	faulty event
$\alpha(ES)$	the initial vertex of a given event sequence
$\delta^-(v)$	indegree of vertex v
$\delta^+(v)$	outdegree of vertex v
$l(ES)$	length of a given event sequence
$\omega(ES)$	the last vertex of a given event sequence
$s \oplus t$	concatenation of two sequences s and t
$s\|\|t$	parallel execution of two sequences s and t
$\langle x_1, x_2, \ldots, x_n \rangle$	finite sequence $x_1, x_2, \ldots, x_n$
$\forall$	for all

Reliability Theory

AIC	Akaike information criterion
BIC	Bayesian information criterion
D	Duane reliability model

D-S	delayed S-shaped reliability model
G-O	Goel-Okumoto reliability model
GGO	generalized Goel-Okumoto reliability model
GoF	goodness of fit
HPP	homogenous Poisson process
IE	impact of each component
K-S	one-sample Kolmogorov-Smirnov test
LLF	log-likelihood function
LP	log-power reliability model
M-O	Musa-Okumoto reliability model
MLE	maximum likelihood estimation
MSE	mean square error
NHPP	non-homogenous Poisson process
R_c	combined reliability
RE	reliability of each component
SRGM	software reliability growth model
UR	usage ratio of each component

Abbreviations

CES	complete event sequence
CPP	Chinese postman problem
CSP	constraint satisfaction problem
DbC	design by contract
DT	decision table

EFG	event flow graph
EP	event pair
ES	event sequence
ESB	enterprise service bus
ESG	event sequence graph
ESG4WSC	ESG for web service compositions
FCES	faulty complete event sequence
FEP	faulty event pair
FES	faulty event sequence
FR	full resolution (testing)
FSM	finite state machine
GUI	graphical user interface
LC	layer-centric (testing)
LC4WSC	layer-centric testing for web service compositions
MBT	model-based testing
PES	partial event sequence
PriFES	private faulty event sequence
PubFES	public faulty event sequence
SLC	selective layer-centric testing
SLC4WSC	SLC for web service compositions
SOA	service-oriented architecture
SR	software reliability
SUC	system under consideration
TSD	Test Suite Designer

TSD4WSC	TSD for web service compositions
TSP	traveling salesman problem
WSC	web service composition

Part I

Preliminaries

Chapter 1

Introduction

Testing is one of the traditional analysis techniques of quality assurance in the software industry [7, 3] in which the software product itself is executed and evaluated, commonly in an environment closely resembling its target environment. The cost of industrial testing may range from 30% up to 60% of the overall development costs [117]; some software developers even spend as much as 80% of their development budget on testing [87]. The aim of this thesis is to considerably reduce these costs by introducing new techniques.

Testing is user-centric since it checks whether the software or *system under consideration* (SUC) does what it is expected to do (*positive testing*) or does not do what it is not expected to do (*negative testing*). However, since all possible tests can potentially be infinite in number, there is no justification for any assessment of the correctness of SUC based on the success or failure of a single test. To overcome this principle shortcoming of testing, which concerns completeness of validation, formal methods have been proposed that *model* the relevant, desirable features of SUC, which has led to *model-based testing* (MBT). MBT does not require the availability of the SUC's source code for test generation—a feature that makes MBT very attractive to the industry since the majority of vendors prefer not to make their products' source codes available.

Once the model is established, one can generate and select *test cases* that are ordered pairs of test inputs and expected test outputs for both positive and negative testing. Such a collection of test cases is commonly referred to as a *test suite*. A

coverage-oriented adequacy criterion [3, 71] is usually used to ascribe a measure to a test suite's effectiveness in revealing faults and to determine the point in time when to terminate the test process. This criterion calculates the ratio of the portion of the selected components of the model (or code) that is covered by the given test suite to the uncovered portion; the higher the degree of test coverage, the lower the risk of having critical software artifacts that have not been sifted through.

The approach proposed in this thesis is model-based and coverage-oriented. The distinction between the correct and faulty functioning of SUC is referred to as the *oracle problem* in literature [8, 71]. In this regard, this thesis represents the behavior of SUC in interacting with the user's actions by means of directed graphs—more precisely, as *event sequence graphs* (ESG) [9, 14] with the vertices representing events and the arcs representing sequences of events. The test is *successful* if and only if a final event can be reached; otherwise the SUC *fails* the test. This view suggests that the model is correct because, for example, it has been validated before the test process is launched; that is, the rules of the game have been clearly stated concerning what is right and what is wrong.

From a knowledge engineering point of view, testing is considered a planning problem that can be solved using a goal-driven strategy [104] such that, given a set of operators, an initial state, and a goal state, the planner is expected to produce a sequence of events by means of which the SUC can run from the initial state to the goal state, producing in the end the final event the user desires as a system responsibility or service. Considering the testing problem described above, this means that an appropriate *test sequence* needs to be constructed upon both the correct inputs to reach a desirable final event (for positive testing), and the faulty inputs to reach an undesirable final event (for negative testing).

Having rudimentarily defined the elementary notions, the test coverage problem now becomes finding a test sequence with the shortest walk that visits each arc at least once in a given graphic model. In this context, the underlying optimization problem is a generalization of the *Chinese postman problem* (CPP) [12]. CPP entails solving the *assignment problem* [30] that deals with the question of how to assign n items (agents) to n other items (tasks), incurring some cost that may vary, depending on the agent-task assignment. However, algorithms need to be constructed to satisfy not only the constraint of a minimum total length of test sequences, but also to cover

all n-tuples (or *sequences of length* n), that is, pairs, triples, quadruples, etc., of events represented graphically. This brings about a substantial improvement in the scalability and solves the test termination problem and thus constitutes one of the benefits of the proposed approach.

Another challenge of testing stems from the fact that nowadays software products are becoming increasingly larger and more and more complex. Therefore, they are not designed and implemented in one single step, but rather in a series of consecutive steps on the basis of the well-known "divide and conquer" principle, leading to a hierarchy of models in several layers. Nevertheless, for generating test sequences, the components, which are refined in lower layers, have to be completely resolved, that is, composed in the top-level layer, leading to a *fully resolved* (FR) model. It is evident that the test generation effort mounts with the increasing size of the model and number of hierarchy layers, which induce high test costs. From our group's previous work, we know that the run-time complexity of finding a minimal test suite for covering each arc is $O(|V|^3)$ [12], where $|V|$ denotes the number of vertices. The run-time complexity for covering sequences of higher length gets even worse. Depending on the chosen sequence length n to be covered, graphs with up to $(|V|^{(n-1)} * (n-1))$ vertices in the worst case have to be covered [12]. As an example, in Figure 1.1 the component represented by vertex *menu* is refined. Figure 1.2 represents the FR model that consists of 14 vertices and 65 arcs. However, calculating a minimal coverage of all sequences of length 5 requires a graph with 1711 vertices and 8516 arcs [12]. This explosive increase of the calculations is the first problem for which this thesis suggests a solution, which is constructed based on a variation of the assignment problem for a *layer-centric* construction of test suites.

A widespread and popular belief in testing is "the longer, the better." This implies test suites covering longer event sequences have the power to detect subtle faults that are hard to find since they only occur in specific contexts. The challenge is then to cope with the commonly exponential increase in the size of the resulting test suites. This is the second problem for which this thesis suggests a solution: combining the layer-centric approach with techniques of software reliability engineering [70].

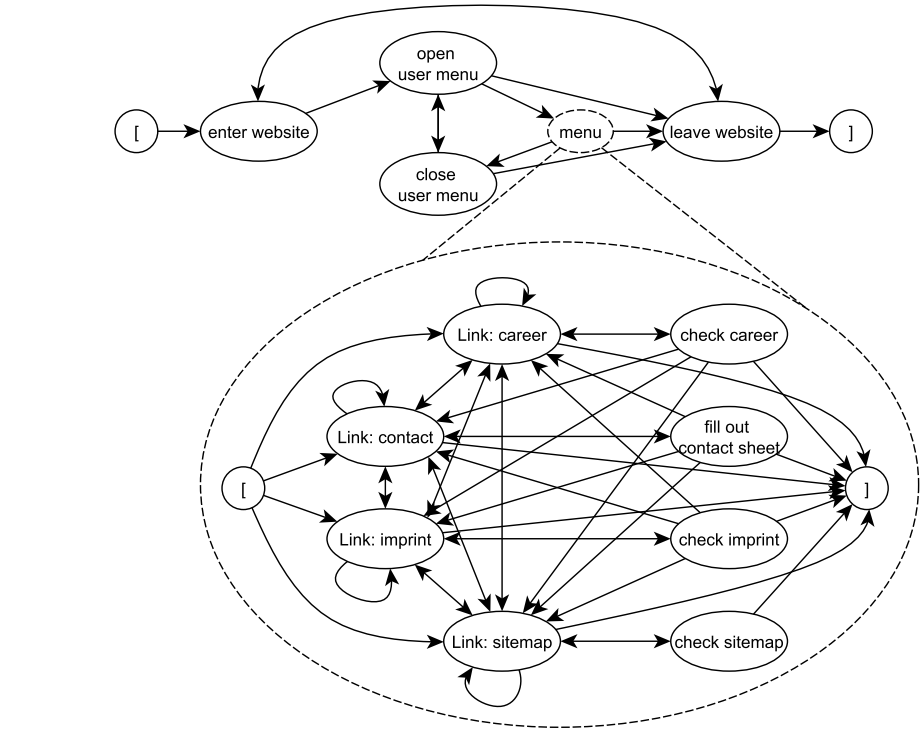


Figure 1.1: A compound vertex *menu* of the model is refined

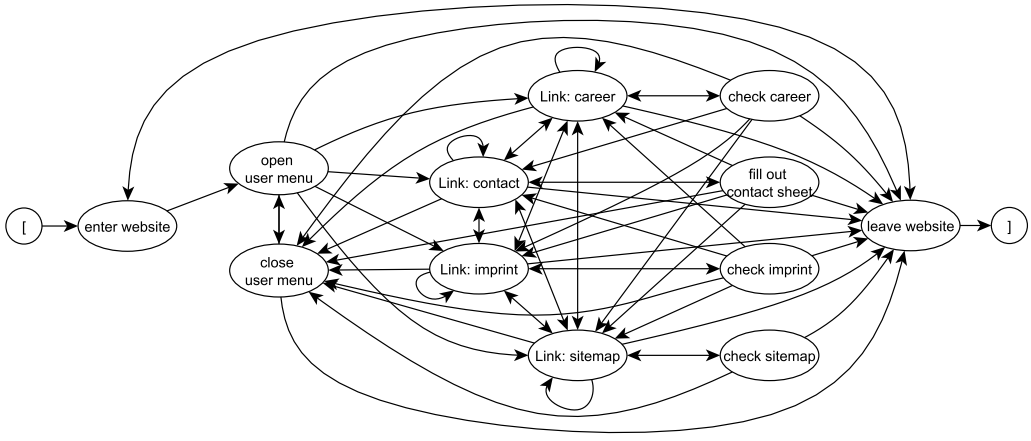


Figure 1.2: Completed (fully resolved) version of the model given in Figure 1.1

1.1 Major Contributions of the Thesis

To sum up, this thesis addresses the following questions:

1. How can the effort of test generation as well as the excessive number of test cases, and thus test costs, be reduced if the modeling process produces a large hierarchical set of models?

2. How can components (in terms of their models) be identified and selected for intensive testing that are likely to provide a better chance of detecting additional "attractive" faults than others?

3. What is the impact and trade-off of this selection process on the overall system reliability?

To answer the first question, this thesis suggests a new strategy called *layer-centric testing* (LC) for test generation and minimization of test suites based on hierarchical models. To answer the second question, a reliability theoretical approach, called *selective layer-centric testing* (SLC), identifies critical components of the SUC that most endanger the overall reliability. Answering the third question requires the comparison of the reliability level achieved by following the new LC/SLC strategy with the one achieved by following the conventional strategy which is based on the fully resolved model.

The cost saving effect of this approach has been evaluated and analyzed in two case studies drawn from completely different domains. The experiments give reasons to claim that the techniques introduced can contribute to an immense reduction in test costs.

1.2 Outline

The present thesis is organized in four parts. Part I summarizes the relevant background and related work before Part II introduces and evaluates the new approach. Part III applies the approach to a different domain to demonstrate its versatility, flexibility, and extendibility. Perspectives for applicability of the approach and hints to future research form the content of Part IV.

In Part I, Chapter 2 summarizes related work on model-based testing techniques, test adequacy, and optimization techniques. Related work on test sequence length, model refinement, and software reliability completes this summary. Chapter 3 introduces the terminology and notions used in the thesis, such as the ESG model for representing system behavior and the corresponding test process. Also the optimization of test generation based on ESG model is described in this chapter.

Beginning Part II, Chapter 4 describes the new LC approach in detail and reviews existing reliability models from which those best suited for the comparison of LC and FR strategies are to be selected. Chapter 5 describes the strategy for selecting a fault-sensitive subset of models and generating tests for them. Chapter 6 concludes Part II by validating the approach and determining its characteristic features in the first case study of the thesis focused on an application drawn from a commercial web-based software system that was developed by our department.

Part III applies the approach to web service compositions. Chapter 7 introduces relevant background and adapts the ESG model to this new domain. Chapter 8 validates the adapted approach and determines its characteristic features in the second case study based on different scenarios of a non-trivial application.

In Part IV, Chapter 9 sketches an approach for semi-automatic detection and correction of boundary overflow faults. Also, this approach has been validated by means of a (third) case study. Furthermore, Chapter 9 presents a technique that combines the merits of positive and negative test techniques leading to a considerable cost reduction. Chapter 10 concludes the thesis by summarizing its significant results and further research planned.

Chapter 2

Related Work

Numerous monographs are dedicated to software testing; e.g., Mathur systematically reviews and presents common existing knowledge [71] whereas Binder summarizes relevant techniques for testing object-oriented systems [26]. The books of Myers and Beizer are well-known as well [84, 85, 7]. A common problem that is described in all the books is the derivation of meaningful test cases. Very often, the usage of models is suggested to fill this gap [117].

A broad variety of formal and informal models exist for testing software as recommended in de-facto standards, such as UML [93] or TTCN-3 [49]. 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 determine the level of granularity of the SUC description and can roughly be divided into *state-based* and *event-based*.

2.1 State-based vs. Event-based Models

State-based models [3, 71] have been in use for a long time, e.g., for conformance testing [28, 105] as well as for specification and testing of system behavior [96, 108, 123]. One of the earliest models based on *finite state machines* (FSM) is described by Chow [35]. A variation of FSMs is given by Petri nets [37]. In addition to state-based models, event-based models have been introduced, e.g., using *event-flow graphs* (EFG) [76] and, in a broader sense, *event sequence graphs* (ESG) [9].

Although nodes are interpreted in both models as operations of an event set [11], EFGs are primarily designed for modeling *graphical user interfaces* (GUI). Therefore, the nodes are enriched by semantics specific to GUIs. However, every EFG can be transformed to an equivalent ESG by taking away the additional semantics used to differentiate the GUI events, and any ESG can be transferred to an equivalent EFG by inclusion of the required semantics [9, 76, 11]. A further difference is that ESGs enable a complementary view, which is necessary to model potential user errors [44, 63].

2.2 Test Adequacy and Test Sequence Length

A common problem of model-based testing is that a very large number of test cases can be derived from a model. This requires a stopping rule for testing, known as test adequacy criterion, which can also be used to determine the "thoroughness" of the testing process [71]. Apart from several model-specific test selection criteria [43], well-known criteria for graph-based models are all-nodes and all-edges [132]. Also, the sequence "length" to be covered has to be taken into account [9, 76]. Arcuri investigated the role played by the length of test sequences for test adequacy, particularly branch coverage, and has empirically shown for white-box testing that longer test sequences can improve the results [5]. Memon and Xie evaluated the fault-detection effectiveness of smoke regression test cases for GUI-based software [77]. One of their observations is that longer sequences have been able to detect more faults than shorter ones, but they did not differentiate and analyze this observation any further, e.g., if the number of faults decreases or increases. Additionally, they stopped testing after executing a subset of their test suite covering length 3 due to the restrictions of their smoke test suite. Therefore, the role of test sequence length due to its fault detection capability in black-box testing is still not answered properly.

2.3 Optimization

Many approaches generate test sets containing redundant and unnecessary tests and thus neglect efficiency [129]. For instance, tests are generated for each state or transition where one test is contained or implicated by another test. Generating large test suites is relatively easy. But, larger test sets do not necessarily result in a better test coverage. Therefore, the test generation method for fulfilling the selected adequacy criterion plays an important role in the test process. For instance, Zeng et al. described an approach on specification-based test generation and optimization such as branch coverage based on model checking [129]. Search-based software testing uses techniques such as hill climbing, simulated annealing, or genetic algorithms to derive test cases [72]. The problem with these techniques is that they are not guaranteed to find the minimum set of test cases, but they do at least provide good approximate solutions.

Algorithms to derive a minimal set of test sequences can often be related to common graph problems, e.g., the *Chinese postman problem* for covering each edge [1, 113] or the *traveling salesman problem* for covering each vertex [30]. Under certain circumstances, it is even possible to form the (sub-)problem as a linear program, which can then be solved by the simplex method if a minimum is desired [112]. An example is the *assignment problem* [30], which has to be solved within the Chinese postman problem although solutions with a better run-time exist [12, 30].

2.4 Model Refinement

An interesting question is the role that model refinement, more precisely the "depth" of the modeling or its granularity, plays in MBT. The principle of "divide-and-conquer" is not new; Parnas was already considering hierarchical structures for modularization of computer programs in 1972 [97]. His thesis that "the effectiveness of modularization depends upon the criteria used in dividing the system into modules" is valid also for test case generation from hierarchical graph-based models as practiced by MBT. As an example, Memon et al. use an automatic planning system to generate test cases from GUI events and their interactions called *planning assisted tester for graphical systems* (PATH) [75]. Paiva et al. presented an ap-

proach based on hierarchical FSMs where the hierarchical structure is given special attention during the test case generation process [94]. The structure of hierarchical FSMs is exploited to reduce the number of states in the "flat" finite state machines, thus providing a way to deal with the state explosion problem. Andrews et al. propose a system-level testing technique that combines test generation based on FSMs with constraints [4]. They use a hierarchical approach to model large web applications and use constraints to select a reduced set of inputs to decrease the state space explosion. Reza et al. use hierarchical predicate transition Petri nets to model the behavior of SUC and to generate adequate test cases [102].

All of the above mentioned approaches deploy hierarchical structures. However, there is no approach comparable to the one the present thesis introduces for making use of this hierarchy for producing *optimized* test suites.

2.5 Software Reliability

Software reliability (SR) is one of the attributes of software quality and is defined as "the probability that software will not cause the failure of a system for a specified time under specified conditions" [58]. Since the early seventies of the last century, probabilistic models have been used to determine the SR based on observations obtained from software testing. SR is usually used to decide when to stop testing. In this context, *non-homogenous Poisson process* (NHPP) models are good candidates because of their compatibility with real world situations and simplicity of computation. They belong to the class of "reliability growth models" since they assume that faults are incrementally detected by tests and immediately (and perfectly) corrected, thus continuously improving the reliability of the SUC [99].

Musa-Okumoto (M-O), *Goel-Okumoto* (G-O), and *Delayed S-Shaped* (D-S) [70] are well-known NHPP models that are recommended by standards [2, 58, 36]. The critical question when applying an NHPP model is that of determining the appropriate mean value function, which eases the derivation of software reliability. This thesis considers NHPP models that follow the Musa-Okumoto classification scheme to cover the different types of the models instead of considering all the numerous existing models [70].

2.6 Component-based Software Reliability

Reliability can be determined twofold: through (i) *system-level reliability estimation* for SUC as a whole, and through (ii) *component-based reliability estimation* using the reliability of the individual components of SUC and their interconnection mechanisms. The following questions thereby arise: How to estimate the reliability of individual components, and how to aggregate and analyze these reliabilities. State-based frameworks for component-based software reliability prediction are available [48]. A different approach identifies critical components and investigates the sensitivity of the application reliability with respect to these components [127]. Krishnamurthy et al. assess the reliability of component-based applications based on test information and test cases [64].

Obviously, component-based reliability estimation techniques are promising candidates to be considered for the selective testing strategy presented in this thesis because the layers of hierarchical models represent components of the SUC. Unfortunately, there is no approach to our knowledge that (i) calculates the reliability on the basis of a hierarchical model used for testing a given SUC, and that (ii) uses this kind of information to detect further faults to increase the overall reliability as is common in reliability growth models. The approach presented in this thesis introduces a solution to this problem.

Chapter 3

Background

ESG notation is preferred for modeling, analyzing, and validating system behavior and user interface requirements prior to implementation and testing of the code because it intensively uses formal notions and algorithms known from graph theory and automata theory. These are relevant to the approach introduced in this thesis, especially hierarchical decomposition [9]. Another reason for this preference stems from the fact that events are externally perceptible and thus objectively observable phenomena, contrary to "states" that are internal to the SUC. Thus, events enable controllability of the test process. This preference causes no loss of generality because ESG, like EFG, is equivalent to FSM since all three can be represented by regular (type-3) grammars and thus can be interchangeably used [11].

3.1 Modeling System Behavior

Vertices of the ESG represent events, that is, environmental or user stimuli or system responses punctuating different stages of system activity. Directed edges connecting two events define allowed sequences among these events. A brief summary of the ESG notions follows; for details see [14].

Definition 3.1 (Event Sequence Graph)**.** An $ESG = (V, E, \Xi, \Gamma)$ is a directed graph where $V \neq \emptyset$ is a finite set of vertices (nodes) uniquely labeled by some input symbols of the alphabet Σ denoting events, $E \subseteq V \times V$ is a finite set of arcs

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

As a convention, a dedicated start vertex, for example, [, denotes the *entry* vertices Ξ of the ESG, whereas a final vertex, e.g.,], represents the *exit* vertices Γ. Note that [and] are not included in Σ. The semantics of an ESG is as follows. Given two vertices a and b in V, a directed edge ab from a to b indicates that event b follows event a, defining an *event pair* (EP) ab. Accordingly an *event triple*, *event quadruple*, etc. can be defined.

Definition 3.2 (Event Sequence)**.** Let V and E be defined as in Definition 3.1. Then any sequence of vertices $\langle v_0, \ldots, v_k \rangle$ is called an *event sequence* (ES) if $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$. The length *l* of an ES $\langle v_0, \ldots, v_k \rangle$ is defined as the number of vertices $| \langle v_0, \ldots, v_k \rangle |$, that is, $l(ES) = | \langle v_0, \ldots, v_k \rangle | = k + 1$. An $ES = \langle v_i, v_k \rangle$ of length 2 is called an *event pair* (EP). Furthermore, an ES is complete (or it is called a *complete event sequence,* CES), if $v_0 \in \Xi$ and $v_k \in \Gamma$. An ES is *partial* (or, it is called a *partial event sequence,* PES), if $v_0 \in \Xi$.

The remaining pairs $\overline{E}$ that can be constructed by all combinations $\widehat{E} = V \times V$ of the nodes given in the alphabet Σ but not in the ESG, that is, $\overline{E} = \widehat{E} \setminus E$, form the set of *faulty event pairs* (FEP). The set of FEPs constitutes the *complement* of the given ESG, which is symbolized as $\overline{ESG}$. Figure 3.1 shows a completed $\widehat{ESG} = (V, \widehat{E}, \Xi, \Gamma)$ with $\widehat{E} = E \cup \overline{E}$ where solid edges represent E (or EPs) and dashed edges represent $\overline{E}$ (or FEPs).

Definition 3.3 (Faulty Event Sequence)**.** Let $ES = \langle v_0, \ldots, v_k \rangle$ be an event sequence of length $k + 1$ of an ESG and $FEP = \langle v_k, v_m \rangle$ a faulty event pair of the corresponding $\overline{ESG}$. The concatenation of the ES and FEP then forms a *faulty event sequence* $FES = \langle v_0, \ldots, v_k, v_m \rangle$. Accordingly, a *faulty event sequence* (FES) of length n consists of $n - 2$ concluding, subsequent EPs and ends with an FEP. An FES is *complete* (or, it is called a *faulty complete event sequence,* FCES) if $ES = \langle v_0, \ldots, v_k \rangle$ is a PES.

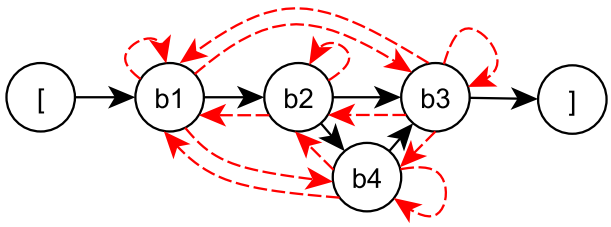


Figure 3.1: A completed $\widehat{ESG}$

A vertex representing a single, self-contained event is called an *atomic* event/ vertex. Alternatively, a vertex can be refined by another ESG (see Figure 3.2), the vertices of which can also be refined, resulting in a hierarchy of models [14, 13]. Events that can be refined are *compound* events/vertices consisting of atomic events and/or even other compound events. Before test cases can be generated, compound vertices are to be resolved according to Definition 3.4.

Definition 3.4 (Refinement)**.** Given an ESG, say $ESG_1 = (V_1, E_1, \Xi_1, \Gamma_1)$, a vertex $v \in V_1$, and another ESG, say $ESG_2 = (V_2, E_2, \Xi_2, \Gamma_2)$. Then replacing v by ESG_2 produces a *refinement* of ESG_1, say $ESG_3 = (V_3, E_3, \Xi_3, \Gamma_3)$ with $V_3 = V_1 \cup V_2 \setminus \{v\}$, and $E_3 = E_1 \cup E_2 \cup E_{pre} \cup E_{post} \setminus E_{1replaced}$ ('$\setminus$': set difference operation), wherein $E_{pre} = N^-(v) \times \Xi_2$ (connections of the predecessors of v with the entry nodes of ESG_2), $E_{post} = \Gamma_2 \times N^+(v)$ (connections of exit nodes of ESG_2 with the successors of v), and $E_{1replaced} = \{(v_i, v), (v, v_k)\}$ with $v_i \in N^-(v)$, $v_k \in N^+(v)$ and where $(v_i, v), (v, v_k) \in E_1$ (replaced arcs of ESG_1). $\Xi_3 = \Xi_1 \cup \Xi_2 \setminus \{v\}$ iff $v \in \Xi_1$, otherwise $\Xi_3 = \Xi_1$. $\Gamma_3 = \Gamma_1 \cup \Gamma_2 \setminus \{v\}$ iff $v \in \Gamma_1$, otherwise $\Gamma_3 = \Gamma_1$.

Note that $N^-(v)$ denotes the predecessors of a vertex v and $N^+(v)$ denotes the successors of v.

Example 3.5. In Figure 3.2, the refinement of vertex b of Model 1 is given as Model 2. The model given in Figure 3.3 is the resolved version of Model 1. More precisely, Model 1 is given as $V_1 = \{a, b, c, d, e, f, x, y, z\}$, $E_1 = \{(a, b), (b, c), (c, d), (c, e), (c, f), (x, y), (y, z), (z, c)\}$, $\Xi_1 = \{a, x\}$ and $\Gamma_1 = \{d, e, f\}$. In the refinement, that is, Model 2 of the compound event b, the predecessors and successors are $N^-(b) = \{a\}$, $N^+(b) = \{c\}$ and the refinement of Model 2 is given by

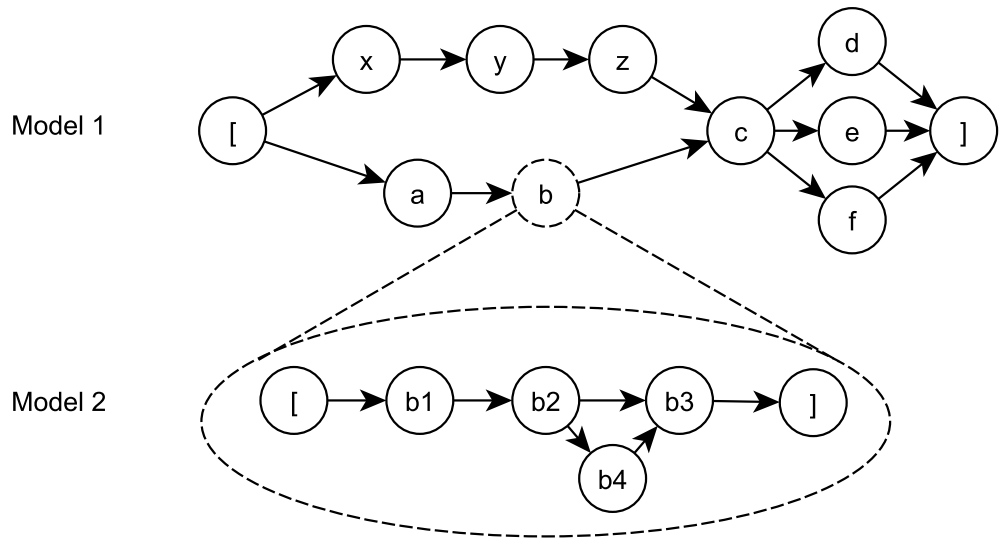


Figure 3.2: A compound vertex *b* of the model is refined

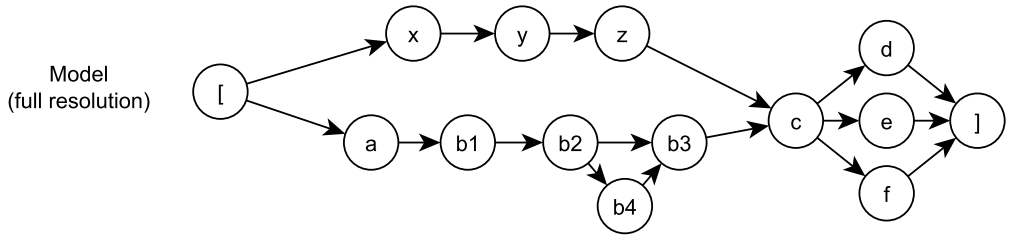


Figure 3.3: Completed (fully resolved) version of the model given in Figure 3.2

$V_2 = \{b1, b2, b3, b4\}$, $E_2 = \{(b1, b2), (b2, b3), (b2, b4), (b4, b3)\}$, $\Xi_2 = \{b1\}$ and $\Gamma_2 = \{b3\}$. The resulting (fully resolved) model shown in Figure 3.3 is represented by

$$
\begin{aligned}
ESG_3 &= (V_3, E_3, \Xi_3, \Gamma_3) \quad \text{with} \\
V_3 &= V_1 \cup V_2 \setminus \{b\} \\
&= \{a, b, c, d, e, f, x, y, z\} \cup \{b1, b2, b3, b4\} \setminus \{b\} \\
&= \{a, b1, b2, b3, b4, c, d, e, f, x, y, z\} \\
E_3 &= E_1 \cup E_2 \cup E_{pre} \cup E_{post} \setminus E_{1replaced} \\
&= \{(a, b), (b, c), (c, d), (c, e), (c, f), (x, y), (y, z), (z, c)\} \\
&\quad \cup \{(b1, b2), (b2, b3), (b2, b4), (b4, b3)\} \\
&\quad \cup \{(a, b1)\} \cup \{(b3, c)\} \setminus \{(a, b), (b, c)\} \\
&= \{(c, d), (c, e), (c, f), (x, y), (y, z), (z, c), (b1, b2), \\
&\quad (b2, b3), (b2, b4), (b4, b3), (a, b1), (b3, c)\} \\
\Xi_3 &= \{a, x\} \\
\Gamma_3 &= \{d, e, f\}
\end{aligned}
$$

3.2 Testing System Behavior

The approach introduced in Section 3.1 uses event sequences, more precisely CESs and FCESs, as test inputs. CESs, as positive tests, are supposed to lead to the exit vertex. If this is not feasible, the corresponding CES is marked as failed (*positive* testing). During a positive test of a web application, an event may not be reachable in certain situations, e.g., if

- a page was not loadable although a previous event, e.g., clicking a hyperlink, was executable,
- an error message has to be acknowledged that was not supposed to show up, or
- an input delivers a different structure of the program than the expected one.

In contrast, an FCES is not supposed to lead to the final event since it ends with an FEP, which should not be executable (*negative* testing) [14, 22]. If this is feasi-

ble, the corresponding FCES is marked as failed. During a negative test of a web application, an event might be reachable in certain situations, e.g., if

- an input was not checked properly by the SUC,
- a hyperlink did not lead to the expected page, or
- elements of a web page have been loaded that should only be available to some specific kind of users.

Hence, by analyzing ESG models, merely faults on events and their order can be detected. Other types of faults, for example the ones likely in database interactions, are usually not within the scope of this testing but they might be detected by chance.

The corresponding ESG-based test process is described by Algorithm 3.1 where $\omega(ES)$ determines the last event of a given ES. The test process is based on sets of CESs and FCESs that cover ESs and FESs of a given length. However, the derivation of minimal sets of test cases to reduce the effort for test execution as much as possible is a complex task. This will be shown in the next section.

Algorithm 3.1: Test process

```
1  cover all ESs of length k by means of CESs;
2  cover all FESs of length k by means of FCESs;
3  foreach ces ∈ CES do
4      apply ces to SUC;
5      if all events applicable in the specified order then
6          mark ces as passed;
7      else mark ces as failed;
8  foreach fces ∈ FCES do
9      apply fces to SUC;
10     if event ω(fces) applicable then
11         mark fces as failed;
12     else mark fces as passed;
```

3.3 Optimizing Test Generation

CESs and FCESs form the test sequences (test cases). For a thorough positive testing of ESGs, all EPs of a given ESG are to be covered by CESs of minimal total length and/or minimal number. Covering EPs is related to the often cited criterion of edge coverage, as in white-box testing [132], since edges represent EPs. This problem is a derivation of the *Chinese postman problem* (CPP) that attempts to find the shortest path or cycle in a graph by visiting each arc [12].

As already mentioned above, hierarchical models are supposed to be resolved completely before CESs are generated. The run-time complexity of finding a minimal solution is $O(|V|^3)$ where $|V|$ denotes the number of vertices [12]. The number of FCESs for negative testing increases with the increasing number of vertices since $|FCES| = |V|^2 - |E|$.

Example 3.6. The strategy explained in [14, 12] delivers the following CESs as test sequences for the model given in Figure 3.3 (on page 18):

```
CES1=[ x y z c d ]
CES2=[ a b1 b2 b3 c e]
CES3=[ a b1 b2 b4 b3 c f ]
```

Below, some of the resulting FCESs are presented as negative test cases.

```
FCES1=[ x a
FCES2=[ x b1
```

Note: the complete set of negative tests has $|FCES| = |V|^2 - |E| = 12^2 - 12 = 132$ elements; adding the three positive test cases above results in a total of 135 test cases for this example.

Solving the Chinese Postman Problem to Generate Test Cases

The set of CESs of Example 3.6 was generated by a solution to the CPP that will be briefly explained in the following. To derive this solution, the given graph is to be extended by additional edges until it forms an *Eulerian graph*, which has a cycle that traverses each edge exactly once and returns to the start vertex. A directed graph is Eulerian if it is strongly connected and each of its vertices $v \in V$ has equal indegree and outdegree defined as follows.

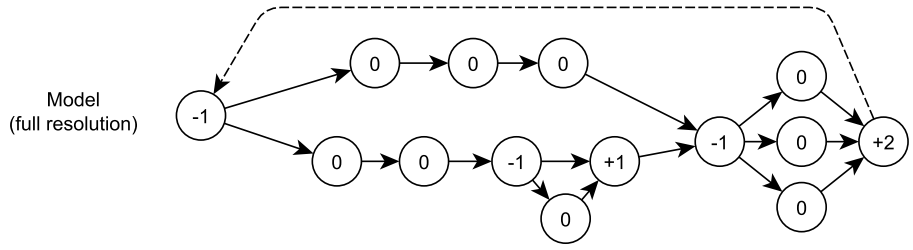


Figure 3.4: Degrees of the vertices of the ESG given in Figure 3.3

Definition 3.7 (Indegree/Outdegree/Balanced)**.** The number of edges going into a vertex v is the *indegree* written $\delta^-(v)$, and the number of edges pointing out of a vertex v is the *outdegree* written $\delta^+(v)$. Let δ be the difference between the in- and outdegrees: $\delta(v) = \delta^-(v) - \delta^+(v)$. If $\delta(v) = 0$, vertex v is called *balanced*.

Following Definition 3.7 leads to the conclusion that a directed graph is Eulerian if every vertex is balanced, that is, $\delta(v) = 0 \, \forall v \in V$, respectively. The resulting Eulerian cycle, which can be obtained by a standard algorithm in $O(|V| * |E|)$ time [122], is a minimal solution to the CPP if the set of added edges is minimal. For determining the minimal set of edges, two sets, A and B, have to be set up with

$$\begin{aligned} A &= \{v_i | i \in \{1, \ldots, \delta(v)\} \wedge \delta(v) > 0\} \\ B &= \{v_i | i \in \{1, \ldots, -\delta(v)\} \wedge \delta(v) < 0\} \end{aligned}$$

Figure 3.4, based on Figure 3.3 (on page 18), shows the degrees for each vertex of Figure 3.3 with set $A = \{],], b3\}$ and set $B = \{[, b2, c\}$. The closing bracket occurs twice in set A since its degree is $+2$. Note that an edge is added from end vertex] to start vertex [in Figure 3.4 to fulfill the requirement of strong connectivity.

Balancing the Graph by Solving the Assignment Problem

The challenge in deriving the minimal set of edges, that is, for balancing the graph, is to assign each element of set A to exactly one element of set B so that there is no unassigned element in either set and there is no other assignment with a lower number of edges to be added (according to the assignment). This leads to *assign-*

ment problems [30], which attempt to answer the question of how to assign n items (agents) to n other items (tasks), incurring some cost that may vary, depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task in such a way that the total cost of the assignment is minimal. Formally, an assignment problem minimizes the objective function 3.1 for a given $n \times n$ cost matrix c_{ij}, which fulfills the given constraints 3.2, 3.3, and 3.4 at the same time.

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{3.1}$$

$$s.t. \sum_{j=1}^{n} x_{ij} = 1 \ (i = 1, \ldots, n) \tag{3.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \ (j = 1, \ldots, n) \tag{3.3}$$

$$x_{ij} \in \{0, 1\} \ (i, j = 1, \ldots, n) \tag{3.4}$$

Considering ESGs, c_{ij} defines the number of edges of the shortest path (as costs) between vertex $i \in A$ and vertex $j \in B$, and n the number of elements of set A or B, respectively. After minimization, $x_{ij} = 1$ indicates that edges along the shortest path from vertex i to vertex j have to be added. Note that set A and B should have the same size since the sum of all degrees in a given graph is zero; that is, $\sum_{v \in V} \delta(v) = 0$ and, hence, $n = |A| = |B|$.

Example 3.8. Table 3.1 shows the resulting cost matrix to be solved for Figure 3.3. The matrix elements grade the minimal number of edges (as costs) if a node represented by row i is assigned to (connected with) a node represented by column j. The goal is to find an assignment of row i to column j so that each row i is assigned exactly once to one column j, and each column j is assigned exactly once to one row i. A minimal assignment is indicated by dark gray boxes in Table 3.1, that is, $x_{ij} = 1$ for the dark gray boxes. Furthermore, there should be no other assignment with a lower sum of costs. According to Table 3.1, the following shortest paths must be added to the ESG given in Figure 3.3 to create a minimal Eulerian cycle: $] \rightarrow [,] \rightarrow b2, b3 \rightarrow c$.

Table 3.1: The resulting cost and x_{ij} matrix out of Figure 3.4

c_{ij}	[	b2	c
]	1	4	5
]	1	4	5
b3	4	7	1

x_{ij}	[	b2	c	∑
]	1	0	0	1
]	0	1	0	1
b3	0	0	1	1
∑	1	1	1	

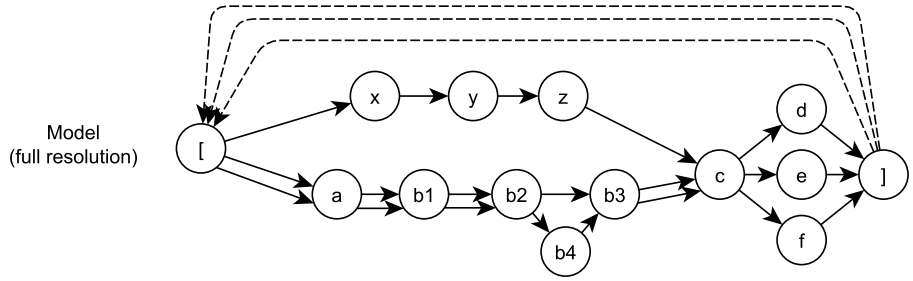


Figure 3.5: The balanced ESG

The balanced ESG is given in Figure 3.5. On the basis of this graph, the resulting Eulerian cycle appears as follows:

```
[ x y z c d ] [ a b1 b2 b3 c e ] [ a b1 b2 b4 b3 c f ][
```

Note that the last vertex [of the resulting Eulerian cycle does not contribute to the desired result and can be deleted. Furthermore, the edge between vertex] and vertex [is traversed more than once in the resulting tour. Thus, the resulting tour is to be split up between every occurrence of the two consecutive vertices] and [to gain the desired set of CESs as given in Example 3.6.

Methods for Solving the Assignment Problem

It is known from graph theory [122] that the construction of a minimal set of edges which creates an Eulerian graph leads to the assignment problem that can be solved in alternative ways.

1. *Complete enumeration method*: All combinations of assignments are prepared and an assignment involving the minimum cost is selected. This method might be suitable for assignment problems of small size, but it is not suitable for real world applications since the number of possible combinations is $n!$; e.g., an assignment matrix with 10 rows/columns would have approximately three and a half million combinations to evaluate.

2. *Simplex method*: The simplex method is a well-known algorithm for solving linear programming problems. Assignment problems can be formulated as a linear programming model, such as given by equations 3.1 to 3.4. Although the simplex method has better run-time complexity than the complete enumeration method, solving assignment problems using the simplex method can also be very time consuming since it has exponential run-time complexity [112].

3. *Hungarian method*: One of the fastest methods for solving assignment problems is the Hungarian method [30], which provides a solution in $O(n^3)$ time. Apart from the Hungarian method, other $O(n^3)$ solutions are given by Dinic-Kronrod [30] or Cycle Canceling [113].

Part II

Layer-centric Testing and its Reliability Analysis

Chapter 4

Approach

The ESG notions and graph-theoretic results summarized in Chapter 3 help to address the first question raised in Section 1.1 (Introduction): How can the effort of test generation as well as the excessive number of test cases be reduced? Furthermore, how the impact of this cost reduction on the overall system reliability can be determined is described. Parts of this chapter have been published in [17, 18].

4.1 Basic Idea

The basic idea for solving the problem of increasing complexity endemic to resolving the hierarchical structure is to generate test cases for each ESG individually, which is called layer-centric testing. This reduces the effort of finding a minimal solution since $O(|V_{resolved}|^3) > O(|V_1|^3) + \cdots + O(|V_k|^3)$ where k is the number of single ESGs forming the hierarchy, that is, $k = 2$ for Figure 3.2. This strategy will also reduce the number of negative test cases significantly since FEPs between different ESG layers are not considered. As a consequence, faults occurring between different layers can not be detected. But generating test cases for each ESG on its own also introduces several problems for test generation, which will be discussed in the following.

Problem 1: Effect of Compound Vertices on Test Generation

Compound vertices, which represent compound events, consist of atomic ones; however, their influence on test generation is not clarified.

Example 4.1. Consider Model 1 and Model 2 of Figure 3.2 (on page 18). The optimization algorithm given in [14, 12] generates the following CESs for Model 1 and Model 2.

```
Model1            Model2
T1=[ x y z c d ]  T4=[ b1 b2 b3 ]
T2=[ a b c e ]    T5=[ b1 b2 b4 b3 ]
T3=[ a b c f ]
```

For positive testing, 5 test cases are generated (instead of 3, Example 3.6). The number of resulting FCESs (negative testing) for Model 1 is 72; for Model 2 this number is 12. Compared to Example 3.6, the total number of test cases has been reduced from 135 to 89.

Analysis of Example 4.1 reveals the following problem. Compound vertices, e.g., b in Example 4.1, have more nodes than the atomic ones do. This implies, if there are many test sequences that include compound vertices, test length will very likely increase and, accordingly, test costs will increase. Therefore, there is a need to determine the weight of the compound events based on the number of events they include.

Definition 4.2 (Weight)**.** The *weight* of a compound vertex is given by the length of the shortest CES.

Example 4.3. The weight of Model 2 of Figure 3.2 is 3 because the shortest CES possible is `[ b1 b2 b3 ]`. Note that the pseudo-events do not contribute to the weight.

Example 4.4. If the weight of the compound event is taken into account, the test set of Example 4.1 modifies as follows:

```
Model1            Model2
T1=[ x y z c d]   T4=[ b1 b2 b3 ]
T2=[ x y z c e ]  T5=[ b1 b2 b4 b3 ]
T3=[ a b c f ]
```

Problem 2: Executing Compound Vertices

The next problem to be considered is how can test sequences be executed that contain compound vertices. An example of this problem can be seen in test case `T3` of Example 4.4 where vertex b represents a compound vertex.

A straightforward strategy is to replace the compound vertices by test case(s) generated from the lower-layer ESG. If this lower-layer ESG also contains compound events, one has to move down to the next lower-layer ESG, etc., and propagate test cases generated in these layers to upper layers.

Example 4.5. In Example 4.4, by replacing b in `T3` by `T4` the following test sequences can be constructed:

```
T1 =[ x y z c d]
T2 =[ x y z c e ]
T3'=[ a b1 b2 b3 c f ]
T4 =[ b1 b2 b4 b3 ]
```

Problem 3: Executing Lower Layer Test Cases

`T1`, `T2`, and `T3'` can be executed using Model 1 at the top layer when considering Example 4.5. `T4` is to be executed using Model 2 at the next lower layer, which is not desirable. In a potential solution, the compound vertex b has to occur in the minimal coverage of Model 1 at least as many times as its atomic refinement has test cases. Model 2 of Example 4.4 has two test cases, namely `T4` and `T5`. Therefore, the solution has to contain at least two occurrences of vertex b. This requirement can be fulfilled through an extension of the assignment matrix of Model 1 by adding columns and rows for the vertex that is needed multiple times. For Example 4.4, this is vertex b. Table 4.1 shows the resulting assignment matrix.

Table 4.1: The extended assignment matrix for Model 1

c_{ij}	**[**	**c**	**b**
]	1	5	5
]	1	5	5
b	3	1	8

Example 4.6. Taking Table 4.1 into account results in the following test sequences for Model 1 and Model 2:

```
Model 1          Model 2
T1=[ x y z c d]  T4=[ b1 b2 b3 ]
T2=[ a b c e ]   T5=[ b1 b2 b4 b3 ]
T3=[ a b c f ]
```

By replacing vertex b in `T2` and `T3` by `T4` and `T5`, respectively, the following test sequences can be constructed for Model 1 and Model 2:

```
Combined test case set:
T1 =[ x y z c d ]
T2'=[ a b1 b2 b3 c e ]
T3'=[ a b1 b2 b4 b3 c f ]
```

Algorithm 4.1 describes the LC testing approach for positive testing. This algorithm differs from the former one [12] in that it generates CESs for each ESG on its own instead of resolving the set of hierarchical ESGs. The run-time complexity depends mainly on balancing the corresponding ESG, which is $O(n^3)$, according to the Hungarian method. However, since this algorithm generates CESs for each ESG on its own, it has a better run-time complexity than solving the fully resolved ESG since $O(|V_{resolved}|^3) > O(|V_1|^3) + \cdots + O(|V_k|^3)$ where k is the number of ESGs forming the hierarchy.

In contrast to generating CESs, calculating FCESs for negative testing is considerably easier. FCESs of ESGs of the lower layer are generated first and then moved to the next higher layer where the shortest path [122] from start vertex [to the corresponding compound vertex $v \in V$ is calculated and concatenated with the given FCESs of the lower layer model. Algorithm 4.2 generates FCESs (see Algorithm A.4 for a formal description). The run-time complexity depends mainly on deriving the shortest path for every vertex. Dijkstra's algorithm can find the shortest path in $O(|V|^2)$. Since the shortest path has to be found for every vertex in the graph, the overall run-time complexity is $O(|V| * |V|^2) = O(|V|^3)$. Algorithm 4.2 also contains the method for deriving FCESs covering FESs of higher length. This will be part of the next section on CESs.

Algorithm 4.1: Determination of CESs for an hierarchical ESG according to LC (see Algorithm A.1 for a formal description)

input : an *ESG*
output : a set of CESs

```
1 foreach compound event of ESG do
2     set weight of compound event in ESG;          // Definition 4.2
3     generate CESs for the corresponding ESG' of the compound event;
4 balance current ESG, considering the generated CESs; // Section 4.1
5 determine CESs on the basis of the balanced ESG;    // Euler Tour
6 replace compound events in the resulting CESs by the CESs of the compound
  events;
```

Algorithm 4.2: Determination of FCESs for an hierarchical ESG according to LC (see Algorithm A.4 for a formal description)

input : an *ESG* and the desired *length* of FESs to be covered
output : a set of FCESs covering faulty event sequences of *length*

```
1  foreach compound event of ESG do
2      generate FCESs for the corresponding ESG' of the compound event;
3      foreach FCES do
4          prepend shortest path from start vertex [ to compound event;
5  set up all FESs of length;
6  foreach FES do
7      prepend shortest path from start vertex [ to first event of FES;
8      foreach compound event in FES do
9          if compound event is last vertex then
10             replace with one of the start vertices of ESG' of the compound
               event;
11         else
12             replace with shortest path through ESG' of the compound event;
```

4.2 Covering Event Sequences of Higher Length

A phenomenon in the testing of interactive systems is that failures can be frequently observed but reproduced only in specific contexts. Coverage of event sequences of length >2 can help here. To achieve such coverage(s), the original graph will be transformed by Algorithm 4.3 to again enable the re-use of algorithms for length 2 coverage. For ease of understanding, Figure 4.1 shows the transformation of Model 1 of Figure 3.2 (on page 18) in two consecutive steps to achieve a coverage of sequence length 3. In Step 1, all sequences of $(length-1)$ are determined as the vertices. Two vertices are connected if the last $(length-2)$ events of one vertex equal the first $(length-2)$ events of the other vertex. The result is a graph where each edge represents the desired length to be covered. Step 2 turns the vertices back into single events, which also is described by lines 3-10 in Algorithm 4.3.

Algorithm 4.3: Transformation of an ESG to cover all ESs of higher length (see Algorithm A.10 for a formal description)

input : an $ESG = (V, E, \Xi, \Gamma)$ and the desired *length* of ESs to be covered
output : a transformed ESG' according to the *length* to be covered

1 $ESG' = (V', E', \Xi', \Gamma')$ with $V' = \emptyset$, $E' = \emptyset$, $\Xi' = \emptyset$, $\Gamma' = \emptyset$;
2 build all ESs of (*length*-1);
3 **foreach** *ES* **do**
4 **if** first event of ES belongs to Ξ **then**
5 add all events of ES as copy to ESG';
6 add edges along the ES to ESG';
7 add copy of first event to Ξ';
8 **else** add last event of ES as copy to ESG';
9 **if** last event of ES belongs to Γ **then**
10 add copy of last event to Γ';
11 **foreach** pair of ESs (ES_1,ES_2) **do**
12 **if** the last ($length$-2) events of ES_1 equal the first ($length$-2) events of ES_2 **then**
13 add edge between the last events of ES_1 and ES_2 in ESG';

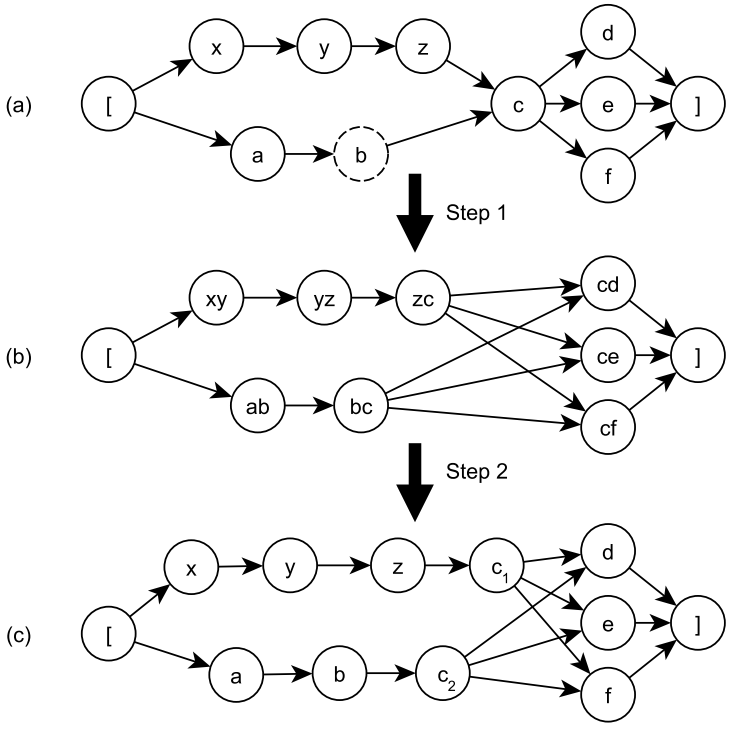


Figure 4.1: Graph transformation for length 3 coverage

Considering Duplicated Vertices

As can easily be seen in Figure 4.1, a vertex like (indexed) vertex c might be contained more than once in the resulting graph. If the transformed graph includes a vertex $v \in V$ of the original graph more than once, and if a multiple occurrence of that vertex v is needed, it is then not obvious which vertex v will minimize the assignment. A solution is to extend the assignment matrix by all occurrences of v and redefine the matching constraints.

Example 4.7. Assuming that vertex c is needed one more time in the solution, then it is not obvious if vertex c_1 or vertex c_2 of Figure 4.1 will minimize the solution. Adding vertex c_1 to the assignment matrix results in costs of 29 (see Table 4.2, left) whereas adding vertex c_2 results in costs of 28 (see Table 4.2, right). That is, vertex c_2 would minimize the solution. Unfortunately, the number of matrices increases very fast if further selections between duplicated vertices are required. In this case, every combination of possible selections has to be built. Assuming that another selection between two duplicated vertices d_1 and d_2 is required, then 4 combinations can be built $((c_1, d_1)\ (c_1, d_2)\ (c_2, d_1)\ (c_2, d_2))$ and 4 matrices need to be solved.

Table 4.2: The cost matrix for Figure 4.1 extended by c_1 (left) and c_2 (right)

c_{ij}	[	c_1	c_1	c_2	c_2	c_1
]	1	5	5	4	4	5
]	1	5	5	4	4	5
d	2	6	6	5	5	6
e	2	6	6	5	5	6
f	2	6	6	5	5	6
c_1	3	7	7	6	6	7

c_{ij}	[	c_1	c_1	c_2	c_2	c_2
]	1	5	5	4	4	4
]	1	5	5	4	4	4
d	2	6	6	5	5	5
e	2	6	6	5	5	5
f	2	6	6	5	5	5
c_2	3	7	7	6	6	6

Unification of the problem

Considering Example 4.7 raises the need for a more straightforward approach that would solve the given problem within one cost matrix, as can be seen in Table 4.3.

Formally, the following set of equations described as a linear program has to be solved in such a case.

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{4.1}$$

$$s.t. \sum_{j=1}^{n} x_{ij} = 1 \ (i = 1, \ldots, k) \tag{4.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \ (j = 1, \ldots, k) \tag{4.3}$$

$$\sum_{j=1}^{n} \sum_{i=s(y)}^{e(y)} x_{ij} = 1 \ (y = 1, \ldots, l) \tag{4.4}$$

$$\sum_{i=1}^{n} \sum_{j=s(y)}^{e(y)} x_{ij} = 1 \ (y = 1, \ldots, l) \tag{4.5}$$

$$\sum_{i=1}^{n} (x_{ij} - x_{ji}) = 0 \ (j = s(y), \ldots, e(y), y = 1, \ldots, l) \tag{4.6}$$

$$x_{ij} \in \{0, 1\} \ (i, j = 1, \ldots, n) \tag{4.7}$$

Table 4.3: The combined cost matrix for Figure 4.1 containing c_1 and c_2 simultaneously

			j						
			1	**2**	**3**	**4**	**5**	**6**	**7**
		c_{ij}	**[**	**c_1**	**c_1**	**c_2**	**c_2**	**c_1**	**c_2**
i	**1**	**]**	1	5	5	4	4	5	4
	2	**]**	1	5	5	4	4	5	4
	3	**d**	2	6	6	5	5	6	5
	4	**e**	2	6	6	5	5	6	5
	5	**f**	2	6	6	5	5	6	5
	6	**c_1**	3	7	7	6	6	7	∞
	7	**c_2**	3	7	7	6	6	∞	6

Example 4.8. In Table 4.3, column/row 6 and 7 have been added and only one of them should be included in the solution for minimization purposes. In this case, column/row 7 minimizes the assignment and column/row 6 has to be skipped. According to Table 4.3, $k = 5$, $l = 1$, $s(1) = 6$, and $e(1) = 7$. Thus, l defines the number of matrix intervals from which only one column/row should be selected (here only one consisting of column/row 6 and 7), $s(y)$ defines the first index of interval $y <= l$, and $e(y)$ defines the last index of interval $y <= l$. k defines the last index of columns/rows, which do not belong to an interval. Equation 4.4 and 4.5 choose "exactly one" of the columns/rows of the "interval" given by column/row 6 and 7. Equation 4.6 guarantees that if a column is selected, the appropriate row will be selected as well; that is, if column 6 in Table 4.3 is selected, row 6 needs to be selected accordingly. Selecting row 7 along with column 6 will otherwise lead to an unbalanced graph.

Algorithm 4.4 improves Algorithm 4.1 for balancing a graph to cover sequences of higher length, according to the findings described above. Unfortunately, it is not a simple matter to adapt the given equation system 4.1 to 4.7 to the Hungarian method. An optimal solution (abbreviated as LC_{opt}) can be calculated using linear programming, but it has exponential run-time complexity.

In case a solution cannot be found in sufficient time, a heuristic approach is given by solving the assignment problem first and then adding the shortest self-cycle for the given vertex v as many times as needed; this strategy is abbreviated as LC_{simple}. LC_{simple} is straightforward and feasible but not necessarily minimal. See Algorithm A.9 for a formal description of LC_{simple}.

4.3 Reliability Analysis

As demonstrated in the previous sections, LC strategy can considerably reduce the size of the test suite and thus the test costs. The critical question is, however, how far this cost reduction will affect the reliability. If the trade-off negatively influences the quality, the saving effect will not persuade to accept the new strategy. Thus, a quantitative comparison of the reliability levels achieved by both strategies, LC and FR, is required. This section briefly summarizes and discusses existing software

Algorithm 4.4: Determination of CESs for an hierarchical ESG according to LC (see Algorithm A.6 for a formal description)

input : an *ESG*
output : a set of CESs

1 **foreach** compound event of ESG **do**
2 set weight of compound event in ESG; `// Definition 4.2`
3 generate CESs for the corresponding ESG' of the compound event;
4 transform ESG for covering the desired length; `// Algorithm 4.3`
5 balance current ESG considering the generated CESs; `// Section 4.2`
6 determine CESs on the basis of the balanced ESG; `// Euler Tour`
7 replace compound events in the resulting CESs by the CESs of the compound events;

reliability models that have been used since the early seventies of the last century for reliability determination based on fault data observed during testing of SUC (see also existing standards and guidelines, e.g., [58, 36]). As there is no prior knowledge about the characteristics of the fault data, several of these models have to be checked to select the one that best fits the data obtained. This section also discusses the criteria given for the selection.

Starting with the same hierarchical model, test suites are separately generated and executed following LC and FR strategies (Figure 4.2). The test results build up the fault data to be analyzed to select the best-fitting SR model. Finally, the reliabilities of the both strategies, R_{LC} and R_{FR}, will be calculated and compared.

Since each of the faults shall be counted only once assuming they were corrected upon detection without causing new faults, *Software Reliability Growth Models* (SRGM) will be used that assume the absolute number of faults remaining in the software decreases and thus SR grows. The assumption made is that the detected faults will be perfectly corrected; that is, they do not induce follow-on faults.

SRGMs can be classified along five different attributes introduced by Musa and Okumoto [70]:

- *Time domain*. Calendar time versus execution time,
- *Type*. The distribution of the number of failures experienced by time t is

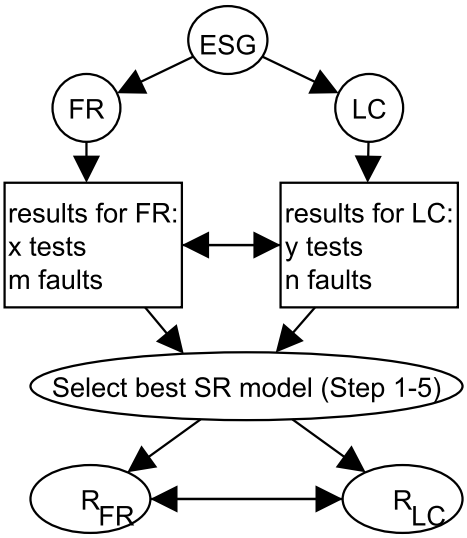


Figure 4.2: Overview of the reliability analysis

Binomial or Poisson.

- *Category*. The total number of failures that can be experienced in infinite time is considered either as finite or as infinite.

- *Class* (in finite failure category only). Functional form of the failure intensity expressed in terms of time, e.g., exponential, Weibull, or gamma class.

- *Family* (in infinite failure category only). Functional form of the failure intensity function expressed in terms of the expected number of failures experienced, e.g., geometric family, power family.

Reliability Determination Process

In order to select proper SRGMs for predicting reliability R of a system, the following steps will be performed.

Step 1: Determine Testing Time and Type of Fault Data There are several ways to measure test time during the testing process, such as calendar time, number of test runs, and number of test cases or execution time. Moreover, there are two types of fault data for SRGMs: time intervals between successive observed faults and the number of faults detected in a specified time interval. The case study (Chapter 6) takes the number of test cases generated by event sequences of length 2, 3, and 4 into account as points in time. Furthermore, the cumulative number of faults is used as fault data.

Step 2: Analyze the Statistical Properties Statistical properties of fault data are analyzed subject to different aspects, for example, whether they follow a specified probability distribution or whether they form a specific stochastic process. *One-sample Kolmogorov Smirnov test* (K-S) [27] is one of statistical nonparametric tests used to determine whether a sample (fault data collected) fits the specified distribution. In the follow-on case study, it will be observed that the cumulative number of faults builds up a Poisson distribution according to K-S test (Table 6.3 on page 64).

Poisson type models can be divided into two groups: *homogenous Poisson process* (HPP) and *non-homogenous Poisson process* (NHPP). HPP models assume that the failure rate does not change during the testing process; in other words, SUC has constant failure intensity. In the present case, failure intensity varies with the time parameter since faults are only counted once and it is assumed that no new faults are inserted. Therefore, NHPP models are favored. As the exact nature of fault data is not known a priori (except that it can be described as NHPP), several NHPP models must be selected to ensure each type (Table 4.4) is covered.

Step 3: Select Parameter Estimation Technique *Maximum likelihood estimation* (MLE) technique [27] will be used for estimating parameters of SRGMs because MLE fulfills most of the favored properties, such as asymptotic normality, robustness and consistency. In addition, MLE simultaneously estimates model parameters and provides for the easy derivation of confidence intervals.

In MLE, it is convenient to use the *log-likelihood function* (LLF) for parameter

Table 4.4: Overview of the selected NHPP models

name of the model	abbr.	ref.	mean value function	parameter	category	class/family
Goel-Okumoto	G-O	[47]	$\mu(t) = a(1 - e^{-bt}),\ b > 0$	a=expected number of faults	finite failure models	exponential class
Generalized Goel-Okumoto	GGO	[46]	$\mu(t) = a(1 - e^{-bt^c})$ $a > 0, b > 0, c > 0$	b, c=some constants		weibull class
Delayed S-Shaped	D-S	[128]	$\mu(t) = a(1 - (1 + bt)e^{-bt})$ $a > 0, b > 0$	a=expected number of faults, b=fault detection rate		gamma class
Log-Power	LP	[130]	$\mu(t) = a \ln^b(1 + t),\ t \geq 0$	a=scaling factor b=shape parameter		power family
Duane	D	[39]	$\mu(t) = a \cdot t^b$		infinite failure models	
Musa-Okumoto	M-O	[83]	$\mu(t) = \frac{1}{\theta} \ln(\lambda_0 \theta t + 1)$	θ=failure intensity decay parameter, λ_0=initial failure intensity		geometric family

estimation

$$\begin{aligned} LLF &= \ln L(\theta_1, \ldots, \theta_k) \\ &= \left(\sum_{i=1}^{p} y_i \cdot \ln\left(\mu(t_i) - \mu(t_{i-1})\right) - \ln(y_i!) \right) - \mu(t_e) \end{aligned} \tag{4.8}$$

where p is the number of groups or time intervals, $\mu(t)$ is the mean value function (Table 4.4), y_i is the detected number of faults in interval i, and $(\theta_1, \ldots, \theta_k)$ are unknown parameters to be estimated according to the maximum likelihood principle. A set of likelihood equations is obtained by taking partial derivative of this log-likelihood with respect to each parameter and setting each of the derivatives to 0. Solving the set of equations delivers the estimates of the model parameters.

The *lower limit* (LL) and *upper limit* (UL) for confidence on the expected number of faults can be solved by the equations 4.9 and 4.10.

$$\mu_{LL}(t_i) = \mu(t_{i-1}) + LL \tag{4.9}$$

$$\mu_{UL}(t_i) = \mu(t_{i-1}) + UL \tag{4.10}$$

LL and UL are obtained by solving equations 4.11 and 4.12 where $1 - \alpha$ is the confidence level or confidence coefficient. α is mostly chosen as 0.05.

$$\sum_{i=0}^{LL-1} \frac{\left(\mu(t_i) - \mu(t_{i-1})\right)^i e^{-(\mu(t_i)-\mu(t_{i-1}))}}{i!} = 1 - \frac{\alpha}{2} \tag{4.11}$$

$$\sum_{i=0}^{UL-1} \frac{\left(\mu(t_i) - \mu(t_{i-1})\right)^i e^{-(\mu(t_i)-\mu(t_{i-1}))}}{i!} = \frac{\alpha}{2} \tag{4.12}$$

Step 4: Calculate Goodness of Fit (GoF) Measures GoF measures describe how well SRGMs fit a set of observations. Therefore, GoF measures can also be used to compare different SRGMs according to their correlation to fault data. In this study, *Akaike information criteria* (AIC) and *Bayesian information criteria* (BIC) are used since these criteria are based on the maximized value of likelihood. In

addition, commonly used *mean square error* (MSE) is selected [109].

$$AIC = -2LLF + 2k \tag{4.13}$$
$$BIC = -2LLF + k\ln(n) \tag{4.14}$$
$$MSE = \sum(y - \hat{y})^2/(n-k) \tag{4.15}$$

where k is number of the model parameters, n is the number of observation, LLF is the log likelihood, y is the observed value, and $\hat{y}$ is the predicted value.

Step 5: Select Best Model The SRGM with the smallest AIC, BIC, and MSE is selected as best fitting model for calculating reliability R in the interval $(t, t+x)$ as follows:

$$R(x|t) = P\{N(t+x) - N(t) = 0\} = e^{-(\mu(t+x)-\mu(t))} \tag{4.16}$$

where $N(t)$ is the cumulative number of faults detected by time t.

Chapter 5

Selective Layer-centric Testing

The conclusion that can be drawn from Chapter 3 and 4 is that the test exhaustiveness and test execution effort can be controlled by appropriate selection of

(i) the ES length to be covered, and

(ii) the strategy for handling model refinement.

The higher the chosen sequence length, the more exhaustive the testing of the underlying SUC, but this also results in greater test execution effort. ESGs allow the generation of a very large set of CESs for testing a given SUC by simply increasing the considered event sequence length step by step, whereby the test effort increases with every step, usually exponentially. As compensation, testing event sequences of higher length facilitate the detection of critical faults that can only be detected in specific contexts. However, there is a need to detect those critical faults with less testing efforts.

Assuming that critical faults are to be detected by a subset of the models forming the hierarchy, it should be possible to increase testing efforts based on this subset of models only. But first it is necessary to identify this subset of models that are expected to have a higher fault detection capability. Reliability estimation provides a solution. The reason for using reliabilities (or impacts) for model selection is that reliabilities describe the likelihood of a system running fault-free during a given time from a statistical point of view. Since it is desirable to select the model with

good chances of detecting the next fault, it is reasonable to select the corresponding system component with a worse reliability/impact on the overall reliability.

5.1 Basic Idea

The basis for an ESG-based test process forms a set of test cases that covers at least event sequence length 2 (or EPs/FEPs, respectively). This set of test cases also forms the basis for analyzing which component is likely to conceal additional faults. For further analysis, it is necessary to execute the underlying test case set covering all EPs and collect the results of their execution. On the basis of the results, detected faults are to be categorized according to the given components (represented by ESGs). This is done by identifying the event which has not been executable in a CES and assigning this fault to the corresponding component. The same is done for an FCES that detected a fault. The event that has been executable should be assigned to the appropriate component. The result of this categorization is a number of faults detected by each component or ESG, respectively. This data will be used to identify the layers that have a worse reliability. After that, the ongoing testing will focus on these layers only. Figure 5.1 gives a summary of the resulting steps to be performed for *selective layer-centric* (SLC) testing strategy. The details on Step 2 and 3 are given in the following.

5.2 Layer Selection Process

To identify the subsystem(s) that most endanger(s) the system reliability, the reliability of each component must be calculated separately. Furthermore, the impact of the individual components on the overall system has to be determined, which requires calculating the *usage ratio of each component* (UR) first [38].

Step 2.1: Determining Usage Ratio (UR) for each Component.

UR contributes to the fact that the single components are tested with a varying amount of effort. Since a single test case may contain events of different components (test cases are merged during LC testing), it is no longer sufficient to use the

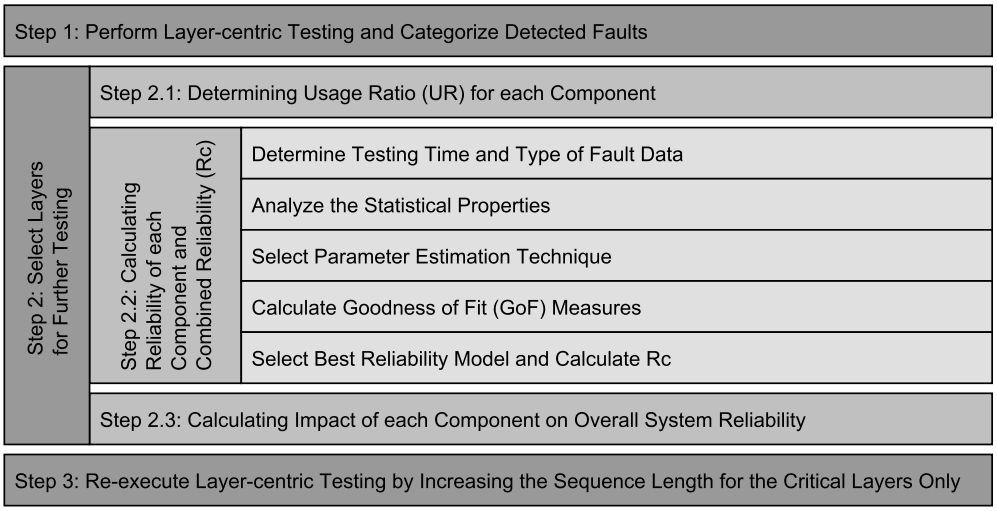


Figure 5.1: Summary of the SLC strategy

number of test cases for reliability calculations. That is why UR parameter represents the ratio of number of events of each component over the number of events of the overall SUC.

$$UR_k = \frac{E_k}{TEOS} \tag{5.1}$$

where E_k is number of events of k-th ESG/component and $TEOS$ is the total number of events of the overall test case set.

Step 2.2: Calculating Reliability of each Component and Combined Reliability ($\mathbf{R_c}$).

Calculating the *reliability of each component* (RE) follows steps similar to those needed for calculating the overall system reliability described in Section 4.3.

- Testing time and type of failure are determined. The cumulative number of events generated for each ESG/component is used here as time parameter. Failure data type is the cumulative number of faults detected in each component.

- Some SRGMs are selected according to the statistical properties of this fault

data. Here, the NHPP models proposed in Section 4.3 are selected again (Table 4.4 on page 42).

- Parameter estimation technique is selected. Here again, MLE is used.
- In order to determine the model that best fits the given fault data, GoF measurements AIC, BIC and MSE are calculated for each selected SRGM.
- The reliability model with the smallest AIC, BIC, and MSE is selected as the best fitting model.

To construct fault data, components are first sorted in descending order in accordance with their usage ratios. Then REs are determined according to the best-fitting SRGM model as follows.

$$RE_k = e^{-\mu(t_k)+\mu(t_k-\Delta t)} \tag{5.2}$$

where RE_k represents the reliability of k-th component of sorted components, t_k is equal to the number of events generated for k-th component, and Δt is a small time interval defined by the user. However, it can be selected as the minimum number of events from among the number of events generated for components. The combined reliability R_c is then defined as follows.

$$R_c = 1 - \sum_{k=1}^{m}(1 - RE_k)UR_k \tag{5.3}$$

where RE_k represents reliability of k-th component, and m is the number of components.

Step 2.3: Calculating Impact of each Component on Overall System Reliability.

The *impact of each component* (IE) [38] on overall system reliability can then be determined as follows.

$$IE_k = 1 - \frac{(1 - RE_k)UR_k}{1 - R_c} \tag{5.4}$$

where IE_k represents the impact of k-th component on overall system reliability. Note that components with a low IE value have a higher (negative) influence on the overall system reliability than those with a higher IE value. Thus, the overall system reliability can be improved by increasing IEs of these components.

Consequently, components with a small IE value have a (statistically) higher fault detection capability and should be considered for further testing. A common method in statistics is to divide a given dataset into four equal groups (called *quartiles*). Quartiles can be used to determine the components for further testing. Here, the 1st quartile of IE values determines the set of components with the worst IE values. That is, further testing can be performed for components that have an IE value equal to or less than the borderline value for the 1st quartile. The 1st quartile of IE values is determined as follows.

- IE values are put in order from smallest to largest.
- 1st quartile is the IE value of component at position $p = (n + 1)/4$. If p is not an integer, e.g., $p = 3.5$, the IE value at position p is as follows: $IE_p = IE_{p_<} + (IE_{p_>} - IE_{p_<}) * (p - p_<)$ where $p_<$ is the (integer) position before p (that is, p is rounded down to the next integer leading to $p_< = 3$) and $p_>$ is the (integer) position after p (that is, p is rounded up to the next integer leading to $p_> = 4$).

Quartiles can be computed by using software packages such as Minitab [79]. In order to demonstrate that the 1st quartile is a good choice in terms of reliability, all components are first of all selected to calculate R_c. Then data related to components is removed step by step from calculating R_c. R_c is re-calculated at every turn and compared with LC testing. After removing them, the changes become apparent. Third and finally, the percentage of changes in the parameters with respect to LC testing are compared, which are calculated as follows.

$$CP = \frac{|p^k - p^{LC}|}{|p^{LC}|} * 100 \tag{5.5}$$

where p^k shows the values of model parameters obtained after removing the k-th component and p^{LC} shows the parameters of LC testing.

5.3 Test Generation Process

On the basis of the analysis in Step 2, the test effort is to be increased only for the components which have been identified as putting the overall system reliability most at risk. The open question is how can test efforts be increased for these components.

LC testing generates test sequences for every individual model of a given model hierarchy to reduce the test generation and execution effort compared to the FR approach. But the LC approach also enables another method of controlling the thoroughness of the test and test execution effort. Based on LC, it is possible to increase event sequence length to be covered only for some selected individual models of the hierarchy. Assuming that a given model hierarchy consists of 3 models, it is possible to generate longer test sequences for just a subset (one or two) of the models whereas the other models are covered by shorter test sequences. The precondition is that the given SUC is represented by a hierarchical set of models.

Arising Problems

Applying the strategy to real-life SUCs uncovers some problems. If tests are generated for the uncritical components as well, this leads to additional test effort since a test case set covering sequence length 2 forms the basis for the identification of critical components. Therefore, tests of the uncritical components are expected to have no additional benefit; that is, they are not likely to detect any additional faults. Hence, it would be worthwhile to generate tests only for the critical components. This, however, leads to the following questions that can easily be answered.

1. How can test sequences containing compound vertices be executed if no tests have been generated for the compound vertex? *Answer*: Since the goal is to minimize the test execution effort, the compound vertices are to be replaced by the shortest path through the corresponding ESG. Note that if some tests have been generated for the compound vertex, they are to be considered during sequence generation as described for LC testing in Chapter 4.

2. How can test sequences of lower layers be executed if no tests are to be generated for upper layers? *Answer*: Move them to the upper layer as well; that

is, generate test sequences for the upper layer so that lower layer tests can be executed. Further details are given in the following.

Executing Lower Layer Test Cases

The second question above refers to the fact that CESs of lower layers are to be executed somehow in the context of the upper layers, even if no tests are generated for the upper layer. To keep the costs as low as possible in such cases, a set of CESs is needed for the upper layer containing the number of compound vertices for which as many tests as needed are generated. Furthermore, this set should not be replaceable by another set of CESs with a lower number of total events.

Example 5.1. Consider Figure 5.2 where an ESG is given with refined vertices c, e, and h that are symbolized as dashed circles. Temporarily ignore the dashed lined arc from vertex] to vertex [. The ESGs of the compound vertices c and e have been selected for further testing so that test sequences (CESs) to cover ESs of higher length have been generated for them. The numbers next to the vertices indicate how many CESs have been generated for them, that is, 2 CESs for vertex c and 3 CESs for vertex e. Thus, the number indicates how often this compound vertex will be needed in a solution. Here the goal is to derive a CES set that contains vertex c at least twice and vertex e at least three times and the total number of events is minimal.

Solving the Traveling Salesman Problem for Testing Lower Layers

Adding an edge from pseudo end vertex] to pseudo start vertex [(as shown in Figure 5.2) produces a strongly connected ESG, which helps to derive the required set of CESs. This edge enables the problem to be transferred to determine a minimal tour that starts and ends at pseudo start [and which visits the compound vertices as often as needed. If every compound vertex needs to be visited only once, this problem forms a derivation of the *traveling salesman problem* (TSP) [30]. The TSP attempts to find the shortest tour that visits each entry (“city”) of a given list. The underlying assumption is that the pairwise distances are known for the cities, e.g., given as a matrix $C = (c_{ij})$. Considering ESGs, the distances consist of the minimal

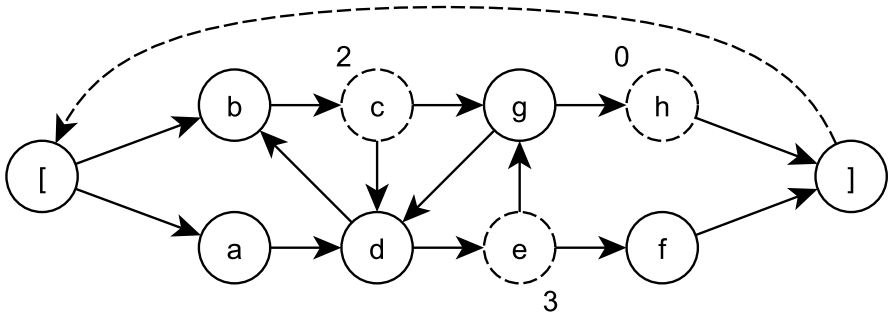


Figure 5.2: Example of an ESG with compound vertices c, e and h

number of edges between two vertices. Solving the TSP can also be seen as solving the assignment problem but where the resulting assignment needs to describe a cyclic permutation. Let $X = (x_{ij})$ describe a cyclic permutation, e.g.,

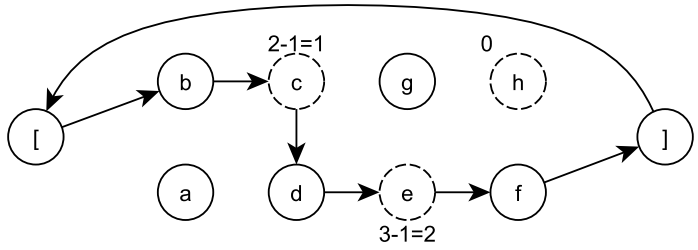
$$X = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Then the TSP can be stated as

$$\min_{X \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

where S_n is the set of all cyclic permutations.

Example 5.2. The right hand side of Figure 5.3 shows the underlying distance matrix for solving the TSP. The numbers of the matrix represent the minimal number of edges to be visited if a vertex represented by row i is assigned to a vertex represented by column j. The dark gray boxes indicate the minimal tour; that is, seven edges have to be visited to follow the minimal tour. According to the table, the following shortest paths must be added to the ESG to denote the shortest tour: $[\rightarrow c$,



c_{ij}	[	c	e
[	-	2	4
c	4	-	2
e	3	4	-

Figure 5.3: The resulting tour through compound vertices c and e (left side) determined by the solution of the TSP on the basis of the distance matrix (right side)

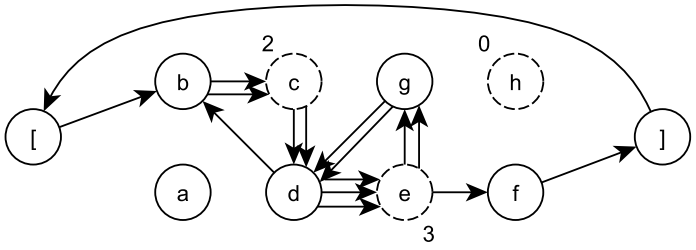
$c \rightarrow e$, $e \rightarrow [$. The left hand side of Figure 5.3 illustrates the minimal tour.

Extending the Tour by Solving the Assignment Problem

On the basis of this initial solution, the next goal is to extend this tour in a minimal way so that the resulting tour contains the desired number of compound vertices needed. This can be achieved by adding this tour into a graph with all edges of the original graph having been deleted (as can be seen in Figure 5.3). After that, this graph is to be extended by additional edges so that the resulting Eulerian cycle contains the compound vertices as many times as needed. As already described in Section 4.1, determining the set of additional edges can be achieved by setting up and solving the assignment problem.

Example 5.3. As can be seen in Figure 5.3, vertex c is needed one more time in the solution and vertex e is needed two more times in the solution. The right hand side of Figure 5.4 shows the cost matrix of the corresponding assignment problem to be solved. A minimal assignment is indicated by dark gray boxes. According to this assignment, the following shortest paths must be added to the ESG given in Figure 5.3: $c \rightarrow e$, $e \rightarrow c$, and $e \rightarrow e$. The resulting ESG can be seen in Figure 5.4. On the basis of this graph, the resulting Eulerian cycle starting in pseudo vertex $[$, looks as follows:

```
[ b c d e g d b c d e g d e f ] [
```



c_{ij}	c	e	e
c	3	2	2
e	4	3	3
e	4	3	3

Figure 5.4: The extended ESG (left side) along the assignment matrix (right side)

In contrast to generating CESs, calculating FCESs for negative testing in SLC is considerably easier. In a way done similarly in LC, FCESs of selected ESGs of lower layer are generated first and then moved to the next higher layer where the shortest path [122] from start vertex [to the corresponding compound vertex $v \in V$ is calculated and concatenated with the given FCESs of the lower layer model.

Chapter 6

Case Study I: Reliability Analysis Concerning Test Length & Model Refinement

A large web application borrowed from an industrial project is used to demonstrate and validate the approach and to analyze its characteristic features, including a comparison of FR with LC and SLC. Parts of this chapter have been published in [17].

6.1 System Under Consideration, Test Setup, and Goals of the Experiment

SUC is a large commercial web portal with 53,000 lines of code called ISELTA ("Isik's System for Enterprise-Level Web-Centric Tourist Applications"). This portal enables travel and tourist enterprises, e.g., hotel owners, to create their own individual search & service offering masks. These masks can be embedded in an existing homepage as an interface between user and system. Customers can then use those masks to select and book services, e.g., hotel rooms, rental cars, etc. See Figure 6.1 for the entry screenshot of ISELTA.

The goal of the experiment is to determine the special characteristics of the FR, LC, and SLC approach with respect to their particular strengths and limitations.

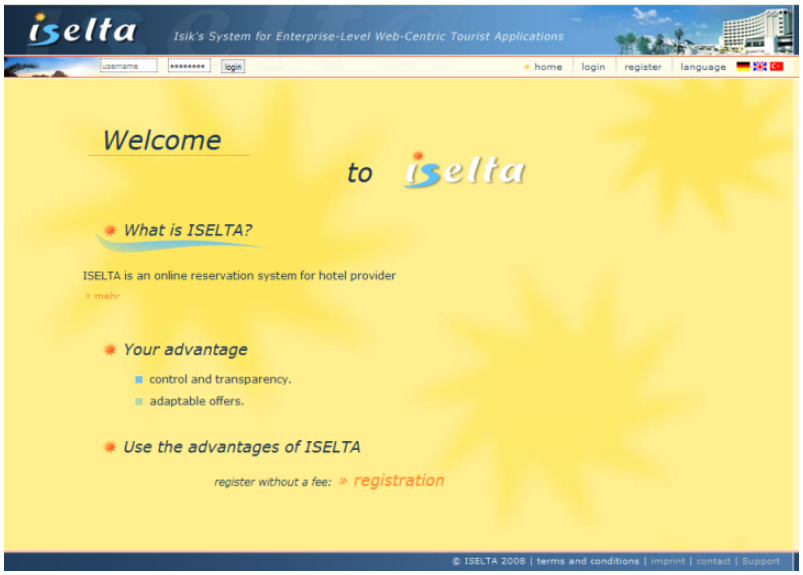


Figure 6.1: Screenshot of ISELTA

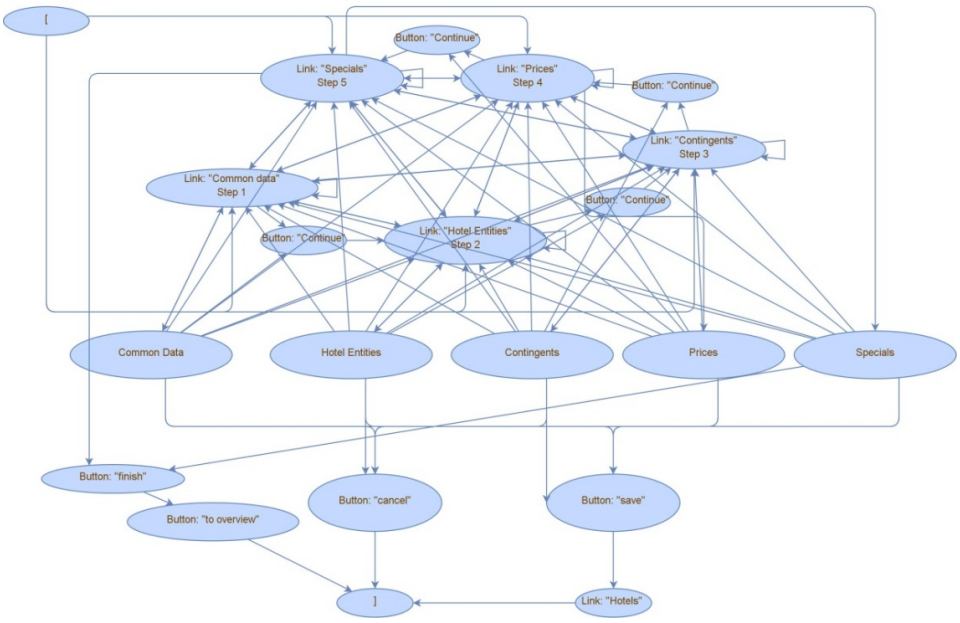


Figure 6.2: ESG modeling the change of hotel data

Special attention will be given to analyzing the impact of test sequence length on the fault detection capability [21, 16]. In addition, the question of how the structure of the model, specifically model refinement, is able to contribute to the fault detection will be clarified. A general concern is the influence of each of the strategies on the overall reliability. A comparison of the ESG approach with random testing is refrained because of previous work [14] that has already demonstrated the overall effectiveness of ESG approach. Instead, this thesis compares the full resolution and the layer-centric approach with the novel selective layer-centric approach to show the advantages and disadvantages of the both approaches. The goal is to find out what kind of faults can be detected by our approach and how the detectability changes with varying the test generation method (FR, LC, SLC).

According to these goals, the sub-system *hotel administration* of ISELTA with the following structure has been selected for performing the case study.

- *Hotel administration* forms a hierarchy of components represented by 7 ESGs with a total of 73 vertices and 207 edges.
- More than 60,000 tests have been generated and executed.
- 3 sets of tests each, varying the length of ES to be covered (2, 3, 4), have been generated for both the FR and LC approaches.

The chosen sub-system is independent of the others so that it can be viewed as a system on its own. Thus, SUC is an impartially-chosen system. One of the 7 ESGs is given in Figure 6.2 (the full set of ESGs of *hotel administration* is given in Appendix B).

6.2 Test Execution and Tool Support

One of the primary goals of a model-based test process is to automate test generation and execution. Figure 6.3 shows the overview of the automated test process performed in this study. On the basis of a given specification, the ESGs have been set up using a newly developed test tool called *Test Suite Designer* (TSD). For test execution, another tool comes into account, which is able to evaluate the generated

test cases by executing them against the SUC automatically. The result is a list of passed and failed test cases. Further details on the tools supporting the test process are given in the following.

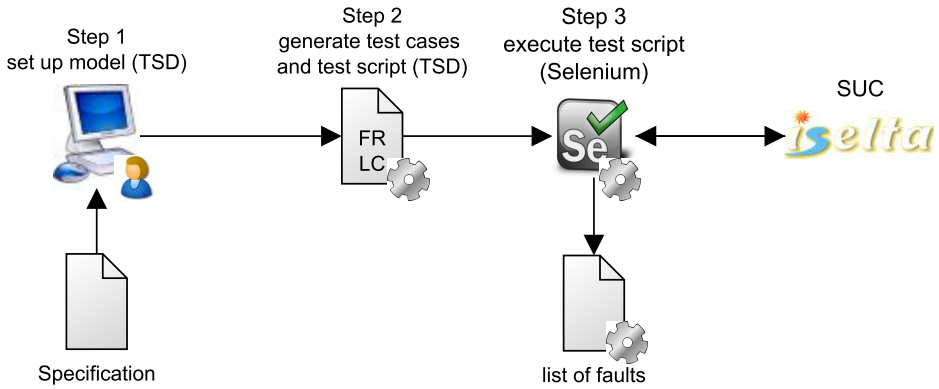


Figure 6.3: Automated test process for testing with ESGs

6.2.1 Modeling & Test Generation: Test Suite Designer

To support the case study, the algorithms created for the LC strategy (see Chapter 4) as well as the SLC strategy (see Chapter 5) have been implemented and integrated in the TSD, which is written in Java®. The TSD automates the following steps:

- the modeling of hierarchical components by ESGs via GUI,
- CES and FCES generation following FR, LC or SLC, and
- the test script generation for test execution.

The GUI for modeling ESGs can be seen on the right side of Figure 6.4. The modeling is supported by the Java Graph Visualization Library offered by JGraph Ltd [60]. An outline of the hierarchical structure of an ESG can be seen in Figure 6.4 on the left side. The outline of the structure can also be used to navigate through the ESG given.

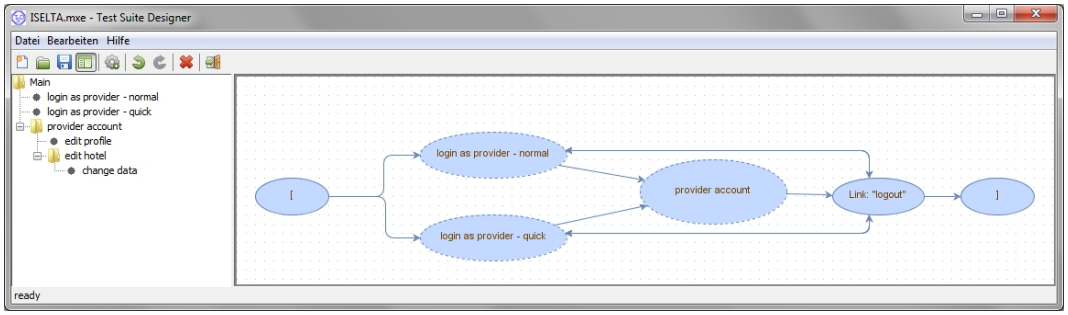
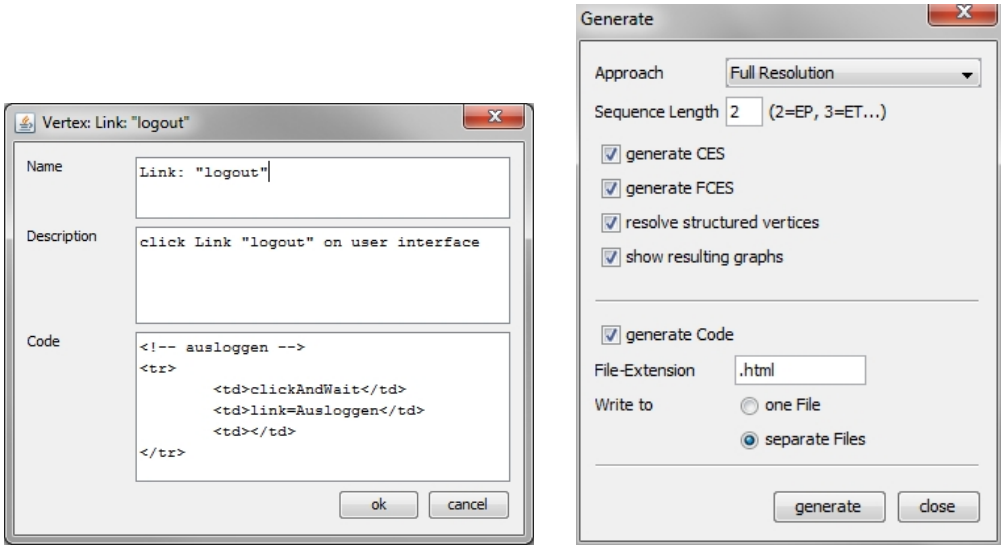


Figure 6.4: Screenshot of Test Suite Designer

Vertex annotations of constructed ESGs refer to source code executing the underlying event within a separate test execution environment (see Section 6.2.2). This enables the automatic generation of test scripts along the calculated test sequences. These test scripts can be loaded into a separate test execution environment. The assumption is that the code snippet can identify the "right" object. Just the name will of course not be enough. Here, object-IDs and more techniques are used to identify objects reliably. In the end, there was no manual conversion necessary to execute the tests which is also one of the strength of the approach. Furthermore, the test oracle defined in Section 3.2 is easily adaptable to test tools like the one used in this study which is described in the next section.

Figure 6.5a shows the dialog for adding the source code to a single event. Figure 6.5b shows the test generation dialog, where the approach for test generation can be selected, and whether test scripts shall be created for test execution. Visit Appendix A for a formal description of the algorithms and models used in the case study.

The linear system of equations 4.1 to 4.7 is solved using the open source *lp_solve* library [69]. An implementation on the basis of the *GNU Linear Programming Kit* (GLPK) [45] also exists but it turned out that the library could not reliably provide a valid solution. This is due to the fact that the solution contained floating point values between 0 and 1 in some cases although the domain of the variables has been set to binary (that is, 0 or 1). But when a valid solution was generated, GLPK was much faster compared to *lp_solve*.



(a) Entities dialog of an event (b) Test generation dialog

Figure 6.5: Screenshots of dialogs in TSD

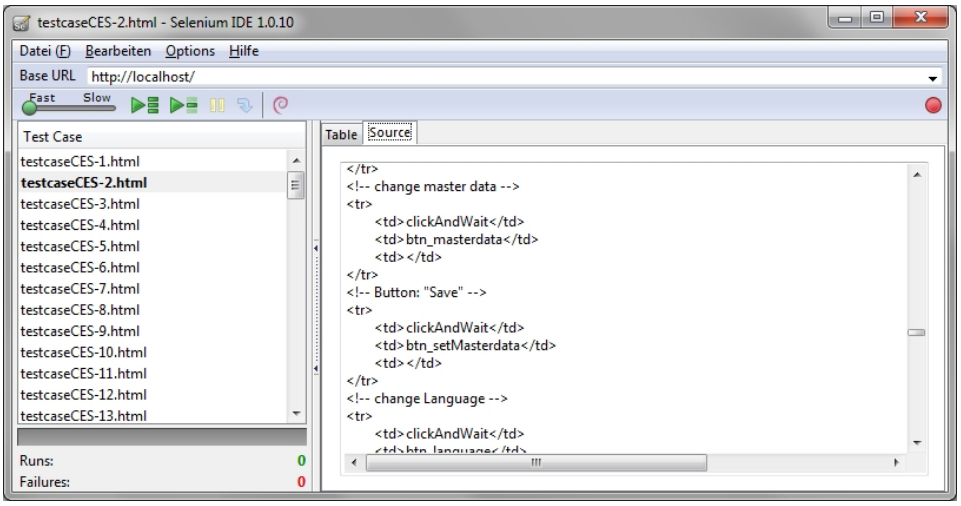


Figure 6.6: Screenshot of Selenium IDE

6.2.2 Test Execution

TSD allows the generation of test scripts along the generated CESs and FCESs. For automated test execution, any tools can be used that support test scripts for running tests against the given SUC.

The freely available web test tool Selenium [107] has been developed for testing web applications. The user interface of Selenium can be seen in Figure 6.6. The right side of Figure 6.6 shows an excerpt of the generated Selenium test script. Thanks to the fact that test scripts can be stored in an HTML text file format, test script files can also be opened and viewed in a web browser.

6.3 Results and Their Analysis for Identifying the Critical Sub-Layers

6.3.1 Results

Based on TSD, CESs and FCESs for positive and negative testing have been generated and executed along LC and FR covering event sequences of length 2, 3, and 4. The results of the tests of this case study are summarized in Table 6.1. It can be seen that the number of CESs in the LC approach is lower than the number of CESs of the conventional FR approach.

Table 6.1: Positive and negative tests subject to ES length

	FR				**LC**			
length	**CES**	**FCES**	$\sum$	**faults**	**CES**	**FCES**	$\sum$	**faults**
2	5	2663	2668	35	2	585	587	29
3	22	9474	9496	35+4	4	1735	1739	29+2
4	93	38768	38861	39+0	6	7005	7011	31+0
$\sum$	120	50904	51024	39	12	9325	9337	31

To enable a more detailed comparison of the test effort reduction, the number of events to be executed according to each approach is shown in Table 6.2. In total, the new approach, LC strategy, reduced the number of events by 78%. Surprising is the

fact that this effort (~20% of the original test effort) was already able to detect 80 % of the faults! 31 faults were detected using the LC approach and 39 faults using the FR approach. Thus, LC detected 20 % less faults. However, the LC approach reduced the test effort by about 80 %. Test execution effort reflected this saving which was reduced from 4 days to round about one day. The data (in Table 6.2) show that much of the 74/78 % saving is accounted for by reduction in the FCES events. The reduction in CES events is much less: it ranges from 3 % to 10 %. However, it was observed that this saving correlates to the detected faults. It is worth noting that these faults are real faults of the system with respect to the model. They are not hand-seeded, or result from mutation operators.

Table 6.2: Number of events to be executed

	FR			**LC**			
length	**CES**	**FCES**	$\sum$	**CES**	**FCES**	$\sum$	**saving**
2	353	17504	17857	341	4370	4711	74 %
3	1589	73053	74642	1430	15150	16580	78 %
4	9580	343969	353549	8710	70268	78978	78 %
$\sum$	11522	434526	446048	10481	89788	100269	78 %

Figure 6.7 shows the number of test cases for event sequences of increasing length. Figure 6.8 shows this on a logarithmic scale. The fact that the curve forms an almost straight line reflects the circumstance that the number of test cases grows exponentially with increasing length. The same holds for the number of events (not shown).

6.3.2 Comparing FR with LC

The reliability determination process of the experiment for comparing FR with LC is carried out in five steps (see Section 4.3).

Step 1: Determine Testing Time and Type of Fault Data

As no time units are given, the cumulative number of test cases (instead of points in time) and number of faults detected by event sequences of length 2, 3, and 4

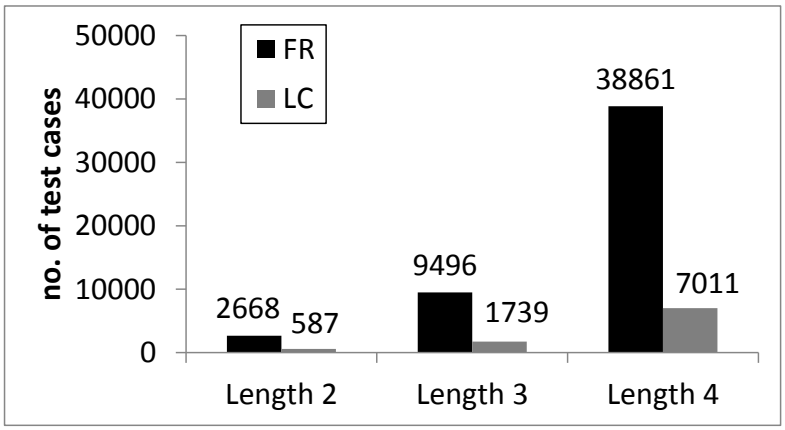


Figure 6.7: Number of test cases

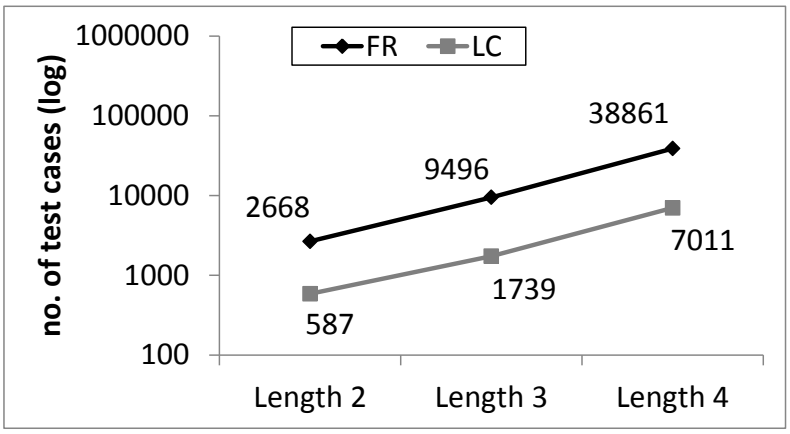


Figure 6.8: Number of test cases on a logarithmic scale

are used for the reliability determination of software (instead of number of faults detected in a time interval). According to Table 6.1, the grouped data consisted of cumulative number of test cases and has three groups, which are defined by: $[0, 587[$, $[587, 2326[$, $[2326, 9337[$; that is, "times" $t_1 = 587$, $t_2 = 2326$, $t_3 = 9337$ with $y_1 = 29$, $y_2 = 2$, and $y_3 = 0$ as faults per interval. Note that t_2 is calculated by $t_2 = t_1 + number\,of\,test\,cases\,covering\,length\,3$, that is, $t_2 = 587 + 1739$ and so on.

Step 2: Analyze the Statistical Properties

To decide whether or not Poisson type models can be used in this study, a K-S test is performed with following hypotheses.

H_0: Cumulative number of faults forms Poisson distribution.

H_1: Cumulative number of faults does not form Poisson distribution.

The analysis of the results of the K-S test (Table 6.3) indicates that the cumulative number of faults follows Poisson distribution with a mean parameter = 30.333 since p-value (0.709) is greater than 0.05.

Table 6.3: One-sample Kolmogorov-Smirnov test

	Cumulative Number of Faults
Poisson Parameter Mean	30.333
Kolmogorov-Smirnov Z	0.707
p-value (2-tailed)	0.709

Step 3: Select Parameter Estimation Technique

Figure 6.9 shows predictions of mean-value and 95% confidence intervals for each SRGM at time t_3. G-O and D-S models gave an exact prediction for the cumulative number of faults at time t_3. In contrast, M-O model performed worse, estimating approximately 4 additional faults until time t_3 by executing 7082 test cases covering length 4; perhaps because the failure intensity does not decrease exponentially with the expected number of faults experienced for this data.

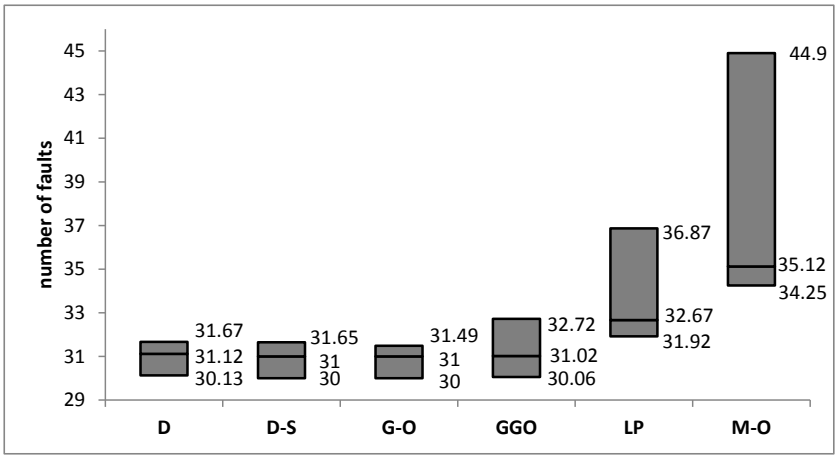


Figure 6.9: Predictions of mean-value and 95 % confidence intervals at time t_3 (thus, for length 4 testing)

Step 4: Calculate Goodness of Fit (GoF) Measures

GoF measures are summarized in Figure 6.10.

Step 5: Select Best Model

G-O gives the best estimation in all GoF measures since it has the smallest AIC, BIC, and MSE values ("the smaller the better"). To compare the reliability of the LC approach with the FR approach, the reliability of the FR approach has also been calculated as $R_{FR} = 0,99870$ and summarized in Table 6.4 for each reliability model, indicating that there is no significant difference between the reliability of FR ($R_{FR} = 0.99876$) and LC ($R_{LC} = 0,99940$) approach.

Table 6.4: Reliability measures

Models	GO	D-S	D	GGO	LP	MO
R_{LC}	0.99940	0.99998	0.8884	0.6092	0.1761	0.00479
R_{FR}	0.99876	0.99997	0.99997	0.63186	0.93659	0.03375

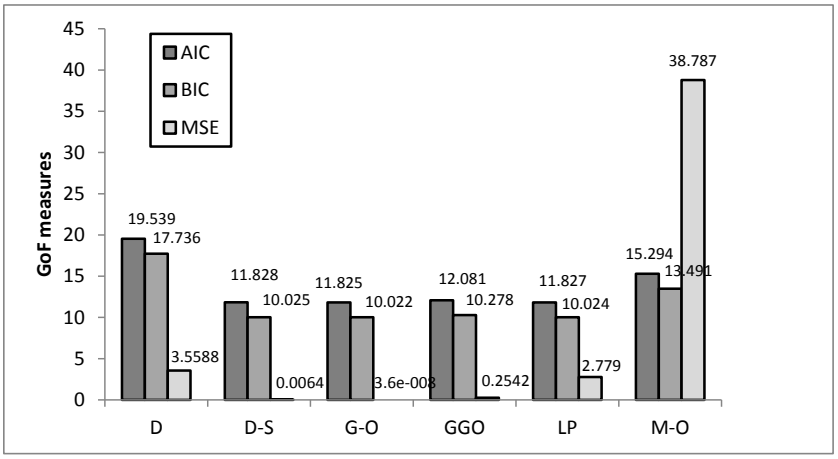


Figure 6.10: GoF measures

Results in a Nutshell

In the case study, 31 faults were detected using the LC approach and 39 faults using the FR approach. Thus, LC detected 20% less faults. However, the new approach reduced the test effort by about 80%. The follow-up reliability analysis found that the LC approach leads to an even slightly better reliability level than the FR approach.

Regarding the test sequence length, the results of the analysis were surprising: Test cases covering event sequences of length 2 detected most of the faults. Test sequences of length 3 contributed very little, and execution of test sequences longer than 4 makes no sense as they obviously have no or only very minor chances of detecting any new faults with respect to the measured reliability level. In the end, the fault detection of test suites covering sequence length 3 and 4 stayed far behind expectation.

6.3.3 Comparing FR and LC with SLC - Identifying the Critical Sub-Layers for Further Testing

In both strategies, LC and FR, testing with higher event sequence length leads to a great deal of additional test effort while detecting fewer faults. The question

that arises now is whether or not it would have been possible to achieve a similar reliability level as in LC and FR with less testing effort. To answer this question, the three steps to be performed for SLC strategy (Chapter 5) will be carried out.

Step 1: Perform Layer-centric Testing and Categorize Detected Faults.

Table 6.5 shows the results of LC testing for each component. The resulting test case set has been analyzed and the events occurring in the resulting test case set have been counted for each of the components. Furthermore, the faults have been categorized according to the components. It is assumed now that only the CESs and FCESs covering sequence length 2 based on LC testing have been generated and executed.

Table 6.5: The number of faults and the number of events categorized according to the components

	ESG 1 main			ESG 2 login normal			ESG 3 login quick			ESG 4 prov. account			ESG 5 edit profile			ESG 6 edit hotel			ESG 7 change data		
	events			events			events			events			events			events			events		
length	CES	FCES	faults	CES	FCES	faults	CES	FCES	faults	CES	FCES	faults	CES	FCES	faults	CES	FCES	faults	CES	FCES	faults
2	6	4	1	5	14	0	24	1163	0	50	1254	7	63	361	9	56	669	1	137	905	11
3	25	13	1	12	24	0	120	3484	1	94	3864	7	161	1354	9	153	2385	1	865	4026	12
4	89	34	1	22	50	0	560	14184	1	290	14957	7	543	4241	9	911	11565	1	6295	25237	12

Step 2: Select Layers for Further Testing

As mentioned in Section 5.2, the first step to identify a subset of components which have a higher fault detection capability is to calculate usage ratio of each component (UR). Then the reliability of each component (RE) and impact of components (IE) on overall system reliability are determined.

Step 2.1: Calculating the Usage Ratio of Components Equation 5.1 is used to calculate UR. According to Table 6.6, the component with the highest usage is represented by ESG 4. The component represented by ESG 1 has the lowest usage.

Table 6.6: Usage ratio of components/ESGs

ESG	ESG 1	ESG 2	ESG 3	ESG 4	ESG 5	ESG 6	ESG 7
URE	0.00212	0.004	0.252	0.2768	0.09	0.154	0.2212

Step 2.2: Calculating Reliability of each Component To determine the fault data used for calculating the reliability of each component, first the number of faults detected during the test process and the corresponding number of events in case of length 2 are sorted in descending order according to their URs (Figure 6.11).

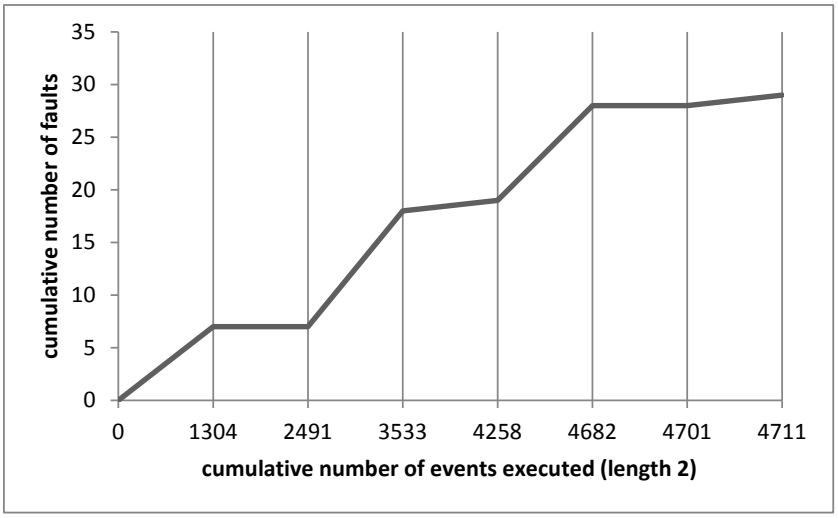


Figure 6.11: Fault data used to calculate the reliability of each component

To decide whether or not Poisson type models can be used in this study, a K-S test is performed with following hypotheses.

H_0: Cumulative number of faults forms Poisson distribution.

H_1: Cumulative number of faults does not form Poisson distribution.

The analysis of the results of the K-S test (Table 6.7) indicates that the cumulative number of faults follows Poisson distribution (mean parameter = 19.4286) since p-value (0.239) is greater than 0.05.

The next step is to apply the NHPP models given in Table 4.4 (on page 42) to the fault data given in Figure 6.11 and to compute GoF measures for each SRGM to determine the best fitting model. Figure 6.12 shows the results of the GoF measures.

Table 6.7: One-Sample Kolmogorov-Smirnov Test

	Cumulative Number of Faults
Poisson Parameter Mean	19.4286
Kolmogorov-Smirnov Z	1.030
p-value (2-tailed)	0.239

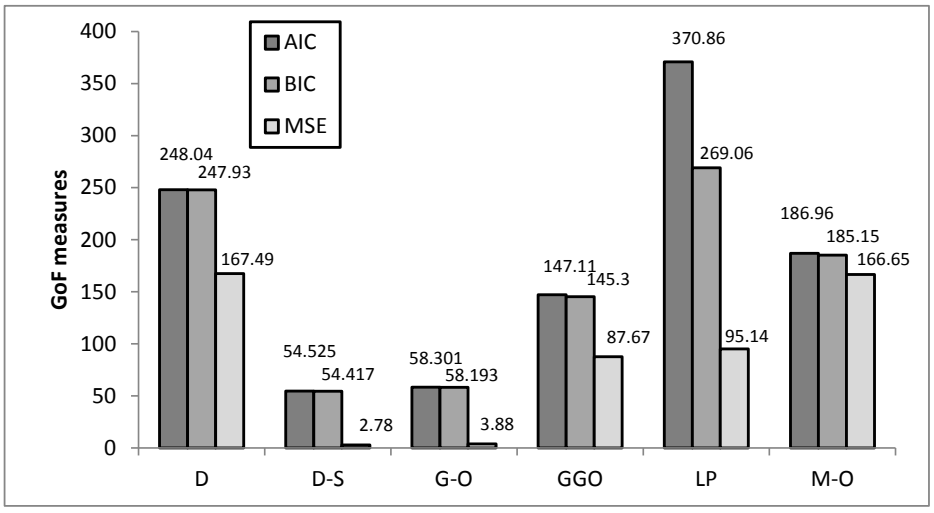


Figure 6.12: GoF measures

It can be seen from Figure 6.12 that D-S model provides the best performance in all GoF measures since it has the smallest AIC, BIC, and MSE values. Therefore, the D-S model has been used to calculate the reliability of each component in this study. Table 6.8 shows reliability results of each component and combined reliability (R_c).

Table 6.8: Reliability results of each component and R_c

ESG	ESG 1	ESG 2	ESG 3	ESG 4	ESG 5	ESG 6	ESG 7	$\mathbf{R_c}$
RE	0.9269	0.9268	0.9329	0.9526	0.9268	0.9261	0.9266	**0.9354**

As can be seen in Table 6.8, R_c (0.9354) is lower than R_{LC} (0.99940) and R_{FR} (0.99876) that have been determined in Section 6.3.2. The goal now is to enhance

R_c. This will be done by performing SLC testing with higher length for components that have a small IE value compared to the overall system reliability.

Step 2.3: Calculating Impact of Components on Overall System Reliability Table 6.9 shows the sorted IE values of components on the overall system reliability derived in line with equation 5.4.

Table 6.9: Impact of each component on overall system reliability

ESG	Sorted - IE
ESG 3	**0.739**
ESG 7	**0.75 (Quartile)**
ESG 4	0.8
ESG 6	0.82
ESG 5	0.9
ESG 2	0.995
ESG 1	0.997

Step 3: Re-execute Layer-centric Testing by Increasing the Sequence Length for the Critical Layers Only

SLC testing with higher length is performed for ESG 3 and ESG 7 since IE values of these components are equal to or less than the 1st quartile of IE values, which is calculated as $IE_p = 0.75$ with $p = (n + 1)/4 = (7 + 1)/4 = 2$.

In order to demonstrate that this selection is adequate in terms of reliability, SLC testing with length 3 was first performed for all ESGs and R_c was calculated. Next data related to components was removed step by step from calculating R_c and then R_c was re-calculated at every turn.

Table 6.10 shows the new combined reliabilities and changes in model parameters CP, indicating that removing ESG 7 has caused a sudden decrease in R_c. When looking at Table 6.10 from the bottom-up, it can be seen that R_c has increased from 0.9353 to 0.985 as a result of SLC testing with length 3 for ESG 3 and ESG 7. Moreover, maximum changes in a (representing the expected number of faults to

be detected) and b (representing the fault detection rate) parameters occur when removing ESG 7 and ESG 3.

Table 6.10: The new combined reliabilities calculated by removing ESGs step by step and changes in model parameters

ESG	R_c	D-S Model Parameters		CP	
		a	b	a	b
All ESGs (equals LC-Testing with length 2+3)	0.998	31	0.0006	0	%36.84
Not ESG1	0.998	31	0.0006	0	%36.84
Not ESG1+ESG2	0.998	31	0.0006	0	%36.84
Not ESG1+ESG2+ESG5	0.996	31	0.0006	0	%36.84
Not ESG1+ESG2+ESG5+ESG6	0.994	31	0.0006	0	%36.84
Not ESG1+ESG2+ESG5+ESG6 +ESG4	0.985	31.2	0.0006	%0.65	%36.84
Not ESG1+ESG2+ESG5+ESG6 +ESG4+ESG7	0.9390	75.07	0.00026	%142.16	%72.63
No ESG (equals LC-Testing with length 2)	0.9353	84.65	0.00025	%173.0645	%73.68
LC Testing (length 2+3+4)	0.9999	31.00	0.00095		

Results in a Nutshell

When performing SLC testing with length 3 for only ESG 3 and ESG 7, SLC reached a reliability level close to the ones achieved by LC and FR with one difference: the test effort could be reduced even further by approximately 30% with length 3 testing for SLC when directly compared to length 3 testing for LC (see Table 6.11). Compared to FR testing, the test effort has even been reduced by 84%.

Table 6.11: Effort comparison of LC and SLC (length 3 testing)

length 3	**LC (length 3)**	**SLC (length 3; ESG 3+7)**	**saving**
# events (CES)	1430	1012	29 %
# events (FCES)	15150	10634	30 %
total	16580	11646	30 %

6.4 Limitations and Threats to Validity

Layer-centric testing (LC) has been introduced to reduce the costs of test case generation and test execution for large hierarchical models. The novel SLC strategy introduced in this thesis brings further valuable steering and cost reduction capabilities for the test process. The results of the case study have been far above expectations. However, there are also some limitations and threats to validity that should be mentioned.

In general, while model-based testing a system, the tester assumes that the underlying model is correct and complete with respect to the entities considered. The same holds for the given case study. By analyzing ESG models, merely faults on events and their order could be detected. Other types of faults, the ones likely in database interactions, for example, are usually not within the scope of this testing, but they might be detected by chance. Data dependencies are also not considered. Usually, SUC which is modeled is more complex than the model and events may have complex side-effects. The focus of the approach is on testing ESGs for detecting faults in sequences of events. However, test data influence test sequences, and thus events may have complex side-effects. In order to compare the true fault detection between the FR, LC and SLC approach, the influence of test data is needed to be eliminated. Thus, the approach creates test data in a way that they have no influence on the detected faults. This can be seen as oversimplifying and thus restrictive; however, the number and severity of the faults detected indicates the strength of the approach (see [14, 10] for further examples of the fault detection effectiveness of ESG approach).

Another concern is about the transferability of the results achieved here to other systems or models since these results heavily depend on the given SUC, its development process, and on many more factors. Furthermore, only one large component of one system, namely ISELTA, is experimentally tested. In addition, reliability estimations depend heavily on the given fault data. Thus, the reliability growth models that are applicable to a given data set might not turn out to be applicable to another data set.

Therefore, the next part of this thesis will address these issues by applying the approach to web service compositions. However, the reliability estimations as per-

formed in this chapter clearly demonstrate the applicability of these models to the given case study. Hence, the reliability analysis should not be left out; it also provides a reasonable indicator for determining the point in time when to stop testing.

Part III

Applying the Approach to Web Service Compositions

Chapter 7

Extending the Approach

Technical and enterprise applications have become more and more complex since they have to cope with strict requirements, such as of business processes and their dynamic evolution, and interaction among different systems. Accordingly, the architecture for integrated enterprise applications has to provide interoperability, scalability, and rapid development. The adoption of technologies that foster the interoperability between different applications has been a recurring solution. *Service-oriented architectures* (SOA) and web services have been developed to enable loosely-coupled, distributed applications by using independent and self-contained services. These services can be combined in a workflow that characterizes a new, *composite* service. The resulting composite service is also called as *web service composition* (WSC). Apart from the adoption of SOA and web services, the use of a service bus to ease the integration process has been advocated for service-oriented applications [106, 61, 95]. The so-called *enterprise service bus* (ESB) controls, routes, and translates messages exchanged by the services involved [95].

SOA testing has gained much attention as a method of ensuring the delivery of high quality and robust service-oriented applications [31]. WSC testing plays an important role in this context [73, 23, 90, 24, 124, 32, 59] because the behavior of the composite services now depends not only on the WSC itself but also on the integrated services, which complicates the testing process. The WSC can present complex communications among the integrated services in which missing or unexpected messages can lead to a failure. Furthermore, the composition may fail due to

undesirable behavior of partner services, such as corrupted messages, unavailable servers, and long timeouts.

Based on [15], this chapter introduces a new approach called *ESG for web service compositions* (ESG4WSC) to generate cost-effective test cases for WSCs. An event-oriented approach is proposed for several reasons. First, message exchanges in a WSC can be viewed as events that follow an order. Second, artifacts (e.g., standardized service descriptions) need not be available to create an ESG or any other model for a service composition. This means that an ESG can easily be constructed in an ad-hoc way by the tester wherever no model is available. It is assumed that the tester can observe and modify the exchanged messages using an ESB (present in several SOAs). The service composition is considered to be a black box, but the tester has control over messages exchanged with the partner services.

To sum up, an event-based approach is proposed to support WSC testing by

- extending the basic notions of ESG (Chapter 3), referred to as ESG4WSC, for testing the WSC behavior in desirable situations (positive testing) and undesirable situations (negative testing) based on one model;
- introducing the concept of "sensitive" events as test oracle for negative test cases;
- introducing new scalable algorithms to generate positive and negative test cases from ESG4WSC;
- enabling the independent modeling of artifacts such as BPEL while allowing simultaneous modeling and implementation;
- enabling to model independent of the type of composition, that is, orchestration or choreography.

To the author's best knowledge, there is no comparable work that performs testing of WSC by modeling and testing not only the interface, which is available to the consumer, but also the internal communication with other services, which is usually hidden from the consumer. Furthermore, the ESG4WSC approach is *holistic,* since it considers positive and negative testing at the same time [9].

The remainder of this chapter is organized as follows. Section 7.1 briefly presents concepts of SOA, web services, and ESBs. In addition, it presents related work and introduces a running example for the explanation of the approach. Section 7.2 introduces the proposed holistic approach to model service compositions. Finally, Section 7.3 presents the underlying testing process for testing service compositions.

7.1 Background, Related Work and Running Example

7.1.1 Background

The information technology landscape of enterprises is mostly heterogeneous, complicating the integration of systems implemented by different technologies. SOA has been introduced to fill this gap and provides a *de facto* standard enabling communication among those systems. SOA is an emerging approach that aims to foster loose coupling among applications. It provides a standardized, distributed, and protocol-independent computing paradigm. Software resources are wrapped as "services" that are well-defined and self-contained modules providing business functionality and are independent from other service states or contexts [95]. Usually, implementation details of a service are hidden and only its interface is available; that is, a service can be viewed as a black box.

A SOA is based on three entities: provider, consumer, and registry. Using the web services technology, a SOA is realized through three main XML standards: WSDL [121], UDDI [91], and SOAP [120]. WSDL (Web Service Description Language) is a W3C standard used to describe the service interface, including details such as operations, data types, and adopted protocols. UDDI (Universal Description, Discovery and Integration) is an OASIS standard that defines a set of functionalities to support description and discovery of services. SOAP (originally for "Simple Object Access Protocol", but no longer an acronym [120]) is also a W3C protocol used to define the structure of messages exchanged among the services. Using SOAP, it is possible to define error messages by using *SOAP Faults*. SOAP Faults are expected by the consumer if they are specified in the WSDL interface

and used to map exceptions that happen within the service. However, the web service frameworks also tend to launch SOAP Faults when internal exceptions are not handled correctly. In this case, it is said that the SOAP Fault is unexpected.

A group of services can be assembled to create a new value-added service via composition. In a service composition, many services can be combined in a workflow to model and execute complex business processes. The services involved in a service composition are usually called *partner services.* Service compositions can be developed either as *orchestration* or as *choreography*. In service orchestration, there is a main entity that is responsible for coordinating the partner services. Currently, the most widespread language used to implement a service orchestration is BPEL [92]. In service choreography, there is no control entity and all partner services work cooperatively to achieve an agreed objective. There are several languages used to describe service choreography, e.g., WS-CDL [119] and WSCI [118]. The service composition is also a service (referred to as a *composite service*) and can be reused by other services. A service that is not a composition is usually called an *atomic* service. The terms *service* and *web service* are used as synonyms in this thesis.

An ESB, which is an intermediate layer among the services, can also be included in a SOA. The ESB works as a backbone that uses web services technology to support many communication patterns over different transport protocols and provides interesting capabilities for service-oriented applications, such as routing, provisioning, service management, integrity, and security [95]. The adoption of ESBs has been considered essential for companies to gain the full advantages provided by SOAs [61]. An ESB enables high interoperability and eases the distribution of business processes using different platforms and technologies [61]. Figure 7.1 presents the transition from a traditional SOA to an ESB-based architecture. According to Schmidt et al. [106], the ESB is an infrastructure that fully supports an integrated and flexible SOA. These features are achieved by receiving, operating, or mediating on the service messages as they flow through the bus. There are many possible uses for mediation, such as load balance, monitoring, and validation. Schmidt et al. [106] describe a set of mediation patterns that are useful in an ESB. In the context of this work, two patterns were used.

- *Monitor pattern* provides the feature for observing messages that pass through

the ESB, not applying any type of change in the messages. This pattern can be applied to logging, audition, monitoring of service levels, measurement of client usages, and so on.

- *Aggregator pattern* provides the feature for monitoring messages from different services over a period of time and generating new messages or events. This pattern can be useful for realizing complex scenarios so that, e.g., a set of events can be mapped to a single event.

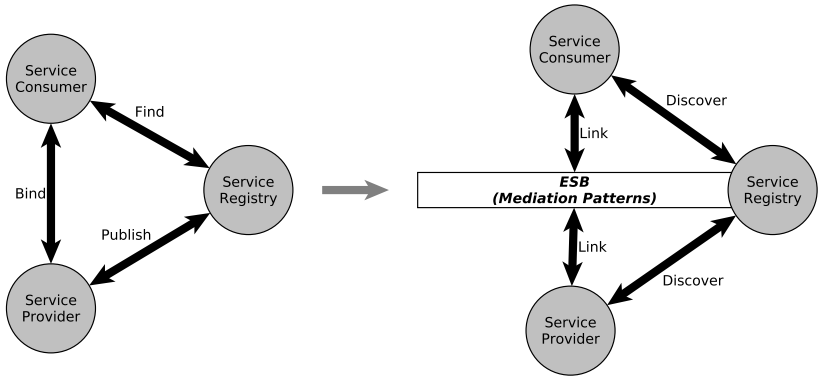


Figure 7.1: ESB representation in a SOA (adapted from [106])

The ESB can be provided by software that implements the concepts of those functionalities. Several ESB applications, from proprietary vendors to open-source solutions, are available [42]. In this work, the open-source version of Mule-ESB [81] is adopted, which is a lightweight Java-based ESB that includes much functionality to integrate existing systems.

7.1.2 Related Work

SOA testing has been studied intensively in recent years [31, 29, 103], with a particular effort on formal testing approaches (for a systematic review of the literature, see [41]).

Benharref et al. [23] propose a multi-observer architecture to detect and locate faults in composite web services. The proposed architecture is composed by

a global observer and local observers that cooperate to collect and manage faults found in the composite service. Mei et al. [73] propose a model to describe a service choreography that manipulates data flow by means of XPath queries. In a choreography, XPath queries can handle different XML schema files. XPath expressions are represented using *XPath rewriting graphs* (XRGs). Based on *labeled transition systems* (LTS), the *LTS-based choreography model* (C-LTS) is proposed with XRGs attached in transitions that represent service invocations. New types of definition-use associations are proposed and test adequacy criteria are presented. The approach introduced in this thesis is focused on test case generation, which is not the subject of the work of Mei et al. Thus, both approaches are complementary since the coverage criteria can be used to give more information about the test suite generated by the ESG4WSC approach.

Transforming composition specifications (such as BPEL and WS-CDL) into formal models to support test case generation has also been researched. Bentakouk et al. [24] propose a mapping from BPEL to symbolic transition systems. Test cases are generated using symbolic execution and applied to the SUC using online testing. Hou et al. [56] model a BPEL program using message sequence graphs and generate message sequences. In an extended version [90], the authors formalize the approach and make an experimental comparison with two other techniques. The present thesis differs from [24, 56, 90] concerning the available artifacts. It is not assumed that the composition was developed using BPEL and the tester has access to this artifact. Although the ESG4WSC approach requires more effort to develop the test model, the SUC is verified from a different black-box point of view.

Wieczorek et al. [124] present *message choreography models* (MCM) for model-based integration testing of service choreographies where test cases are generated using model checking. In [125], the authors present a case study about the application of this MBT approach to test service choreographies in a real-world project. MCM models were designed to support the tests. Wieczorek et al.'s work is close to the ESG4WSC approach, with two main differences: (i) ESG4WSCs are more abstract models compared to MCMs, which form a domain-specific language created to design service choreography; (ii) the ESG4WSC approach also performs the tests using a test execution environment based on ESBs, which can be adapted to the MCM approach.

The negative testing included in this thesis can be associated with fault injection techniques for WSCs. Chan et al. [33] describe a fault taxonomy for WSCs. The authors identified a set of fault classes that are classified into physical, development and interaction faults. In a matrix, these faults are related to six elementary fault classes to explain observed effects in the composition. This taxonomy has been used in fault injection for WSCs. However, Chan et al. do not describe how to systematically detect the faults related to the different fault classes.

Cavalli et al. [32] propose the framework WebMov, which is composed by a methodology and tools for modeling, validation, and testing of WSCs. It is mainly based on variations of timed extended finite state machines to model BPEL compositions. The methodology also includes fault injection to test the robustness of the composition. Ilieva et al. [59] propose a similar framework, named TASSA, for robustness testing of BPEL orchestrations using fault injection mechanisms. Both papers approach the use of fault injection techniques for negative/robustness testing of BPEL compositions. Nevertheless, they do not deal with faulty message sequences as tested by the ESG4WSC approach.

To sum up, the ESG4WSC approach differs from existing approaches by combining positive and negative testing at the same time. Furthermore, the presented approach is independent of artifacts such as BPEL, which are regularly a result of the implementation process. This enables to start the modeling and implementation process at the same time. A further difference is the fact that the introduced model is not restricted to one of the two existing types of composition, that is, orchestration or choreography. It is also worth noting that the presented approach introduces the concept of "sensitive" events as the test oracle; a concept that has not been introduced before to solve the oracle problem.

7.1.3 Running Example

The business process to grant loans called `xLoan` proposed in [24] is the running example used to illustrate the approach over the following sections. Note that this example is not the case study described in Chapter 8, which is a non-trivial, commercial web application.

The example involves three services: `LoanService` (`LS`), `BankService`

(BS), and BlackListInformationService (BLIS). LoanService represents the xLoan business process, whose workflow is implemented using BPEL. It contains three operations: request, cancel, and select. BankService represents the financial agency that approves (or not) loans, providing loan offers to its clients. The operations used in the example are approve, offer, confirm, and cancel. BlackListInformationService provides an operation checkBL to check if a client has debits with some financial organization.

The example is extended to add parallel flow (a common entity of WSCs) in the process by including a new service called CommercialAssociationService (CAS). Similar to BLIS, CAS provides an operation inDebtorsList to check whether a client has debits with some commercial organization. In the extension, both services are supposed to be called in parallel. If the client has debit according to one of them, the client requires bank approval.

7.2 Modeling Web Service Compositions

This section introduces an event-based model, named ESG4WSC, that represents the request and response messages exchanged between services involved in a WSC. Requests regularly require some kind of input data, which has in turn an influence on the returned response. Furthermore, the execution of events can be bound to specific constraints that need to be fulfilled.

Constraint Modeling

Decision tables (DT) are introduced to augment the graph representation with constraint modeling capabilities that enable a systematic evaluation of constraints. DTs are widely employed in information processing and are also traditionally used for testing, e.g., in cause and effect graphs [86]. A DT logically links constraints (“if”) with events (“then”) that are to be triggered, depending on combinations of constraints (“rules”). Therefore, DTs are powerful mechanisms for

- handling sequences of events which depend on constraints; and
- refining data modeling of calls to invoked services [20].

DTs are formally defined as follows.

Definition 7.1 (Decision Table)**.** A (simple/binary) *decision table* $DT = \{C, E, R\}$ represents events that depend on certain constraints where

- C is the nonempty finite set of constraints (conditions) that can be evaluated as either true or false,
- E is the nonempty finite set of events, and
- R is the nonempty finite set of rules, each of which forms a Boolean expression connecting the truth/false configurations of constraints and determines the executable or awaited event.

Definition 7.2 (Rule)**.** Let R be a set of rules as in Definition 7.1 where in a *rule* $R_i \in R$ is defined as $R_i = (C_{True}, C_{False}, E_x)$ with

- $C_{True}, C_{False} \subseteq C$ being the disjoint sets of constraints that have to be evaluated as **true** and as **false**, respectively; and
- $E_x \subseteq E$ being the set of events that should be executable if all constraints $t \in C_{True}$ are resolved to true and all constraints $f \in C_{False}$ are resolved to false. In this work, $|E_x| = 1$ for all rules to avoid non-determinism.

Special care has to be taken in the formation of rules. For anyone interested in a complete coverage of all constraint combinations, the DT should be complete.

Definition 7.3 (Complete)**.** A decision table $DT = (C, E, R)$ is *complete*, if $\mathcal{P}(C) \subseteq S$ with $S = \{x|(x, y, z) \in R\}$ and $\mathcal{P}(C) = \{M|M \subseteq C\}$. $\mathcal{P}(C)$ is also called as the *power set* of C.

To prevent contradictions, a DT should be redundance-free and consistent and is defined as follows.

Definition 7.4 (Redundance-free)**.** A decision table $DT = (C, E, R)$ is *redundance-free* if $S = \emptyset$ with $S = \{x|(x, y, z) \in R \wedge (a, b, c) \in R \backslash (x, y, z) \wedge x = a \wedge z = c\}$.

Definition 7.5 (Consistent)**.** A decision table $DT = (C, E, R)$ is *consistent* if $S = \emptyset$ with $S = \{x|(x, y, z) \in R \wedge (a, b, c) \in R \wedge x = a \wedge z \neq c\}$.

The conclusion that follows from Definition 7.3, 7.4, and 7.5 is that a decision table $DT = (C, E, R)$ is *complete*, *redundance-free*, and *consistent* at the same time if $\mathcal{P}(C) = S$ with $S = \{x|(x, y, z) \in R\}$. Such a decision table has always $2^{|C|}$ rules according to $\mathcal{P}(C)$ where $|\mathcal{P}(C)| = 2^{|C|}$.

Note that under regular circumstances C_{True} and C_{False} partition C, that is, $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$. In certain cases, it is inevitable to have constraints with a *don't care* (denoted as '-' in a DT). In this case, such a constraint is not considered in a rule and is neither in C_{True} nor in C_{False}.

Table 7.1 presents a DT that models the invoking process of operation `checkBL` of `BLIS`. It contains two constraints on input parameter `uniqueID`, three next events (`inBList`, `notInBList`, `SOAPFault`), and three rules (R1, R2, R3). Constraints in bold model possible domains of input parameters and constraints in italics represent additional constraints to that input data. Rules are used to evaluate which is the next event. For instance, R1 means that if `uniqueID` is valid and also in the blacklist (that is, both constraints are true), then the next event (namely, event `inBList` standing for `uniqueID` being in the blacklist) should be completed in less than 200ms. In R3, if `uniqueID` is not valid and the other constraint does not matter (namely, '-'), then the next event is `SOAPFault`. As illustrated in Table 7.1, the events can be extended by time constraints whenever a response is expected within a certain time range.

Table 7.1: A decision table with 3 rules, 2 constraints, and 3 events

		Rules		
	checkBL(uniqueID)	R1	R2	R3
Constr.	**uniqueID is valid**	T	T	F
	uniqueID in Blacklist	T	F	-
Events	inBList < 200ms	✓		
	notinBList < 100ms		✓	
	SOAPFault - invalid identification			✓

Interaction Modeling

DTs are useful to describe constraints, but they are not appropriate for describing WSC interactions. Hence, the ESG notion is extended and combined with DTs in order to consider additional aspects, such as communication, parallel flow, and conditional activities.

Definition 7.6 (ESG4WSC)**.** An event sequence graph for web service compositions $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$ is a directed graph where

- V is a nonempty finite set of vertices (representing events),
- $E \subseteq V \times V$ is a finite set of arcs (edges),
- M is a finite set of refining ESG4WSC models,
- $R \subseteq V \times M$ is a relation that specifies which ESG4WSCs are connected to a refined vertex,
- DT is a set of DTs that refine events according to function f,
- $f : V \rightarrow DT \cup \{\varepsilon\}$ is a function that maps a decision table $dt \in DT$ to a vertex $v \in V$. If $v \in V$ is not associated with a $dt \in DT$, then $f(v) = \varepsilon$, and
- $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$ and $\gamma \in \Gamma$ called entry nodes and exit nodes, respectively, wherein for each $v \in V$ there exists at least one sequence of vertices $\langle \xi, v_0, \ldots, v_k \rangle$ from $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

Definitions 7.7 and 7.8 elaborate Definition 7.6, formalizing the set of vertices and the set of DTs, respectively.

Definition 7.7 (Set V)**.** Let V be as in Definition 7.6. The set of vertices V is then partitioned into V_e, $V_{refined}$, V_{req} and V_{resp}, that is, $V = V_e \cup V_{refined} \cup V_{req} \cup V_{resp}$ and V_e, $V_{refined}$, V_{req} and V_{resp} are pairwise disjoint where

- V_e is a set of generic events;

- $V_{refined} = \{v \in V \mid \exists m \in M \wedge (v, m) \in R\}$ is a set of vertices refined by one or more ESG4WSCs. A refinement with more than one ESG4WSC represents behavior running in parallel.
- V_{req} is a set of vertices modeling a request to its own interface/operations (*public*) or an invoked service (*private*); and
- V_{resp} is a set of responses to a public or private request. Therefore, it is also marked as public or private.

Definition 7.8 (Set DT)**.** Let DT be defined as in Definition 7.6. The set of decision tables DT is then partitioned into DT_{seq} and DT_{input}, where

- DT_{seq} is the set of decision tables that model the execution restrictions for successor events; and
- DT_{input} is the set of decision tables that model constraints for input parameter of invoked operations.

Since WSCs always initiate with one or more request events, the set Ξ contains only vertices $v \in V_{req}$. Just like in ESGs, all $\xi \in \Xi$ are preceded by a pseudo vertex $[\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex $] \notin V$ to mark the entry and exit of an ESG4WSC. For two events $v, v' \in V$, the event v' can follow the execution of v if and only if $(v, v') \in E$. In this case v' is also called *successor* of v and v is called the *predecessor* of v'.

Semantics of the Model and an Example

The semantics of an ESG4WSC is as follows: Any $v \in V$ represents an event, e.g., a request or a response that occurs during the invocation of another service. In general, requests and responses can be *public* or *private*. A public request is to be activated by the tester; that is, it is an operation call to the WSC itself, which is supposed to be sent by a consumer or the tester. A public response is expected to be an answer by the WSC to a public request and therefore should be received by the

consumer/tester. The opposite is true for private requests and responses that represent partner services of the WSC. They are usually not observable by the consumer; however, it is assumed that they are observable by the tester. Private requests are to be monitored by the tester and the tester should check and (if necessary) send back the appropriate response.

Example 7.9. Figure 7.2 represents an ESG4WSC for `xLoan` (Section 7.1.3). Operations `checkBL` and `inDebtorsList` are executed concurrently. Requests are represented by light gray vertices, responses by dark gray vertices. Vertices with a bold line represent public requests and responses. Vertices refined by DTs are double-circled. The corresponding ESG4WSC looks like this:

$$
\begin{aligned}
ESG4WSC &= (V, E, M, R, DT, f, \Xi, \Gamma) \text{ with} \\
V_e &= \{Timeout > 2h\} \\
V_{refined} &= \{check\} \\
V_{req} &= \{LS : requestLoan, BS : approveBank, \\
&\quad BS : offer, LS : cancel, LS : SelectOffer, \\
&\quad BS : cancelBank, BS : confirmBank\} \\
V_{resp} &= \{BS : approved, BS : Notapproved, \\
&\quad LS : notAprovedMSG, BS : Offers, \\
&\quad LS : replyOffers, LS : wrongOffer, \\
&\quad LS : replySelect\} \\
E &= \{(LS : requestLoan, \\
&\quad BS : approveBank), \ldots\} \\
M &= \{M_{BLIS}, M_{CAS}\} \\
R &= \{(check, M_{BLIS}), (check, M_{CAS})\} \\
DT &= DT_{seq} \cup DT_{input} = \{dt_{check}\} \\
&\quad \cup \{dt_{LS:requestLoan}, dt_{BS:approveBank}, \ldots\} \\
f(check) &= dt_{check} \\
f(LS : requestLoan) &= dt_{check} \\
f(BS : approveBank) &= dt_{check} \\
\Xi &= \{LS : requestLoan\} \\
\Gamma &= \{LS : notAprovedMSG, BS : cancelBank, \\
&\quad LS : replySelect\}
\end{aligned}
$$

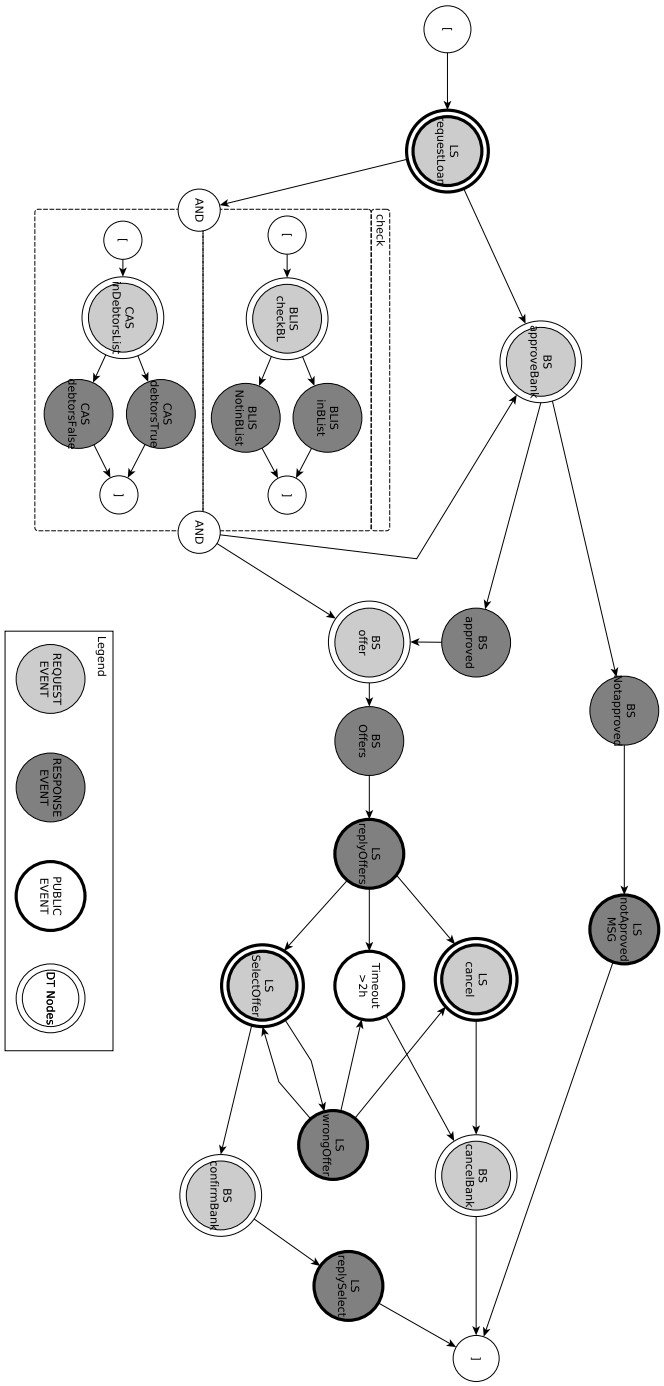


Figure 7.2: ESG4WSC for the `xLoan` example

The ESG4WSC model can be seen as a simplified representation of possible and expected executions of a WSC. It aggregates sequences of events (Definition 3.2 on page 16) which describe walks of execution and message exchanges along the execution. Furthermore, some parts of the execution can be performed in parallel.

Definition 7.10 (Parallel)**.** Two events or ESs a and b that are to be executed in parallel are denoted as $a||b$.

The operator $||$ is commutative, that is, $a||b = b||a$, and associative, that is, $(a||b)\,||c = a||\,(b||c)$.

Example 7.11. For the `xLoan` example given in Figure 7.2, services `CAS` and `BLIS` are to be executed in parallel, e.g., following sequence holds

```
⟨BLIS:checkBL, BLIS:inBList⟩ ||
        ⟨CAS:inDebtorsList, CAS:debtorsTrue⟩
```

7.3 Testing Web Service Compositions

Positive and negative testing can be performed on the basis of the ESG4WSC model introduced in the previous section. Both, positive and negative testing, are described in the following.

7.3.1 Testing the Desirable Behavior

This section introduces the underlying fault model and test process for positive testing a WSC. Furthermore, it is explained how test cases are generated.

7.3.1.1 Fault Model and Test Process

A CES (see Definition 3.2 on page 16) describes a specific execution of a WSC that has to be enforced during testing. Thus, it is expected that precisely those events will be executed in the specified order. According to this, the following faults might occur during the execution.

- There are calls to partner services that are *not defined* in the CES.

- There are *missing* calls to partner services that are defined in the CES.
- The *sequence* of calls is different from the sequence given by the CES.
- The *parameter* of calls to partner services do not correspond to the expected ones.

In order to test a specific CES of the WSC, it is often inevitable that control is taken over the partner services since they communicate with the SUC and the flow of the WSC might depend on a returned response. A simple example for sending back the expected response would be a search operation by a partner service. If the test sequence evaluates a path where an empty result is returned by the partner service, it is required to send back a response to the WSC with no result, even if the invoked service returns a result. However, testing the WSC where a result is returned by the partner service will be part of another test sequence. Thus, taking control of the partner service does not affect the fault uncovering capabilities of the ESG4WSC approach. Considering the lack of controllability of the partner services, Algorithm 7.1 shows the overall test process.

7.3.1.2 Test Case Generation

A test suite is generated that covers at least all EPs again to detect faults in the WSC (due to the fault model described above). Moreover, the cost should be minimal; that is, CESs generated to cover all EPs should have a minimal total length. Unfortunately, the algorithms described in Section 3.3 are not directly applicable. The problem is that refined vertices $v \in V_{refined}$ might contain more than one refining model expressing behavior executed in parallel. Therefore, resolving the hierarchy according to the FR approach (as in Definition 3.4 on page 17) is not possible. A further limitation is that refined vertices can have a DT that restricts the ongoing execution. However, an adapted version of the LC approach called *layer-centric testing for web service composition* (LC4WSC) can help here. The algorithm for deriving CESs from an ESG4WSC is described in following steps.

1. Generate CESs for the refined vertices first (recursive call). If a refined vertex has more than one refining model, then build all combinations of CESs of the single models as parallel sequences.

Algorithm 7.1: Test process for positive testing

```
1  cover all ESs of length k by means of CESs;
2  foreach ces ∈ CES do
       // apply ces to SUC
3      foreach event ∈ ces do
4          if event is public request then
5              send request to the WSC;
6          if event is private request then
7              observe request to invoked service;
8              if call is missing or deviates from the expected one then
9                  mark ces as failed and continue with next ces ∈ CES;
10         if event is public response then
11             observe response from WSC;
12             if response is missing or deviates from the expected one then
13                 mark ces as failed and continue with next ces ∈ CES;
14         if event is private response then
15             observe response from invoked service;
16             send back expected response to WSC if necessary;
17     if all events applicable in the specified order then
18         mark ces as passed;
19     else mark ces as failed;
```

2. Add multiple edges to the ESG4WSC.
   (a) If a refined vertex has a DT restricting the ongoing execution,
      i. identify the valid successor for each CES with respect to the DT, and
      ii. add an edge from the refined vertex to the allowed successor.
   (b) If a refined vertex has not a DT, then add an edge from the refined vertex to the successor (there should be only one) for each CES.
3. Generate CESs according to the CPP algorithm.
4. Replace refined vertices in the resulting CES set with respect to their allowed successors.

Note that Step 2 adds multiple edges to the underlying ESG4WSC; that is, every edge represents a CES of the refined vertex and its valid successor. The benefit of this approach is that the resulting CES set derived in Step 3 contains a sufficient amount of the refined vertex and its corresponding successor so that the CESs of Step 1 can be combined completely with the CESs of Step 3 (recall that every EP/edge is to be covered in Step 3) [65]. Algorithm A.12 along with Table A.2 (in the Appendix), gives a detailed description of the CES generation process. Example 7.12 shows the test generation process for the running example.

Example 7.12. According to Figure 7.2, the test generation process looks as follows.

Step 1: Generate CESs for the Refined Vertices First In this step, the CPP algorithm is applied to the two refining ESG4WSCs in event `check` (Figure 7.2). The event sequences of each ESG4WSC are combined with operator ||. The following sequences for refined vertex `check` have been generated.

```
S1:  ⟨⟨BLIS:checkBL, BLIS:inBList⟩ ||
     ⟨CAS:inDebtorsList, CAS:debtorsTrue⟩⟩
S2:  ⟨⟨BLIS:checkBL, BLIS:inBList⟩ ||
     ⟨CAS:inDebtorsList, CAS:debtorsFalse⟩⟩
```

```
S3:  ⟨⟨BLIS:checkBL, BLIS:NotinBList⟩ ||
     ⟨CAS:inDebtorsList, CAS:debtorsTrue⟩⟩
S4:  ⟨⟨BLIS:checkBL, BLIS:NotinBList⟩ ||
     ⟨CAS:inDebtorsList, CAS:debtorsFalse⟩⟩
```

Step 2: Add Multiple Edges to the ESG4WSC Event `check` has a DT that restricts the execution of its successor events `BS:offer` and `BS:approveBank` in Table 7.2. To cover all rules in this table, the event pair `(check, BS:approveBank)` needs to be covered three times (R1, R2, R3) and the event pair `(check, BS:offer)` once (R4). Thus, the algorithm adds the following edges according to Table 7.2.

- 3 edges `(check, BS:approveBank)` for sequences S1 to S3
- 1 edge `(check, BS:offer)` for sequence S4

An intermediate ESG4WSC is produced (with extra edges added) as illustrated in Figure 7.3.

Table 7.2: Decision table for vertex check of Figure 7.2

dt$_{check}$	**R1**	**R2**	**R3**	**R4**
event: BLIS:inBList happens	T	T	F	F
event: BLIS:NotInBList happens	F	F	T	T
event: CAS:DebtorsTrue happens	T	F	T	F
event: CAS:DebtorsFalse happens	F	T	F	T
BS:offer				✓
BS:approveBank	✓	✓	✓	

Step 3: Generate CESs According to the CPP Algorithm The CPP algorithm is applied on the intermediate ESG4WSC (produced in Step 2) to derive CESs. In this step, the refined events (e.g., `check`) are considered as simple vertices (Figure 7.3). The following CESs have been generated (refined vertices and their successor are emphasized with a bold font).

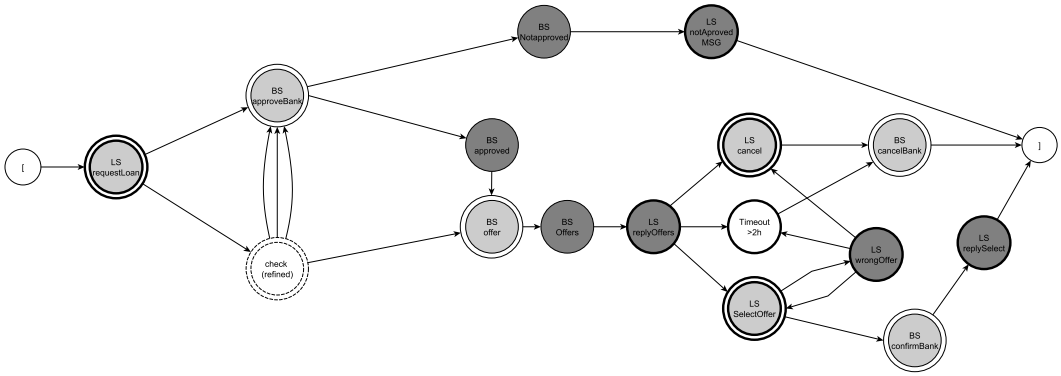


Figure 7.3: ESG4WSC for the `xLoan` example extended by additional edges

CES1: ⟨LS:requestLoan, BS:approveBank, BS:Notapproved, LS:notApprovedMSG⟩

CES2: ⟨LS:requestLoan, check, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:cancel, BS:cancelBank⟩

CES3: ⟨LS:requestLoan, check, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:cancel, BS:cancelBank⟩

CES4: ⟨LS:requestLoan, check, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:SelectOffer, LS:wrongOffer, Timeout > 2h, BS:cancelBank⟩

CES5: ⟨LS:requestLoan, check, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, BS:confirmBank, LS:replySelect⟩

CES6: ⟨LS:requestLoan, check, BS:offer, BS:Offers, LS:replyOffers, Timeout > 2h, BS:cancelBank⟩

Step 4: Replace Refined Vertices in the Resulting CES Set with Respect to Their Allowed Successors Refined events are searched in the CES set generated

in Step 3 and are replaced by the corresponding CESs derived from the refining ESG4WSCs in Step 1. In the final test suite, event pair (check, BS:approve-Bank) is covered exactly three times in CES2, CES3, and CES4 using S1, S2, and S3, respectively, to replace check. Event pair (check, BS:offer) is covered twice in CES5 and CES6. S4 is used in both CESs to replace check.

CES1: ⟨LS:requestLoan, BS:approveBank, BS:Notapproved, LS:notApprovedMSG⟩

CES2: ⟨LS:requestLoan, ⟨⟨BLIS:checkBL, BLIS:inBList⟩ || ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:cancel, BS:cancelBank⟩

CES3: ⟨LS:requestLoan, ⟨⟨BLIS:checkBL, BLIS:inBList⟩ || ⟨CAS:inDebtorsList, CAS:debtorsFalse⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:cancel, BS:cancelBank⟩

CES4: ⟨LS:requestLoan, ⟨⟨BLIS:checkBL, BLIS:NotinBList⟩ || ⟨CAS:inDebtorsList, CAS:debtorsTrue⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:SelectOffer, LS:wrongOffer, Timeout > 2h, BS:cancelBank⟩

CES5: ⟨LS:requestLoan, ⟨⟨BLIS:checkBL, BLIS:NotinBList⟩ || ⟨CAS:inDebtorsList, CAS:debtorsFalse⟩⟩, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, BS:confirmBank, LS:replySelect⟩

CES6: ⟨LS:requestLoan, ⟨⟨BLIS:checkBL, BLIS:NotinBList⟩ || ⟨CAS:inDebtorsList, CAS:debtorsFalse⟩⟩, BS:offer, BS:Offers, LS:replyOffers, Timeout > 2h, BS:cancelBank⟩

Deriving Input Data

After generating CESs, a DT for the initial WSC call is to be evaluated. Algorithm A.12 along with Table A.2 (in the Appendix) gives a detailed description of the CES generation process. The generation of data based on a $dt \in DT_{input}$ is related to the *constraint satisfaction problem* (CSP). A CSP is defined by a set of variables X_1, X_2,...,X_n and a set of constraints, C_1,C_2,...,C_m. Each variable X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset (see [104]). Each rule of the DT under consideration represents a CSP and will be evaluated for test execution.

Covering Event Sequences of Higher Length

The described algorithm is used to generate a test suite that covers all edges, namely EPs. In other words, the test suite covers all ESs with length 2. Similar processes can be performed to cover ESs with higher length k. For this purpose, it is necessary to transform the ESG4WSC model after Step 1 (see Section 4.2 and Algorithm 4.3 for further details on the transformation) so that the coverage of the transformed graph will deliver the desired test suite. Unfortunately, it might happen that a specific event appears more than once in the resulting model (see also Section 4.2). In this case, it might not be obvious where to add the multiple edges in Step 2 since the corresponding edge could have doubled as well. However, the solution is to remove each doubled edge and add a pseudo vertex instead, which is connected to the source and target of the removed edge. This enables to generate a coverage which contains as many vertices (representing the edges) as needed and in a minimal way as described in Section 4.2. However, note that after Step 3 the pseudo vertices are to be removed from the solution.

7.3.2 Testing the Undesirable Behavior

The previous section described the testing process for expected/desirable situations. However, it is also important to test undesirable situations where partner services do not function as expected. Thus, a holistic approach is worthwhile that generates

positive (desirable) and negative (undesirable) tests.

The negative testing checks separately unexpected behavior in public events and private events. These two cases are represented by *public faulty event sequences* (PubFESs) and *private faulty event sequences* (PriFESs). Subsections 7.3.2.1 and 7.3.2.2 present the definitions and algorithms to generate PubFESs and PriFESs from an ESG4WSC model.

7.3.2.1 Negative Testing for Public Events

The negative testing for public events involves generating sequences that cover unspecified event pairs for public events, that is, request and response messages of the WSC interface. This part considers that a WSC can be viewed and tested as an atomic service where the private communication with its partner services is hidden from the consumer's perspective. Accordingly, the ESG4WSC is reduced to a model representing only public events of the WSC interface for negative testing of public events. Since the resulting model can be used to represent any (atomic) web service, it is called *ESG for web services* (ESG4WS) [20, 40].

Definition 7.13 (ESG4WS)**.** An $ESG4WS = (V, E, \Xi, \Gamma)$ is an ESG where the set of vertices is partitioned into two disjoint sets V_{req} and V_{resp}; that is, $V = V_{req} \cup V_{resp}$, where

- V_{req} is a set of vertices modeling a (public) request, and
- V_{resp} is a set of responses to a (public) request.

In a web service, a consumer can call any (public) operation at any time. However, this might not be appropriate, e.g., the xLoan requires a consumer to request some offers before he can select one of them. Missing edges in the ESG4WS represent this undesirable behavior, which needs to be handled by the SUC. Hence, missing edges between

- response events $v_i \in V_{resp}$ and request events$v_j \in V_{req}$,
- two request events $v_i, v_j \in V_{req}$, and

- pseudo start vertex [and all request events $v \in V_{req}$

define the undesirable situations where a request is critical. Given an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$, let F be a special event used to represent any *faulty event*, such that $F \notin V$.

Definition 7.14 (Public FES)**.** The ordered pair $(pes; F)$ is a *public faulty event sequence* (PubFES), if pes is a PES $\langle v_0, \ldots, v_k, v_{k+1} \rangle$ (see Definition 3.2 on page 16) that is expected to produce a faulty event F and where the last two events v_k, v_{k+1} represent missing edges $(v_k, v_{k+1}) \notin E$ of the corresponding ESG4WS between

- response events $v_k \in V_{resp}$ and request events $v_{k+1} \in V_{req}$,
- two request events $v_k, v_{k+1} \in V_{req}$, or
- pseudo start vertex $v_k = [$ and request events $v_{k+1} \in V_{req}$.

Example 7.15. For the `xLoan` example given in Figure 7.2, the following pair is a PubFES:

```
(⟨LS:requestLoan, BS:approveBank, BS:approved, BS:offer,
   BS:offers, LS:replyOffers, LS:cancel, LS:SelectOffer⟩; F)
```

Since there is no edge between the last two events `LS:cancel` and `LS:SelectOffer`, they were selected as a faulty pair. For this undesirable case, it is expected that the composition produces a faulty event F. If no faulty event is produced, this test sequence fails; otherwise it passes.

The algorithm for deriving PubFESs from an ESG4WSC is as follows. The detailed algorithms for the transformation and test generation can be found in the Appendix (Algorithms A.13 and A.15).

1. Transformation from ESG4WSC to ESG4WS: An ESG4WS can be obtained from an ESG4WSC by removing the private events and keeping edges between any public events v_i and v_j if there exists an ES from v_i to v_j.
2. Inclusion of faulty edges: In the produced ESG4WS, faulty edges are added between events with no edges along Definition 7.14.

3. Generation of test sequences: For each faulty edge (v_i, v_j) in the ESG4WS, find a PES pes that leads to v_i in the ESG4WSC. Next append v_j to pes referred to as $pes \oplus v_j$. Finally, create the PubFES $(pes \oplus v_j; F)$.

Example 7.16. Using `xLoan`, the ESG4WS in Figure 7.4 is obtained after the transformation. The faulty edges are represented by grey dashed lines. The dotted lines connect the request events with the fault that must be produced afterwards. The faulty edges are created by

- connecting all response events with request events,
- connecting request events with request events, and
- connecting the start vertex [with all request events,

in case that there is no edge connecting them. The self-loop from `LS:cancel` to `LS:cancel` was also considered since it represents a one-way operation without response.

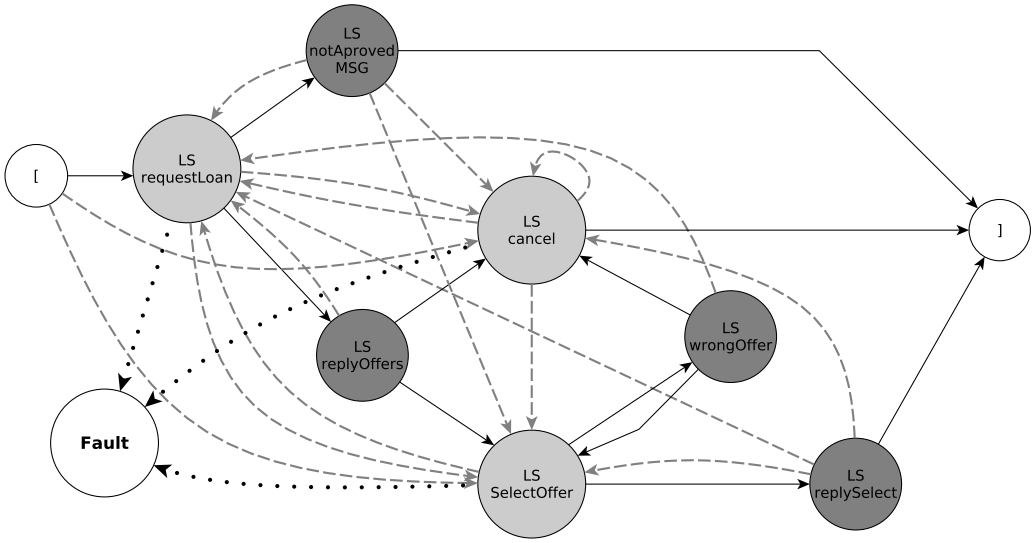


Figure 7.4: ESG4WS for the `xLoan` public interface

After obtaining the ESG4WS, test cases have to be generated to cover the following faulty edges:

(LS:notApprovedMSG, LS:requestLoan); (LS:notApprovedMSG, LS:cancel); (LS:notApprovedMSG, LS:SelectOffer); (LS:replyOffers, LS:requestLoan); (LS:wrongOffer, LS:requestLoan); (LS:replySelect, LS:requestLoan); (LS:replySelect, LS:SelectOffer); (LS:replySelect, LS:cancel); (LS:cancel, LS:requestLoan); (LS:cancel, LS:SelectOffer); (LS:cancel, LS:cancel); (LS:requestLoan, LS:cancel); (LS:requestLoan, LS:SelectOffer); (LS:SelectOffer, LS:cancel); (LS:SelectOffer, LS:requestLoan); ([, LS:cancel); ([, LS:SelectOffer).

Then a PES from the ESG4WSC (Figure 7.2) is derived for each faulty edge (v_i, v_j) to reach the event v_i. The event v_j is included after v_i and the faulty event F is added to the tuple. The following PubFES is obtained for the faulty edge (LS:replySelect, LS:cancel):

(⟨LS:requestLoan, ⟨⟨BLIS:checkBL,BLIS:NotinBList⟩ || ⟨CAS:inDebtorsList, CAS:debtorsFalse⟩⟩, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, BS:confirmBank, LS:replySelect, LS:cancel⟩; F)

The same procedure is repeated for each faulty edge to generate the final test suite of public negative tests.

7.3.2.2 Negative Testing for Private Events

The proper functioning of a WSC does not depend only on its correct implementation but also on the partner services. It is often not clearly defined what happens when an invoked service is not working as expected.

In this work, seven fault classes are defined based on fault taxonomy and fault injection literature [33, 32, 59]. The fault classes are:

No response: The invoked service does not send back a response for a request-response operation, e.g., due to internal problems or modified behaviors.

Long time response: The invoked service needs an inappropriately long time to send back a response.

Missing service: The service is missing; e.g, the server hosting the service is not available or the service address (URL) has changed.

Unexpected fault: An unexpected SOAP Fault is returned either by the invoked service or due to a fault in the environment, e.g., a fault caused by pre/post-processing in the ESB.

Wrong XML schema: The invoked service sends back an unexpected XML schema, e.g., due to some (untold) changes of the service by the provider.

Wrong XML syntax: The response of the invoked service contains a corrupted XML file, e.g., due to some noise in the network.

Right schema, wrong data: The response is a well-formed message that contains invalid data, e.g., an invalid date.

Testing these undesirable situations is important to the robustness of the given WSC. Depending on the number of partner services, this results in some additional testing efforts. Fault classes "no response," "missing service," and "unexpected fault" can be tested for every private request vertex of the ESG4WSC model. Fault classes "longtime response," "wrong XML schema," "wrong XML syntax," and "right schema, wrong data" can be tested for every private response of an ESG4WSC. Table 7.3 summarizes this information and also includes the symbols used to represent each class in PriFESs.

Table 7.3: Fault classes and their relation to events

			event of an ESG4WSC	
		symbol	request	response
fault class	no response	F_{NR}	✓	
	longtime response	F_{LR}		✓
	missing service	F_{MS}	✓	
	unexpected fault	F_{UF}	✓	
	wrong XML schema	F_{WSc}		✓
	wrong XML syntax	F_{WSy}		✓
	right schema, wrong data	F_{WD}		✓

7.3.2.3 "Sensitive" Events as Test Oracle for Negative Testing of Private Events

When one of the fault classes is provoked, an automated test oracle needs to be established to determine the expected test outputs. A tester can, of course, decide to evaluate and define the expected behavior for every single situation by hand. However, this would mean tedious manual work and does not scale well. A more straightforward approach is to mark events of the given ESG4WSC as *sensitive*; that is, these events are not allowed to show up after provoking one of the faulty situations. For instance, a loan must not be approved if some fault happens during the verification process of a client's reputation. Defining a set of sensitive events is much easier and enables automation. To the best of the author's knowledge, there is no other approach that solves the oracle problem for negative testing of WSCs (or any other application) in this way.

Definition 7.17 (Sensitive Events)**.** Given an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$, the nonempty set $S \subset V$ represents the events marked as *sensitive*. The sensitive events are not allowed to show up after provoking a faulty situation.

Definition 7.18 (Private FES)**.** Given that pes is a PES $\langle v_0, \ldots, v_k \rangle$, F is any faulty event, and $s \subseteq S$ is a set of sensitive events, the triple $(pes; F; s)$ is a *private faulty event sequence (PriFES)* if $v_k \in V_{req} \cup V_{resp}$ is a private event and s is not empty.

Example 7.19. For the `xLoan` example given in Figure 7.2, let `LS:replyOffers` be a sensitive event; the following triple is a PriFES:

```
(⟨LS:requestLoan, ⟨⟨BLIS:checkBL⟩ || ⟨CAS:inDebtorsList,
                    CAS:debtorsFalse⟩⟩; F_MS; {LS:replyOffers})
```

In this example, F_{MS} represents the fault class "missing service" that must be provoked; that is, `BLIS` is not available. This sequence passes if no sensitive event (`LS:replyOffers`) is produced by the composition after executing the PES and provoking F_{MS}; otherwise it fails.

The negative testing for private events generates sequences that cause some unexpected behavior in the partner services, that is, in the private events, and check the service composition in these cases.

The algorithm for deriving PriFESs from an ESG4WSC is as follows.

1. Let V_{RR} be the set of all private request and response events in the ESG4WSC model. Then find the shortest PES pes that reaches e for each $e \in V_{RR}$.

2. If e is a private request event,

   (a) copy pes and mark e with F_{NR} to provoke the "no response" fault;

   (b) copy pes and mark e with F_{MS} to provoke the "missing service" fault;

   (c) copy pes and mark e with F_{UF} to provoke the "unexpected fault" fault.

3. If e is a private response event,

   (a) copy pes and mark e with F_{LR} to provoke the "longtime response" fault;

   (b) copy pes and mark e with F_{WSc} to provoke the "wrong XML schema" fault;

   (c) copy pes and mark e with F_{WSy} to provoke the "wrong XML syntax" fault;

   (d) copy pes and mark e with F_{WD} to provoke the "right schema, wrong data" fault.

4. For all sequences produced in previous steps, add the set of sensitive events s that is not covered by pes in the PriFES, that is, (pes; F; s).

The detailed algorithms can be found in the Appendix (Algorithms A.17 and A.14).

Example 7.20. Consider private request event `CAS:inDebtorsList` of the `xLoan` example. The first step is to find the shortest PES that reaches `CAS:inDebtorsList`.

pes=⟨`LS:requestLoan, ⟨⟨BLIS:checkBL, BLIS:NotinBList⟩ || ⟨CAS:inDebtorsList⟩⟩⟩`.

Notice that `CAS:inDebtorsList` is part of the refined event `check`. In this case, CESs must be generated for the other parallel ESG4WSCs so that there is no influence on event `CAS:inDebtorsList` and the provoked fault.

Next, pes is copied to pes_1 and event `CAS:inDebtorsList` is marked with F_{NR}. This PriFES tests the scenario in which the WSC calls operation `inDebtorsList` and no answer/response is sent back. The same procedure is performed for F_{MS} and F_{UF} with the copies pes_2 and pes_3, respectively.

Let $s = \{\texttt{LS:replyOffers}\}$ be the set of sensitive events, the resulting PriFESs look as follows: (pes_1; F_{NR}; s), (pes_2; F_{MS}; s), and (pes_3; F_{UF}; s). As the client reputation must be good in both services, `BLIS` and `CAS`, any fault in event `CAS:inDebtorsList` must not produce a successful approval represented by `LS:replyOffers` marked as sensitive. The same steps are repeated for all other private request and response events.

7.3.3 Selective Layer-Centric Testing

To show the extendability of the approach, the derivation of test cases as described above follows the basic idea of LC testing. Apart from this exemplary derivation, the SLC approach for test generation described in Section 5.3 can also be applied for testing WSC, referred to as *selective layer-centric testing for web service compositions* (SLC4WSC). Similar as in SLC, test cases are generated for the critical layers only.

For positive testing, test cases of lower layers are moved to the upper layer for test execution. If no test cases are to be generated for the upper layer, a tour will be derived that visits the corresponding refined vertex as often as needed. The tour can be derived by following the descriptions given in Section 5.3, that is, solving the traveling salesman problem and extending the tour by solving the assignment problem.

For negative testing of private events, PriFESs of selected ESG4WSCs of lower layer are first generated and then returned to the next higher layer where the shortest path [122] from start vertex [to the corresponding compound vertex $v \in V$ is calculated and concatenated with the given test case of the lower layer model. The negative testing of public events is similar to that described in Section 7.3.2.1. The difference is that only events of the selected layers are considered and put into the ESG4WS.

Chapter 8

Case Study II: Reliability Analysis Concerning Test Length & Model Refinement

This chapter describes the evaluation of the ESG4WSC approach for positive and negative testing of WSCs (Chapter 7). It presents the application used as the subject in the case study, as well as configuration and results of the experiments. Limitations and threats to validity conclude the chapter. The purpose of this chapter is to compare LC with SLC for a different type of application and determine their characteristics, especially the influence and contribution of test length and model refinement to the test process. In addition, two tools are introduced to support automation. *Test Suite Designer for Web Service Compositions* (TSD4WSC) provides a graphical user interface, which allows to model the SUC and to generate test cases. *Event Runner for Test Execution* (ERunTE) automates test execution by composing three modules: a web service, a test runner, and an ESB component.

8.1 System Under Consideration, Test Setup, and Goals of the Experiment

The case study was conducted using the `xTripHandling` application, which is based on different scenarios proposed in technical and research literature [73, 126, 88]. The application was developed using SOA concepts and web services and provides a set of facilities to query and book a trip. It also includes facilities to buy train tickets, rent a car, book sightseeings, and order maps. The application consists of eight services: six atomic services and two composite services (see Section 7.1.1). The atomic services are:

1. *ISELTA-hotel Service* is a web service provided by the commercial system ISELTA that enables travel and touristic enterprises to create their individual search and service offering masks. It provides operations to query hotels and manage bookings.

2. *Airlines Service* provides a set of operations to manage flight tickets, which are similar to ISELTA-hotel service.

3. *Map Service* provides operations to locate places close to a city (e.g., airports, train stations) and to order maps for certain cities.

4. *Car Rental Service* provides operations to search and rent vehicles to be used in a pre-defined city.

5. *Train Service* provides operations to check train lines between cities and to buy train tickets.

6. *Sightseeing Service* provides operations to list available cities in which the service operates and to buy tickets for sightseeing.

The composite services are:

1. *Travel Agent Service* provides a set of facilities to query and book a trip, interacting with two services, ISELTA-hotel and Airlines services. It combines these two services, providing operations to search and book a travel involving

flight and hotel reservation. As the flight ticket and hotel reservation are essential in any travel, a successful booking using this service guarantees hotel and flight reservations.

2. *Customer Service* combines the services *Travel Agent, Airlines*, *Map*, *Car Rental*, *Sightseeing*, and *Train* to provide a centralized resource for customers to manage all their travel plans, including hotels, flights, maps, trains, cars, and sightseeing.

Figure 8.1 illustrates the services, their interfaces, and the interactions of composite services. The figure presents a summarized version of the information available in the WSDL interfaces. The dashed edges represent the interaction between composite services and partner services. The specification of the composite services *Travel Agent Service* and *Customer Service* can be found in the appendix. The *Customer Service* will form the SUC in this case study.

Model Information

Table 8.1 summarizes the information about the ESG4WSC model. Lines 1-4 refer to the number of each type of event. Lines 5 and 6 show the total number of events and edges, respectively. Line 7 refers to the number of refining ESG4WSCs and those that are to be executed in parallel in Line 8. Lines 9-11 show the number of DTs, constraints, and rules, respectively. Line 12 refers to the elapsed time to study the specification and interfaces and produce the first model version.

Refinements were used in *Customer Service* to simplify the modeling process. For instance, after booking a basic trip (hotel + flight), different ESG4WSCs were included for maps, cars, sightseeing, and trains. Thus, this model contains 34 events that are refined by 68 ESG4WSCs, 44 of which are in parallel. Notice that 24 of the 68 ESG4WSCs are not in parallel and were used to modularize the model. Thus, refinements were used not only to represent parallelism but also to ease modeling. For instance, after booking a basic trip (hotel + flight), the client can search and book a car. This workflow is abstracted as a refined event “rentCar” and its details are expressed in an associated refining ESG4WSC. Similar refined events were defined for maps, sightseeing, and trains using the hierarchy of refining ESG4WSCs to organize the model.

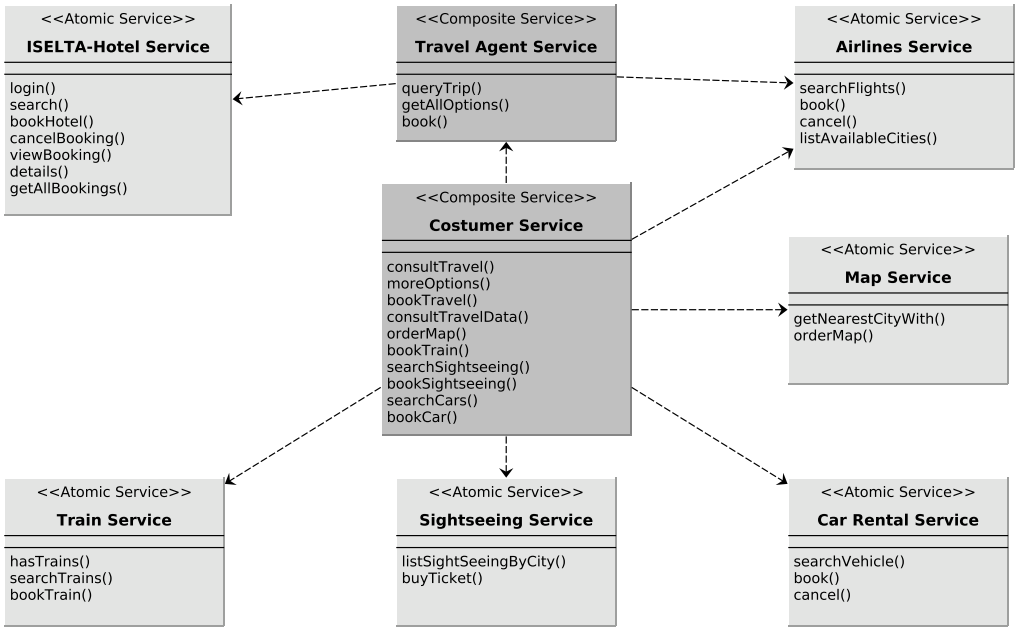


Figure 8.1: Service interfaces in `xTripHandling`

Table 8.1: Test model information

		Customer Service
1:	# request events	204
2:	# response events	449
3:	# generic events	13
4:	# refined events	34
5:	**# events (total)**	**700**
6:	**# edges (total)**	**947**
7:	# refining ESG4WSCs	68
8:	# ESGs in parallel	44
9:	# DTs	108
10:	# constraints	197
11:	# rules	300
12:	initial modeling time	~20h

In the model there are more response events than request events since a request message can have several relevant response messages and instances of response messages. For example, a search request can return one of the following responses: (i) a message with zero items, (ii) a message with one or more items, or (iii) an expected fault. Generic events facilitate the description of time constraints or changing points. DTs mainly supplement public request events for which input data must be generated. The constraints are defined over request parameters and rules test different combinations of these constraints.

Goals of the Experiment

The purpose of the experiment is to apply LC and SLC to WSCs in order to gain the special characteristics of both approaches with respect to their strengths and limitations for a different type of application. Of primary interest is the impact of test sequence length on its fault detection capability and overall reliability. In addition, the question of how the structure of the model, specifically model refinement, is able to contribute to the fault detection will be investigated. A secondary goal of the experiments is to demonstrate the applicability of the ESG4WSC approach.

In the course of the experiment more than 57,000 tests following the LC4WSC strategy were derived from the model covering sequences of length 2, 3, and 4 and were applied to the given WSC. After that, the SLC4WSC strategy was performed and the results were compared with the ones achieved by the LC4WSC strategy using a reliability theoretical analysis.

8.2 Test Execution and Tool Support

In this case study, the specification has been set up first (see Appendix C). After that, the implementation and test model creation was done in parallel by two different persons. Thus, the tester did not need to wait for the implementation to set up a model and derive tests since the new approach is not based on artifacts such as BPEL or WS-CDL as in [73, 56, 24].

For large models like the one created for this case study, test generation and execution can hardly be done by hand. Therefore, two tools have been developed

and used to support test case generation and test execution in the conducted case study. The tool for test case generation generates test files, which are loaded into the tool for automated test execution. However, it takes some time before test cases can be executed. Automated test execution requires some manual work by writing two kinds of adaptors, one for invoking and checking the messages of the WSC interface, that is, the public request and response events, and another one for checking SOAP messages of the private events produced during the test case execution. This step can hardly be automated due to the characteristics of web service technology. For testing *Customer Service*, the adaptors consisted of approximately ~2,300 lines of code. Compared to the number of events, this is negligible—especially because most of the code for implementing single events is similar, which allows reuse of code fragments. In the case study, the tester designed the entire models before implementing the adaptors for test execution. This strategy fits cases in which the development phase is ongoing and the implementation is not yet available. However, if the implementation has a preliminary version, the test model and adaptors can be built partially and iteratively to obtain some executable test cases sooner. Detailed information on the tools supporting the case study are given in the following.

8.2.1 Modeling & Test Generation

TSD (see Section 6.2) is adapted to provide a graphical user interface for the tester to model all features of an ESG4WSC that are necessary for test generation. Figure 8.2 shows a screenshot of the *TSD for web service compositions* (TSD4WSC) as well as the model set up for the running example `xLoan`. TSD4WSC implements the algorithms described in Section 7.3 for generating positive and negative test suites. Furthermore, TSD4WSC allows to consider constraints on input data (for test data generation) and the execution of events by DTs (see Figure 8.3). That is, the test generation process is fully automated.

An XML format was defined to describe test cases generated by TSD4WSC. The resulting XML files are used as input to the test execution environment. This integrates the test generation and test execution and reduces the dependency between both environments. Thus, new tests can be generated and no extra effort is necessary for concretization, except for the adaptors.

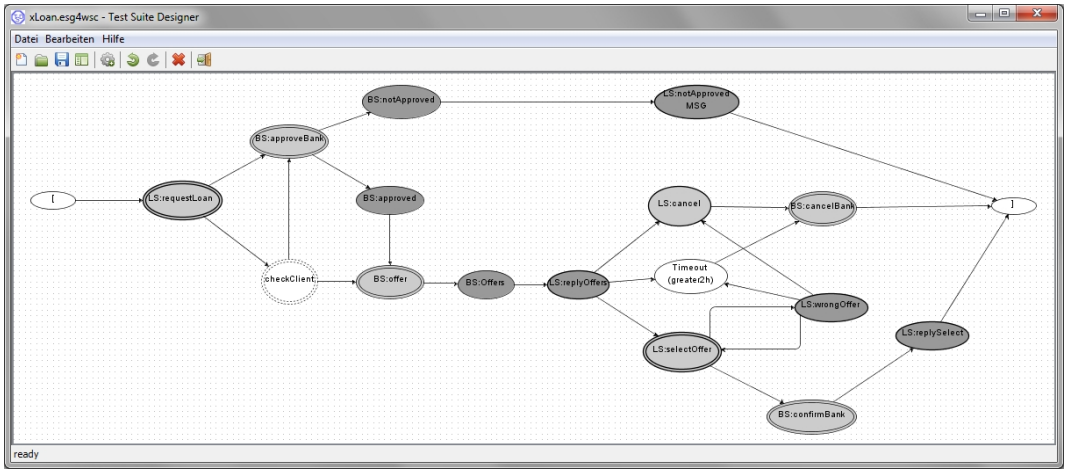


Figure 8.2: ESG4WSC for `xLoan` in TSD4WSC

Figure 8.4 presents an example of a test case for *Travel Agent Service* in the XML format. The file starts with an element `<TestCase>`, followed by an element `<CompleteEventSequence>` that represents a CES. Events are represented by the element `<Event>` with attributes to label and classify (type and public) the event. Events with associated DT (Line 3) have an attribute to define which rule must be tested and may also include the child elements `<Param>` to provide input data (Lines 4 and 5). A refined event is represented by the element `<RefinedEvent>` (Line 7) and can include one or more CESs. In the example, there are two CESs in parallel, the first in Lines 8-15 and the second in Lines 16-21. When the response event is private and is supposed to answer a specific message, the element `<Message>` can be used within the event, such as in Line 31. A predefined SOAP message can be provided in TSD4WSC and will be available within a CDATA section[1]. The negative test cases are also represented with special elements and attributes for sensitive events, faulty edges, and fault classes.

[1] All text within a CDATA section is ignored by the XML parser.

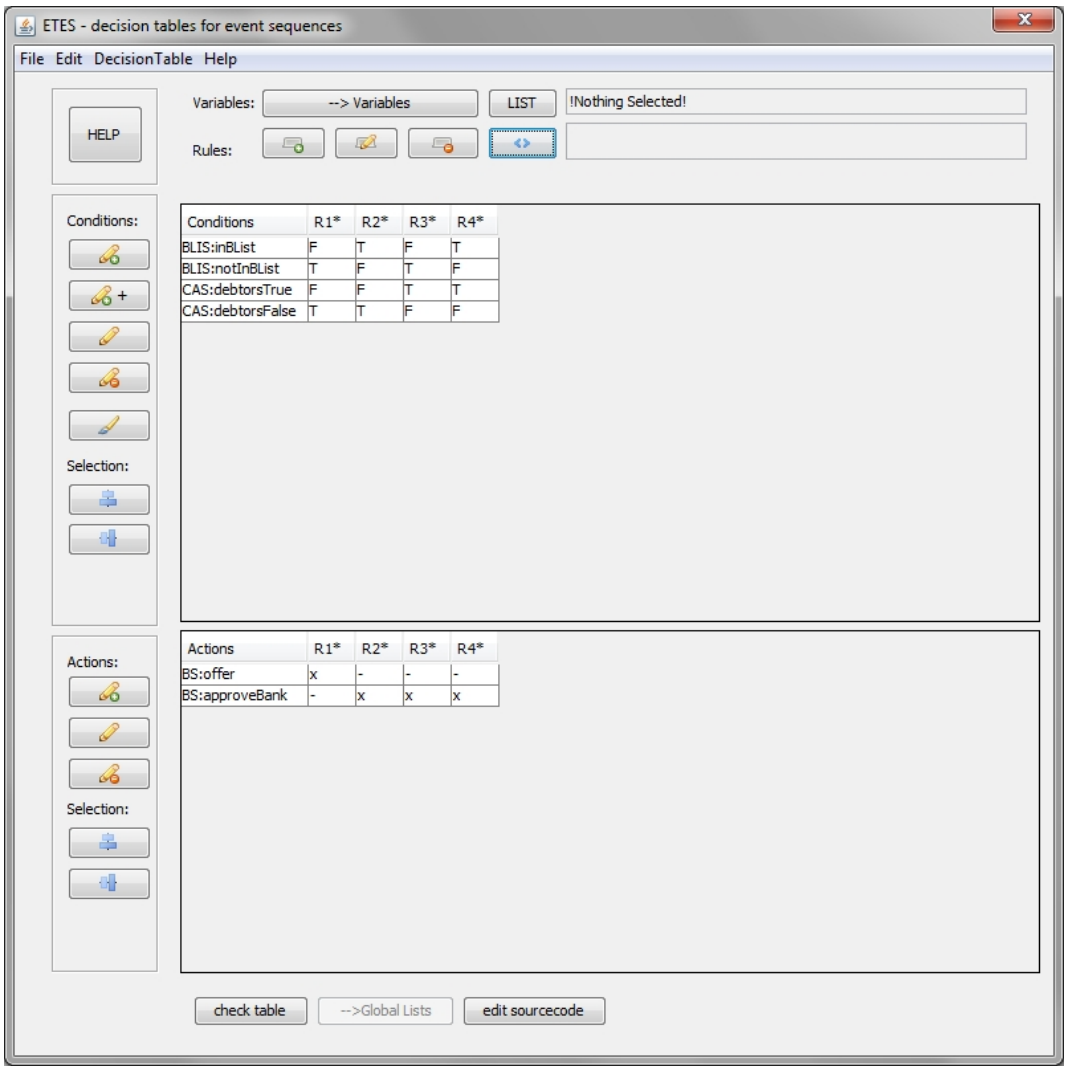


Figure 8.3: Decision tables in TSD4WSC

```
01:<TestCase>
02: <CompleteEventSequence>
03:  <Event label="TA:queryTrip" type="request" public="true" rule="R1" >
04:   <Param name="departureDate">31.12.2011</Param>
05:   <Param name="toCity">Sao Carlos</Param>
        ...
06:  </Event>
07:  <RefinedEvent>
08:   <CompleteEventSequence>
09:    <Event label="IS:login" type="request" public="false"/>
10:    <Event label="IS:login_Response" type="response" public="false" />
11:    <Event label="IS:search" type="request" public="false"/>
12:    <Event label="IS:searchResults_greaterEqThanOne" type="response" public="false" >
13:     <Message><![CDATA[ ... ]]></Message>
14:    </Event>
15:   </CompleteEventSequence>
16:   <CompleteEventSequence>
17:    <Event label="FL:search" type="request" public="false"/>
18:    <Event label="FL:searchResults_greaterEqThanOne" type="response" public="false" >
19:     <Message><![CDATA[ ... ]]></Message>
20:    </Event>
21:   </CompleteEventSequence>
22:  </RefinedEvent>
23:  <Event label="TA:queryTrip_Response" type="response" public="true" />
24:  <Event label="TA:book" type="request" public="true" rule="R1" >
25:   <Param> ... </Param>
26:  </Event>
27:  <RefinedEvent>
28:   <CompleteEventSequence>
29:    <Event label="FL:book" type="request" public="false"/>
30:    <Event label="FL:bookingSuccess" type="response" public="false" >
31:     <Message><![CDATA[ <soap:Envelope><soap:Body> ... </soap:Envelope> ]]></Message>
32:    </Event>
33:   </CompleteEventSequence>
34:   <CompleteEventSequence>
35:    <Event label="IS:login" type="request" public="false"/>
36:    <Event label="IS:login_Response" type="response" public="false" />
37:    <Event label="IS:search" type="request" public="false"/>
38:    <Event label="IS:searchResults_sameHotelPrice" type="response" public="false" >
39:     <Message><![CDATA[ ... ]]></Message>
40:    </Event>
41:    <Event label="IS:book" type="request" public="false"/>
42:    <Event label="IS:bookingSuccess" type="response" public="false" >
43:     <Message><![CDATA[ ... ]]></Message>
44:    </Event>
45:   </CompleteEventSequence>
46:  </RefinedEvent>
47:  <Event label="TA:bookingConfirmation" type="response" public="true" />
48:  <Event label="TA:getAllOptions" type="request" public="true" rule="R1" >
49:   <Param name="searchCode">[$validSearchCode$]</Param>
50:  </Event>
51:  <Event label="TA:TripInputException" type="response" public="true" />
52: </CompleteEventSequence>
53:</TestCase>
```

Figure 8.4: XML file for a test case

8.2.2 Test Execution

Mule-ESB [81] is used as the infrastructure software. Initially, all services involved in the composition (including the composite service) are deployed in the bus; that is, the entire communication (SOAP messages) passes through the ESB before reaching the destination service. The test execution is supported by a tool named ERunTE (*Event Runner for Test Execution*), which is composed of three modules: a web service (ERunTE-service), an ESB component (ERunTE-esbcomp), and an event runner (ERunTE-runner). Figure 8.5 summarizes the corresponding architecture adopted to execute the tests in this case study.

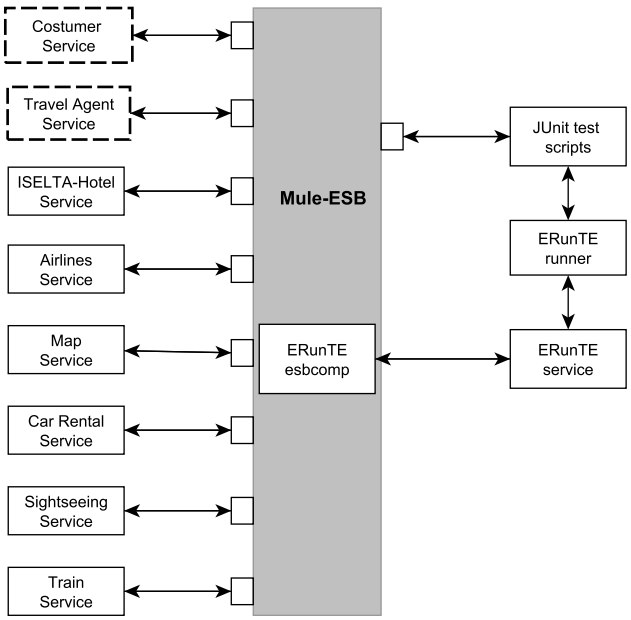


Figure 8.5: Architecture to execute the tests

ERunTE-service can be used directly in the test code and provides operations such as `startObservation`, `modifyMessage`, and `getAllMessages`. The second module, ERunTE-esbcomp, is the ESB component that implements the monitor and aggregator patterns. This component is integrated with mule-ESB and is able to interact with ERunTE-service. The component records all messages that pass through the ESB and also modifies exchanged messages according to opera-

tion `modifyMessage`. For test execution, ERunTE-service has been integrated in ERunTE-runner, which contains the aforementioned adaptors to execute the test cases using Java/JUnit. ERunTE-runner works in three phases. First, it sets up all private messages according to the test case using ERunTE-service. Next, a public request is send to the WSC to initiate the process. As a last step, all messages are checked according to the test case.

The negative testing requires special configurations of the test execution environment. ERunTE-esbcomp implements a configurable delay for fault classes "no response" and "longtime response." ERunTE-service has an operation to shut down service proxies, helping to reproduce fault class "missing service." Likely unexpected fault messages have been identified and simulated for fault class "unexpected fault." An example is SOAP Faults thrown by web service frameworks when exceptions are not correctly handled in the application. ERunTE-runner makes small modifications in the original messages to reproduce the fault classes "wrong XML syntax" and "wrong XML schema." To sum up, the test generation and execution is fully supported by TSD4WSC and ERunTE tools.

8.3 Results and Their Analysis for Identifying the Critical Sub-Layers

8.3.1 Results

Using the designed test models, the supporting tool (described in Section 8.2) generated test suites according to the holistic ESG4WSC approach. Table 8.2 summarizes the information about the test suites, divided into positive and negative testing. The number of executed events for each test suite is also provided. Furthermore, the test suites are divided by the length of covered ESs ($k = 2$, $k = 3$, and $k = 4$). The information about faults detected using positive and negative test suites is included as well.

19 faults were detected for the *Customer Service* while 13 faults were revealed by the positive test suite and 6 faults by the negative test suite. All positive test suites were applied first and faults were corrected at once. The faults in the im-

Table 8.2: Positive and negative test cases and their number of events subject to ES length

	test suite (as test cases)			test suite (as events)			faults		
length	**CES**	**FCES**	$\sum$	**CES**	**FCES**	$\sum$	**CES**	**FCES**	$\sum$
2	1054	7596	8650	89520	174805	264325	12	6	18
3	998	9977	10975	139388	235541	374929	12+1	6+0	18+1
4	20537	16953	37490	3406148	429139	3835287	13+0	6+0	19+0
$\sum$	22589	34526	57115	3635056	839485	4474541	13	6	19

plementation were mainly identified by testing different rules (from the DTs) and checking expected events and their order. The correction of these faults was not critical and was performed directly in the implementation.

8.3.2 Comparing LC4WSC and SLC4WSC - Identifying the Critical Sub-Layers for Further Testing

The result of the present case study and its analysis confirms the result achieved in the previous experiment with LC testing (Chapter 6). Testing with higher event sequence length also leads here to a great deal of additional test effort while detecting fewer faults. Therefore, this section analyzes the test effort reduction capabilities of the SLC4WSC approach from a reliability point of view.

Step 1: Perform Layer-centric Testing and Categorize Detected Faults

The resulting test case set has been analyzed and the events occurring in the resulting test case set have been counted for each of the components. Furthermore, the faults have been categorized along the components. It is assumed now that only the CESs and FCESs covering sequence length 2 have been generated and executed. The Customer Service consists of 69 models in total. Therefore, Table 8.3 shows only the results for each component/ESG4WSC that revealed at least one fault. The complete table can be found in the appendix (Table C.1).

Table 8.3: The number of faults categorized according to the number of events

	ESG 1		ESG 2		ESG 3		ESG 4		ESG 8		ESG 12		ESG 14	
length	events	faults	events	faults	events	faults	events	faults	events	faults	events	faults	events	faults
2	10386	1	28314	3	28242	1	7241	3	3395	1	18134	2	2338	1
3	13161	0	35870	0	35798	0	6980	1	4820	0	24746	0	2862	0
4	41430	0	129136	0	129064	0	16024	0	28541	0	103997	0	75533	0

	ESG 15		ESG 18		ESG 44		ESG 46		ESG 57		ESG 60	
length	events	faults	events	faults	events	faults	events	faults	events	faults	events	faults
2	2338	1	6253	1	1112	1	877	1	989	1	4264	1
3	3701	0	8834	0	1628	0	1367	0	1577	0	6035	0
4	74068	0	203473	0	22228	0	20289	0	24987	0	71907	0

Step 2: Select Layers for Further Testing

As mentioned in Section 5.2, the first step to identify a subset of models which have a higher fault detection capability is to calculate their usage ratios UR. Next, the reliability of each component RE and their impacts IE are determined.

Step 2.1: Calculating the Usage Ratio of Components Equation 5.1 is used to calculate UR. According to Table 8.4, the component with the highest usage is represented by ESG 2. The components represented by ESG 10, ESG 37, ESG 52, and ESG 67 have the lowest usage (not contained in Table 8.4). The complete table is given in the appendix (Table C.2 on page 211).

Table 8.4: Usage ratio of Components/ESG4WSCs

ESG4WSC	ESG 1	ESG 2	ESG 3	ESG 4	ESG 8	ESG 12	ESG 14
UR	0.0392	0.1071	0.1069	0.0274	0.0128	0.0686	0.0088

ESG4WSC	ESG 15	ESG 18	ESG 44	ESG 46	ESG 57	ESG 60
UR	0.0088	0.0237	0.0042	0.0033	0.0037	0.0161

Step 2.2: Calculating Reliability of each Component To determine the fault data used for calculating the reliability of each component, first, the number of faults detected by each component and the corresponding number of events in case of length 2 are sorted in descending order according to their URs (Figure 8.6).

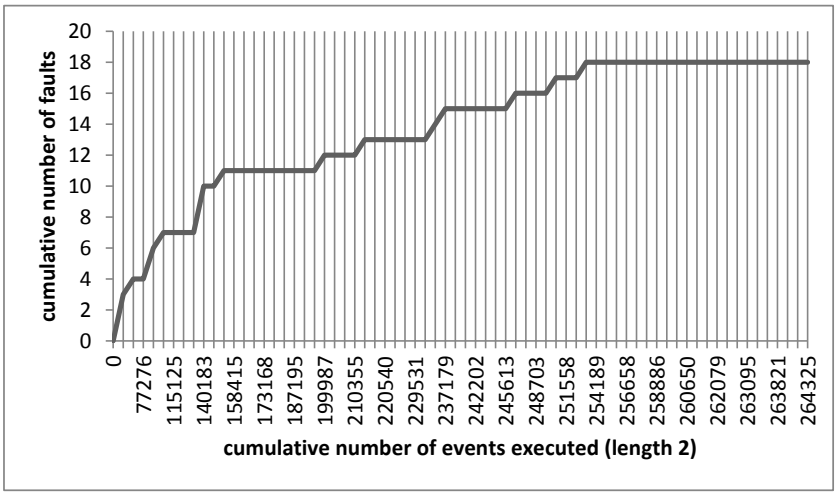


Figure 8.6: Fault data used to calculate the reliability of each component

A K-S test (Table 8.5) indicates that the cumulative number of faults follows Poisson distribution (mean parameter = 13.9275) since p-value (0.085) is greater than 0.05. Therefore, the NHPP models given in Table 4.4 (on page 42) are applied to the fault data given in Figure 8.6 and GoF measures are computed for each SRGM to determine the best-fitting model. Figure 8.7 visualizes the results of the GoF measures showing that G-O model provides the best performance in all GoF measures since it has the smallest AIC, BIC, and MSE values. Therefore, the G-O model has been used to calculate the RE (equation 5.2 on page 48) in this study. Table 8.6 shows the corresponding subset of reliability results of each component and combined reliability (R_c). The complete table can be found in the appendix (Table C.3 on page 212). The next goal is to enhance R_c. This will be done by performing SLC4WSC testing with higher length for components that have a small IE value compared to the overall system reliability.

Table 8.5: One-Sample Kolmogorov-Smirnov Test

	Cumulative Number of Faults
Poisson Parameter Mean	13.9275
Kolmogorov-Smirnov Z	1.256
p-value (2-tailed)	0.085

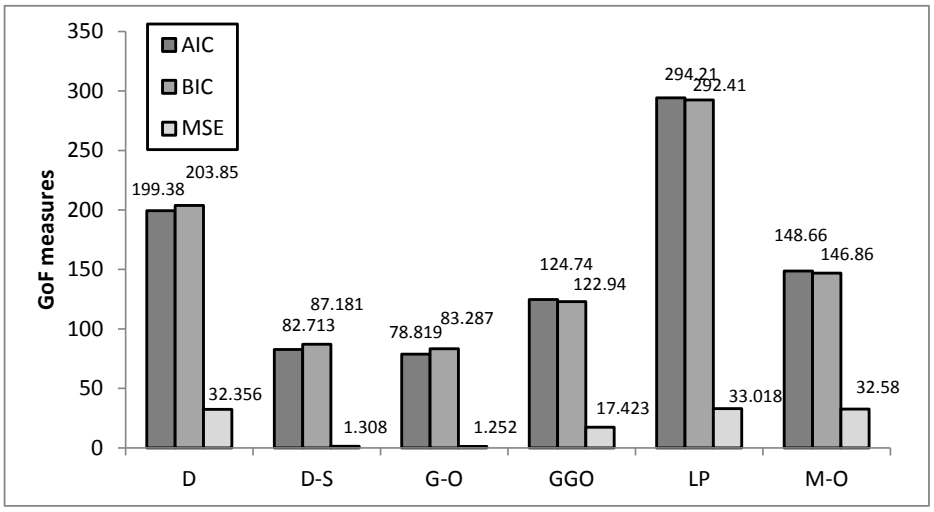


Figure 8.7: GoF measures

Table 8.6: Reliability results of each component and combined reliability R_c

ESG4WSC	**ESG 1**	**ESG 2**	**ESG 3**	**ESG 4**	**ESG 8**	**ESG 12**	**ESG 14**
RE	0.99933	0.99926	0.99929	0.99937	0.99944	0.99933	0.99945
ESG4WSC	**ESG 15**	**ESG 18**	**ESG 44**	**ESG 46**	**ESG 57**	**ESG 60**	R_c
RE	0.99945	0.99938	0.99946	0.99947	0.99946	0.99942	0.9993

Step 2.3: Calculating Impact of Components on Overall System Reliability Table 8.7 shows the sorted IE values of components on the overall system reliability derived in line with equation 5.4 (on page 48). The IE-value for determining the Quartile has been calculated as $IE_p = IE_{p<} + (IE_{p>} - IE_{p<}) * (p - p_<) = IE_{17} + (IE_{18} - IE_{17}) * (17.5 - 17) = 0.983708426$ with $p = (n+1)/4 = (69+1)/4 = 17.5$ and $p_< = 17$ and $p_> = 18$. The complete table can be found in the appendix (Table C.3).

Table 8.7: Impact of each component on overall system reliability

ESG4WSC	**Sorted - IE**
ESG 2	**0.875234468**
ESG 3	**0.880494158**
ESG 38	**0.914891862**
ESG12	**0.927427172**
ESG 1	**0.959049777**
ESG 40	**0.963706467**
ESG 55	**0.964189084**
ESG 13	**0.967811853**
ESG 4	**0.972824563**
ESG 39	**0.974068847**
ESG 18	**0.976973913**
ESG 53	**0.981719601**
ESG 68	**0.981850293**
ESG 31	**0.98204088**
ESG 20	**0.982925822**
ESG 41	**0.983042295**
ESG 56	**0.983157974 (Quartile)**
ESG 35	0.984258879
...	...

Step 3: Re-execute Layer-centric Testing by Increasing the Sequence Length for the Critical Layers Only

SLC4WSC testing with length 3 is performed for the 17 components/ESG4WSCs given in Table 8.7 since IE values of these components are equal or less than the 1st quartile of IE values. The testing with the SLC4WSC test suite detected one more fault with respect to LC4WSC testing with length 2. The reliability after executing SLC4WSC has been calculated as $R_c^{SLC} = 0.999654$ and is close to the one calculated for LC4WSC $R_c^{LC} = 0,999999$ on the basis of G-O.

Results in a Nutshell

When performing SLC4WSC testing with length 3 for the selected models, SLC4WSC reached a reliability level close to the one achieved by LC4WSC testing. The difference is that the test effort could be reduced by $1 - (398132/\ 4474541) = 91\%$ (Table 8.8) while detecting the same number of faults. The results of Case Study I could be confirmed regarding the test sequence length. Test cases covering event sequences of length 2 detected most of the faults. Test cases covering length 3 contributed very little, and the execution of test sequences longer than 4 makes no sense, especially with respect to the measured reliability level.

Table 8.8: SLC4WSC compared to LC4WSC

	SLC4WSC as events				LC4WSC as events			
length	**CES**	**FCES**	$\sum$	**faults**	**CES**	**FCES**	$\sum$	**faults**
2	89520	174805	264325	18	89520	174805	264325	18
3	2308	131499	133807	+1	139388	235541	374929	+1
4	-	-	-	-	3406148	429139	3835287	+0
$\sum$	91828	306304	398132	19	3635056	839485	4474541	19

8.4 Limitations and Threats to Validity

In the previous sections, it has been described how to apply the proposed approach. The case study demonstrated that the approach is applicable to a non-trivial web

service-oriented application. It is not necessary to distinguish between orchestration and choreography for modeling and testing since the approach can be applied in both contexts. The only restriction is that the tester has control over messages exchanged by the partner services. The tool which was used to support the test execution (Section 8.2.2) requires that all messages pass through an ESB. Although ESBs may not be part of the service-oriented application under test, deploying the services in an ESB is a simple task.

The proposed approach assumes that ESG4WSCs in a refined event are independent. This enables a simple way to model some parallelism, and this was sufficient for the example and case study. It is possible that more complex scenarios occur in WSCs and the tester might also want to test combinations of message interleaving. Although the approach can be adapted to test these scenarios, it is recommended to use specific models and testing techniques for concurrent programs [54, 67].

Experiments of this case study addressed seven fault classes for negative testing. Additional classes can be defined and implemented using the current infrastructure. Notice that covering all fault classes generates a high number of negative test cases. If the cost of test execution is critical in the current project, a subset of the negative test suite can be selected. Based on the case study experience, a strategy is to concentrate on the fault classes "longtime response" and "unexpected fault." They are usually sufficient to test the WSC robustness since the fault correction for those classes indirectly handles other fault classes.

The evaluation of negative test results should be performed carefully. The information for undesirable situations is usually misleading, scarce, and even missing. For instance, the "longtime response" class can expose unplanned issues that must be handled by the composition, such as timeouts in the implementation, missing specification, and incomplete workflows. This fact hinders an accurate and automatic evaluation of test sequences. Therefore, a mechanism was presented to handle this issue using *sensitive events*. This strategy avoids false positives, but faults can be missed by the test cases. Thus, it is recommended that the tester inspects a subset of each fault class to avoid false negatives.

Part IV

Further Perspectives and Conclusions

Chapter 9

Further Perspectives

LC and SLC reduce the test generation and execution effort. The underlying test process executes positive test cases first and negative test cases afterwards. In some very restrictive cases it is possible to reduce the test execution effort by combining positive and negative test cases and therefore reduce the test execution effort significantly [19].

However, the correction of observed failures costs additional (routinely manual) effort to detect and correct the corresponding faults. Thus, further costs can be saved if the fault correction step is automated. This chapter presents preliminary work to automate this step using *design by contract* (DbC) patterns [115, 116]. The approach is evaluated by experiments on boundary overflows, which occur when numerical input values violate the range of specified values. Furthermore, a tool is presented that implements the presented approach, enabling a semi-automatic detection of boundary overflow errors and suggesting correction steps based on DbC.

To sum up, this chapter presents further cost reduction capabilities by

1. presenting an approach for semi-automatic detection and correction of boundary overflow errors to reduce the manual fault correction effort, and

2. introducing a method for combining positive and negative test cases to further reduce the effort of test execution.

9.1 Correcting Numerical Input Faults and a Case Study

Input validation testing chooses test data that attempt to show the presence or absence of specific faults endemic to input tolerance [51]. Decision tables (see Section 7.2) will be used to visualize the Boolean algebraic constraints on input data and are supplemented with DbC patterns so that rules of the decision table are refined to precondition rules for numerical input validation. Equivalence class partitioning and boundary value approaches support the test data generation process [114, 68].

The approach combines input validation with static analysis for evaluating given constraints. Input validation checks the syntax and, to a degree, the semantics of the information provided by user via GUI [52]. Because input validation errors may lead to malfunctions of the entire system as well as to vulnerabilities for attacks [80], various specification-based and implementation-based test techniques exist to validate user interfaces [51].

Static analysis techniques are used to handle buffer overflow problems, which are one of the common security issues since they may lead to vulnerabilities such as system crash, corruption of data, or undesirable system access. They occur when a programmer implements incorrect bound checks on buffer size or even fails to perform bounds checking when data is written into a fixed length buffer [74]. By definition, buffer overflow is similar to boundary overflow, which is an input error that occurs when values are entered which violate the range of values. Such entries exceed the implicitly or explicitly specified but not implemented boundary values.

According to Chess and McGraw [34], static analysis tool BOON applies integer range analysis to determine whether a C program can index an array outside its bounds. UNO, another static analysis tool, accepts user-defined properties of application specific requirements to overcome specific problems [55]. Kolmonen [62] introduces taint propagation as a technique used by static analysis tools to find software vulnerabilities caused by failed or missing input validation. In taint propagation, the tool tracks the tainted data, including also the parts of the program on which the tainted data has an effect. A taint analysis is performed to find the places where data is read from an untrusted source [110], e.g., by using Patterson's value range propagation algorithm for calculating the range of possible values for each

variable [98].

There exist some approaches that adopt the DbC-idea for testing. Zheng et al. [131] introduced an UML-based software component testing technique called *test by contract*. There are also some contract-based testing techniques focused on web service testing [53]. Languages such as Phyton, C++, Java are extended to comply with DbC for catching bugs [50]. In [100], the DbC concept is integrated into the programming language Python and adopted by adding mechanisms for dynamic type checking of method parameters and instance variables. Guerreiro used DbC in C++ by using and inheriting the `Assertions` class [50].

However, all of these techniques lack clearly arranged representations that enable a systematic evaluation. Therefore, this work attempts to provide a simple, nonetheless powerful, representation of contracts for checking an SUC on numerical input vulnerabilities.

9.1.1 Fault Correction Using Design by Contract Patterns

The numerical input validation approach proposed here is composed of three phases: (i) modeling DbC patterns by DTs, (ii) testing the SUC with the test cases generated by DbC supplemented DTs, and (iii) detecting/correcting deficient input validation code if errors are found during the test phase.

9.1.1.1 Modeling Design by Contract Patterns by Decision Tables

DbC is an object-oriented design technique that was first introduced by Meyer in 1992 [78]. DbC focuses on the extension of the source code, in this case a function, by pre-conditions, post-conditions, and invariants that can be evaluated during run-time (similar to a legal contract). Pre-conditions have to be fulfilled before the function is executed; post-conditions have to be ensured after the function has been executed. Invariants are conditions that must hold anytime the function is invoked [53]. Software components are extended by those pre-conditions, post-conditions, and invariants so that compliance with them can be verified during run-time.

Although DTs (see Section 7.2) can contain a wide variety of constraints, they are classified here into three groups—namely pre-conditions, post-conditions, and

invariants—by utilizing DbC patterns for automation purposes. Now, the definition of rules given by Definition 7.2 (on page 85) is redefined for considering DbC concepts and, moreover, exception messages to be thrown.

Definition 9.1 (Rule)**.** Let C_{True} and C_{False} be defined as in Definition 7.2 and E be defined as $E = E_{xcpt} \cup E_{ui}$ with E_{xcpt} containing exception messages and E_{ui} containing possible user interactions. Then a *rule* is defined as $R_i = (t, C_{True}, C_{False}, E_x)$ where

- $t \in \{t_<, t_>, t_{<>}\}$ is a time marker with
  - $t_<$ indicating a pre-condition,
  - $t_>$ indicating a post-condition,
  - $t_{<>}$ indicating an invariant,
- $E_x \subseteq E_{ui} \times \{E_{xcpt} \cup \varepsilon\}$ with ε defining an empty exception.

Example 9.2. The following sets and rules are given for the DT presented in Table 9.1:

$E_{ui} = \{accept,\ abort,\ btn_3\}$
$E_{xcpt} = \{Exception1,\ Exception2,\ Exception3\}$
$R_1 = (\{t_<\},\ \{Condition\,1,\ Condition\,2\},\ \{\},\ \{(accept,\ \varepsilon),\ (abort,\ \varepsilon),\ (btn\_3,\ \varepsilon)\}$
$R2 = (\{t_<\}, \{Condition\,1\}, \{Condition\,2\}, \{(accept,\ Exception\,1),\ (abort, \varepsilon),\ (btn_3,\ Exception\,3)\})$
$R3 = (\{t_<\}, \{Condition\,2\}, \{Condition\,1\}, \{(accept, Exception\,2),\ (abort, \varepsilon), (btn_3, \varepsilon)\})$
$R4 = (\{t_<\}, \{\}, \{Condition\,1, Condition\,2\}, \{(accept, Exception\,1),\ (accept, Exception\,2), (abort, \varepsilon), (btn_3, Exception\,3)\})$

As an example, rule R3 of Table 9.1 reads as follows: If $Condition\,1$ is resolved to false and $Condition\,2$ is resolved to true, a press of $accept$ button results in $Exception\,2$, a press of $abort$ or btn_3 will throw no exception and therefore the input is accepted. Pre-conditions, post-conditions, and invariants are used to supplement DTs with specific classes of rules.

Table 9.1: Example of a refined DT with exceptions

$t_<$	R1	R2	R3	R4
Condition 1	T	T	F	F
Condition 2	T	F	T	F
accept	✓			
Exception 1		✓		✓
Exception 2			✓	✓
abort	✓	✓	✓	✓
btn_3	✓		✓	
Exception 3		✓		✓

9.1.1.2 Testing Design by Contract Patterns by Decision Tables

In general, input validation requires to consider three validation types: isolated validation, interdependency validation, and service-specific validation [111], as depicted in Figure 9.1. Isolated validation checks boundary conditions (restrictions) and interdependency validation checks relations between the variables (dependencies). Service-specific validation considers conditions related to business or service.

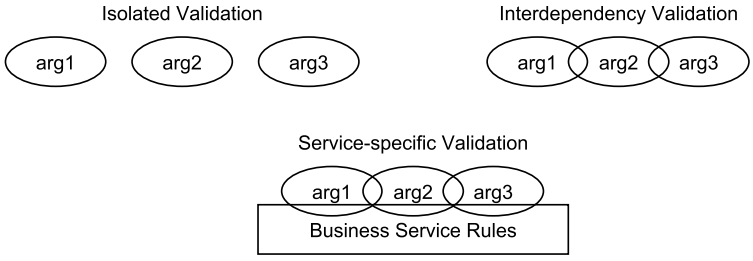


Figure 9.1: Input validation types

As an example, consider port values. For isolated validation, the considered variables should be between the port ranges (0-65535). The restriction that the minimum port value should be lower than the maximum port value is associated with interdependency validation. The dynamic and/or private ports are from 49152 to 65535 [57]. No ports can be registered in the dynamic range because they are com-

monly used by operating system kernels. The port allocations are only valid for the duration of the session of the connection. The port values between the dynamic ranges are not valid for service-specific validation when the session is closed, although the values are inside the port ranges and the dependency requirement holds for the port values. DTs enable to express all three of the validation types, leading to a complex system of constraints to be solved.

In general, there exist two different approaches to solve a system of constraints: constraint solving and constraint satisfaction. Whereas constraint solving tries to find a solution mathematically, constraint satisfaction is based on search-based algorithms. As already mentioned in Section 7.3, the automatic generation of data out of DTs can be done by a solution of the constraint satisfaction problem (CSP). CSPs are distinguished by the type of variables and constraints [104, 6]. Variables can be (1) *discrete* with finite or infinite domains or (2) *continuous*. Note that variables with infinite domains can be transformed into variables with finite domains whenever it is possible to set an upper and lower bound to the infinite domain.

Each rule of the DT represents a CSP. Note that constraints have to be set to true or false according to the sets C_{True} and C_{False} before submitting them to the CSP. But how is this data to be generated? A simple solution is to enumerate all possible combinations and check for valid solutions. But this is a very inefficient solution and not even possible for variables with infinite or continuous domains. Another solution is to build a constraint graph $G = \{V, E\}$ where nodes $v \in V$ symbolize the variables and edges $e \in E = \{(v_i, v_j) | v_i, v_j \in V\}$ are annotated with the constraints [104]. An example can be seen in Figure 9.2. The underlying constraint set is as follows:

$$C = \{A \geq 2B; B < C - 2; B \leq D - 1; C = D + 5; E \geq 3D\}$$

On the basis of this graph, different search algorithms based on depth-first search or breadth-first search can be run. The advantage of depth-first search algorithms is that they are able to return a single solution faster than breadth-first search algorithms. The basic idea of both variants is to start with a single variable by assigning a value and extending the solution by assigning values step by step to the other variables. If the assignment of a value is not possible due to previously selected values,

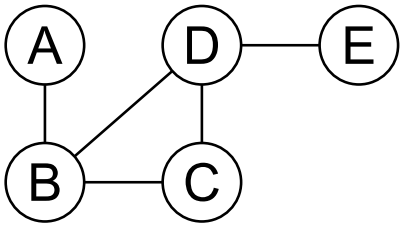


Figure 9.2: Constraint graph

the algorithms go one step back and assign another value to the previous variable.

Further details and techniques to improve this process are also given in [6]. It should be noted that a CSP can be transformed into a Boolean satisfiability problem [25] (also known as SAT-Problem) as described by Le Berre [66]. But this will not be part of this thesis.

Equivalence class testing partitions the input space into equivalence classes according to the input conditions. Each rule of the DT characterizes an equivalence class by means of the underlying constraint set. The whole set of possible solutions to a given constraint set (or rule) forms the corresponding equivalence class for this constraint set. It is assumed that the SUC has the same behavior on all elements [71] within an equivalence class. This assumption allows to select exactly one solution from each equivalence class to represent this class. The selection can be strengthened according to boundary value analysis [114, 68], which complements the equivalence partitioning by selecting a value at the edges of a class [68].

DTs also provide the oracle for the selected input values because they are defined along the cause-effect testing approach where the input conditions represent the causes and events represent the effects [71]. Algorithm 9.1 shows the test case generation algorithm where a test case consists of input values and the expected behavior (namely the expected events) of the system. Test cases are generated for all the rules in the DT; that is, the input values of a single test case are selected according to a rule's conditions and the expected output is defined by the allowed successor event of the same rule.

Algorithm 9.1: Test case generation algorithm

input : decision table
output: test cases

1 **foreach** $rule = (t, C_{True}, C_{False}, E_x) \in decision\ table$ **do**
2 set each constraint $c \in C_{True}$ to true;
3 set each constraint $c \in C_{False}$ to false;
4 build a constraint graph;
5 derive a solution by solving the CSP;
6 combine the solution with E_x;

9.1.1.3 Detection and Correction

A detection algorithm is proposed to check the error handling mechanism of the SUC related to validation of numerical inputs. The algorithm scans the source code statically, detects the points that may cause problems (possible violation of boundaries), and checks the error handling mechanism of the SUC against validation errors. The deficient parts of the error handling mechanism related to numerical input validation are identified first. Once detected, a mechanism is required to correct the deficiencies. The correction mechanism relies upon the DbC technique discussed in Section 9.1.1.1. The correction algorithm provides an error handling mechanism through extension of the source code by pre-condition contract methods where control for the numerical input validation vulnerability does not exist. The suggested algorithm inserts an error handling mechanism with following features:

- insertion of necessary control conditions into the source code,
- generation of related error messages when inputs are given that lead to boundary values, and
- termination of the currently executed function.

The boundary overflow vulnerability detection algorithm given by Algorithm 9.2 consists of five steps. In Step 1, the variable definitions with the specified types are obtained and the variables (as type, name, defined file, defined line, defined function) are entered into the hash table. In Step 2, the variables in the hash table are

Algorithm 9.2: Detection and correction algorithm

input : a *decision table*
output : corrected code

```
1  n := number of lines;
2  hashtable := ∅ ;                                        // a hash table
   // Step 1. Obtain the variable definitions with their types
      and add them to the variable list
3  for line i := 1 to n do
4      if line i contains a variable definition of specified types then
5          add the variable to hashtable with (type,name,definedfile,definedline);
   // Step 2. Match the conditions to variables
6  Match the variables with the conditions defined in decision table;
   // Step 3. Detect GUI input lines
7  for line i := 1 to n do
8      if line i contains a GUI input then
9          if assigned variable is of string type then
10             trace the string variable and find the string to integer or double conversion line;
11             var := lookup the assigned variable from hashtable;
12             if var.boundarycondition = true then
13                 var.usedfile := currentfile;
14                 var.traceline := currentline;
   // Step 4. Trace the variables
15 for variable.traceline i := 1 to n do
16     find the first line where the variable is used;
17     if line i contains a control statement (if, while, for) then
18         parse the expression(s);
19         var := lookup the variable(s) used in the expression from hashtable;
20         if var.boundarycondition = true AND
           expression = complement(var.condition) then
21             var.condition.check := true;
22         else if line i contains a GUI input then
23             go back to Step 3;
   // Step 5. Apply correction mechanism
24 for i = 1 to number of all the variables in hashtable do
25     if variable.boundarycondition = true AND variable.condition.check = false
       then insert the related error handling code after the conversion line;
```

matched with the conditions defined in the DT. Matched variable's boundary condition is set to true. Step 3 of the algorithm detects the points that may cause problems (input assignments from the user interface input) and sets the trace line of the variable as the current line. Step 4 traces the variables from their trace lines and detects the lines where the variables are used first. If a variable is used in a control statement (such as *if*, *while,* or *for*), the expression(s) in the statement are parsed. After that, the conditions are compared with the defined conditions of the variable. If the parsed expression complies with the defined condition of the variable, condition check of the variable is set to true (that is, if the defined condition is $a > 0$, the expression should be $a <= 0$ to catch the undesired input). Step 5 of the algorithm applies the correction mechanism. The error handling code is inserted after the trace line of the variable if the condition check of the variable is missing.

9.1.2 Case Study III: Fault Correction of Three Port Scanners

A case study evaluates the characteristics of the approach with respect to its strength and limitations. Furthermore, it analyzes its applicability to real-world applications.

9.1.2.1 System Under Consideration, Test Setup, and Goals of the Experiment

The approach has been evaluated by means of three port scanners. A port scan function analyzes a single port or a range of ports, that is, ports between a given minimum and maximum, to check whether they are open or not. The case study is exemplified on the basis of the port scanner part of the open source firewall software Netdefender (version 1.5) [89]. Its GUI is shown in Figure 9.3. The boundary restrictions of the port scan function are modeled by Table 9.2.

The second and third evaluations were performed on port scanners named Multiscan (version 0.8.5) [82] and Pscan [101]. They are open source port scanners coded in C++, which allow you to scan a range of IP addresses and ports.

The goals of the experiment are, on the one hand, to demonstrate the applicability of the approach introduced in Section 9.1.1 and, on the other hand, to measure its effectiveness in detecting and correcting numerical input faults. Hence, test values in the experiment are derived from Table 9.2. These test values are evaluated on

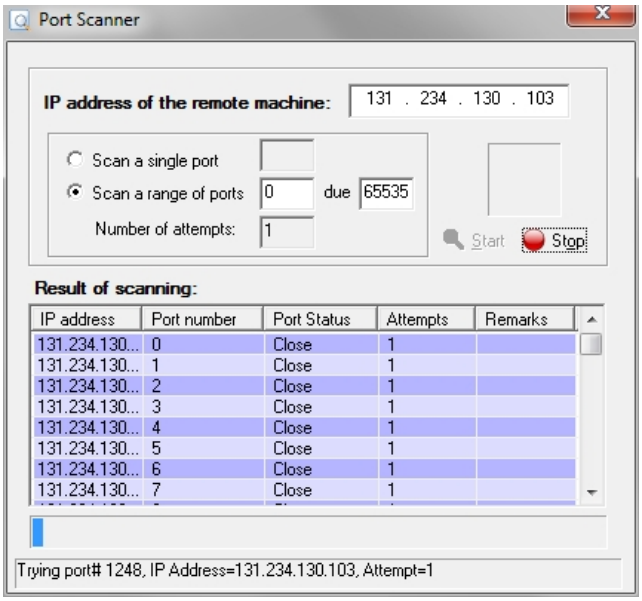


Figure 9.3: Netdefender Port Scanner

Table 9.2: Decision table for entering ports

	Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
C1	min >= 0	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
C2	min <= 65535	T	T	T	T	T	T	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T
C3	max >= 0	F	F	F	T	T	T	F	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T
C4	max <= 65535	T	T	T	F	T	T	T	F	F	F	T	T	T	T	F	F	T	T	T	T	T	T	T	T
C5	min < max	F	F	T	T	T	T	T	F	F	T	F	F	F	F	T	T	F	F	F	F	F	T	T	T
C6	min = max	F	T	F	F	F	F	F	F	T	F	F	F	F	F	F	F	F	F	F	T	T	F	F	F
C7	min > max	T	F	F	F	F	F	F	T	F	F	T	T	T	T	F	F	T	T	T	F	F	F	F	F
C8	min < 49152	T	T	T	T	T	T	F	F	F	F	F	F	F	T	F	T	F	F	T	F	T	F	T	T
C9	max < 49152	T	T	T	F	F	T	T	F	F	F	F	T	T	T	F	F	F	T	T	F	T	F	F	T
	Events																								
E1	button „Start“																								✓
E2	Exception 1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								
E3	Exception 2	✓	✓						✓	✓		✓	✓	✓	✓			✓	✓	✓	✓	✓			
E4	Exception 3				✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓		✓		✓	✓	

the port scanners before and after applying Algorithm 9.2, and the number of faults detected are compared with each other.

9.1.2.2 Test Execution and Tool Support

The test process for each of the port scanners is summarized in Figure 9.4. As a first step, the approach generates test cases by using DTs. Equivalence class testing supplemented with boundary value analysis is used to generate test cases, which are selected from the values that are at the edges of each equivalence class. As a second step, the SUC is tested in a real environment by entering these values in its user interface. The faults are obtained and recorded. After applying the proposed detection and correction method, the new corrected version of the SUC is tested in the real environment again. The faults detected before and after applying the correction method are compared.

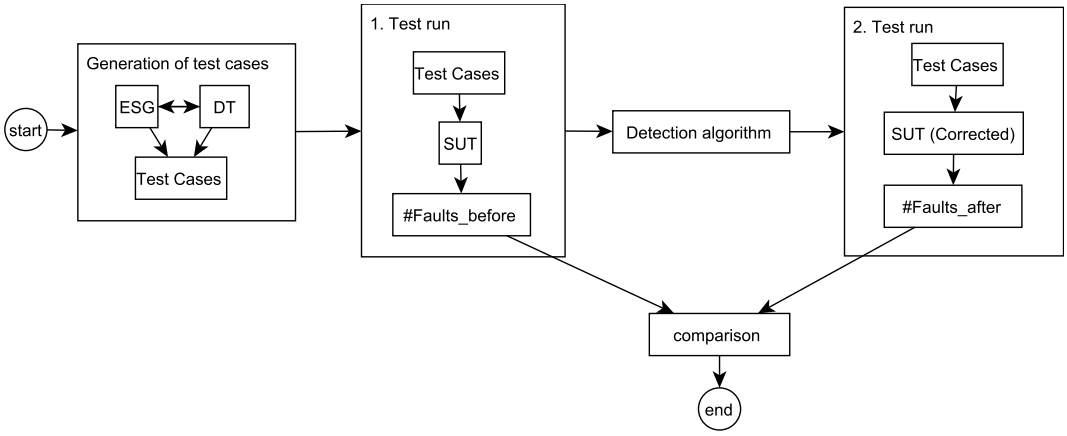


Figure 9.4: Summary of the approach

A numerical input validation analysis tool that is able to analyze software developed in C++ was developed in Java for the implementation of the approach. Functioning as a static analysis tool, it analyzes the source code of SUC, finds the deficient parts that may cause numerical input validation vulnerabilities, and extends the source code by inserting pre-condition functions to ensure that the specified conditions hold before the inputs are processed. The class `Assertions` [50] that

provides the functions required for emulating pre-conditions and post-conditions is used for exception handling where, in the current case, only the pre-conditions are considered. The tool inserts its `Require` function for numerical input validation wherever a control mechanism is absent. The numerical input validation analysis tool accepts two inputs: (1) the directory of the software to be analyzed and (2) DbC supplemented DTs for the GUI. The given implementation requires a manual matching of the listed variables with the pre-conditions of the DT. The tool identifies the variables that have the boundary condition and displays the conditions of the variables as well as whether or not condition checks exist in the source code relative to the numerical input validation. The correction mechanism is applied by informing the user about the insertion of the exception handling code where the pre-condition checks do not exist. Figure 9.5 shows the corresponding GUI of the tool that enables to input the source directory of the software to be checked, shows the detection steps, suggests corrections, and displays the outputs.

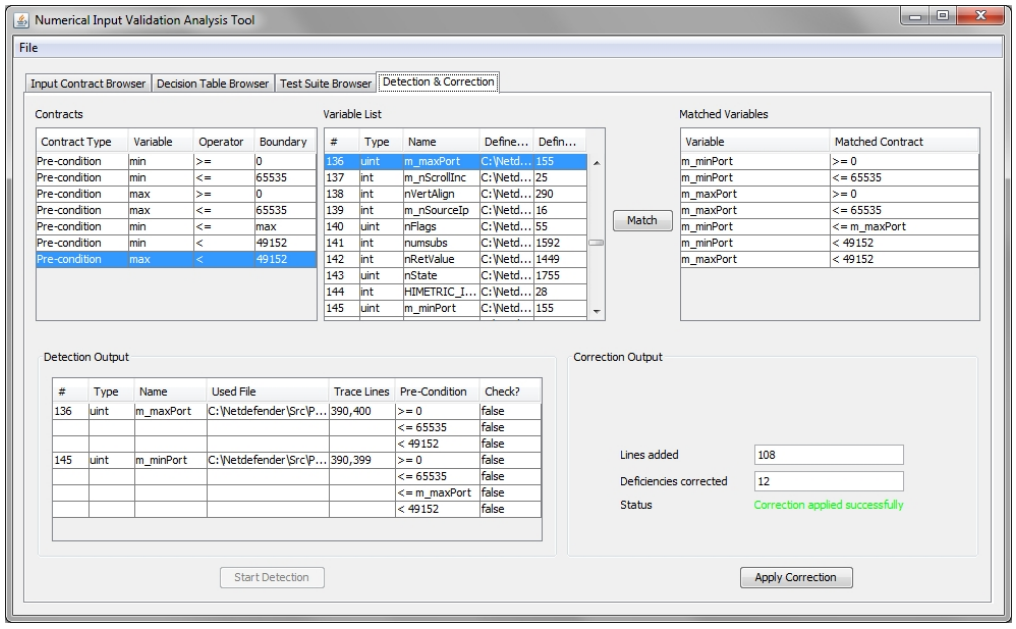


Figure 9.5: Numerical input validation analysis tool - graphical user interface screen

9.1.2.3 Results and Their Interpretation

Test cases are generated for the minimum and maximum port from Table 9.2 using Algorithm 9.1. The generated test values can be seen in Table 9.3.

The port scanner is evaluated in a local area network (LAN) and the generated test values are applied as inputs to the GUI of the port scanner. The user interface outputs are obtained and the network packet outputs are captured. Table 9.4 summarizes the results by showing the test values as input pair, GUI, and network packet outputs, state of the case (erroneous or not), and the error message.

To sum up, the cases with "out of boundary" input pairs give rise to problems in the network environment. In certain cases (2, 3, 7, 9, 10, 13, 14, 15, 16), there are faulty input pairs that are out of boundary values, but the program behaves as if they were not faulty. This is critical because the program does not abandon processing the related task; hence the resulting situation forces the program to work erroneously. In some cases (2, 3, 7, 13, 14), the client does not stop sending the TCP packets to the target computer, but rather continues sending the packets in an infinite loop, generating a flood in the LAN.

After the insertion of control statements related to the boundary constraints in the port scanner of Netdefender firewall, the software is evaluated in the LAN again and the generated test cases are applied as inputs to the GUI of the port scanner. The outputs considerably differ from the ones in Table 9.4. In erroneous cases (1-19), the software outputs the right error message and aborts sending the packets.

As in the first evaluation, it has been observed that also the second and third SUC have no exception handling mechanisms. The control mechanisms against out of boundary values are deficient for the three port scanners. Hence, in all three cases, the tool inserted control statements to fulfill the deficiencies of the software.

Results in a Nutshell

An overview of the three test runs comparing the number of faults detected before and after the detection algorithm can be seen in Table 9.5. It is evident that the tool has successfully carried out detection and correction operations. Analysis of the results encourages the generalization that boundary overflow vulnerabilities were not considered and thus countermeasure actions were neglected during soft-

Table 9.3: Test cases generated from rules of the DT given in Table 9.2

rule	test values
R1	(-1,-2)
R2	(-1,-1)
R3	(-2,-1)
R4	(-1,65536)
R5	(-1,49152)
R6	(-1,0)
R7	(65536,-1)
R8	(65537,65536)
R9	(65536,65536)
R10	(65536,65537)
R11	(65536,49152)
R12	(65536,0)

rule	test values
R13	(49152,-1)
R14	(0,-1)
R15	(49152,65536)
R16	(0,65536)
R17	(49153,49152)
R18	(49152,0)
R19	(1,0)
R20	(49152,49152)
R21	(0,0)
R22	(49152,49153)
R23	(0,49152)
R24	(0,1)

Table 9.4: Outputs of the test cases for Netdefender firewall

#	input pair	GUI output	network packet	erroneous?	error message?
1	(-1,-2)	no output	no packet	yes	message 1
2	(-1,-1)	(-1,0,1,2,3,...∞)	65535,1,2,...65535…	yes	no
3	(-2,-1)	(-2,-1,0,1,2,3,...∞)	65534,65535,1,2,...65535…	yes	no
4	(-1,65536)	no output	no packet	yes	message 1
5	(-1,49152)	no output	no packet	yes	message 1
6	(-1,0)	no output	no packet	yes	message 1
7	(65536,-1)	(65536...∞)	1,2,...65535,1,2,....65535…	yes	no
8	(65537,65536)	no output	no packet	yes	message 1
9	(65536,65536)	(65536)	no packet	yes	no
10	(65536,65537)	(65536,65537)	1	yes	no
11	(65536,49152)	no output	no packet	yes	message 1
12	(65536,0)	no output	no packet	yes	message 1
13	(49152,-1)	(49152...∞)	49152,49153,...65535,1,2,...65535…	yes	no
14	(0,-1)	(0...∞)	1,2,...65535,1,2,...65535…	yes	no
15	(49152,65536)	(49152...65536)	49152,49153,…65535	yes	no
16	(0,65536)	(0...65536)	1,2,…65535	yes	no
17	(49153,49152)	no output	no packet	yes	message 1
18	(49152,0)	no output	no packet	yes	message 1
19	(1,0)	no output	no packet	yes	message 1
20	(49152,49152)	(49152)	49152	no	
21	(0,0)	(0)	no packet	no	
22	(49152,49153)	(49152,49153)	49152,49153	no	
23	(0,49152)	(0,49152)	1,2,…49152	no	
24	(0,1)	(0,1)	1	no	

Table 9.5: Comparison of the three test runs

		# faults detected		benefit of the approach
software	**# test cases**	**before**	**after**	**(% of faults corrected)**
Netdefender	24	19	0	100 %
Multiscan	24	10	0	100 %
Pscan	24	4	0	100 %

ware development. Therefore, tools as introduced in this work might be useful in preventing likely failures or undesirable situations that may occur as a consequence of deficiency control mechanisms in the software.

9.1.3 Conclusions

This section proposed a solution for the numerical input validation problem and reported the experience gained through experiments as described in the case study. DTs supplemented with DbC patterns were used for modeling input data restrictions and generating test cases for input validation. An algorithm was introduced to validate the exception handling mechanism of SUC related to invalid numerical inputs and provide the necessary exception handling mechanism where none existed. A tool supported the deployment of the algorithm introduced. Three port scanners were tested to evaluate this tool. Results of those tests show that the approach is very effective for finding deficiencies in the exception handling mechanism of SUC concerning boundary overflow problems. Moreover, the approach includes appropriate checks to compensate for those deficiencies of SUC.

However, the approach reveals a limitation when considering numerical inputs. Modern systems collect different types of data, such as strings, dates, and so on, which need to be considered as well. Furthermore, the matching of variables to their appropriate constraints has to be done manually and needs further improvements for a complete automation of the approach. The case study showed that the fault detection and correction works well for input data. Thus, it is desirable to extend the approach to other types of faults, e.g., to the automatic detection and correction of faults in sequences of events as given by ESGs.

9.2 Positive and Negative Testing Revisited: Cost Reduction Through Combination

The test process introduced in Section 3.2 requires that each FEP forms its own FCES because the execution of an FEP as a test case leads the SUC into an undefined state. Very often, events correspond to elements on the GUI that are executed, e.g., clicking a hyperlink, button, etc. Nowadays, test tools for automatic test execution support validation of the element-properties of a GUI. An example of such a tool is Selenium [107], which can check whether an element, e.g., a button or link, is present or not in a web application. Instead of activating an element, this tool can check whether an element is available and/or (if necessary) enabled. Thus, it is not necessary to execute the faulty event itself; instead, preconditions for its execution can be defined and checked. This helps to reduce test costs. If the faulty (second) event of an FEP is executable, the test is judged as failed, otherwise as passed.

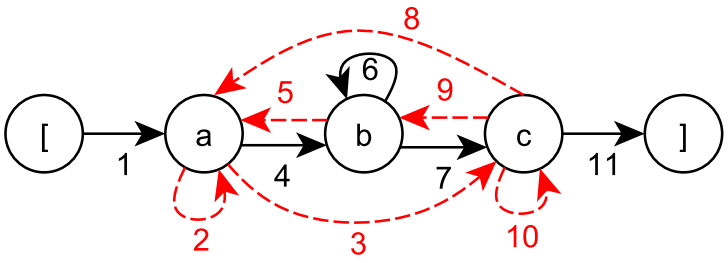


Figure 9.6: Improved negative test execution

A simple ESG is shown in Figure 9.6. There exists exactly one CES that covers all EPs. The six dashed edges represent FEPs. As in the previous approach, six FCESs must be generated. The resulting test case set would be as follows:

```
CES1 =[ a b b c ]
FCES1=[ a a
FCES2=[ a c
FCES3=[ a b a
FCES4=[ a b c a
FCES5=[ a b c b
FCES6=[ a b c c
```

Now, instead of applying the negative test cases `FCES1` to `FCES6` one by one to the SUC, their corresponding FEPs are checked during the execution of the positive test case `CES1`:

1. execute event a
2. *check, if event a would be executable*
3. *check, if event c would be executable*
4. execute event b
5. *check, if event a would be executable*
6. execute event b
7. execute event c
8. *check, if event a would be executable*
9. *check, if event b would be executable*
10. *check, if event c would be executable*.

For this example, instead of executing seven test sequences in total, execution of one test sequence is sufficient. Furthermore, the number of events could be reduced from 23 to 10.

Algorithm 9.3 sketches the corresponding test process. Note that this algorithm considers only FEPs, that is, FESs of length 2. Furthermore, this test process is based on the assumption that events can be checked without executing them. This assumption strongly depends on the SUC and the test execution environment used; e.g., an adaptation of this approach to the ESG4WSC approach is not possible. The reason is that the fault classes given in negative testing can not be checked without execution.

Algorithm 9.3: Test process

```
1  cover all ESs of length k by means of CESs;
2  FEP := Ē with Ē = Ê \ E and Ê = V × V;
3  foreach ces ∈ CES do
4      m := length of the ces;
5      for i := 1 to m do
6          if event i executable then
7              execute event i;
8              foreach (i, j) ∈ FEP do
9                  check, if event j would be executable;
10                 if event j would be executable then
11                     mark faulty event pair (i, j) as failed;
12                 else mark faulty event pair (i, j) as passed;
13                 FEP = FEP \ (i, j);
14         else mark ces as failed and continue with next ces ∈ CES;
15     mark ces as passed;
```

Chapter 10

Conclusions

Model-based testing is an attractive approach to testing since, depending on the underlying model features and the test criteria considered, test cases can be derived systematically, even automatically. Additionally, no source or binary code is necessary to generate tests for the system under consideration (SUC).

Most existing test approaches generate test cases as sequences of events of different length. The cost of the test process mainly depends on the number and total length of those test sequences. One of the interesting questions currently discussed in software testing, both in practice and academia, is the role of test sequences and their length on software testing, especially on fault detection. The prevailing belief is "the longer, the better"; that is, the longer the test sequences, the more faults will be detected. However, there is no evidence of this "length" hypothesis; that is, that an increase in the test sequence length really affects the fault detection.

A second problem accompanies model refinement. Most of the modeling techniques require creating a hierarchy of models since the state space of large, complex systems can be impracticable to be modeled in one, single step. Nevertheless, this hierarchy has to be resolved before test cases are generated, which is called full resolution (FR). This can, however, lead to a large model and an unfeasibly great number of test cases and substantial test generation effort. Thus, the deeper the model hierarchy is, the more time and labor consuming are the test case generation and test execution. Nevertheless, the belief is still "the deeper the better," meaning the more layers fully resolved, the more faults will be detected. However, like the

"length" hypothesis, there is no evidence of this "depth" hypothesis; that is, that an increase in the considered hierarchy depth, e.g., a thorough testing of all layers, actually affects the fault detection.

This thesis analytically and empirically investigated the "length" and "depth" hypotheses; that is, the impact of the test length and model refinement given through the hierarchy depth on effectiveness and efficiency of test generation and test process. The layer-centric testing (LC) approach was introduced to solve these problems. Based on event sequence graphs (ESG), the LC strategy makes use of specific features of the hierarchical structure of the model and contributes to considerably reducing costs for test generation and execution. In order to select critical components that are likely to hide more faults than others, the LC approach was extended, leading to selective layer-centric testing (SLC). To the author's best knowledge, there is no approach available in literature that simultaneously solves the "length" and "depth" problem.

SLC is flexible and can be applied to many domains. This was demonstrated by extending ESG for modeling and testing web service compositions. The strength of the extension stems from its potential to fit cases well for which no technical specifications, e.g., via BPEL or WS-CDL, are available. Other testing approaches strictly require the availability of these artifacts. Thus, the approach introduced is independent of the type of composition which can be either orchestration or choreography and, therefore, allows to simultaneously perform the steps for implementation and testing of the SUC. To the author's best knowledge to date, the present work is the only one that describes how an enterprise service bus can support the test execution in performing the necessary observations and modifications of exchanged messages as well as provoking unexpected situations.

Apart from the above described analytic research, the thesis conducted two large case studies to determine the trade-off of the LC and SLC strategies, subject to the overall reliability of the SUC. Significant results of these empiric studies are as follows.

1. "Length" hypothesis: In both case studies, I and II, the fault detection capability of test suites of higher length lagged far behind expectation. Test suites covering event sequences of length 2 detected most of the faults. Test sequences for covering event sequences of length 3 contributed very little, and

the execution of test sequences for covering event sequences longer than 4 made no sense as they obviously have no or only very minor chances of detecting new faults. This result was independent of the chosen strategy, that is, FR, LC, and SLC.

2. "Depth" hypothesis:

   (a) Case study I showed that LC testing, which avoids a full resolution by making use of the hierarchical structure, leads to a cost reduction of about 80% at a reliability level that is comparable with the one achieved by FR testing.

   (b) Additionally, case study I showed that SLC strategy, which selects critical components instead of considering them all, further reduced costs by approximately 30% compared to LC testing, and 85% compared to FR testing at a reliability level close to the ones achieved by LC and FR testing.

   (c) In case study II, SLC even reduced the test effort by 90% and achieved a similar reliability level.

3. The approach introduced in this thesis and the reliability models applied, along with the case studies, allow the decision of when to stop testing, that is, without having to perform all potentially feasible number of tests while achieving the same reliability level.

The results of both case studies show that SLC testing can especially be recommended for cases with limited test budgets that can only afford a small number of test cases, e.g., in smoke testing. The optimization problem for generating a minimal test suite along LC and SLC leads to the assignment problem. It turns out, however, that the specific constellation of "length" and "depth" problems requires a derivation of the assignment problem to consider cases in which only *one* assignment of an item (agent) to another item (task) is required within a given set of *several* items (instead of finding an assignment for *all* items in the set). Here, this specific assignment problem was solved by linear programming—more precisely, by the simplex method.

In sketching out the further perspectives of the proposed approach, a concept of combining positive and negative testing was introduced to reduce the test costs. Finally, a rudimentary concept for automating the correction of faults in the source code was presented, which was evaluated by the third case study.

It is this author's observation that the assignment problem, along with Chinese postman problem, has received relatively little attention in software testing literature. It is the hope of the author that this thesis might have demonstrated that this class of optimization problems contains further valuable potential for cost minimization and thus deserves the attention of researchers in software testing.

Bibliography

[1] A. H. Aho, A. T. Dahbura, D. Lee, and M. Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615, Nov. 1991.

[2] AIAA. Recommended practice for software reliability. *AIAA R-013-1992*, 1992.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[4] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, 2005.

[5] A. Arcuri. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Transactions on Software Engineering*, 38:497–519, 2012.

[6] R. Bartak. Constraint propagation and backtracking-based search. *First International Summer School on Constraint Programming*, 2005. `http://kti.mff.cuni.cz/~bartak/downloads/CPschool05notes.pdf`.

[7] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition, 1990.

[8] F. Belli. Methoden und Hilfsmittel für die systematische Prüfung komplexer Software. *Informatik Spektrum*, 21(6):337–346, 1998.

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

[10] F. Belli, M. Beyazit, A. Hollmann, M. Linschulte, and S. Padberg. Ereignisbasierter Test grafischer Benutzeroberflächen - ein Erfahrungsbericht. *GI-Softwaretechnik-Trends*, 30(2):21–23, 2010.

[11] F. Belli, M. Beyazit, and A. Memon. Testing is an event-centric activity. In *Proceedings of the 6th IEEE International Conference on Software Security and Reliability (SERE)*, pages 198–206. IEEE Computer Society, 2012.

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

[13] F. Belli, C. J. Budnik, M. Linschulte, and I. Schieferdecker. Testen Web-basierter Systeme mittels strukturierter, graphischer Modelle - Vergleich anhand einer Fallstudie. In *GI Jahrestagung (2)*, volume 94 of *Lecture Notes in Informatics*, pages 266–273. Gesellschaft für Informatik (GI), 2006.

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

[15] F. Belli, A. T. Endo, M. Linschulte, and A. S. Simao. Model-based testing of web service compositions. In *Proceedings of the 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, pages 181–192. IEEE Computer Society, 2011.

[16] F. Belli, N. Güler, and M. Linschulte. Are longer test sequences always better? - a reliability theoretical analysis. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C)*, pages 78–85. IEEE Computer Society, Jun. 2010.

[17] F. Belli, N. Güler, and M. Linschulte. Does "depth" really matter? On the role of model refinement for testing and reliability. In *Proceedings of the*

35th IEEE International Computer Software and Applications Conference (COMPSAC), pages 630–639. IEEE Computer Society, Jul. 2011. Best Paper.

[18] F. Belli, N. Güler, and M. Linschulte. Layer-centric testing. In *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS)*, pages 88–95. VDE Verlag, 2011.

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

[20] F. Belli and M. Linschulte. Event-driven modeling and testing of real-time web services. *Service Oriented Computing and Applications*, 4(1):3–15, 2010.

[21] F. Belli, M. Linschulte, C. J. Budnik, and H. A. Stieber. Fault detection likelihood of test sequence length. In *Proceedings of the 3rd IEEE International Conference on Software Testing Verification and Validation (ICST)*, pages 402–411. IEEE Computer Society, 2010.

[22] F. Belli, M. Linschulte, R. Zirnsak, and G. Hofmann. "Negativ"-Tests interaktiver Systeme und ihre Automatisierung. In *Workshop Proceedings of the Conference on Software Engineering (SE)*, volume 106 of *Lecture Notes in Informatics*, pages 35–44. Gesellschaft für Informatik (GI), 2007.

[23] A. Benharref, R. Dssouli, R. Glitho, and M. A. Serhani. Towards the testing of composed web services in 3rd generation networks. In *Proceedings of the 18th International Conference on Testing of Communicating Systems (TESTCOM)*, pages 118–133. Springer-Verlag, 2006.

[24] L. Bentakouk, P. Poizat, and F. Zaïdi. A formal framework for service orchestration testing based on symbolic transition systems. In *Proceedings of the 21st International Conference on Testing of Software and Communication Systems (TESTCOM)*, pages 16–32. Springer-Verlag, 2009.

[25] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Feb. 2009.

[26] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[27] A. Birolini. *Reliability Engineering*. Springer-Verlag, Berlin, 5th edition, 2007.

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

[29] M. Bozkurt, M. Harman, and Y. Hassoun. Testing web services: A survey. Technical Report TR-10-01, Department of Computer Science, King's College London, Jan. 2010.

[30] R. E. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 2009.

[31] G. Canfora and M. Di Penta. Software engineering. chapter Service-Oriented Architectures Testing: A Survey, pages 78–105. Springer-Verlag, 2009.

[32] A. Cavalli, T.-D. Cao, W. Mallouli, E. Martins, A. Sadovykh, S. Salva, and F. Zaidi. Webmov: A dedicated framework for the modelling and testing of web services composition. In *Proceedings of the 17th IEEE International Conference on Web Services (ICWS)*, pages 377–384. IEEE Computer Society, Jul. 2010.

[33] K. S. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A fault taxonomy for web service composition. In *Workshop Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC)*, pages 363–375. Springer-Verlag, 2007.

[34] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, Nov. 2004.

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

[36] DIN. Guidance on software aspects of dependability (IEC 56/1349A/CD: 2009). *DIN IEC 62628*, 2010.

[37] DIN. Analysis techniques for dependability–Petri net techniques (IEC 56/1476/FDIS:2012-07). *DIN IEC 62551*, 2012.

[38] J. Dolbec and T. Shepard. A component based software reliability model. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 19–29. IBM Press, 1995.

[39] J. T. Duane. Learning curve approach to reliability monitoring. *IEEE Transactions on Aerospace*, 2(2):563–566, Apr. 1964.

[40] A. Endo, M. Linschulte, A. da Silva Simao, and S. do Rocio Senger de Souza. Event- and coverage-based testing of web services. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C)*, pages 62–69. IEEE Computer Society, 2010.

[41] A. T. Endo and A. S. Simao. A systematic review on formal testing approaches for web services. In *Proceedings of the 4th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, pages 89–98, 2010.

[42] Enterprise Service Bus (ESB). `http://en.wikipedia.org/wiki/Enterprise_service_bus`.

[43] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591 –603, Jun. 1991.

[44] M.-C. Gaudel. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT)*, pages 82–96. Springer-Verlag, 1995.

[45] GNU Linear Programming Kit (GLPK). `http://www.gnu.org/s/glpk/`.

[46] A. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, SE-11(12):1411–1423, Dec. 1985.

[47] A. L. Goel and K. Okumoto. Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE Transactions on Reliability*, R-28(3):206–211, Aug. 1979.

[48] S. Gokhale and K. Trivedi. Analytical models for architecture-based software reliability prediction: A unification framework. *IEEE Transactions on Reliability*, 55(4):578–590, Dec. 2006.

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

[50] P. Guerreiro. Simple support for design by contract in c++. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 24–34. IEEE Computer Society, 2001.

[51] J. H. Hayes and A. J. Offutt. Increased software reliability through input validation analysis and testing. In *Proceedings of the 10th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 199–209. IEEE Computer Society, 1999.

[52] J. H. Hayes and J. Offutt. Input validation analysis and testing. *Empirical Software Engineering*, 11(4):493–522, 2006.

[53] R. Heckel and M. Lohmann. Towards contract-based testing of web services. *Electronic Notes in Theoretical Computer Science*, 116:145–156, 2005.

[54] C. A. R. Hoare. *Communicating Sequential Processes*. 2004. `http://www.usingcsp.com/cspbook.pdf`.

[55] G. J. Holzmann. Uno: Static source code checking for userdefined properties. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT)*, 2002.

[56] S.-S. Hou, L. Zhang, Q. Lan, H. Mei, and J.-S. Sun. Generating effective test sequences for BPEL testing. In *Proceedings of the 9th International Conference on Quality Software (QSIC)*, pages 331–340. IEEE Computer Society, 2009.

[57] IANA. Service name and transport protocol port number registry. `http://www.iana.org/assignments/port-numbers`.

[58] IEEE Recommended Practice on Software Reliability. *IEEE Std 1633-2008*, 2008.

[59] S. Ilieva, D. Manova, I. Manova, C. Bartolini, A. Bertolino, and F. Lonetti. An automated approach to robustness testing of BPEL orchestrations. In *Proceedings of the 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, pages 193–203. IEEE Computer Society, 2011.

[60] JGraph. `http://www.jgraph.com`.

[61] N. Josuttis. *SOA in practice: the art of distributed system design*. O'Reilly Media, Inc., 2007.

[62] L. Kolmonen. Securing network software using static analysis. *Seminar on Network Security*, 2007. `http://www.tml.tkk.fi/Publications/C/25/papers/Kolmonen_final.pdf`.

[63] I. Koufareva, A. Petrenko, and N. Yevtushenko. Test generation driven by user-defined fault models. In *Proceedings of the 12th International Workshop on Testing Communicating Systems: Method and Applications (IWTCS)*, pages 215–236. Kluwer, B.V., 1999.

[64] S. Krishnamurthy and A. P. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *Proceedings of the 8th*

IEEE International Symposium on Software Reliability Engineering (ISSRE), pages 146–155. IEEE Computer Society, 1997.

[65] B. Krüger and M. Linschulte. Cost reduction through combining test sequences with input data. In *Proceedings of the 6th International Conference on Software Security and Reliability (SERE)*, pages 207–216. IEEE Computer Society, 2012.

[66] D. Le Berre. CSP2SAT4J: A simple CSP to SAT translator. In *Proceedings of the 2nd International Workshop on Constraint Propagation and Implementation*, pages 59–66, 2005.

[67] Y. Lei and R. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, Jun. 2006.

[68] H. Liu and H. B. Kuan Tan. Covering code behavior on input validation in functional testing. *Information and Software Technology*, 51(2):546–553, 2009.

[69] lp_solve - a Mixed Integer Linear Programming (MILP) solver. `http://lpsolve.sourceforge.net/5.5/`.

[70] M. R. Lyu, editor. *Handbook of software reliability engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[71] A. P. Mathur. *Foundations of software testing*. Addison-Wesley Longman, Amsterdam, 2008.

[72] P. McMinn. Search-based software testing: Past, present and future. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 153–163. IEEE Computer Society, Mar. 2011.

[73] L. Mei, W. K. Chan, and T. H. Tse. Data flow testing of service choreography. In *Proceedings of the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 151–160. ACM, 2009.

[74] P. Mell and M. C. Tracy. Procedures for handling security patches. Technical Report NIST Special Publication 800-40, National Institute of Standards and Technology, 2002. `http://www.iwar.org.uk/comsec/resources/patches/sp800-40.pdf`.

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

[76] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 256–267. ACM, 2001.

[77] A. M. Memon and Q. Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for GUI-based software. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, pages 8–17. IEEE Computer Society, 2004.

[78] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[79] Minitab. `http://www.minitab.com`.

[80] MSDN. Design guidelines for secure web application. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod77.asp`.

[81] MuleSoft. Mule ESB: Open source ESB and integration platform. http://www.mulesoft.org/.

[82] Multiscan port scanner (version 0.8.5). `http://www.sourceforge.net/projects/multiscan`.

[83] J. D. Musa and K. Okumoto. A logarithmic poisson execution time model for software reliability measurement. In *Proceedings of the 7th International*

Conference on Software Engineering (ICSE), pages 230–238. IEEE Computer Society, 1984.

[84] G. J. Myers. *Software Reliability: Principles and Practices*. John Wiley & Sons, Inc., New York, NY, USA, 1976.

[85] G. J. Myers. *The art of software testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[86] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The art of software testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, USA, 2nd edition, 2004.

[87] National Institute of Standards and Technology (NIST). Software errors cost u.s. economy $59.5 billion annually. `http://www.abeacha.com/NIST_press_release_bugs_cost.htm`.

[88] NetBeans SOA Project Home. `http://soa.netbeans.org/`.

[89] Netdefender firewall (version 1.5). `http://www.codeplex.com/netdefender`.

[90] Y. Ni, S. Hou, L. Zhang, J. Zhu, Z. Li, Q. Lan, H. Mei, and J. Sun. Effective message-sequence generation for testing BPEL programs. *IEEE Transactions on Services Computing*, PrePrints(99), 2011.

[91] OASIS. *Universal Description, Discovery and Integration (UDDI)*. Organization for the Advancement of Structured Information Standards, 2004. `http://uddi.org/pubs/uddi_v3.htm`.

[92] OASIS. *Web Services Business Process Execution Language (WS-BPEL) v2.0*. Organization for the Advancement of Structured Information Standards, 2007. `http://docs.oasis-open.org/wsbpel/2.0/`.

[93] OMG. *Unified Modeling Language (UML)*. Object Management Group, 2012. `http://www.uml.org/`.

[94] A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal. Modeling and testing hierarchical GUIs. In *Proceedings of the 12th International Workshop on Abstract State Machines (ASM)*, pages 329–344, 2005.

[95] M. P. Papazoglou and W.-J. Heuvel. Service oriented architectures: approaches, technologies and research issues. *The International Journal on Very Large Databases*, 16(3):389–415, Jul. 2007.

[96] 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 National Conference*, pages 379–385. ACM, 1969.

[97] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[98] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 67–78. ACM, 1995.

[99] H. Pham. *Software Reliability*. Springer-Verlag, Singapore, 2000.

[100] R. Plosch. Design by contract for python. In *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC)*, pages 213–219. IEEE Computer Society, Dec. 1997.

[101] Pscan port scanner. `http://www.codeproject.com/KB/cpp/pscan.aspx`.

[102] H. Reza, S. Endapally, and E. Grant. A model-based approach for testing GUI using hierarchical predicate transition nets. In *Proceedings of the 4th International Conference on Information Technology: New Generations (ITNG)*, pages 366–370. IEEE Computer Society, 2007.

[103] H. M. Rusli, M. Puteh, S. Ibrahim, and S. G. H. Tabatabaei. A comparative evaluation of state-of-the-art web service composition testing approaches. In

Proceedings of the 6th International Workshop on Automation of Software Test (AST), pages 29–35. ACM, 2011.

[104] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

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

[106] M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797, Oct. 2005.

[107] SeleniumHQ. `http://seleniumhq.org/`.

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

[109] K. Shibata, K. Rinsaka, and T. Dohi. Metrics-based software reliability models using non-homogeneous poisson processes. In *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 52–61. IEEE Computer Society, 2006.

[110] A. I. Sotirov. Automatic vulnerability detection using static source code analysis. Master's thesis, The University of Alabama, 2005. `http://gcc.vulncheck.org/sotirov05automatic.pdf`.

[111] A. Stevenson. Aspect-oriented smart proxies in java rmi. Master's thesis, University of Waterloo, Ontario, Canada, 2008. `http://plg.uwaterloo.ca/~stevem/students/AndrewStevenson08/AndrewStevenson_MMath08.pdf`.

[112] L. Suhl and T. Mellouli. *Optimierungssysteme: Modelle, Verfahren, Software, Anwendungen*. Springer-Verlag, 2005.

[113] H. W. Thimbleby. The directed chinese postman problem. *Software: Practice and Experience*, 33(11):1081–1096, 2003.

[114] T. Tuglular. Test case generation for firewall implementation testing using software testing techniques. In *Proceedings of the 1st International Conference on Security of Information and Networks (SIN)*, pages 196–203. Trafford Publishing, 2007.

[115] T. Tuglular, C. Muftuoglu, F. Belli, and M. Linschulte. Event-based input validation using design-by-contract patterns. In *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 195–204. IEEE Computer Society, 2009.

[116] T. Tuglular, C. A. Muftuoglu, O. Kaya, F. Belli, and M. Linschulte. GUI-based testing of boundary overflow vulnerability. In *Proceedings of the 33rd IEEE International Conference on Computer Software and Applications (COMPSAC)*, pages 539–544. IEEE Computer Society, 2009.

[117] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[118] W3C. *Web Service Choreography Interface (WSCI)*. The World Wide Web Consortium, 2002. `http://www.w3.org/TR/wsci/`.

[119] W3C. *Web Services Choreography Description Language (WS-CDL)*. The World Wide Web Consortium, 2005. `http://www.w3.org/TR/ws-cdl-10/`.

[120] W3C. *SOAP Version 1.2*. The World Wide Web Consortium, 2007. `http://www.w3.org/TR/soap12-part1/`.

[121] W3C. *Web Services Description Language (WSDL)*. The World Wide Web Consortium, 2007. `http://www.w3.org/TR/wsdl20/`.

[122] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2000.

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

[124] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Proceedings of the 21st International Conference on Testing of Software and Communication Systems (TESTCOM)*, pages 179–194. Springer-Verlag, 2009.

[125] S. Wieczorek, A. Stefanescu, and A. Roth. Model-driven service integration testing - a case study. In *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 292–297. IEEE Computer Society, 2010.

[126] WS-CDL Eclipse (with examples). `http://sourceforge.net/projects/wscdl-eclipse/`.

[127] S. Yacoub, B. Cukic, and H. Ammar. A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability*, 53(4):465–480, Dec. 2004.

[128] S. Yamada, M. Ohba, and S. Osaki. S-shaped reliability growth modeling for software error detection. *IEEE Transactions on Reliability*, R-32(5):475–484, Dec. 1983.

[129] H. Zeng, H. Miao, and J. Liu. Specification-based test generation and optimization using model checking. In *Proceedings of the 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 349 –355. IEEE Computer Society, Jun. 2007.

[130] M. Zhao and M. Xie. On the log-power NHPP software reliability model. In *Proceedings of the 3rd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 14–22. IEEE Computer Society, 1992.

[131] W. Zheng and G. Bundell. Test by contract for UML-based software component testing. In *Proceedings of the International Symposium on Computer Science and its Applications (CSA)*, pages 377–382. IEEE Computer Society, 2008.

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

List of Figures

List of Tables

List of Algorithms

Part V

Appendix

Appendix A

Algorithms

A.1 Layer-centric Testing

Algorithm A.1: Determination of CESs for an hierarchical ESG covering EPs

function : generateCES_LC(ESG)
input : an $ESG = (V, E, \Xi, \Gamma)$
output : a set of CESs covering event pairs

1 sets $CES[], weight[], curCESs, resCES := \emptyset$;

`// generate CESs for the compound vertices first`

2 **foreach** $v \in \Psi(ESG)$ **do**
3 $CES[v]$:= generateCES_LC(v); `// recursive call`
4 $weight[v]$:= length(ShortestPath(v));

5 ESG := balanceESG_LC_opt($ESG, CES, weight$) ; `// Algorithm A.2`
6 $curCESs$:= determineCESs(ESG) ; `// Algorithm A.3`

`// replace compound vertices`

7 **foreach** *ces*$\in$*curCESs* **do**
8 **foreach** $v \in \Psi$*(ces)* **do**
9 **if** $|CES[v]| > 0$ **then**
10 replace v by $x \in CES[v]$;
11 $CES[v] := CES[v] \setminus \{x\}$;
12 **else** replace v by ShortestPath(v);
13 $resCES := resCES \cup \{ces\}$;
14 **return** *resCES*;

Table A.1: Legend

$\Psi(s)$, $\Psi(g)$	a function determining the set of compound vertices for a given sequence s or ESG g
$\Phi(v, ESG)$	a function determining the set of all occurrences of a given vertex v in an ESG
ShortestPath(ESG)	determines the shortest path from $\varepsilon = [$ to $\gamma =]$ for a given ESG (note that compound vertices are also ESGs)
computeEulerTour(ESG)	determines the Eulerian cycle of the given ESG
computeShortestPaths(v,B,D,weight)	determines all shortest paths from vertex v to all $b \in B$ with Breadth-First-Search algorithm and stores these shortest distances in the distance matrix D taking weights into account
computeShortestPath(v, w)	determines the shortest path from vertex v to vertex w
length()	determines the length of a path or list
solveAssignmentProblem(D)	solves the assignment problem according to the Hungarian algorithm
solveAssignmentProblemLC_opt(D)	solves the assignment problem described in Section 4.2 according to equations 4.1 to 4.7
getElement(list,i)	returns the i-th element of the given list or sequence
getPartList(list,start,i)	returns the sublist from start to i of the given list or sequence

Algorithm A.2: Balancing an ESG according to LC_{opt}

function : balanceESG_LC_opt(ESG, CES[], weight[])
input : an $ESG = (V, E, \Xi, \Gamma)$ with $\varepsilon = [, \gamma =]$
input : an array $CES[]$ containing the CESs for each compound vertex of ESG
input : an array $weight[]$ containing the weights for each compound vertex of ESG
output : a balanced ESG

```
1  E := E ∪ {(γ, ε)} ;                                   // insert arc from ] to [
2  sets A, B, M := ∅ ;                                   // empty sets
   // determine set A and B
3  foreach v ∈ V do
4      if δ(v) > 0 then
5          for s := 1 to δ(v) do
6              A := A ∪ {v};
7      if δ(v) < 0 then
8          for s := 1 to −δ(v) do
9              B := B ∪ {v};
10 foreach v ∈ Ψ(ESG) do
11     n := |CES[v]| − Max(δ⁻(v), δ⁺(v));
12     for s := 1 to n do
13         A := A ∪ {v};
14         B := B ∪ {v};
15 m := |A| := |B| ;                                     // cardinality
16 D[1..m][1..m] ;                                       // distance matrix D
   // compute all shortest paths from v to all b ∈ B
17 foreach v ∈ A do
18     computeShortestPaths(v, B, D, weight) ;           // shortest distances are
       saved in D
19 M := solveAssignmentProblem(D) ;                      // Hungarian Algorithm
   // M = {(i, j)| one-to-one mapping: i ∈ {1, .., |A|} → j ∈ {1, .., |B|}}
20 foreach (i, j) ∈ M do
21     Path := computeShortestPath(i, j);
22     foreach e ∈ Path do                               // e = (i, j) with i, j ∈ V
23         E := E ∪ {e};
24 return ESG;
```

Algorithm A.3: Algorithm to determine the CES set for a balanced ESG

function : determineCESs(ESG)
input : a balanced $ESG = (V, E, \Xi, \Gamma)$ with $\varepsilon = [, \gamma =]$
output : a set of CESs

1 $EulerTourList$:= computeEulerTour(ESG); ; // tour starts in ε
// EulerTourList=$(\varepsilon, \ldots, \gamma, \varepsilon, \ldots, \gamma, \varepsilon, \ldots, \gamma, \varepsilon)$
2 $CES := \emptyset$;
3 $start := 1$;
4 **for** $i := 2$ **to** (length($EulerTourList$)-1) **do**
5 **if** getElement($EulerTourList$,i) $= \gamma$ **then**
6 $CES := CES \cup$ getPartList($EulerTourList, start, i$);
7 $start := i + 1$; // CES={$(\varepsilon, \ldots, \gamma)$, $(\varepsilon, \ldots, \gamma)$, $(\varepsilon, \ldots, \gamma)$, ...}
8 **return** *CES*;

Algorithm A.4: Generate FCES for an ESG and its compound vertices

function : generateFCES_LC(ESG, length)
input : an $ESG = (V, E, \Xi, \Gamma)$ with $\varepsilon = [, \gamma =]$
input : the desired event sequence *length* to be covered
output : a set of FCESs covering the given *length* of FESs

1 sets FES, resFCES := $\emptyset$; // empty sets
// generate FCESs for the compound vertices first
2 **foreach** $v \in \Psi(ESG)$ **do**
3 $FES :=$ generateFCES_LC(v, $length$);
4 **foreach** $fes \in FES$ **do**
5 $fes :=$ computeShortestPath(ε,v)$\oplus fes$; // set start sequence
6 $resFCES := resFCES \cup \{fes\}$;
7 $resFCES := resFCES \cup$ determineFCES($ESG, length$) ; // Algorithm A.5
// replace compound vertices in resFCES
8 **foreach** $fes \in resFCES$ **do**
9 **foreach** $v \in \Psi(fes)$ **do**
10 **if** $\omega(fes) = v$ **then**
11 replace v by $v' \in \Xi(v)$;
12 **else**
13 replace v by ShortestPath(v);
14 $resFCES := resFCES \cup \{fes\}$;
15 **return** *resFCES*;

Algorithm A.5: Generate FCES for a single ESG

function : determineFCES(ESG, length)
input : an $ESG = (V, E, \Xi, \Gamma)$ with $\varepsilon = [, \gamma =]$
input : the desired event sequence *length* to be covered
output : a set of FCESs covering the given *length* of FESs

```
1  FCES := V;
   // set up event sequences of (length-1)
2  while length > 2 do
3      FCES' := ∅;
4      foreach fes ∈ FCES do
5          foreach (ω(fes), v) ∈ E with v ∈ V do
6              FCES' := FCES' ∪ {fes ⊕ v};
7      FCES := FCES';
8      length := length − 1;
   // add invalid transitions
9  FCES' := ∅;
10 foreach fes ∈ FCES do
11     foreach (ω(fes), v) ∉ E with v ∈ V do
12         FCES' := FCES' ∪ {fes ⊕ v};
13 FCES := FCES';
   // set start sequence
14 FCES' := ∅;
15 foreach fes ∈ FCES do
16     fes:=computeShortestPath(ε,α(fes))⊕fes;
17     FCES' := FCES' ∪ {fes};
18 FCES := FCES';
19 return FCES;
```

Algorithm A.6: Improved version of Algorithm A.1 for determining the set of CESs for an hierarchical ESG covering event sequences of any length

function : generateCES_LC(ESG, length)
input : an $ESG = (V, E, \Xi, \Gamma)$
input : the desired event sequence *length* to be covered
output : a set of CES covering event sequences of *length*

```
1  sets CES[], curCESs, resCES := ∅;
   // generate CESs for the compound vertices first
2  foreach v ∈ Ψ(ESG) do
3      CES[v] := generateCES_LC(v, length);                    // recursive call
4      weight[v] := length(ShortestPath(v));
   // generate CESs for the given ESG
5  ESG := transformESG(ESG, length) ;                    // see Algorithm A.10
6  ESG := balanceESG_LC_opt(ESG, CES, weight) ;            // Algorithm A.7
7  curCES := determineCESs(ESG) ;                          // Algorithm A.3
   // replace compound vertices
8  foreach ces∈curCES do
9      foreach v ∈ Ψ(ces) do
10         if |CES[v]| > 0 then
11             replace v by x ∈ CES[v];
12             CES[v] := CES[v] \ {x};
13         else replace v by ShortestPath(v);
14     resCES := resCES ∪ {ces};
15 return resCES;
```

Algorithm A.7: Improved version of Algorithm A.2 for balancing an ESG according to LC_{opt} and length>2

function : balanceESG_LC_opt(ESG, CES[], weight[])
input : an $ESG = (V, E, \Xi, \Gamma)$ with $\varepsilon = [, \gamma =]$
input : an array $CES[]$ containing the CESs for each compound vertex of ESG
input : an array $weight[]$ containing the weights for each compound vertex of ESG
output : a balanced ESG

```
1  E := E ∪ {(γ, ε)};
2  sets A, B, M := ∅;
3  hasArea := determineSetAandB_LC_opt(ESG, CES, A, B) ;    // Algorithm A.8
4  m := |A| := |B| ;                                          // cardinality
5  D[1..m][1..m] ;                                          // distance matrix
   // compute all shortest paths from v to all b ∈ B
6  foreach v ∈ A do
7      computeShortestPaths(v,B,D,weight) ; // shortest distances are saved
       in D
8  if !hasArea then
9      M := solveAssignmentProblem(D) ;                  // Hungarian Alhgorithm
10 else M := solveAssignmentProblemLC_opt(D) ;          // Equations 4.1 to 4.7
   //  M = {(i, j)| one-to-one mapping:  i ∈ {1, .., |A|} → j ∈ {1, .., |B|}}
11 foreach (i, j) ∈ M do
12     Path := getShortestPath(i,j);
13     foreach e ∈ Path do                                  // e = (i, j) with i, j ∈ V
14         E := E ∪ {e};
15 return ESG;
```

Algorithm A.8: Determine set A and B for LC_{opt}

function : determineSetAandB_LC_opt(ESG, CES[], A, B)
input : an $ESG = (V, E, \Xi, \Gamma)$
input : CES[] := an array containing the CES set for each compound vertex
input : empty sets A and B to be populated
output : populates set A and B and returns true if problem has an area, otherwise false

```
1  foreach v ∈ V do
2      if δ(v) > 0 then
3          for s := 1 to δ(v) do
4              A := A ∪ {v};
5      if δ(v) < 0 then
6          for s := 1 to −δ(v) do
7              B := B ∪ {v};
8  hasArea := false;
9  foreach v ∈ Ψ(ESG) do
10     n := 0;
11     if |Φ(v, ESG)| = 1 then
12         n := |CES[v]| − Max(δ⁻(v), δ⁺(v));
13     else
14         foreach v ∈ Φ(v, ESG) do
15             n := n + Max(δ⁻(v), δ⁺(v));
16         n := |CES[v]| − n;
17         if n > 0 then
18             hasArea := true;
19     for s := 1 to n do
20         A := A ∪ Φ(v, ESG);
21         B := B ∪ Φ(v, ESG);
22 return hasArea;
```

Algorithm A.9: Balancing an ESG according to LC_{Simple}

function : balanceESG_LC_simple(ESG, CES[], weight[])
input : an $ESG = (V, E, \Xi, \Gamma)$
input : an array $CES[]$ containing the CESs for each compound vertex of ESG
input : an array $weight[]$ containing the weights for each compound vertex of ESG
output : a balanced ESG

```
1  V' := ∅;
2  foreach v ∈ Ψ(ESG) do
3      if |Φ(v, ESG)| > 1 then
4          CES2[v] := CES[v];
5          CES[v] := ∅;
6          V' := V' ∪ {v};
7  ESG := balanceESG_LC_opt(ESG, CES, weight) ;          // Algorithm A.2
8  foreach v' ∈ V' do
9      n := 0;
10     foreach v ∈ Φ(v', ESG) do
11         n := n + δ⁻(v);
12     n := |CES2[v']| − n;
13     if n > 0 then
14         l := ∞;
15         foreach v ∈ Φ(v', ESG) do
16             Path_tmp := computeShortestPath(v,v);
17             if length(Path) < l then
18                 l := length(Path);
19                 Path := Path_tmp;
20     foreach e ∈ Path do                              // e = (i, j) with i, j ∈ V
21         E := E ∪ {e};
22 return ESG;
```

A.2 Model Transformation

Algorithm A.10: Algorithm to transform an ESG for higher length coverage (improved version in one step)

function : transformESG(ESG, length)
input : an ESG=(V,E,Ξ,Γ)
input : the desired event sequence *length* to be covered
output : a transformed ESG according to the *length* to be covered

```
1  if length>2 then
2      ESG' := (V', E', Ξ', Γ') with V' = ∅, E' = ∅, Ξ' = ∅, Γ' = ∅;
       // add vertices
3      foreach es ∈ getSequencesOfLength(ESG, length-1) do     // Algorithm A.11
4          V' := V' ∪ {ω(es)};
       // add edges
5      foreach es_1 ∈ getSequencesOfLength(ESG, length-1) do
6          foreach es_2 ∈ getSequencesOfLength(ESG, length-1) do
7              if es_1 ⊕ ω(es_2) = α(es_1) ⊕ es_2 then
8                  E' := E' ∪ {(ω(es_1), ω(es_2))};
       // set Ξ' and Γ'
9      foreach es ∈ getSequencesOfLength(ESG, length-1) do
10         if α(es) ∈ Ξ then
               // add all vertices of es to new ESG
11             V' := V' ∪ {α(es)};
12             last := α(es);
13             foreach event e ∈ es do
14                 if e ≠ α(es) AND e ≠ ω(es) then
15                     V' := V' ∪ {e};
16                 if e ≠ α(es) then
17                     E' := E' ∪ {(last, e)};
18                     last := e;
19             Ξ' := Ξ' ∪ {α(es)};
20         if ω(es) ∈ Γ then Γ' := Γ' ∪ {ω(es)};
21     return ESG';
22 return ESG;
```

Algorithm A.11: Algorithm to retrieve all ESs of a given length

function : getSequencesOfLength(ESG, length)
input : an $ESG = (V, E, \Xi, \Gamma)$
input : the desired *length* of event sequences
output : a set of all ESs of the given *length*

```
1 ES := V;
2 while length > 1 do
3     ES' := ∅;
4     foreach es ∈ ES do
5         foreach (ω(es), v) ∈ E with v ∈ V do
6             ES' := ES' ∪ {es ⊕ v};
7     ES := ES';
8     length := length − 1;
9 return ES;
```

A.3 WSC Testing

Table A.2: Legend

$\Psi(s)$	a function determining the set of compound vertices for a given sequence s
$s[i]$	the i-th event of sequence s
$s[i..j]$	the sequence from i to j
ShortestPath(ESG4WSC)	determines the shortest path from $\varepsilon = [$ to $\gamma =]$ for a given ESG4WSC
computeEulerTour(ESG)	determines the Eulerian cycle of the given ESG
computeShortestPath(v, w)	determines the shortest path from vertex v to vertex w
length()	determines the length of a path or list
getAllowedSuccessor(ces, DT_{seq})	determines the allowed successor event for a given sequence ces according to DT_{seq}
solveCPP(ESG4WSC)	solves the Chinese postman problem for the given ESG4WSC

Algorithm A.12: Algorithm to generate CESs for a given ESG4WSC

function : generateCESs(ESG4WSC)
input : an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma,)$
output : a set of CESs covering event pairs

1 $resCES := \emptyset$; // Step 1
2 **foreach** $re \in V_{refined}$ **do**
3 **foreach** $esg \in re$ **do**
4 $CES :=$ generateCESs(esg); // recursive call
5 **if** $resCES = \emptyset$ **then** $resCES := resCES \cup \{(re \times CES)\}$;
6 **else**
7 **foreach** $ces_1 \in \{ces|(re, ces) \in resCES\}$ **do**
8 **foreach** $ces_2 \in CES$ **do** $resCES := resCES \cup \{(re, (ces_1||ces_2))\}$;
9 $resCES := resCES \setminus \{(re, ces_1)\}$;
10 **foreach** $re \in V_{refined}$ AND $f(re) \neq \epsilon$ **do** // Step 2
11 $\text{DT}_{seq} := f(re)$;
12 **foreach** $ces \in \{ces|(re, ces) \in resCES\}$ **do**
13 $v :=$ getAllowedSuccessor(ces,DT_{seq});
14 $E := E \cup \{(re, v)\}$;
 // store a Mapping, i.e., $Map \subseteq E \times ces$
15 $Map := Map \cup \{((re, v), ces)\}$;
16 $resCES := resCES \setminus \{(re, ces)\}$;
 // add multiple edges for each dataset to be tested
17 **foreach** $DT_{input,public} \in V$ **do**
18 **foreach** $e \in E_{DT}$ **do** $E := E \setminus \{(DT_{input}, e)\}$;
19 **foreach** $(C_{true}, C_{false}, E_x) \in R$ **do** $E := E \cup \{(DT_{input}, E_x)\}$;
20 $CES :=$ solveCPP($ESG4WSC$); // Step 3
21 **foreach** $ces \in CES$ **do** // Step 4
22 **for** $i = 1$ **to** length*(ces)* **do**
23 **if** $ces[i] \in V_{refined}$ **then**
24 **if** $|\{ces|((ces[i], ces[i+1]), ces) \in Map]\}| > 0$ **then**
25 $new :=$ es with $es \in \{ces|((ces[i], ces[i+1]), ces) \in Map]\}$;
26 $Map := Map \setminus \{((ces[i], ces[i+1]), ces)\}$;
27 $ces := ces[1..(i-1)] \oplus new \oplus ces[(i+1)..\text{length}(ces)]$;
28 **else if** $|\{ces|(ces[i], ces) \in resCES\}| > 0$ **then**
29 $new :=$ es with $es \in \{ces|(ces[i], ces) \in resCES]\}$;
30 $resCES := resCES \setminus \{(ces[i], ces)\}$;
31 $ces := ces[1..(i-1)] \oplus new \oplus ces[(i+1)..\text{length}(ces)]$;
32 **return** *CES*;

Algorithm A.13: Algorithm to generate PubFESs

function : generatePubFESs(ESG4WSC)
input : an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$
output : the test suite PubFES

1 $PubFES := \emptyset$;
2 $ESG4WS :=$ transformToESG4WS($ESG4WSC$); `// Algorithm A.15`
3 **foreach** $(e_i, e_j) \in$ obtainFEP($ESG4WS$) **do** `// Algorithm A.16`
4 $pes_i :=$ generatePES($ESG4WSC, e_i$) ; `// Algorithm A.17`
5 $pes_i := pes_i \oplus e_j$;
6 $PubFES := PubFES \cup \{(pes_i; F)\}$;
7 **return** *PubFES*;

Algorithm A.14: Algorithm to generate PriFESs

function : generatePriFESs()
input : an ESG4WSC, set of sensitive events s
output : the test suite PriFES

1 $PriFES := \emptyset$;
2 Let V_{REQ} be the union of sets V_{req} for the ESG4WSC and their refining ESG4WSCs;
3 Let V_{RESP} be the union of sets V_{resp} for the ESG4WSC and their refining ESG4WSCs;
4 **foreach** $e_i \in V_{REQ}$ **do**
5 $pes_i :=$ generatePES($ESG4WSC, e_i$) ; `// Algorithm A.17`
6 $fes_1 := (pes_i; F_{\text{NR}}; s)$;
7 $fes_2 := (pes_i; F_{\text{MS}}; s)$;
8 $fes_3 := (pes_i; F_{\text{UF}}; s)$;
9 $PriFES := PriFES \cup \{fes_1, fes_2, fes_3\}$;
10 **foreach** $e_j \in V_{RESP}$ **do**
11 $pes_j :=$ generatePES($ESG4WSC, e_j$) ; `// Algorithm A.17`
12 $fes_1 := (pes_j; F_{\text{LR}}; s)$;
13 $fes_2 := (pes_j; F_{\text{WSc}}; s)$;
14 $fes_3 := (pes_j; F_{\text{WSy}}; s)$;
15 $fes_4 := (pes_j; F_{\text{WD}}; s)$;
16 $PriFES := PriFES \cup \{fes_1, fes_2, fes_3, fes_4\}$;
17 **return** *PriFES*;

Algorithm A.15: Algorithm to transform an ESG4WSC to an ESG4WS

function : transformToESG4WS(ESG4WSC)
input : an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$
output : an $ESG4WS = (V', E', \Xi', \Gamma')$

```
1  ESG4WS = (V', E', Ξ', Γ');
2  foreach v ∈ V do
3      if (v ∉ V_req AND v ∉ V_resp) OR v is private then
4          foreach pre ∈ N⁻(v) do                    // predecessors of v
5              foreach post ∈ N⁺(v) do                // successors of v
6                  if (pre, post) ∉ E then
7                      E := E ∪ {(pre, post)};
8                  E := E \ {(v, post)};
9              E := E \ {(pre, v)};
10         V := V \ v;
11         if v ∈ Ξ then Ξ := Ξ ∪ N⁺(v) \ {v};
12         if v ∈ Γ then Γ := Γ ∪ N⁻(v) \ {v};
13 V' := V; E' := E; Ξ' := Ξ; Γ' := Γ;
14 return ESG4WS;
```

Algorithm A.16: Algorithm to obtain the faulty event pairs of an ESG4WS

function : obtainFEP(ESG4WS)
input : an $ESG4WS = (V, E, \Xi, \Gamma)$ with $V = V_{req} \cup V_{resp}$
output : a set of FEPs for the given $ESG4WS$ according to Definition 7.14

```
1  FEP := ∅;
2  foreach v_req ∈ V_req do
3      foreach v_resp ∈ V_resp do
4          if (v_resp, v_req) ∉ E then
5              FEP := FEP ∪ {(v_resp, v_req)};
6      foreach v_req2 ∈ V_req do
7          if v_req2 ≠ v_req AND (v_req2, v_req) ∉ E then
8              FEP := FEP ∪ {(v_req2, v_req)};
9      if v_req ∉ Ξ then
10         FEP := FEP ∪ {([, v_req)};
11 return FEP;
```

Algorithm A.17: Algorithm to generate a partial event sequence that covers an event e_i

function : generatePES(ESG4WSC, e_i)
input : an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma,)$ with $\varepsilon = [, \gamma =]$
input : a non-refining event e_i
output : a PES to event e_i

```
1  re := ε;
2  if e_i ∈ V then
3      e_k := e_i;
4  else
5      select re ∈ V_refined such that e_i is within re;
6      e_k := re;
7  pes := computeShortestPath(ε, e_k);
   // replace the refined events in pes
8  foreach v ∈ Ψ(pes) do
9      replace v by ShortestPath(v);
   // replace last event of pes if e_i is in a lower layer
10 if re ≠ ε then
11     par := ε;
12     foreach esg4wsc ∈ re do
13         if e_i ∈ esg4wsc then
14             pes_i := generatePES(esg4wsc,e_i) ;        // recursive call
15         else
16             pes_i := ShortestPath(esg4wsc);
17         par := par||pes_i;
18     replace ω(pes) by par;
19 return pes;
```

Appendix B

Supplementary Material
Case Study I

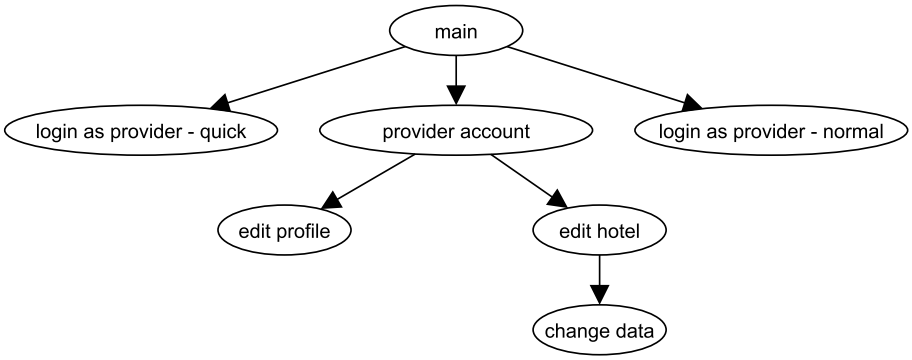


Figure B.1: Hierarchical structure of the ESG used in the case study

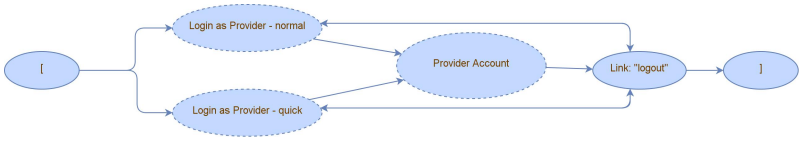


Figure B.2: Main

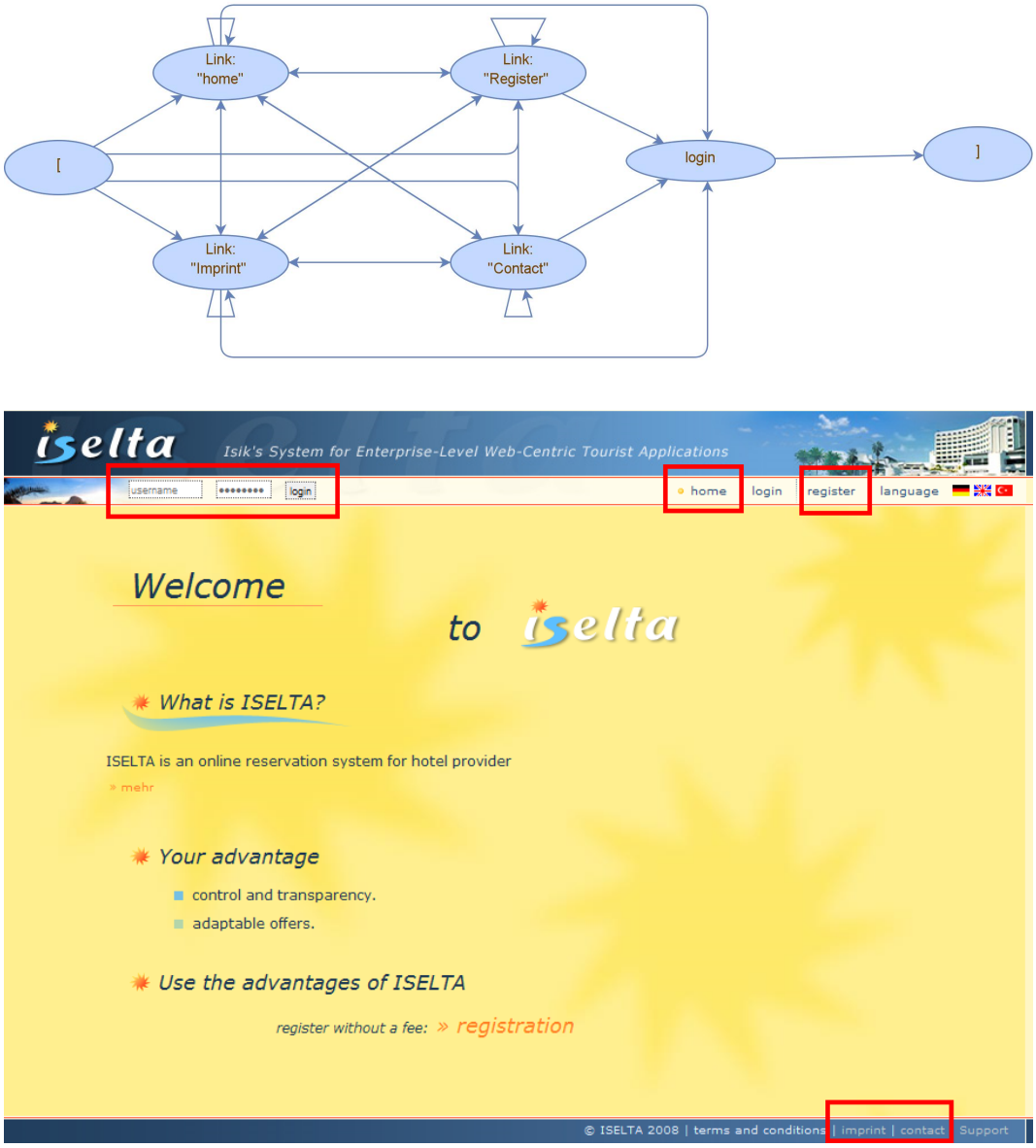


Figure B.3: Refinement of vertex "Login as Provider - quick" and the corresponding screenshot (not expected to show every event)

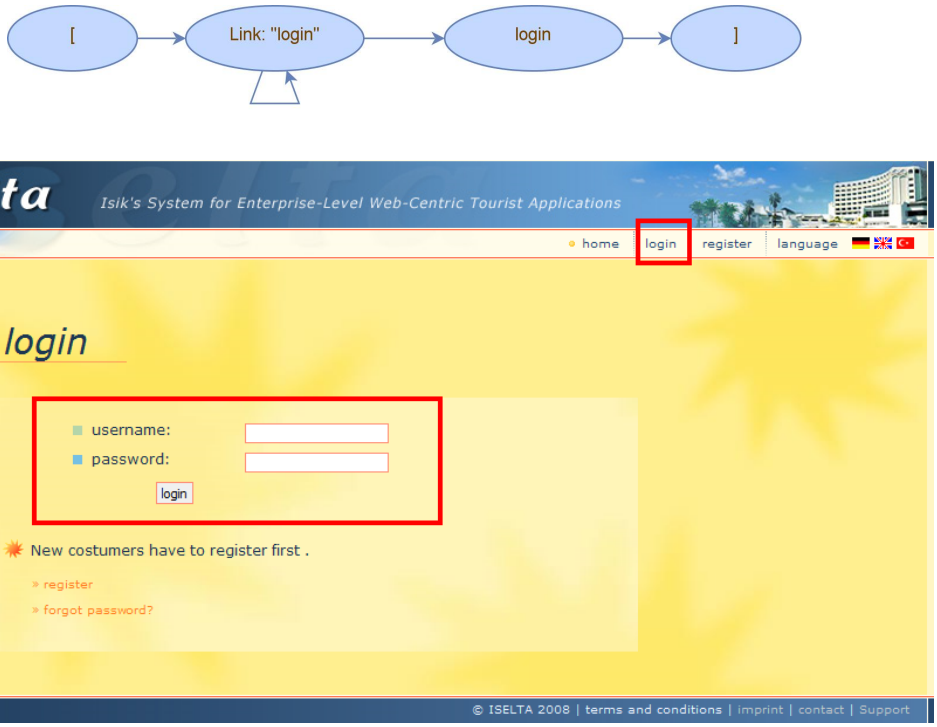


Figure B.4: Refinement of vertex "Login as Provider - normal" and the corresponding screenshot (not expected to show every event)

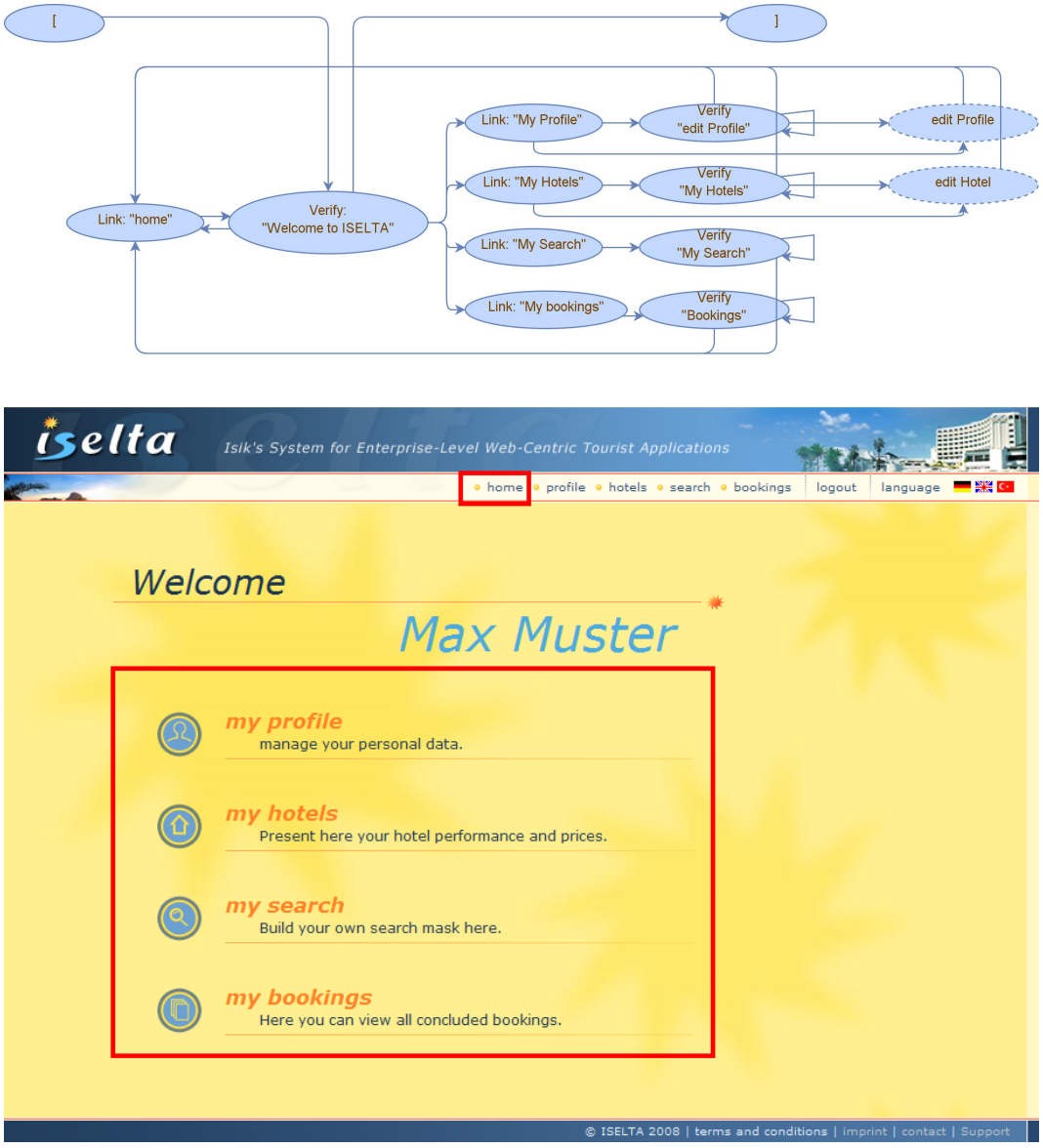


Figure B.5: Refinement of vertex "Provider Account" and the corresponding screenshot (not expected to show every event)

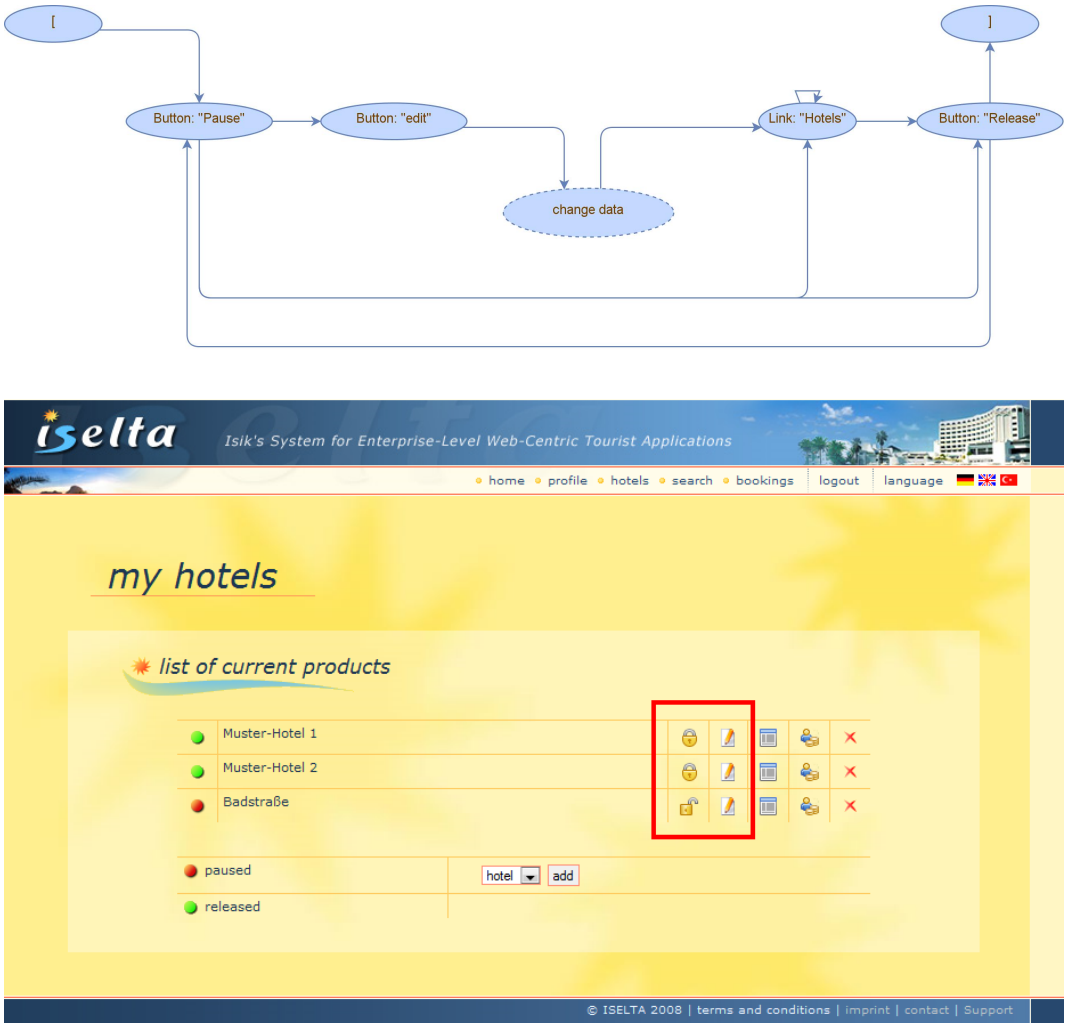


Figure B.6: Refinement of vertex "edit-Hotel" and the corresponding screenshot (not expected to show every event)

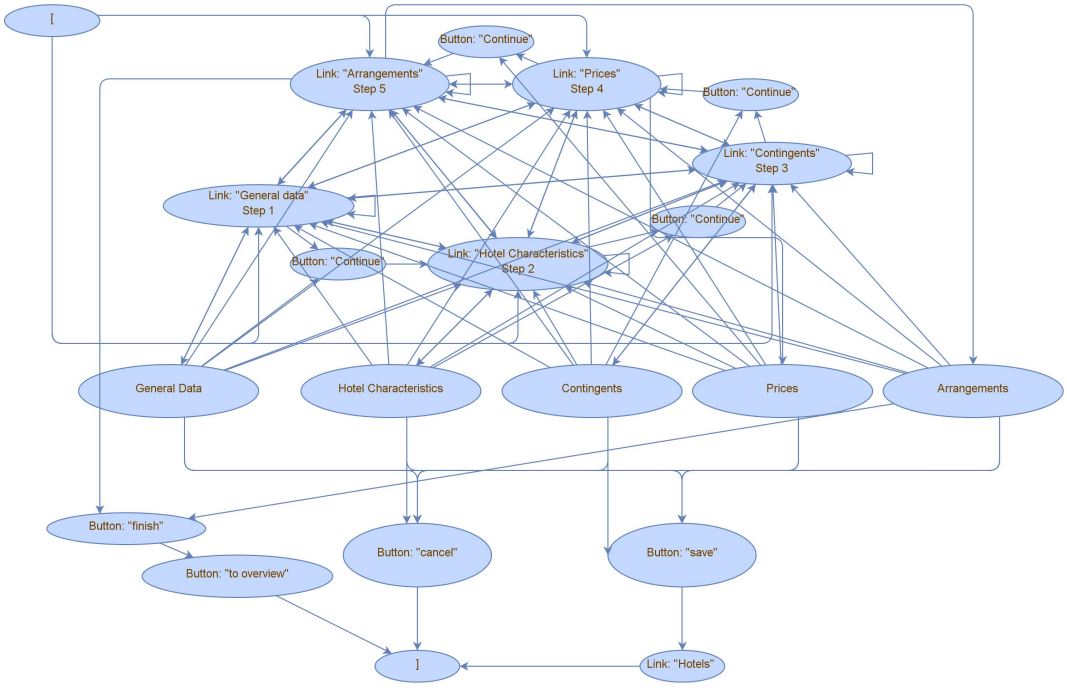


Figure B.7: Refinement of vertex "change data" and the corresponding screenshot (not expected to show every event)

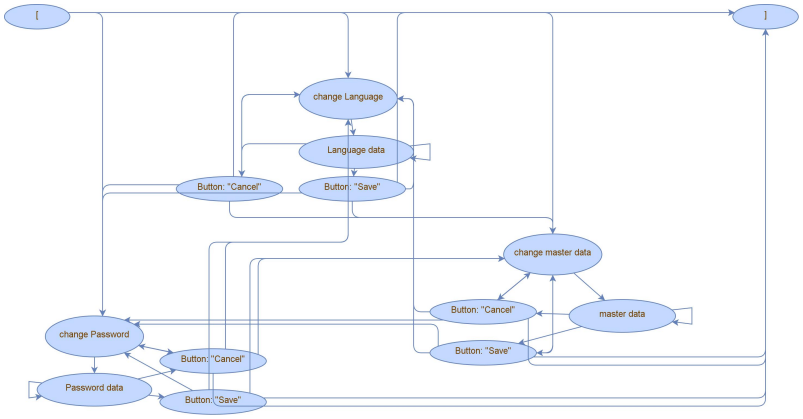


Figure B.8: Refinement of vertex "edit Profile" and the corresponding screenshot (not expected to show every event)

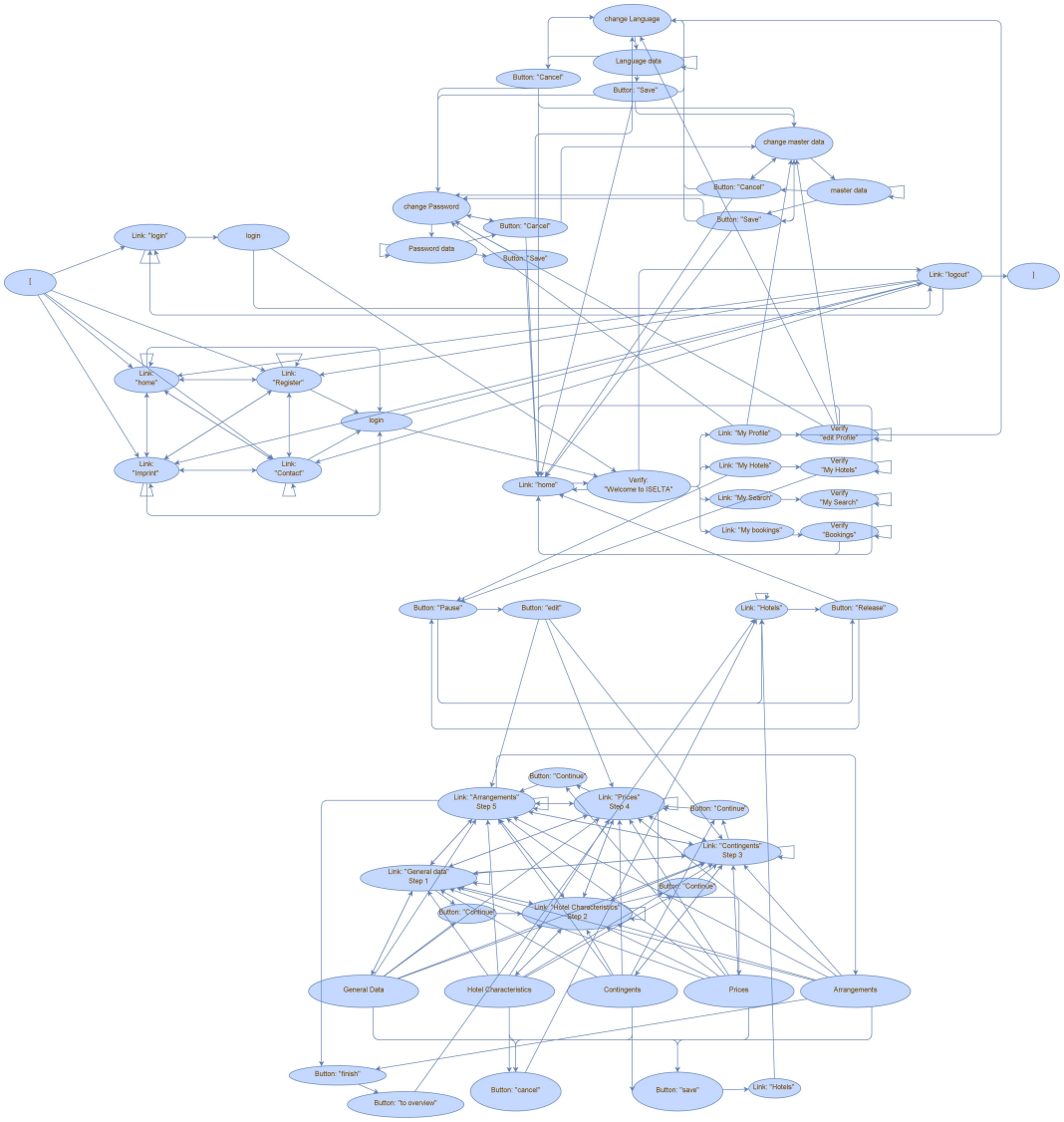


Figure B.9: The full resolution model

Appendix C

Supplementary Material Case Study II

C.1 xTripHandling Description

This appendix describes the application xTripHandling that provides a set of facilities to query and book a trip. xTripHandling is developed using SOA concepts and web services and was based on available examples of service compositions, such as WS-CDL, Netbeans BPEL, and so on. The application is composed of different services. First, the atomic (single) services are presented as follows.

- Airlines service
- ISELTA hotel service
- Car Rental service
- Map service
- Sightseeing service
- Train service

Further information about each service can be found in the interface description (Java code and associated documents). The composite services are described in the following.

C.1.1 Travel Agent Service

It is composed by three operations: queryTrip, getAllOptions, and book.

Workflow description (functional):

1. *queryTrip* is invoked with searchData.
   (a) If there is some invalid data, TripInputException is launched and the process finishes (see 7.).
2. The searchData is mapped to search operations for hotel and flight.
3. Both search operations are called concurrently.
   (a) HotelService requires login before the search.
   (b) If one of the services (hotel, flight) launches an exception, *TripInputException* is launched and the process finishes (see 7.).
4. Check if at least one hotel and flight was returned. Otherwise it launches a *TripBookingException* with a message showing that there is no hotel or flight or both. The process finishes (see 7).
5. Generate a search code and return the operation *queryTrip* with the five cheapest prices for flights and hotels.
6. At this point, the process will wait for 3 events, *getAllOptions* or *book* calls or a timeout (5 min).
   (a) After 5 min without a new request, the process finishes (see 7.).
   (b) *getAllOptions* request will reply all options for hotel and flight. If the searchCode is invalid (process finishes or does not exist), *TripInputException* is launched. Timeout is restarted.
   (c) *book* request.
      i. *TripInputException* if some input data is invalid or the process finished.

ii. All input data is ok, the bookings for hotel and flight are called concurrently and (successfully) a booking message is returned. The process finishes (see 7.).

A. for booking a hotel, a login and a search is required first, since there is a timeout of 30 sec. If the search does not contain the same combination of hotel and price to be booked, a *TripBookingException* is launched.

iii. If some fault happens in hotel and flight, *TripBookingException* is launched with the problem description. The client can try again back to step 6.

A. Successful hotel or flight booking should be canceled. To cancel a hotel, you need to login again first. If the cancellation fails, a *TripBookingException* is launched with a message "Contact administrator ($e-mail), code: $searchCode, $bookingInformation!". The process finishes (see 7.).

7. If the process finishes:

(a) *getAllOptions* leads to a *TripInputException* (always, that is, independent of input values)

(b) *boo*k leads to a *TripInputException* (always, that is, independent of input values)

C.1.1.1 Non-Functional Requirements

This section presents some non-functional requirements for the Travel Agent Service.

1. The timeout for invoking external services is 60 seconds.

C.1.2 Customer Service

This service maps the workflow followed by a traveler. It interacts with airline service, map service, car rental, sightseeing, train, and travel agent services. It is com-

posed by ten operations: *consultTravel*, *moreOptions*, *bookTravel*, *consultTravelData*, *orderMap*, *bookTrain*, *searchSightseeing*, *bookSightseeing*, *searchCars*, and *bookCar*.

Workflow description (functional):

1. The client starts a trip search, invoking the operation *consultTravel*.
   (a) Input data is validated.
   (b) Launch an *CSInvalidInputDataException* if there is some invalid input data. The process finishes.
2. The origin and destiny cities are checked in the airline service.
   (a) The Airline service has the operation *listAvailableCities* that returns one or more items if there is any airport in the city passed by parameter. First, check the origin city. If there is no item returned, call mapservice operation *getNearestCityWith* with the city and item "AIRPORT" as parameter.
      i. If map service cannot find the city, an CSSearchException is launched, the process finishes.
   (b) The same procedure (previous step) is done concurrently for the destiny city. The origin and destiny airport cities are updated if necessary.
3. The operation *queryTrip* of travel agent service is called.
   (a) If some exception is launched, launch an CSSearchException with some message.
4. Return to the client the five cheapest flights and the five cheapest hotels returned by Travel Agent Service and the travelSearchCode.
5. At this point, the process will wait for 3 events, *moreOptions, bookTravel,* or a timeout (3 min).
   (a) After 3 min without a new request, the process finishes.

(b) *moreOptions* request. *GetAllOptions* of travel agent service is called and new results are returned to the client. The timeout is restarted. Back to step 5.

i. If the travelSearchCode is invalid (process finishes or does not exist), *CSInvalidInputDataException* is launched.

(c) The client chooses a hotel and flight and requests a booking using *bookTravel* operation.

i. If some input data is invalid, *CSInvalidInputDataException* is launched. Back to step 5. The *book* operation of Travel Agent service is called.

A. If some exception is launched, *CSBookingException* is launched. Back to step 5.

ii. If the airport and origin city are in different cities, the operation *hasTrains* of Train service is called. If it is true, the operation *searchTrains* is called to list the lines. Then, the same is done for the destiny city. The returned train lines, if they exist, are refered to as originLines (origin city) and destinyLines (destiny city). The originLines and destinyLines are empty if the city and airport city are the same.

A. List just the successfully returned train lines.

6. Booking is successful returning a bookingID + (hotel and flight data). If the destiny or/and origin city are different from the airport, some train lines are returned (if it there exist).

7. Using the bookingID, the client can make the following operations. There are some period restrictions: (a) until the return date and (b,c,d,e) until one day before the departure day.

(a) Query the Trip data using *consultTravelData* operation

i. If the bookingID is invalid (process finishes or does not exist), *CSInvalidInputDataException* is launched.

(b) *bookTrain* request. Book some train (if some lines were returned).

i. If the bookingID is invalid (process finishes or does not exist), *CSInvalidInputDataException* is launched.

ii. If the confirmation message returned from train service is not "Your train booking code is $Code", *CSBookingException* is launched.

iii. The trains can only be successfully booked once for origin train and destiny train (if there were listed after *bookTravel*).

(c) Search and book some sightseeing in the destiny city using *searchSighteeing* and *bookSightseeing* operations.

i. If some input data is invalid, *CSInvalidInputDataException* is launched.

ii. In the *searchSighteeing* operation, the operation *listCities* from the sightseeing service is called first. Then, the code for the destiny city is used to call the *listSightSeeingByCity* operation from the sightseeing service.

iii. If there is some exception with the sightseeing service, the sightseeing is already booked or the destiny city is not listed by the operation *listCities*, *CSBookingException* is launched.

iv. The client can book zero, one, or more sightseeings. The same sightseeing cannot be booked twice.

(d) search and rent some car in the destiny city using *searchCars* and *bookCar* operations.

i. If some input data is invalid, *CSInvalidInputDataException* is launched.

ii. If there is some exception with the carrental service, *CSBookingException* is launched.

iii. A car can only be booked once. A new try to book must launch *CSBookingException* if the bookingID is correct. If the bookingID is not correct *CSInvalidInputDataException* is launched. A search using *searchCars* is still possible.

(e) *orderMap* request. Order a map of the destiny city (send by email to the client) to the map service.

i. If the bookingID is invalid (process finishes or does not exist), *CSInvalidInputDataException* is launched.

ii. If the confirmation message returned from Map service is not "OK", *CSBookingException* is launched.

iii. The client can just order successfully once. A new try to book must launch *CSBookingException* if the bookingID is correct. If the bookingID is not correct *CSInvalidInputDataException* is launched.

8. The process finishes after the returnDate.

C.1.2.1 Non-Functional Requirements

This section presents some non-functional requirements for Customer Service.

1. The timeout for invoking external services is 60 seconds.

C.1.3 Implementation Details

An ESB is included to intermediate the services in the application. Thus, the application can employ a set of functionalities provided by the ESB, such as routing, security, service discovery, and integration with other technologies.

C.2 Data

Table C.1: The number of failures categorized according to the number of events

	Events (pos+neg)				Failures			
	length 2	length 3	length 4	Σ	length 2	length 3	length 4	Σ
ESG 1	10386	13161	41430	64977	1	0	0	1
ESG 2	28314	35870	129136	193320	3	0	0	3
ESG 3	28242	35798	129064	193104	1	0	0	1
ESG 4	7241	6980	16024	30245	3	1	0	4
ESG 5	2972	5596	73397	81965	0	0	0	0
ESG 6	279	538	5272	6089	0	0	0	0
ESG 7	820	1144	8527	10491	0	0	0	0
ESG 8	3395	4820	28541	36756	1	0	0	1
ESG 9	588	882	7270	8740	0	0	0	0
ESG 10	168	172	188	528	0	0	0	0
ESG 11	1781	2305	5423	9509	0	0	0	0
ESG 12	18134	24746	103997	146877	2	0	0	2
ESG 13	8488	16391	622613	647492	0	0	0	0
ESG 14	2338	2862	75533	80733	1	0	0	1
ESG 15	2338	3701	74068	80107	1	0	0	1
ESG 16	829	1762	44476	47067	0	0	0	0
ESG 17	1756	2672	64170	68598	0	0	0	0
ESG 18	6253	8834	203473	218560	1	0	0	1
ESG 19	1486	2380	59516	63382	0	0	0	0
ESG 20	4770	9081	210413	224264	0	0	0	0
ESG 21	989	2416	24987	28392	0	0	0	0
ESG 22	458	940	15046	16444	0	0	0	0
ESG 23	1112	1733	22228	25073	0	0	0	0
ESG 24	4264	6309	71907	82480	0	0	0	0
ESG 25	877	1367	20289	22533	0	0	0	0
ESG 26	2972	5596	73397	81965	0	0	0	0
ESG 27	279	538	5272	6089	0	0	0	0
ESG 28	820	1144	8366	10330	0	0	0	0
ESG 29	3395	4820	28541	36756	0	0	0	0
ESG 30	588	882	7431	8901	0	0	0	0
ESG 31	4983	9086	210579	224648	0	0	0	0
ESG 32	989	1577	26452	29018	0	0	0	0
ESG 33	513	1027	15048	16588	0	0	0	0
ESG 34	1112	1628	22228	24968	0	0	0	0
ESG 35	4487	6143	71910	82540	0	0	0	0
ESG 36	877	1367	20289	22533	0	0	0	0
ESG 37	168	176	192	536	0	0	0	0
ESG 38	20720	25492	129228	175440	0	0	0	0
ESG 39	6979	8665	39225	54869	0	0	0	0
ESG 40	9329	12162	37601	59092	0	0	0	0
ESG 41	4770	9081	210413	224264	0	0	0	0
ESG 42	989	1577	24987	27553	0	0	0	0
ESG 43	458	940	15046	16444	0	0	0	0
ESG 44	1112	1628	22228	24968	1	0	0	1
ESG 45	4264	6035	71907	82206	0	0	0	0
ESG 46	877	1367	20289	22533	1	0	0	1
ESG 47	2972	5848	73777	82597	0	0	0	0
ESG 48	279	538	5413	6230	0	0	0	0
ESG 49	820	1144	8366	10330	0	0	0	0
ESG 50	3470	4822	28951	37243	0	0	0	0
ESG 51	588	882	7270	8740	0	0	0	0
ESG 52	168	172	188	528	0	0	0	0
ESG 53	5000	6298	21288	32586	0	0	0	0
ESG 54	3449	4385	13505	21339	0	0	0	0
ESG 55	9329	12982	36211	58522	0	0	0	0
ESG 56	4770	9081	210413	224264	0	0	0	0
ESG 57	989	1577	24987	27553	1	0	0	1
ESG 58	458	940	15046	16444	0	0	0	0
ESG 59	1187	1630	22230	25047	0	0	0	0
ESG 60	4264	6035	71907	82206	1	0	0	1
ESG 61	877	1367	20289	22533	0	0	0	0
ESG 62	3047	5598	73399	82044	0	0	0	0
ESG 63	279	538	5272	6089	0	0	0	0
ESG 64	820	1144	8366	10330	0	0	0	0
ESG 65	3395	4820	28541	36756	0	0	0	0
ESG 66	588	882	7270	8740	0	0	0	0
ESG 67	168	172	188	528	0	0	0	0
ESG 68	5000	6298	21288	32586	0	0	0	0
ESG 69	3449	4385	13505	21339	0	0	0	0
	264325	**374929**	**3835287**	**4474541**	**18**	**1**	**0**	**19**

Table C.2: Usage ratio of ESGs

	URE		URE
ESG 1	0.03929254	ESG 36	0.00331789
ESG 2	0.10711813	ESG 37	0.00063558
ESG 3	0.10684574	ESG 38	0.07838835
ESG 4	0.02739431	ESG 39	0.0264031
ESG 5	0.01124373	ESG 40	0.03529367
ESG 6	0.00105552	ESG 41	0.01804597
ESG 7	0.00310224	ESG 42	0.00374161
ESG 8	0.01284404	ESG 43	0.00173272
ESG 9	0.00222453	ESG 44	0.00420694
ESG 10	0.00063558	ESG 45	0.01613166
ESG 11	0.00673792	ESG 46	0.00331789
ESG 12	0.06860494	ESG 47	0.01124373
ESG 13	0.03211198	ESG 48	0.00105552
ESG 14	0.00884517	ESG 49	0.00310224
ESG 15	0.00884517	ESG 50	0.01312778
ESG 16	0.00313629	ESG 51	0.00222453
ESG 17	0.00664334	ESG 52	0.00063558
ESG 18	0.02365648	ESG 53	0.01891611
ESG 19	0.00562187	ESG 54	0.01304833
ESG 20	0.01804597	ESG 55	0.03529367
ESG 21	0.00374161	ESG 56	0.01804597
ESG 22	0.00173272	ESG 57	0.00374161
ESG 23	0.00420694	ESG 58	0.00173272
ESG 24	0.01613166	ESG 59	0.00449068
ESG 25	0.00331789	ESG 60	0.01613166
ESG 26	0.01124373	ESG 61	0.00331789
ESG 27	0.00105552	ESG 62	0.01152748
ESG 28	0.00310224	ESG 63	0.00105552
ESG 29	0.01284404	ESG 64	0.00310224
ESG 30	0.00222453	ESG 65	0.01284404
ESG 31	0.01885179	ESG 66	0.00222453
ESG 32	0.00374161	ESG 67	0.00063558
ESG 33	0.00194079	ESG 68	0.01891611
ESG 34	0.00420694	ESG 69	0.01304833
ESG 35	0.01697531		

Table C.3: Reliability results of each ESG, their impacts and R_c

Pos.	ESG	# cum. events	cum. no. of failures	RE	Impact
		0	0		
1	ESG 2	28314	3	0.99926311	**0.875234468**
2	ESG 3	56556	4	0.99929238	**0.880494158**
3	ESG 38	77276	4	0.99931311	**0.914891862**
4	ESG 12	95410	6	0.99933075	**0.927427172**
5	ESG 1	105796	7	0.99934065	**0.959049777**
6	ESG 40	115125	7	0.99934942	**0.963706467**
7	ESG 55	124454	7	0.99935807	**0.964189084**
8	ESG 13	132942	7	0.99936584	**0.967811853**
9	ESG 4	140183	10	0.99937239	**0.972824563**
10	ESG 39	147162	10	0.99937865	**0.974068847**
11	ESG 18	153415	11	0.9993842	**0.976973913**
12	ESG 53	158415	11	0.9993886	**0.981719601**
13	ESG 68	163415	11	0.99939297	**0.981850293**
14	ESG 31	168398	11	0.9993973	**0.98204088**
15	ESG 20	173168	11	0.99940141	**0.982925822**
16	ESG 41	177938	11	0.99940549	**0.983042295**
17	ESG 56	182708	11	0.99940955	**0.983157974**
18	ESG 35	187195	11	0.99941334	0.984258879
19	ESG 24	191459	11	0.99941692	0.985132451
20	ESG 45	195723	11	0.99942047	0.985223146
21	ESG 60	199987	12	0.99942401	0.985313288
22	ESG 50	203457	12	0.99942687	0.98810747
23	ESG 54	206906	12	0.9994297	0.988237802
24	ESG 69	210355	12	0.99943251	0.988295874
25	ESG 8	213750	13	0.99943527	0.988535114
26	ESG 29	217145	13	0.99943802	0.988590834
27	ESG 65	220540	13	0.99944075	0.988646284
28	ESG 62	223587	13	0.99944319	0.98985454
29	ESG 5	226559	13	0.99944556	0.990146379
30	ESG 26	229531	13	0.99944792	0.990188314
31	ESG 47	232503	13	0.99945027	0.990230071
32	ESG 14	234841	14	0.99945211	0.992339978
33	ESG 15	237179	15	0.99945394	0.992365636
34	ESG 11	238960	15	0.99945534	0.994199274
35	ESG 17	240716	15	0.99945671	0.994295094
36	ESG 19	242202	15	0.99945787	0.995182556
37	ESG 59	243389	15	0.99945879	0.996158429
38	ESG 23	244501	15	0.99945965	0.996406895
39	ESG 34	245613	15	0.99946051	0.996412625
40	ESG 44	246725	16	0.99946137	0.996418345
41	ESG 21	247714	16	0.99946214	0.996819035
42	ESG 32	248703	16	0.9994629	0.996823546
43	ESG 42	249692	16	0.99946366	0.996828051
44	ESG 57	250681	17	0.99946442	0.99683255
45	ESG 25	251558	17	0.9994651	0.997194783
46	ESG 36	252435	17	0.99946577	0.997198311
47	ESG 46	253312	18	0.99946644	0.997201835
48	ESG 61	254189	18	0.99946711	0.997205354
49	ESG 16	255018	18	0.99946775	0.997361452
50	ESG 7	255838	18	0.99946837	0.997393166
51	ESG 28	256658	18	0.999469	0.997396232
52	ESG 49	257478	18	0.99946962	0.997399294
53	ESG 64	258298	18	0.99947024	0.997402353
54	ESG 9	258886	18	0.99947069	0.998138868
55	ESG 30	259474	18	0.99947114	0.998140438
56	ESG 51	260062	18	0.99947158	0.998142006
57	ESG 66	260650	18	0.99947203	0.998143573
58	ESG 33	261163	18	0.99947242	0.998381554
59	ESG 22	261621	18	0.99947277	0.998556021
60	ESG 43	262079	18	0.99947311	0.99855697
61	ESG 58	262537	18	0.99947346	0.998557918
62	ESG 6	262816	18	0.99947367	0.999121878
63	ESG 27	263095	18	0.99947388	0.99912223
64	ESG 48	263374	18	0.99947409	0.999122581
65	ESG 63	263653	18	0.9994743	0.999122932
66	ESG 10	263821	18	0.99947443	0.999472001
67	ESG 37	263989	18	0.99947455	0.999472128
68	ESG 52	264157	18	0.99947468	0.999472255
69	ESG 67	264325	18	0.99947481	0.999472382
			R_c=	**0.99936734**	