Layer-Centric Testing

Fevzi Belli[+], Nevin Güler[*], Michael Linschulte[+]

[+]) Department of Software Engineering, University of Paderborn, Paderborn, Germany
[*]) Department of Statistics, University of Mugla, Mugla, Turkey

Summary / Abstract

Model-based testing attempts to generate test cases from a model focusing on relevant aspects of a given system under consideration (SUC). When SUC becomes too large to be modeled in a single step, existing design techniques usually require a modularization of the modeling process. Thereby, the refinement process results in a hierarchical decomposition of the SUC in several hierarchical layers. Conventional testing requires the refined components be completely replaced by these subcomponents for test case generation. We present a new test case generation strategy based on the different layers of the model to reduce (i) the number of test cases, and thus (ii) the costs of test case generation and test execution. A case study based on a complex application component of a large web-based commercial system validates the approach and discusses its characteristics regarding the number of test cases and their fault detection ability. Surprisingly, most of the faults could be detected by a considerably reduced test case set.

1 Introduction

Software testing is the process that aims to increase confidence in the quality of software by checking whether the software does what is expected to do or not. In recent years, many approaches have been developed that are addicted to software testing. Model-based testing (MBT) has several advantages: it enables testers to concentrate on the relevant aspect of the given system under consideration (SUC) and, depending on the underlying model features and the test criterion considered, the ability of automatic generation of test cases (forming *test suites*) even if the source code of SUC is not available as well as update and reuse of these test cases with evolving requirements.

MBT creates a model of SUC's behavior out of the specification to generate test cases. Most of these models are graph-based, as collected in UML [1]. A common problem with deriving tests from the model is that the complexity of SUC can cause a state space explosion. Therefore, most of the existing techniques enable refinement of the starting model by additional models resulting in a hierarchical decomposition of the SUC **(Figure 1)**. For testing, the refined components have to be resolved completely before test sequences are generated **(Figure 2)**. Very often, a complete resolution of the refined vertices results in a large model where test case generation and test execution will take much longer since the generation and execution effort raises with increasing number of vertices. Thus, the deeper the model hierarchy, the more expensive becomes the test generation and execution process.

Our previous work analyzed the effect of test sequence length on the fault detection capability of model-based testing for a single layer model, i.e., we did not consider model refinement [2]. This paper suggests a new test generation technique to further reduce test costs if modeling process produces a hierarchical set of models.

The next section summarizes related work. Section 3 introduces the terminology and notion used in our approach and describes the new test case generation strategy for hierarchical models. Section 4 validates the approach and determines its characteristic features over a case study based on a commercial web-based software system. Section 5 concludes the paper summarizing the results and outlines our research work planned.

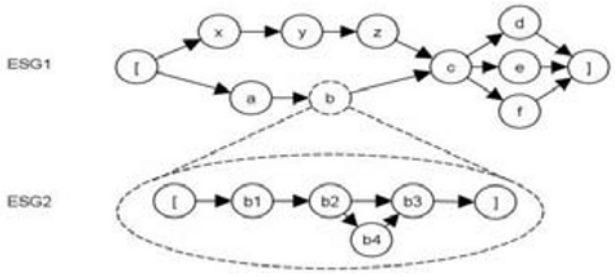

Figure 1 ESG where the compound vertex *b* of ESG1 is refined by ESG2.

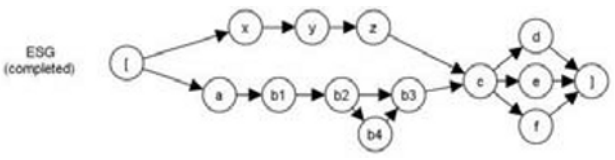

Figure 2 Resolved version of ESGs given in **Figure 1**.

2 Related work

A broad variety of formal or informal models exist for modeling and testing software as recommended in standards such as UML [1] or TTCN-3 [3]. 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 SUC description. Event sequence graphs (ESG) [4], similar to the concept of event flow graphs [5], can be used for modeling, analysis and validation of system behavior and user interface requirements prior to implementation and testing of the code. The present paper chooses ESG notation because it intensively uses formal notions and algorithms known from graph theory and automata theory, which are relevant to the approach introduced in this paper.

Memon et al. [6] use an automatic planning system to generate test cases from GUI events and their interactions called Planning Assisted Tester for graphical systems (PATHs). Paiva et al. [7] presented an approach for modeling with Hierarchical Finite State Machines (HFSMs) and to generate test cases from those models in an optimized way taking advantage of the hierarchical structure. Andrews et al. [8] proposed a system-level testing technique that combines test generation based on FSMs with constraints. Reza et al. [9] use a high level Petri nets known as Hierarchical Predicate Transition Nets (HPrTNs) to model the behavior of SUC and to generate adequate test cases using this model. Memon et al. [10] define a new class of coverage criteria using event sequences.

All of the mentioned approaches focus on hierarchical structures. However, none of them is comparable with our approach that makes use of this hierarchy for producing optimized test suites.

3 Test generation and minimization

This section summarizes the technique used for graph-based modeling of SUC and introduces the new strategy to generate and select test cases.

3.1 Event Sequence Graphs

Belli et al. introduced an event-based approach based on modeling with *event sequence graphs* (*ESG*) for testing human-computer systems [4]. Vertices of the ESG represent externally observable phenomena ("events"), that is, an environmental or a user stimulus, or a system response, punctuating different stages of system activity. Directed edges connecting two events define allowed sequences among these events.

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. The remaining pairs $\bar{E}$ that can be constructed by all combinations $\hat{E}=V\times V$ of the nodes given in the alphabet Σ, but not in the ESG, that is, $\bar{E}= \hat{E}\backslash E$, form the set of *faulty event pairs* (*FEP*). The set of FEPs constitutes the *complement* of the given ESG, which is symbolized as $\overline{ESG}$.

As a convention, a dedicated start vertex, e.g., *[*, remarks 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.

A vertex representing a single, self-contained event is called *atomic* event/vertex. Besides, vertices can be refined by another ESG **(Figure 1)** resulting in a hierarchy of models, i.e., they are *compound* events/vertices consisting of atomic events and/or even other compound events.

A sequence of *n+1 consecutive events* that represents the sequence of n edges is called an *event sequence (ES)* of length $n+1$, e.g., an ES of length 2 is an EP. An ES is *complete* if it starts at the initial event of the ESG and ends at the final event; in this case it is called a *complete ES* (*CES*). Occasionally, CES are also called *walks* (or *paths*) through the ESG given. 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* if it starts at the initial vertex of the ESG; in this case it is called *faulty complete ES*, abbreviated as *FCES*. An FCES must not necessarily end at the final event. FESs that are not complete, can be completed by ESs (*starters*) that start at the entry and end at the first node of the considered FES.

3.2 Test case generation

CESs and FCESs form the test sequences (as test cases). Note that a CES is supposed to lead to the exit vertex. If this is not feasible, the corresponding CES is marked as failed (*positive* testing). 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). If this is feasible, the corresponding FCES is marked as failed. 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. This problem is a derivation of the *Chinese Postman Problem* (*CPP*) that attempts to find the shortest path or circuit visiting each arc in a directed or even undirected graph [11].

As already mentioned before, 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 [11]. The number of FCESs for negative testing increases with increasing number of vertices since $|FCES| = |V|^2-|E|$. For enabling the solution of the CPP, the given graph is to be extended by additional edges until it forms an *Eulerian graph*. A directed graph is Eulerian if it is strongly connected and each of its vertices $v\in V$ has the same number of incoming and outgoing edges. The resulting *Eulerian cycle*, which can be obtained by a standard algorithm in $O(|V|*|E|)$ time [12], is a minimal solution to the CPP if the set of added edges is minimal.

Determining the set of minimal edges leads to a solution of the *assignment problem* [13] which attempts 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 minimized.

Formally, an assignment problem minimizes the o*bjective function* (1) for a given $n \times n$ cost matrix c_{ij} which fulfills the given constraints (2), (3) and (4) at the same time.

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \quad (1)$$

$$\text{s.t.} \sum_{j=1}^{n} x_{ij} = 1 \quad (i = 1,...,n) \quad (2)$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad (j = 1,...,n) \quad (3)$$

$$x_{ij} \in \{0,1\} \quad (i, j = 1,...,n) \quad (4)$$

In our case, c_{ij} defines the number of edges of the shortest path (as costs) between vertex *i* with |*incoming edges*|>|*outgoing edges*| and a vertex *j* with |*incoming edges*|<|*outgoing edges*|. After minimization, x_{ij}=1 indicates that edges along the shortest path from vertex *i* to vertex *j* have to be added.

Example 1: Table 1 shows the resulting cost matrix to be solved for **Figure 2**. According to **Table 1**, following shortest paths (indicated by dark grey boxes) have to be added to the ESG given in **Figure 2** to create a minimal Eulerian cycle: *]→[,]→b2, b3→c*. ■

Table 1 The resulting cost matrix out of **Figure 2**

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

One of the most well-known solutions of the assignment problem, *Hungarian algorithm*, provides a solution with $O(n^3)$ time [13]. Other $O(n^3)$ solutions are given by Dinic-Kronrod [13] or Cycle Canceling [14].

3.3 Minimizing the generated test case set

Example 1 is intentionally constructed in such a way that the determination of test cases becomes very simple. In large hierarchical models the determination of test cases can become considerably more complex. Therefore, we suggest (instead of resolving compound vertices) to generate test cases for each ESG on its own, i.e., we suggest a *layer-centric* (*LC*) test generation method instead of a *full resolution* (*FR*) test generation method. In other words, we leave the compound vertices unresolved within LC-testing. This will reduce the effort of finding a minimal solution since

$O(|V_{resolved}|^3) > O(|V_1|^3)+\ldots+ O(|V_k|^3)$

where *k* is the number of single ESGs forming the hierarchy, that is k=2, for **Figure 1**. This strategy will reduce also the number of negative test cases significantly since FEPs between different ESG layers are not considered.

Example 2. Consider ESG1 and ESG2 of **Figure 1**. The optimization algorithm given in [4, 11] generates for ESG1 and ESG2 following CESs:

```
ESG1:              ESG2:
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 ]
```

Thus, we have 5 test cases for positive testing. The number of resulting FCESs (negative testing) for ESG1 is 72 and for ESG2 this number is 12. ■

Analysis of Example 2 reveals following problem: Compound vertices, e.g., `b` in Example 4, 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, also test costs will increase. Therefore, there is a need to weight the compound events based on the number of events of the shortest ES through the graph since this will be the minimum effort to execute the compound vertex.

Example 3. The weight of ESG2 of **Figure 1** is *3*, because the shortest ES possible is "`[ b1 b2 b3 ]`". Note that the pseudo events do not contribute to the weight. ■

Example 4. If we take the weight of the compound event into account, the test set of Example 3 modifies as follows.

```
ESG1:              ESG2:
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 ]                        ■
```

Emerging Problem

Now, another problem has to be considered: How can test sequences be executed that contain compound vertices?

A straightforward strategy is to replace the compound vertices by test case(s) generated from the lower-layer ESG. If also this lower-layer ESG 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 5. In Example 4, replacing *b* in T3 by T4, 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 ]                    ■
```

Note that T1, T2, and T3' can be executed at the top layer, using ESG1. T4 is to be executed at the next lower layer, using ESG2. This can be seen as inconsequent. A solution

is needed in which the minimal coverage of ESG1 contains node *b* as many times as test cases exist for the compound vertex *b*. This is achieved by an extension of the cost matrix of ESG1 by additional columns and rows for each vertex that is needed more often. **Table 2** shows the resulting cost matrix extended by vertex b.

Table 2 The extended cost matrix for ESG1

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

Example 6. Taking **Table 2** into account results in the following test sequences for ESG1 and ESG2:

```
ESG1:                    ESG2:
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 ]
```

Replacing vertex *b* in T2 and T3 by T4 and T5, respectively, following test sequences can be constructed for ESG1 and ESG2:

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 ]
```

■

In contrast to generating CESs, calculating FCESs for negative testing is considerably easier. FCESs of ESGs of lower layer are generated first and then returned to next higher layer where the shortest ES from start vertex *[* to the corresponding compound vertex $v \in V$ is calculated and concatenated with the given FCESs of the lower layer model. The shortest ES can be constructed in $O(|V|^2)$ time, using, for example, Dijkstra's shortest path algorithm [15].

3.4 Generating event sequences of length>2

A phenomenon in testing of interactive systems is that faults can be frequently detected but reproduced only in specific context(s). Coverage of event sequences of length >2 can help here. To achieve such coverage(s), the original graph will be transformed to enable the usage of algorithms [4, 11] that are introduced in Section III.C. **Figure 3** shows the transformation of ESG1 of **Figure 1** to achieve a coverage of sequence length 3. As it can easily be seen in **Figure 3** for (indexed) vertex *c*, a vertex might be contained more than once in the resulting graph.

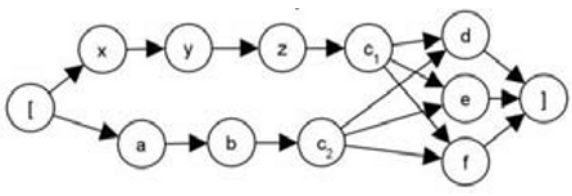

Figure 3 ESG1 transformed for length 3 coverage

If the transformed graph includes a vertex $v \in V$ more than once in the original graph 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. Formally, following set of equations described as a linear program has to be solved in such a case:

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \tag{5}$$

$$\text{s.t.} \sum_{j=1}^{n} x_{ij} = 1 \quad (i = 1,...,k) \tag{6}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad (j = 1,...,k) \tag{7}$$

$$\sum_{j=1}^{n}\sum_{i=s(y)}^{e(y)} x_{ij} = 1 \quad (y = 1,...,l) \tag{8}$$

$$\sum_{i=1}^{n}\sum_{j=s(y)}^{e(y)} x_{ij} = 1 \quad (y = 1,...,l) \tag{9}$$

$$\sum_{i=1}^{n}(x_{ij} - x_{ji}) = 0 \quad (j = s(y),...,e(y), y = 1,...,l) \tag{10}$$

$$x_{ij} \in \{0,1\}, \quad (i, j = 1,...,n) \tag{11}$$

Example 7. Table 3 shows an example where column/row 4 and 5 have been added and only one of both should be put to the solution w.r.t. minimization purposes. In this case, column/row 5 minimizes the assignment and column/row 4 has to be skipped. According to **Table 3**, *k*=3, *l*=1, *s*(1)=4, *e*(1)=5. Thus, *l* defines the number of matrix intervals from which only one should be selected (here only one consisting of column/row 4 and 5), *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. ■

Table 3 An assignment Matrix

		j				
		1	2	3	4	5
i	1	5	6	5	4	5
	2	5	6	5	4	5
	3	5	6	5	4	5
	4	7	8	7	6	∞
	5	7	1	7	∞	7

The given equation system (5)-(11) cannot be solved by the Hungarian method. An optimal solution (abbreviated as LC_{opt}) can be calculated using linear programming, but current algorithms like the simplex method have exponential runtime complexity.

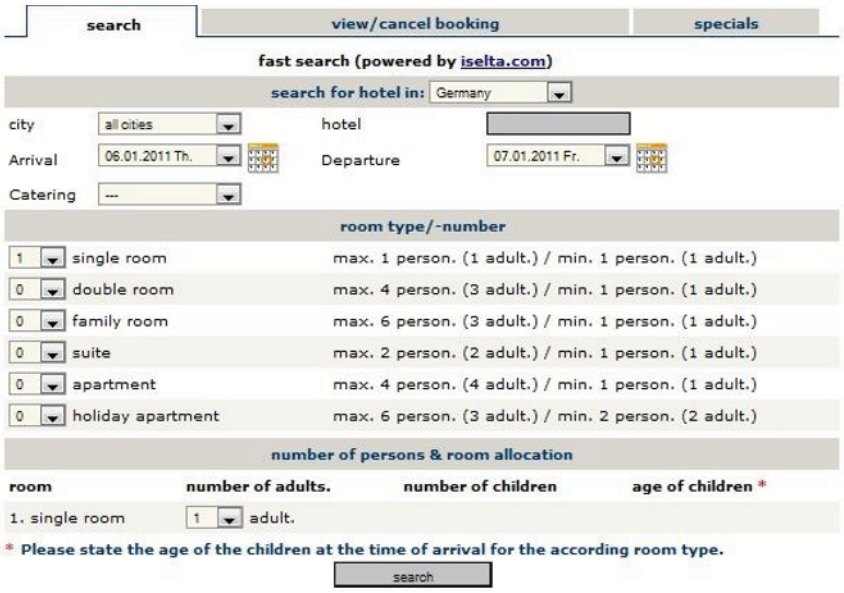

Figure 4 Screenshot of ISELTA

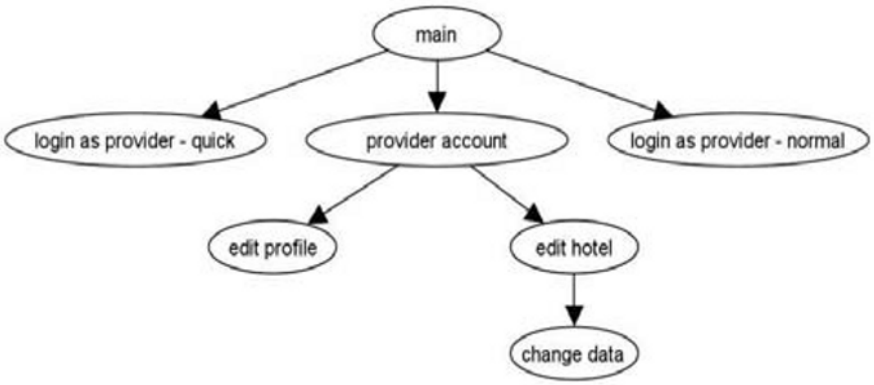

Figure 5 Hierarchical structure of the ESG used in the case study

In case that a solution cannot be found in appropriate 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 time as needed; this strategy is abbreviated as LC_{simple}. LC_{simple} is straightforward and feasible, but not necessarily minimal.

4 Case study

This chapter analyzes and validates the LC testing approach presented in Section 3. SUC is a large commercial web portal with 53K LOC (lines of code): ISELTA ("Isik's System for Enterprise-Level Web-Centric Tourist Applications"). ISELTA enables travel and tourist enterprises e.g., hotel owners, to create their individual search & service offering masks. These masks can be embedded in an existing homepage as an interface between customers and system. Potential end users can then use those masks to select and book services, e.g., hotel rooms, rental cars, etc. A screenshot of ISELTA can be seen in **Figure 4**.

For testing a large sub-system of ISELTA, 7 ESG with totally 73 vertices and 207 edges have been constructed. **Figure 5** shows the hierarchical structure of the model of ISELTA. The main (or top) ESG contains three vertices that are refined by ESGs of the 2nd layer. One of these ESGs, "provider account", is also a compound vertex and contains two sub-vertices, "edit profile" and "edit hotel", as its refinements.

4.1 Results of the case study

On the basis of this model, CESs and FCESs have been generated covering event sequences of length 2, 3 and 4. The results of the case study are summarized in **Table 4**. It can easily be seen that the number of CESs of our LC approach is lower than the number of CESs of the conventional FR approach. In total, the number of test cases

is reduced by ~80 %. Surprising is the fact that still 80 % of the faults could be detected by the test cases generated by a test suite the test case number of which has been reduced by 80%! Some of the faults detected during the case study are listed in Table 5.

Table 4 Results of the test process

	FR				LC			
length	CES	FCES	∑	Faults	CES	FCES	∑	Faults
2	5	2668	2673	35	2	585	587	29
3	22	9474	9496	35+4	4	1735	1739	29+2
4	93	38768	38861	39+0	77*	7005	7082	31+0
∑	120	50904	51024	39	83	9325	9408	31

* LC_{simple}

Table 5 Excerpt of faults detected

No.	Fault Description
1	missing "back" button
2	"back" button available where it is not allowed
3	welcome page is shown instead of the list of hotel page after insertion of an hotel
4	the quick login form in the navigation was shown on the regular login page

4.2 Threats to validity

In general, testing a system by a model-based technique assumes that the underlying model is correct and complete with respect to the considered features. The same holds for ESGs. We can just draw a conclusion about faults that can be detected by analyzing our model, that is, on faults in executing events and their order but not, e.g., on faults in database interactions. To keep the example easy to understand, the underlying ESG model we use is very simple in its fault modeling; it solely visualizes sequences of events that are considered. Data dependencies are not considered. This can be seen as restrictive; however, this view is straightforward which explains the strength of the approach (for further explanation and fault detection effectiveness of ESG approach see [16] and [4]). In spite of its simplicity, also ESG used in the case study revealed numerous faults. We believe that the simplicity of the underlying ESG concept does not influence the validity of the new test case generation approach but rather underlines its characteristic features.

Another concern is about the generalization of the results achieved which heavily depend on the given SUC, its development process and on many more factors. In our case, a web application (ISELTA) is tested on the basis of one large hierarchical model. But we are not able to conclude that the same results hold for other systems or models. In contrast, we believe that the test suite reduction capabilities will not change substantially.

4.3 Test process and tool support

The case study is supported by a test tool called TestSuiteDesigner (TSD) which can be seen in **Figure 6**. TSD allows creating hierarchical ESGs on the basis of a graphical user interface. The hierarchical structure of an ESG can be seen in **Figure 6** on the left side. The outline of the structure can also be used to navigate through the ESG given. TSD is extended by the LC testing approach described in Section 3.3 and 3.4 to generate test cases. The linear equation system (5)-(11) is solved using the open source *lp_solve* library [17]. An implementation on the basis of the GNU Linear Programming Kit (GLPK) [18] 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 values between 0 and 1 in some cases although the domain of the variables has been set to binary (i.e., 0 or 1). But if a valid solution was generated, GLPK was much faster compared to *lp_solve* although both libraries promise to implement the same algorithm.

The results of Table 4 show that a great number of test sequences had to be executed. Manual execution of these test sequences would be a time-consuming and an error-prone task. To enable automatic execution of these test sequences, TSD allows to annotate pieces of code to every node according to a given test script language. This enables to generate test scripts automatically along the generated test sequences. These test scripts can be loaded into a separate test execution environment.

In our case we used the freely available web test tool Selenium [19]. The user interface of Selenium can be seen in **Figure 7**. The right side of **Figure 7** shows an excerpt of the generated test script.

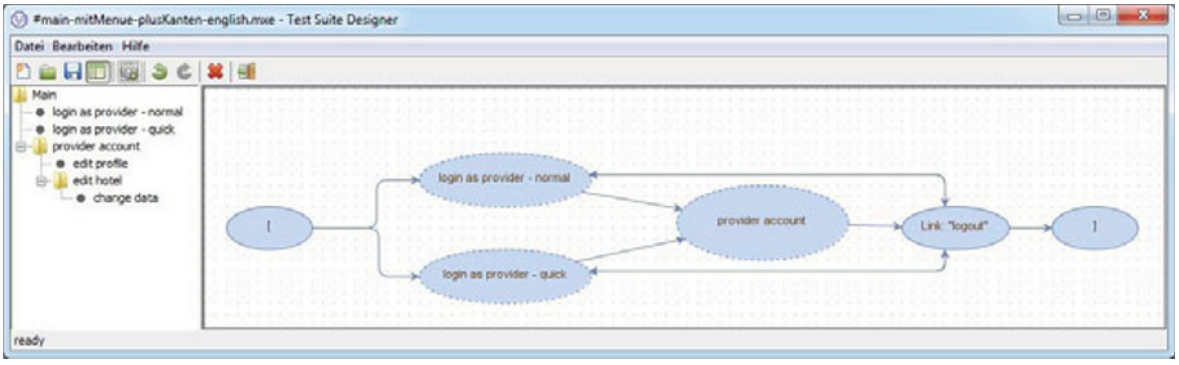

Figure 6 Screenshot of TestSuiteDesigner

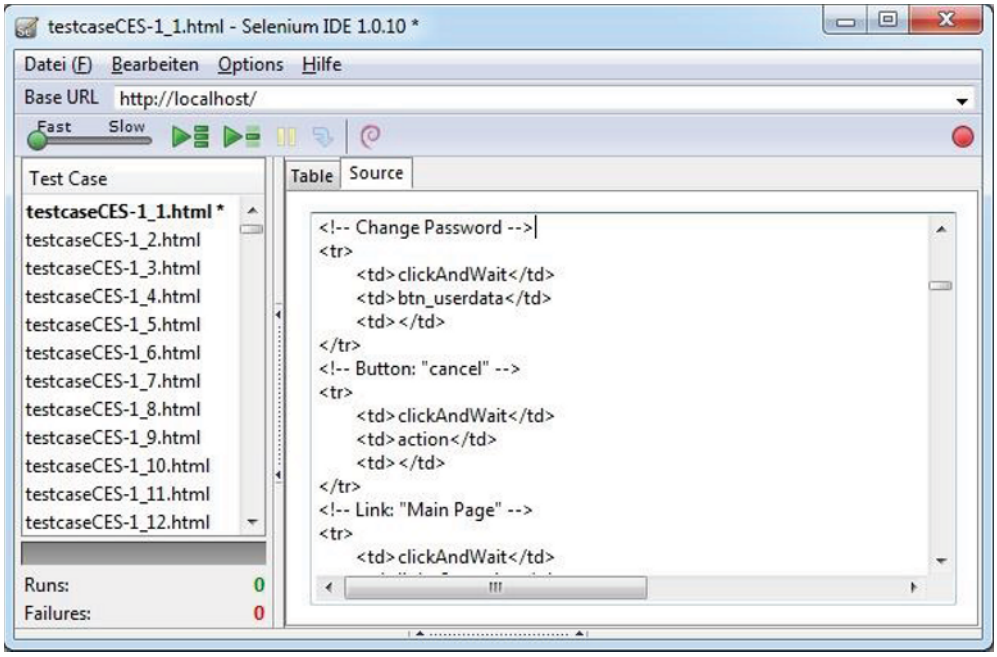

Figure 7 Screenshot of Selenium

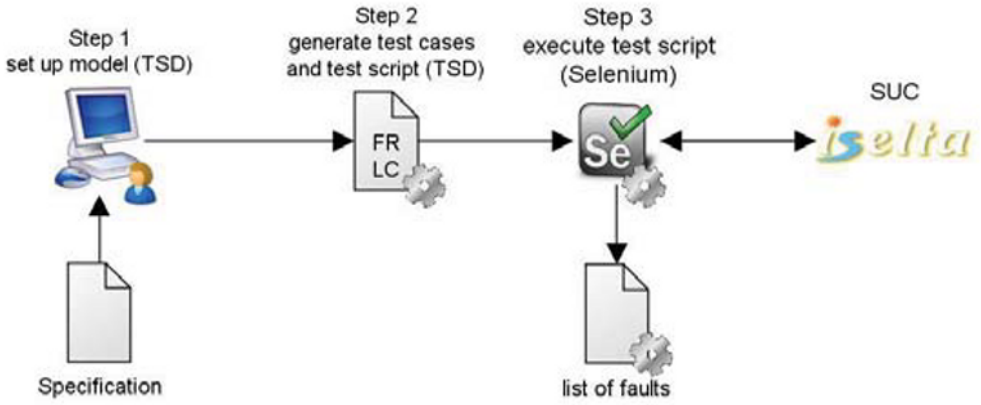

Figure 8 Test process

To conclude, **Figure 8** shows the overall test process for the case study. The first step was to develop a hierarchical ESG out of the given specification. Note that step 1 is the only step which has to be done by hand. Step 2 was to generate the test sequences and test scripts out of the given ESG model. Step 3 was to load the generated test scripts into Selenium and execute them against our SUC, namely ISELTA. The result is a list of test sequences which could be executed successfully or not.

5 Conclusions

This paper introduced a new strategy called *layer-centric* testing (*LC*) based on event sequence graphs (ESG) to overcome the problem of time consuming and labor-intensive test case generation and execution. LC makes use of the hierarchical structure of the model by generating test cases based on the single models so that the test generation and execution effort is reasonably be reduced. A case study validated and demonstrated the approach and analyzed its characteristics. Compared to the FR testing, the LC testing reduced the test suite by 80 % at a fault detection rate of 80 %. This fact shows that especially in cases where a smaller number of test cases is needed, the new strategy will achieve good results.

Nevertheless, generating sequences of higher length according to the new approach is still not satisfactory w.r.t. to its runtime complexity. Therefore, future work will concentrate on the generation of test cases of length >2.

6 Literature

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

[2] Belli, F.; Güler, N.; Linschulte, M.: Are longer test sequences always better? - A Reliability Theoretical Analysis, 4th IEEE Intern. Conf. on Secure Software Integration and Reliability Improvement, 2010, pp. 78-85

[3] Grabowski, J.; Hogrefe, D.; Réthy, G.; Schieferdecker, I.;Wiles, A.; Willcock, C.: An Introduction to the Testing and Test Control Notation (TTCN-3), The International Journal of Computer and Telecommunications Networking, Vol. 42, No. 3, 2003, pp. 375-403

[4] Belli, F.; Budnik, C.J.; White, L.: Event-based modeling, analysis and testing of user interactions: Approach and case study, Journal of Software Testing, Verification and Reliability (STVR), Vol. 16, No. 1, John Wiley & Sons, Ltd., 2006, pp. 3-32

[5] Memon, A.: An event-flow model of GUI-based applications for testing, Softw. Test. Verif. Reliab., Vol. 17, No. 3, 2007, pp. 137–157

[6] Memon, A.; Pollack, M.E.; Soffa, M.L.: Hierarchical GUI Test Case Generation Using Automated Planning, IEEE Trans. Software Eng., Vol. 27, No. 2, 2001, pp. 144-155

[7] Paiva, C.R.; Tillmann, N.; Faria, J.C.P.; Vidal, R.F.A.M.: Modeling and Testing Hierarchical GUIs, in Proc. of 12th Intern. Workshop on Abstract State Machines, Paris France, 2005, pp. 8-11

[8] Andrews, A.; Offutt, J.; Alexander, R.T.: Testing Web applications by modeling with FSMs, Software and Systems Modeling, Vol. 4, No. 3, 2005, pp. 326-345

[9] Reza, H.; Endapally, S.; Grant, E.: A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets, Intern. Conf. on Information Technology (ITNG'07), 2007, pp. 366-370

[10] Memon, A.; Soffa, M.L.; Pollack, M.E.: "Coverage criteria for GUI testing," In: 8th Europe conference and 9th ACM SIGSOFT foundation of software engineering (FSE-9), 2001, pp. 256–267

[11] Belli, F.; Budnik, C.J.: Test Minimization for Human-Computer Interaction, International Journal of Applied Intelligence, Vol. 26, No. 2, Springer, 2007, pp. 161-174

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

[13] Burkard, R.; Dell'Amico, M.; Martello, S.: Assignment Problems, Society for Industrial and Applied Mathematics, Philadelphia, 2009

[14] Thimbleby, H.: The directed Chinese Postman Problem, Journal of Software – Practice and Experience, 2003

[15] Dijkstra, E.W.: A note on two problems in connexion with graphs, Numerische Mathematik 1, 1959, pp. 269-271

[16] Belli, F.; Budnik, C.J.; Linschulte, M.; Schieferdecker, I.: Testen Web-basierter Systeme mittels strukturierter, graphischer Modelle – Vergleich anhand einer Fallstudie, GI Jahrestagung (2), 2006, pp. 266-273

[17] lp_solve: http://lpsolve.sourceforge.net/5.5

[18] GNU Linear Programming Kit: http://www.gnu.org/software/glpk

[19] Selenium: http://seleniumhq.org