




Research Article

A new approach to event- and model-based feature-driven software testing and comparison with similar approaches

Fevzi Belli ^{a,b} , Tugkan Tuglular ^{b,*}  and Ekinan Ufuktepe ^c 

^a University of Paderborn, Paderborn, Germany

^b Izmir Institute of Technology, Izmir, Türkiye

^c University of Missouri – Columbia, Columbia, MO, USA

ARTICLE INFO

Article history:

Received 26 June 2022

Revised 28 November 2022

Accepted 15 December 2022

Keywords:

Event sequence graphs
Feature-oriented software
development
Model-based testing

ABSTRACT

A software can be thought as a composition of features. Feature-oriented software development (FOSD) builds the development process on features. Part of the FOSD process is testing, and accordingly, it should be feature-driven. In model-based testing, test cases are systematically generated using the model. This research concentrates on event-based graphical models and utilizes event sequence graphs (ESGs). We develop a new test sequence generation algorithm for ESGs and named it short and frequent test sequences (SFT). Then we compare it with the existing test sequence generation algorithm called TSD. Moreover, we introduce two model-building approaches, namely daisy and swim lane, for ESGs and analyze their effects on feature-driven testing. For the evaluation, we use five different feature-driven software models. The evaluation results shows that both modeling approaches are advantageous in certain test objectives. For testing the software product as a whole, test sequence(s) should be generated by TSD from daisy modeled ESG. If a certain feature within the software product or its interaction with another feature is to be tested, then test sequence(s) should be generated by SFT from swim lane modeled ESG.

1. Introduction

A feature is a semantically cohesive entity of a software [1]. Feature-oriented software development (FOSD) aims for the configuration and composition of features to obtain a software [2]. FOSD enables software to be composed from features with respect to configuration. This approach enables reuse of features and managed variation of software, which is highly beneficial in case of software product families.

An important part of the FOSD process is testing. Although there are various approaches in testing, our scope in this research is model-based testing. In model-based testing, test cases are systematically generated using the model. The models are the behavioral specification of the software. This systematic approach enables tester to define test coverage criteria, which is important if only a feature is to be covered by the test case(s) instead of the whole software product.

Event sequence graphs (ESGs) are an event-based modeling approach for representing software under test (SUT) and generating tests case(s) or test sequence(s) [3]. Event sequence graphs can be obtained from finite state

machines (FSMs) by taking events and putting them into the vertices of a graph, where each walk on this graph, which is an ESG, can also be obtained by the corresponding walk on the FSM. The details can be found in [UYMS 2016]. ESGs are not the only approach that utilizes events as the core concept in modeling software. Event flow models [4] and event process chains [5] are two other examples.

Test case, or test sequence, generation can be seen as an optimization problem, where possible event sequences are tried to be covered with minimum number of test cases. For the building ESG models and for test sequence generation from them, a tool called TSD, which can be downloaded at <http://download.ivknet.de/>, was developed. The test sequence generation algorithm in TSD is optimized for end-to-end testing and feature-oriented testing was not a goal at its design time. However, there is a need for feature-oriented testing in model-based testing. We propose to use ESGs, where not only SUT but also features can be represented formally. As the first novelty of this research, we develop a new test sequence generation algorithm for ESGs and named it short and

* Corresponding author. Tel.: +90-232-7507875; Fax: +90-232-7507862.

E-mail addresses: belli@upb.de (Fevzi Belli), tugkantuglular@iyte.edu.tr (Tugkan Tuglular), eh46@missouri.edu (Ekinan Ufuktepe)

ORCID: 0000-0002-8421-3497 (Fevzi Belli), 0000-0001-6797-3913 (Tugkan Tuglular), 0000-0002-0156-4321 (Ekinan Ufuktepe)

DOI: [10.35860/iarej.1135989](https://doi.org/10.35860/iarej.1135989)

© 2022, The Author(s). This article is licensed under the CC BY-NC 4.0 International License (<https://creativecommons.org/licenses/by-nc/4.0/>).

frequent test sequences (SFT), which is suitable for feature-oriented testing. We compare it with the existing TSD test sequence generation algorithm.

While building ESG models to represent SUT, we observed that there could be two different model building approaches, namely daisy and swim lane, for ESGs. As the second novelty of this research, we present them and compare them to each other as well as investigate which one suits better with TSD and SFT. As a result of this research, we conclude that test sequence(s) should be generated by TSD from daisy modeled ESG if the test objective is product testing. If the test objective is feature or feature interaction testing, then test sequence(s) should be generated by SFT from swim lane modeled ESG.

The manuscript is structured as follows. After Introduction section, the methods section presents the fundamentals and explains the newly proposed SFT algorithm whereas the following section outlines and exemplifies the newly introduced daisy and swim lane model building approaches. In the results section, we present our findings and discuss them in the following section with comparison to related work. The final section concludes the paper.

2. Methods

2.1 Event Sequence Graphs

Event sequence graphs are a practical event-based behavioral modeling approach for representing software under test (SUT) and generating tests. They have a formal foundation, the formal definitions and the detailed explanations can be found in [3] and [6]. An ESG, which is a directed graph, starts with pseudo entry node vertex '[' and ends with pseudo exit vertex ']'. These pseudo vertices and their edges are not included in the vertex set and in the edge set, respectively [3]. For the ESG given in Figure 1, the vertex set V is $\{A, B, C\}$, and the edge set E is $\{(A, B), (A, C)\}$. For the ESG given in Figure 1, A could be a *Select* event, B be *Play Classical Music* event, and C be *Play Pop Music* event. So, the SUT behaves either *Select - Play Classical Music* or *Select - Play Pop Music*. Various examples can be found in [3], [6], [7], and [8].

A test sequence, or complete event sequence (CES), starts with the entry of the ESG and ends at its exit. One or more CES can be used for feature testing, but all CESs are required for product testing. We differentiate feature testing from product testing so that covering feature vertex set and edge set is sufficient for feature testing. We are not interested in interaction among the features for feature testing, whereas feature interaction is critical in product testing.

The following subsection outlines the existing test generation algorithm for ESGs, which we will compare with our newly developed test sequence generation algorithm for ESGs, which is explained in Section 3.

2.2 Existing Test Generation Algorithm for Event Sequence Graphs

One approach to generate CESs from ESG solves the Chinese Postman Problem (CPP) [7]. Solving CPP means finding the Euler cycles on the graph, i.e., starting from and returning to the same vertex by visiting each edge exactly once [8]. To achieve this, ESG is converted to a Euler graph by creating a pseudo edge from exit vertex to entry vertex [7]. Then this graph is balanced by assigning a positive degree vertex partition to a negative degree vertex partition and this assignment problem is solved by the Hungarian Matching Algorithm [9]. Further details can be found in [7] and [8]. The existing test sequence generation algorithm for ESGs is referred as TSD, since it is used by the TSD tool. In summary, this algorithm aims to cover all edges in ESG while trying to avoid using a previously passed edge. The algorithm achieves minimum number of tests.

2.3 New Test Generation Algorithm for Event Sequence Graphs

We present our new test generation algorithm for ESGs, which aims generating frequent but shorter test sequences from ESG. For generating short test sequences, the algorithm takes advantage of the well-known shortest path finding Dijkstra algorithm. Furthermore, our test generation algorithm trade upon the structure and nature of ESG. The ESG graphs can be defined as Hammock graphs [10],[11], which means that the graph has only one entry point/vertex and one exit point/vertex. The brief and general strategy of generating short and frequent test sequences is by randomly selecting one vertex (except the starting and ending vertex), then finding the shortest path from the starting vertex to the randomly selected vertex and finding the shortest path from the randomly selected vertex to the exit vertex. Finally, the two paths are connected from the randomly selected vertex, which represents the test sequence.

In Algorithm 1, we have given the formal algorithm to generate frequent and short test sequences from ESGs. The algorithm receives the ESG as an input to the algorithm and returns a test suite as an output. The generated test suite ensures that there is a 100% event-pair (edge) coverage. However, before generating the test cases for the test suite, the algorithm goes through a graph construction phase. The Dijkstra algorithm [12] finds the shortest path based on the vertices (events). Dijkstra algorithm has many applications in graph theoretic problems such as route planning [13] and path planning [14]. In graph theory, edges can be represented as vertices, which transforms the original graph into a new graph such as given in Figure 1. Once the graph is transformed graph, it will be beneficial in several ways; (1) enabling to perform the Dijkstra algorithm based on the event-pairs from the original graph, (2) allowing to select source and target event-pairs for finding shortest path, (3)

simplifying and guaranteeing the 100% event-pair coverage.

After the ESG is transformed, we provide the transformed ESG as an input to Algorithm 1 shown Figure 2. However, we recall that the entry node to the original ESG is vertex “[”, and the exit vertex is “]”. Therefore, in the transformed graph of ESG, it is likely to have multiple options of starting vertices and multiple options of exit vertices. The naive approach would be randomly selecting among the multiple starting vertices and multiple exit vertices. However, random selection has the risks of selecting a longer path, which violates the main objective of generating short test sequences. To eliminate the possibility of generating longer test sequences, while we have the chance to generate a shorter path, we reconstruct the graph by inserting one *pseudo starting vertex* (v_s), and one *pseudo exit vertex* (v_e). The v_s vertex has outgoing edges to the starting vertices from the transformed graph, and the v_e vertex has incoming edges from the exit vertices from the transformed graph, such as given in Figure 3. The graph reconstruction enables to have a Hammock graph, which will simplify test sequence generation process.

Once we have completed the graph reconstruction (inserting one pseudo starting and one exit vertex) we are able initiate the test generation process. Our test generation algorithm has an option to generate feature-oriented test sequences, or simply generate test sequences regardless of any feature information given in prior. If feature-oriented test sequences wanted to be generated, this information should be given. The required information is simply by providing which event-pairs (edges of the original ESG) are mapped to the desired feature of the variant. Then, this information is acquired by the “*getFeatureRelatedEdges()*” function in SFT algorithm for ESG. However, if no such information is provided, the function will return an empty set, and will not generate feature specific test sequences.

Now we assume that we have two edges namely, “[\rightarrow A” and “A \rightarrow B” that are related to a feature. Therefore, initially we must randomly select one edge among the feature related edges. Such as given in Figure 4, assume that the “A \rightarrow B” vertex is randomly selected. Then, we find the shortest path from v_s to “A \rightarrow B” (path P_0), and the shortest path from “A \rightarrow B” to v_e (path P_1).

- $P_0: v_s \rightarrow ([\rightarrow A) \rightarrow (A \rightarrow B)$
- $P_1: (A \rightarrow B) \rightarrow (B \rightarrow] \rightarrow v_e$

After finding the paths P_0 and P_1 , we connect the two paths from the end of P_0 to the beginning of P_1 . Finally, the connected two paths represent a test sequence t , which is given below. We also notice that the generated test sequence also contains the edge “[\rightarrow A” among feature related edges. Since that “[\rightarrow A” edge is already covered in the generated test sequence we do not specifically generate another test sequence that targets the “[\rightarrow A” edge. However, other generated test sequences might cover “[\rightarrow A” or even “A \rightarrow

B”, but they will be covered by coincidence.

- $t: v_s \rightarrow ([\rightarrow A) \rightarrow (A \rightarrow B) \rightarrow (B \rightarrow] \rightarrow v_e$

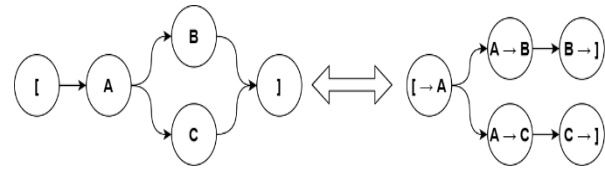


Figure 1. Graph transformation

Algorithm 1: SFT Algorithm for ESG

```

Input: Event Sequence Graph (ESG)  $G$ 
Output: Test Suite  $T$ 
 $V \leftarrow getVertices(G)$ 
 $E \leftarrow getEdges(G)$ 
// let  $v_0$  be the entry vertex/event for  $G$ 
 $v_0 \in V$ , where  $v_0 = “[$ ”
let  $v_s$  be a pseudo start vertex, where  $v_s \in V$ 
let edge  $e_s \leftarrow (v_s \rightarrow v_0)$ , where  $e_s \in E$ 
// let  $v_n$  be the exit/sink vertex for  $G$ 
 $v_n \in V$ , where  $v_n = “]$ ”
let  $v_e$  be a pseudo end vertex, where  $v_e \in V$ 
let edge  $e_e \leftarrow (v_n \rightarrow v_e)$ , where  $e_e \in E$ 
 $E' \leftarrow getFeatureRelatedEdges(E)$ 
 $E'' \leftarrow E - E'$ 
while  $E \neq \emptyset$  do
    let  $e_k$  be a randomly selected edge
    if  $E' \neq \emptyset$  then
        // Select a random event from  $E'$ 
         $e_k \leftarrow selectRandomEdge(E')$ 
        // remove edge from  $E'$ 
         $E' \leftarrow E' - \{e_k\}$ 
    end
    else
        // Select a random event from  $E''$ 
         $e_k \leftarrow selectRandomEdge(E'')$ 
    end
    // find shortest path from edge  $e_s$  to  $e_k$ 
     $P_0 \leftarrow shortestPath(e_s, e_k)$ 
    // find shortest path from edge  $e_k$  to  $e_e$ 
     $P_1 \leftarrow shortestPath(e_k, e_e)$ 
    // Connect paths  $P_0$ , and  $P_1$  to generate test sequence  $t$ 
     $t \leftarrow connectPaths(P_0, P_1)$ 
     $T \leftarrow T \cup \{t\}$  // Add test case  $t$  to test suite  $T$ 
     $F \leftarrow coveredEdges(t)$  // find covered edges
     $E \leftarrow E - F$  // removes covered edges from set  $E$ 
end
    
```

Figure 2. SFT algorithm for ESG

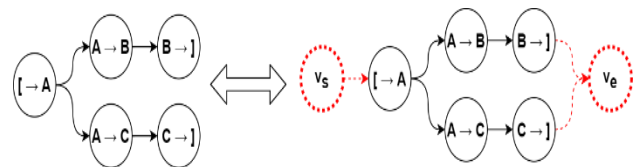


Figure 3. Inserting one *pseudo starting vertex* (v_s) and one *pseudo exit vertex* (v_e). On the left we have the transformed graph, and on the right we have reconstructed graph, which is now a Hammock Graph

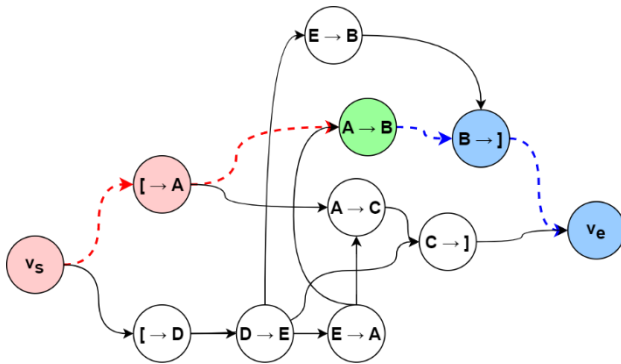


Figure 4. Running example for SFT Algorithm 1

In the next iteration, we check if there is any remaining edges that are not covered yet, and find that there are still 6 more edges that are not covered in the ESG, which are: ($[\rightarrow D)$, ($D \rightarrow E$), ($E \rightarrow A$), ($E \rightarrow B$), ($A \rightarrow C$), ($C \rightarrow]$). This means, in the next iteration the randomly selected edge will be among the remaining six edges.

This new test sequence generation algorithm for ESGs is referred as SFT, since it aims short and frequent test sequences. As opposed to TSD algorithm, its objective is not an optimized solution rather it covers all edges in ESG with short test sequences. Short test sequences have two advantages. First, they are fast and second, if there is a failure in the test sequence, other tests can still be executed.

In Algorithm 2 shown in Figure 5, we present the test minimization that is applied after Algorithm 1 composes the initial test suite. The test minimization has a straightforward approach to minimize the test suite. The SFT algorithm is likely to come up with test cases that can cover another. These types of scenarios occur if there are cycles or self-loop edges in the ESG. The number of generated tests and events are reduced by first finding the covered edges for each test cases. If the covered edges of a test are contained by another test, the contained test is removed from the test suite. Thereby, we cut down the number of tests and events to get rid of any redundancy.

2.4 Model Building Approaches for Event Sequence Graphs

We present that model building technique makes a difference in test sequence generation and in the use of the generated sequences. We utilize the bank account ESG as the running example. The terms model and ESG will be used interchangeably from this point on. Before explaining two different model building approaches, we introduce the definition of feature in ESGs.

A feature in ESG is a subgraph $F_G = (F_v, F_e)$, where F_v is the vertex set of the vertices exist in the feature and F_e is the edge set of the edges exist in the feature. Examples are given in the following two sub-sections, where daisy and swim lane model building approaches are outlined and exemplified.

Algorithm 2: Test suite minimization

```

Input: Test Suite  $T$ 
Output: Test Suite  $T'$ 
let  $T' \leftarrow \emptyset$ 
let  $C$  and  $C'$  be sets of covered edges
// Remove the test sequences that is
// covered by another test sequence
for each  $t_i$  in  $T$  do
  for each  $t_k$  in  $T$  do
    if  $t_i \neq t_k$  then
      // Gets the covered edges for test  $t_i$ 
       $C \leftarrow \text{coveredEdges}(t_i)$ 
      // Gets the covered edges for test  $t_k$ 
       $C' \leftarrow \text{coveredEdges}(t_k)$ 
      if  $C \subseteq C'$  then
        |  $T' \leftarrow T' \cup \{t_i\}$ 
      end
      else if  $C' \subset C$  then
        |  $T' \leftarrow T' \cup \{t_k\}$ 
      end
    end
  end
end
end

```

Figure 5. Test suite minimization algorithm

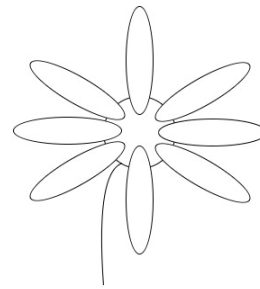


Figure 6. Daisy model

2.4.1 Daisy Model

In the daisy model, the features are attached as daisy leaves to the core of the model as given in Figure 6. The core is the main operation existing in all possible products. In the bank account ESG, it is a show menu event where all the operations of the features start and end. Therefore, each feature looks like a daisy leaf.

Figure 7 shows the bank account ESG as daisy model. An example feature as daisy leaf is *Credit* feature, of which edges are drawn in thick red. For the *Credit* feature,

F_v is {enter a credit amount, confirm credit approved, confirm credit disapproved} and

F_e is {(show menu, enter a credit amount), (enter a credit amount, confirm credit approved), (confirm credit approved, show menu), (enter a credit amount, confirm credit disapproved), (confirm credit disapproved, show menu)}.

Connection or variability point or vertex is show menu event. The features are connected through the show menu vertex.

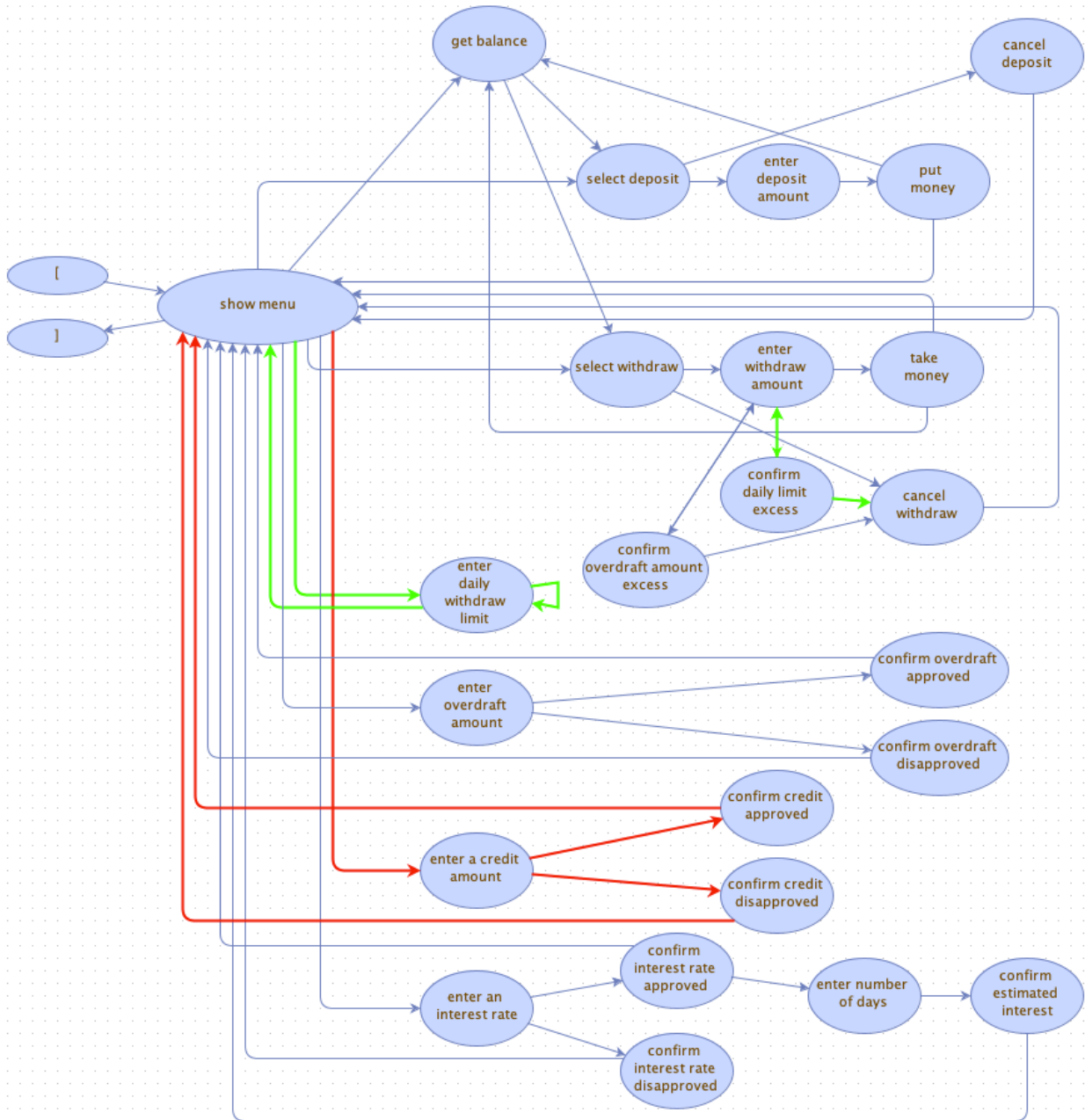


Figure 7. Bank account ESG as daisy model

A feature may interact with other features. As seen in bank account model, the DailyLimit feature, of which edges are drawn in thick green, interacts with Withdraw feature, but this does not affect the daisy leaf structure. For the features interacting with other features, special care should be taken in the process of ESG design in such a way that feature interaction is loosely coupled so that addition and removal of features do not affect the validity of ESG. For the DailyLimit feature,

F_v is {enter daily withdraw limit, confirm daily limit excess} and

F_e is {(show menu, enter daily withdraw limit), (enter daily withdraw limit, enter daily withdraw limit), (enter daily withdraw limit, show menu), (enter withdraw amount, confirm daily limit excess), (confirm daily limit excess, entry

withdraw amount), (confirm daily limit excess, cancel withdraw)}.

2.4.2 Swim Lane Model

In the swim lane model, the features are not attached as daisy leaves to the core of the model. Instead, they go from start, i.e., entry vertex of ESG, to finish, i.e., exit vertex of ESG, as given in Figure 8. Figure 9 shows swim lane model of the running example. The Credit feature, of which edges are drawn in thick red, flows from its own lane without any interaction with other features. For the Credit feature,

F_v is {enter a credit amount, confirm credit approved, confirm credit disapproved} and

F_e is {([, enter a credit amount), (enter a credit amount, confirm credit approved), (confirm credit approved,)}, (enter

a credit amount, confirm credit disapproved), (confirm credit disapproved,)}).

As seen in the swim lane ESG, the connection vertex is the pseudo start event as opposed to the show menu vertex in the daisy ESG.

The *DailyLimit* feature, of which edges are drawn in thick green, interacts with *Withdraw* feature, but this does not affect the swim lane structure. The necessary caution in ESG design explained in Section 4.1 to achieve loosely coupled features should also be taken in the swim lane model building approach.

In the swim lane ESG,

F_v is {enter daily withdraw limit, confirm daily limit excess} and

F_e is {(, enter daily withdraw limit), (enter daily withdraw limit, enter daily withdraw limit), (enter daily withdraw limit,)}, (enter withdraw amount, confirm daily limit excess), (confirm daily limit excess, entry withdraw amount), (confirm daily limit excess, cancel withdraw)} for the *DailyLimit* feature.

3. Results

In this section, we show that model building approach makes a difference in test sequence generation and the use of the generated sequences. We continue to utilize the bank account ESG as the running example.

Table 1 outlines the number of test sequences generated by both algorithms for two bank account (BA) ESGs, namely daisy BA ESG and swim lane BA ESG, as well as the total number of events in these test sequences, which is considered as the length of the test suite. Table 1 shows that TSD covers daisy BA ESG with one test sequence achieving its objective of a minimum number of test sequences. On the other hand, SFT covers daisy BA ESG with 18 test sequences with an average of 5.28 events per test, achieving its objective of short test sequences.

The reason is that due to the structure of the daisy model, specific events are covered more than once in every test sequence. These specific events inevitably cover or reach other events in the ESG. Therefore, this causes to generate test sequences with events that are already covered more than once. On the other hand, due to the structure of the swim lane model, there are alternative paths that could be reached by other events that are not covered yet. Thereby, it is more likely to generate a test suite with fewer events with fewer duplicate events.

We repeat the experiments with other four models, namely email, elevator, online shopping, and smart home. Like bank account, they model feature-based software. All the model drawings used in evaluation are available at <https://github.com/esg4aspl/comparison-of-event-based-modeling-approaches/tree/master/models>. Table 2 outlines their number of features, events, and edges. The models are from various domains with different number features. They

are sorted with respect to their number of events. Although the difference between daisy model and swim lane model is zero or just one event, the modeling approach critically affects certain choices in testing, which are discussed after delineating all the facts about the experiments.

Table 3 presents test sequence generation times for all models with respect to the modeling approach. It is observed that based on average SFT works faster than TSD. In Figure 10, we show the boxplot of execution times for SFT, which includes the outliers as well. Even with the outliers, SFT either faster or almost the same as TSD’s average execution time.

Table 1 shows that TSD covers swim lane BA ESG with 15 test sequences and 49 events. Here, we see the effect of the modeling approach. In the daisy model, the features are like daisy leaves attached to the core feature, which enables loops in the ESG. Because of these loops, TSD can cover daisy BA ESG in one test sequence. However, since there are no loops in the swim lane modeling approach and features run (swim) to completion, we observe that TSD results in a minimum of 15 test sequences with an average of 3.27 events. On the other hand, SFT covers swim lane BA ESG with 18 test sequences with an average of 3.06 events. The loop property of the daisy modeling approach affects SFT in the total number of events.

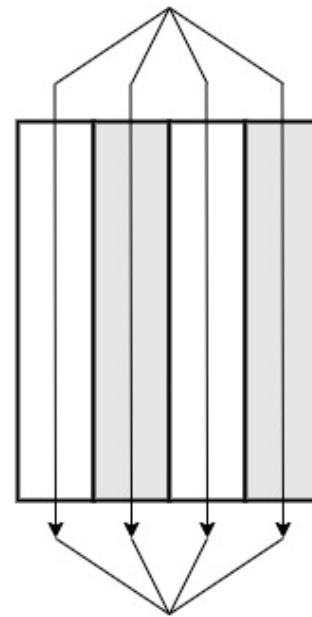


Figure 8. Swim lane model

Table 1. Bank account ESGs

Modeling Approach	TSD		SFT	
	No of test seq	No of events	No of test seq	No of events
daisy BA ESG	1	64	18	95
swim lane BA ESG	15	49	18	55

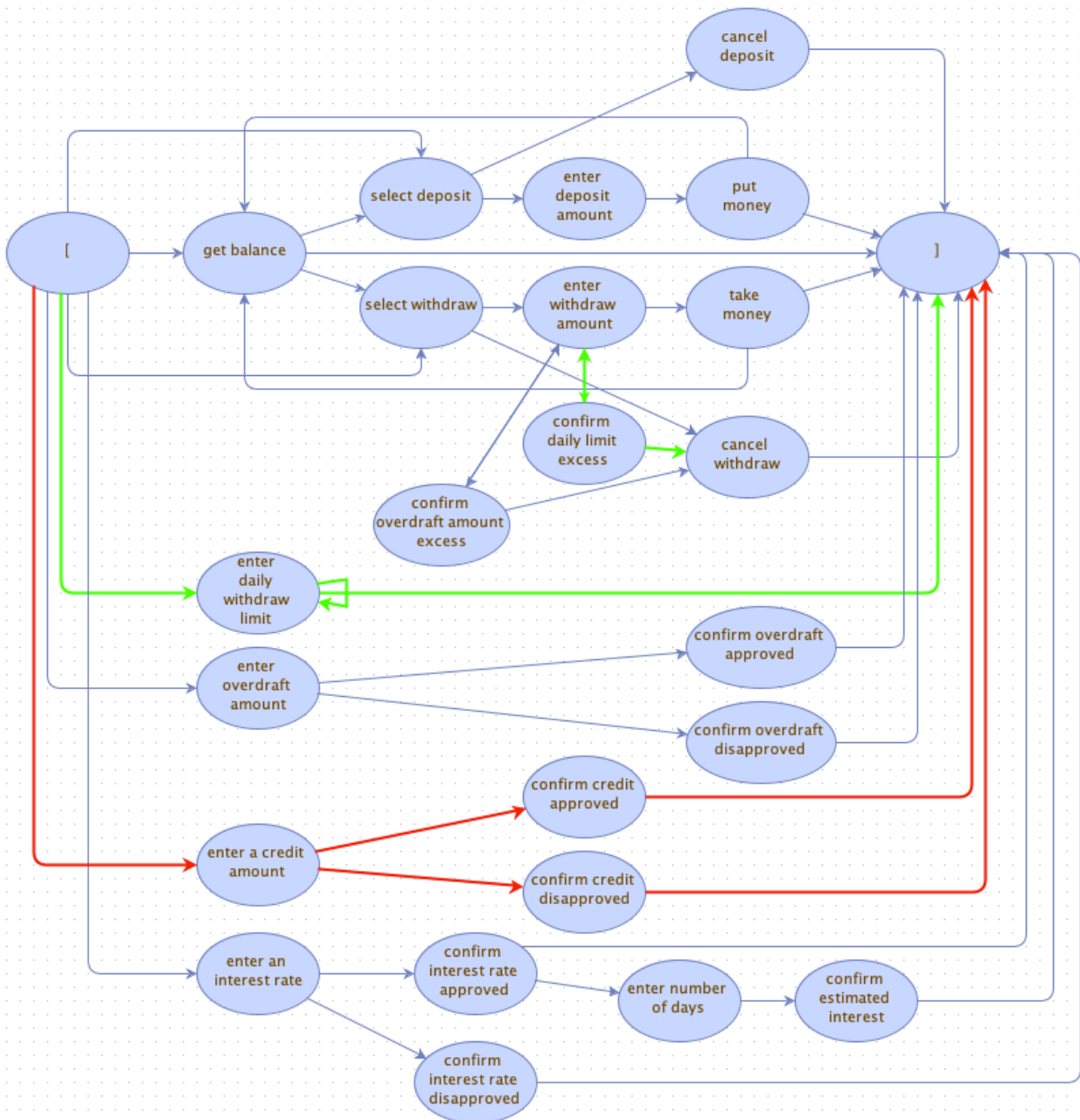


Figure 9. Bank account ESG as swim lane model

Table 2. Models under experiment

Models with No of Features	Modeling Approach	No of events	No of edges
Elevator EL (3 features)	daisy	16	26
	swim lane	15	24
Email EM (5 features)	daisy	19	38
	swim lane	18	36
Online Shopping OS (4 features)	daisy	23	37
	swim lane	23	37
Bank Account BA (9 features)	daisy	26	46
	swim lane	25	45
Smart Home SH (16 features)	daisy	41	70
	swim lane	41	70

Table 3. Test sequence generation times

Model	Modeling Approach	TSD (s)	SFT (s)
Elevator	daisy	0.095	0.082
	swim lane	0.097	0.082
Email	daisy	0.100	0.092
	swim lane	0.108	0.088
Online Shopping	daisy	0.098	0.089
	swim lane	0.099	0.890
Bank Account	daisy	0.103	0.098
	swim lane	0.106	0.095
Smart Home	daisy	0.113	0.112
	swim lane	0.120	0.106

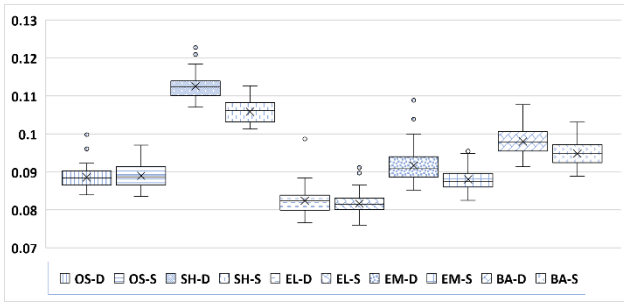


Figure 10. Boxplot of execution times for each case study

Table 4. Number of test sequences and total number of events in test sequences

Models	TSD		SFT	
	No of test seq	No of events	No of test seq	No of events
EL-D	1	43	8	78
EL-S	4	38	8	60
EM-D	1	47	16	90
EM-S	9	37	16	57
OS-D	1	53	10	81
OS-S	5	53	10	75
BA-D	1	64	17	92
BA-S	15	49	19	56
SH-D	1	83	30	170
SH-S	26	170	30	189

Table 4 presents the number of tests produced by TSD for all ten ESGs, five software modeled with two different approaches. The models are ordered on the X axis by the number of events shown in Table 2. Table 4 also presents the number of tests produced by SFT for all ten ESGs. In Table 4, we observe that the number of events for the *swim lane* model for SFT is less than the number of events from the *daisy* model. The reason is because of the algorithm of SFT and the structure of the *daisy* model. SFT aims to generate short test sequences, and when used on a *daisy* model, it generates test sequences with events and event pairs already covered.

The number of test sequences and events given in Table 4 for the SFT algorithm is rounded up to integer average values. Unlike the TSD algorithm, SFT has randomness, which may generate a different number of tests with other events. Therefore, in Figure 11 and Figure 12, respectively, we present the boxplots of the number of generated test sequences and the number of events for SFT.

4. Discussion

These experiments indicate that if the objective of testing is to test the software product as a whole, then test sequence(s) should be generated by TSD from daisy modeled ESG. If a certain feature within the software product is to be tested, then test sequence(s) should be generated by SFT from swim lane modeled ESG.

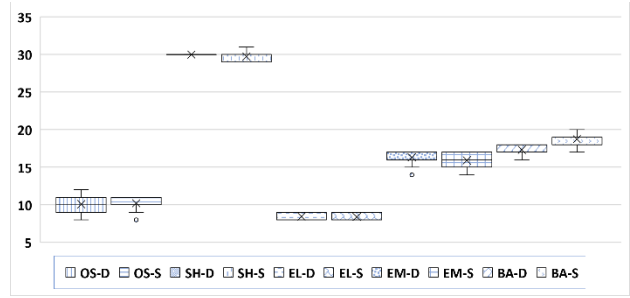


Figure 11. Boxplots of the number of generated test cases for each case study

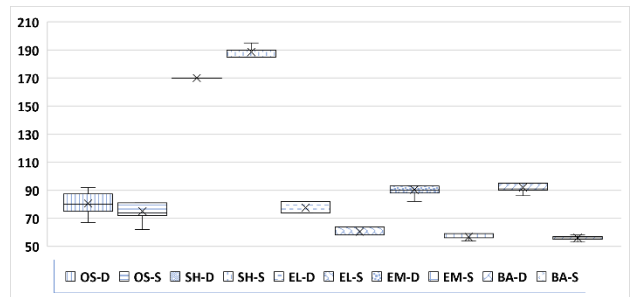


Figure 12. Boxplots of the number of events from the generated test cases for each case study

Moreover, if certain feature interactions are the goal of testing, then again test sequence(s) generated by SFT from swim lane modeled ESG should be preferred. Since both model building approaches are advantageous in certain test objectives, it would be favorable to prepare and keep ready both daisy and swim lane ESG models. This would be a tedious effort. Therefore, there should be a transformation between two models, which we plan as a future work.

4.1 Threats to Validity

We discuss the limitations of our evaluations that involves the internal and external threats to validity.

Internal Threats to Validity: The novel test generation approach SFT is based on finding the shortest paths between the start node to the selected node, and the selected node to the exit node. However, the node between the start and the exit node is selected randomly, which can lead to generating a different number of tests, and events on each run. Therefore, to evaluate if our test generation approach generates a significantly different number of tests and events, in Figure 11 and Figure 12, we show the distribution of the number generated tests and events of 50 different executions. For 10 case studies, we have observed that there are no outliers in terms of the number of events.

For the generated tests, among ten case studies, eight of them did not have any outliers. However, the remaining two case studies (OS-S and EM-D) had outliers only that generated a few numbers of test cases. We have carefully investigated the two ESG models and noticed that the two

models had cycles, including self-loops. In other words, there were edges defined in the ESG that caused feedback. During random selection, if the feedback edges are not initially selected and left for last, the SFT algorithm is likely to generate more test cases. If the feedback edges were selected earlier, it would produce fewer tests since the test sequence included the self-loop will already cover the test sequence without the self-loop. The outliers that generated fewer test cases are scenarios in which the self-loop edges were selected first compared to the other edges. Therefore, to minimize the test cases, the random selection process can assign a higher priority to self-loop edges in an ESG.

External Threats to Validity: Even though we have studied 10 case studies, there could still be different scenarios that may have not been included in this study. For instance, our study is limited to two graph models which we have defined as a daisy and swim lane. However, there could be different graph structures or types that could result in different.

4.2 Comparison with Related Work

We summarize the research on model-based testing in FOSD. Olimpiew and Gomma [15] proposed an approach for mapping the UML models, namely use case and sequence diagrams, so that functional tests are systematically produced. In Lamancha et al.'s work [16], feature scenarios are described UML sequence diagrams. Through model transformations, the sequence diagrams are converted into test cases. These approaches utilize UML models that are not formal and, therefore, error-prone compared to our proposed method.

Petry et al. [17] conducted a systematic mapping study and built a roadmap from 44 selected studies. Some of their results concerning our research are as follows: "Finite State Machines is the most used model to test SPLs" and "Behavioral-based and Scenario-based are the most used models" [17].

Lity et al. [18] utilized finite state machine models for delta-oriented testing of SPLs. Uzuncaova et al. [19] and Neto et al. [20] proposed repeated extensions through FSM deltas for delta-oriented test generation. Lochau et al. [21] proposed an integrated delta-oriented architectural test modeling and testing approach for component as well as integration testing. Their approach is component-based and aimed for integration testing. Dukaczewski et al. [22] proposed requirements-based delta-oriented SPL testing, which takes requirements into focus and uses them to define deltas.

Varshosaz et al. [23] proposed to utilize deltas for an incremental structure to formulate FSM-based test models. Devroey et al. [24] utilized featured transition systems for test generation for SPL products. Although these approaches are formal, they do not utilize a formal definition of features, and software composition is incremental with deltas. In contrast, we utilize a formal definition of features, and our composition does not require any deltas.

Belli et al. [25] mapped feature models to ESGs. This approach enabled holistic testing for the SPL and its variants. Tuglular et al. [26] introduced featured event sequence graphs, where there are distinct ESGs for each feature. They proposed a test generation technique for each product from any other smaller product, which is different than delta-oriented testing. Both research [25] and [26] used original TSD algorithm for test sequence generation and therefore are different than this research. None of the above research have introduced any model building approach.

5. Conclusion

Testing in feature-oriented software development requires validation of features alone, validation of feature interactions, and validation of the whole product. This research addresses this problem from model-based testing perspective and presents two novelties, a new test sequence generation algorithm developed considering feature and feature interaction testing and two model building approaches. An evaluation on five feature-oriented software models is performed and the results show that SFT with swim lane model building fits well to feature testing whereas TSD with daisy model building suits product testing. As seen with the examples the model building approach makes a difference in test generation. Moreover, depending on the test objective different combinations of model building approach and test generation algorithm should be used for efficient test generation in model-based testing. In the future, we are going to work on the formal definitions of the *daisy* and *swim lane* modeling techniques and algorithms for *daisy* to *swim lane* and vice versa model transformations.

Declaration

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article. The authors also declared that this article is original, was prepared in accordance with international publication and research ethics, and ethical committee permission or any special permission is not required.

Author Contributions

Tugkan Tuglular and Ekinan Ufuktepe developed the methodology and performed the evaluation. Fevzi Belli supervised and improved the study. All authors wrote and proofread the manuscript together.

References

1. Apel, S., Batory, D., Kästner, C., Saake, G., *Feature-oriented software product lines*. 2016, USA: Springer-Verlag Berlin Heidelberg.
2. Apel, S., Kästner, C., *An overview of feature-oriented software development*. J. Object Technol, 2009. 8(5): p. 49-84.

3. Belli, F., *Finite state testing and analysis of graphical user interfaces*. Proceedings of the 12th IEEE International Symposium on Software Reliability Engineering, 2001. Washington, DC, USA: p. 34-43.
4. Memon, A.M., *An event-flow model of GUI-based applications for testing*. Software testing, verification and reliability, 2007. **17**(3): p. 137-157
5. Amjad, A., Azam, F., Anwar, M.W., Butt, W.H., Rashid, M., *Event-driven process chain for modeling and verification of business requirements—a systematic literature review*. IEEE Access, 2018. **6**: p. 9027-9048.
6. Belli, F., Budnik, C.J., White, L., *Event based modelling, analysis and testing of user interactions: approach and case study*. Software Testing, Verification and Reliability, 2006. **16**(1): p. 3-32.
7. Belli, F., Budnik, C.J., *Minimal Spanning Set for Coverage Testing of Interactive Systems*. Proceedings of the International Colloquium on Theoretical Aspects of Computing, 2005. Springer Berlin Heidelberg: p. 220-234.
8. Belli, F., Guler, N., Linschulte, M., *Does" depth" really matter? on the role of model refinement for testing and reliability*. Proceedings of the IEEE 35th Annual Computer Software and Applications Conference (COMPSAC), 2011: p. 630-639.
9. Burkard, R., Dell'Amico, M., Martello, S., *Assignment problems: revised reprint*. 2012, SIAM.
10. Kas'yanov, V.N., *Distinguishing hammocks in a directed graph*. Proceedings of the Doklady Akademii Nauk, 1975.
11. Ferrante, J., Ottenstein, K.J., Warren, J.D., *The program dependence graph and its use in optimization*. ACM Transactions on Programming Languages and Systems, 1987. **9**(3): p. 319-349.
12. Dijkstra, E.W., *A note on two problems in connexion with graphs*. Numerische Mathematik, 1959. **1**: p. 269-271.
13. Kim, S., Jin, H., Seo, M., Har, D., *Optimal path planning of automated guided vehicle using dijkstra algorithm under dynamic conditions*. Proceedings of the 7th IEEE International Conference on Robot Intelligence Technology and Applications, 2019. p. 231-236.
14. Luo, M., Hou, X., & Yang, J., *Surface optimal path planning using an extended Dijkstra algorithm*. IEEE Access, 2020. **8**: p. 147827-147838.
15. Olimpiew, E.M., Gooma, H., *Model-based testing for applications derived from software product lines*. ACM SIGSOFT Software Engineering Notes, 2005. **30**(4): p. 1-7.
16. Lamanca, B.P., Diaz, O., Azanza, M., Polo, M., *Software product line testing: A feature oriented approach*. Proceedings of the IEEE International Conference on Industrial Technology, 2012. p. 298-305.
17. Petry, K. L., Oliveira Jr, E., Zorzo, A. F., *Model-based testing of software product lines: Mapping study and research roadmap*. Journal of Systems and Software, 2020. **167**: p. 110608.
18. Lity, S., Lochau, M., Schaefer, I., Goltz, U., *Delta-oriented model-based SPL regression testing*. Proceedings of the Third International Workshop on Product Line Approaches in Software Engineering, 2012.
19. Uzuncaova, E., Khurshid, S., Batory, D., *Incremental test generation for software product lines*. IEEE transactions on software engineering, 2010. **36**(3): p. 309-322.
20. Neto, P.A. da M.S., Machado, I. do C., Cavalcanti, Y.C., Almeida, E.S. de, Garcia, V.C., Meira, S.R. de L., *A Regression Testing Approach for Software Product Lines Architectures*. Proceedings of the Fourth Brazilian Symposium on Software Components, Architectures and Reuse, 2010. p. 41–50.
21. Lochau, M., Schaefer, I., Kamischke, J., Lity, S., *Incremental Model-Based Testing of Delta-Oriented Software Product Lines*. in Tests and Proofs, 2012. p. 67–82.
22. Dukaczewski, M., Schaefer, I., Lachmann, R., Lochau, M., *Requirements-based delta-oriented SPL testing*. Proceedings of the 4th International Workshop on Product Line Approaches in Software Engineering, 2013. p. 49-52.
23. Varshosaz, M., Beohar, H., Mousavi, M.R., *Delta-oriented FSM-based testing*. Proceedings of the International Conference on Formal Engineering Methods, 2015: p. 366-381.
24. Devroey, X., Perrouin, G., Schobbens, P.-Y., *Abstract test case generation for behavioural testing of software product lines*. Proceedings of the 18th ACM International Software Product Line Conference, Companion Volume for Workshops, Demonstrations and Tools, 2014: p. 86-93.
25. Belli, F., Tuğlular, T., Ufuktepe, E., *Heterogeneous Modeling and Testing of Software Product Lines*. Proceedings of the 21st IEEE International Conference on Software Quality, Reliability and Security Companion, 2021: p. 1079-1088.
26. Tuğlular, T., Beyazıt, M., Öztürk, D., *Featured Event Sequence Graphs for Model-Based Incremental Testing of Software Product Lines*. Proceedings of the 43rd IEEE International Conference on Computer, Software and Applications, 2019. Milwaukee, Wisconsin, USA: p. 197-202.