A MODEL-BASED TEST GENERATION APPROACH FOR AGILE SOFTWARE PRODUCT LINES

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Dilek ÖZTÜRK

July 2020
İZMİR

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere thanks to my supervisor Assoc. Prof. Dr. Tugkan Tuglular for his support, motivation and patience. I am grateful for sharing his experiences and knowledge with me. I also would like to thank Assist. Prof. Dr. Mutlu Beyazit sincerely for his support, motivation, advices and comments on my studies.

My thanks also go to my friends for all of their support and motivation. Finally, I would like to offer my infinite thanks to my family who gave me moral support unconditionally all of these years. This thesis study is dedicated to my dearest family.

This thesis is supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under the grant 117E884.

ABSTRACT

A MODEL-BASED TEST GENERATION APPROACH FOR AGILE SOFTWARE PRODUCT LINES

Achieving fast development of good-quality software products is as important as achieving pure functionality. Qualified software development provides client satisfaction, reduces post-deployment costs and certificates the products. In addition to increasing quality, clients expect to tailor the products according to their needs and therefore, product configurability becomes more and more critical. Hence, the software manufacturing is required to adapt this configurable development process correspondingly. Software product line is a paradigm that purposes faster development of qualified software products that belongs to a particular domain. This thesis concentrates on quality assurance in software product lines and provides novel model-based approaches which are full test sequence composition and incremental test sequence composition approaches that aim to reuse existent test artefacts. Full test sequence composition approach reuses the existing test models and the test sequences are composed from scratch each time a product variant's test sequences are generated. Incremental test sequence composition approach reuses both of the test models and the existing test sequences of product variants. Whenever a product variant's test sequences are generated, existing test sequences and features which are incrementing the existing product are composed. The proposed approaches and the classical test generation of ESGs are compared, the results show that the incremental test sequence composition is the best in terms of both test set size and test generation time, the full test sequence composition is better than the single model ESG test generation in terms of test suite size but not in terms of test generation time.

ÖZET

ÇEVİK YAZILIM ÜRÜN HATLARI İÇİN BİR MODEL TABANLI TEST ÜRETİM YAKLAŞIMI

Yazılım ürünleri geliştirilmesinde, kaliteli ve hızlı bir şekilde ürün geliştirebilmek, ürünlerden beklenen işlevselliği elde etmek kadar önemlidir. Yüksek kaliteli yazılım ürünleri, müşteri memnuniyetini artırırken dağıtım sonrası maliyetleri azaltır. Günümüzde, yazılım ürünlerinin alıcıları, sadece yüksek kaliteli değil, aynı zamanda ihtiyaçlarına göre uyarlanabilen ürünleri de beklemektedir. Bu nedenle, ürün yapılandırılabilirliği, ensdütriyel anlamda daha önemli hale gelmiştir. Yazılım üretiminin, bu yapılandırılabilir geliştirme sürecine uyum sağlaması gerekmektedir. Yazılım ürün hattı (YÜH), belirli bir alana ait yazılım ürünlerinin, yüksek kaliteli bir şekilde ve daha hızlı geliştirilmesini amaçlayan bir paradigmadır. Bu tez, yazılım ürün hatlarında kalite güvencesi üzerine yoğunlaşmakta ve mevcut test paydaşlarını yeniden kullanmayı amaçlayan, tam test sırası birleştirme ve artırımlı test sırası birleştirme isimli model tabanlı yaklaşımlar sunmaktadır. Tam test sırası birleştirme yaklaşımında, yalnızca mevcut test modelleri yeniden kullanılmaktadır ve bir ürün varyantının test sıraları her oluşturulduğunda, test modellerine ait test sıraları sıfırdan birleştirilmektedir. Artırımlı test sırası birleştirme yaklaşımında ise hem test modelleri hem de ürün varyantlarının mevcut test sıraları yeniden kullanılabilir ve bir ürün varyantının test sıraları birleştirilirken, bu yeni ürünün elde edilmesini sağlayan taban ürüne ait test sıraları ve bu mevcut ürünü artıran özelliklere ait test sıraları birleştirilmektedir. Bu tez kapsamında önerilen iki yaklaşımı ve klasik, tek-model test üretim yaklaşımlarını karşılaştırırken, sonuçlar, artırımlı test sırası birleştirme yaklaşımının hem test kümesi boyutu hem de test üretim süresi açısından en iyi olduğunu; ful test sırası birleştirme yaklaşımının ise test kümesi boyutu bakımından tek-model test üretimi yaklaşımından daha iyiyken, test üretim süresi açısından daha kötü olduğunu göstermektedir.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

Figure **Page**

Figure **Page**

LIST OF TABLES

Table **Page**

Table **Page**

CHAPTER 1

INTRODUCTION

System quality is one of the most important aspects of software development as much as achieving pure functionality (Mistrik et al. 2014). It provides client satisfaction, reduces post-deployment costs and certificates the products (Mistrik et al. 2014) In addition to increasing quality, customers expect to tailor the products according to their needs therefore, product configurability becomes more and more critical (Mistrik et al. 2014). Hence, the software manufacturing is required to adapt this configurable development process correspondingly. Paradigm for software product lines is proposed for configurable systems production (Mistrik et al. 2014) hence, this thesis focuses on quality assurance in software product lines.

Software product line paradigm states that most software systems are not new (Kang, Sugumaran, and Park 2009). More commonalities are shared by software systems in application domain (Kang, Sugumaran, and Park 2009). In fact, most companies tend to build modular software systems from reusable parts instead of designing software systems from scratch (Apel et al. 2013). For example, world-wide known companies Boeing, Bosch, Toshiba, General Motors etc. have software product line development success stories (Apel et al. 2013). Software product lines promises different advances such as tailor-made products, reduced costs, shorter development cycles and higher quality through the systematic reuse of software assets (Apel et al. 2013; Devroey et al. 2012). Agile software product lines are obtained from the transformation of classical software product lines with agile methods.

Comparing the monitored behavior of system to the expected one by stimulating the system with the pre-defined inputs is the idea behind the testing (Mistrik et al. 2014). User-centered testing examines the behavior of the system as it checks whether the software does what it is expected to do (positive testing) or not what it is not expected to do (negative testing) (Linschulte 2013). Since the number of to-be-applied tests can be infinite, the distinction between positive functioning and negative functioning, which is known as Oracle Problem (Memon, Pollack, and Soffa 2001),

comes into question. In order to overcome this problem, formal methods which *model* the behavior that is desired and undesired are proposed. These methods are called as model-based testing (MBT) methods. In model-based testing, a formal model is derived from the requirements (Mistrik et al. 2014). Furthermore, it is required to employ models from which test cases are obtained automatically (Mistrik et al. 2014). The software system's source code for test case generation does not have to be available for MBT and this makes it attractive to the industry since most of the products' source codes are not shared (Linschulte 2013).

In this thesis, a model-based approach is proposed for systematic and automatic testing of agile software product lines (SPLs). The software systems' behavior is represented by using the *event sequence graphs* (ESG) under consideration of user actions, i.e., events and features of a software product line are represented *featured event sequence graphs* (FESG).

1.1. Motivation

For majority of the projects, the testing costs range from 20 to 50 per cent of the comprehensive system development costs (Mistrik et al. 2014). Moreover, these costs can reach up to 80% easily for safety-critical systems (Jones 1991). One study on the ground of data gathered from Rolls-Royce shows that nearly 52 per cent of the overall development activities of a system are on testing activities (Mistrik et al. 2014; Nolan et al. 2011). It has been argued that software product line paradigm reduces the development costs but not necessarily the testing costs, in the aforementioned study. Also, it has been reported that, 72% of overall product development activities are on validation and verification in software product line context and this percentage can reach 90 in theory, because of high reuse (Nolan et al. 2011). It has been concluded that this percentage rises not because the testing effort has increased but because the development effort has decreased thanks to the reusable assets and this raises the testing effort compared to the overall effort, nevertheless, testability becomes more important for software product lines (Nolan et al. 2011). Hence, automatic and systematic testing methods become a must-have in software product line development.

Automation could be applied to test execution and test design. To implement test design automation, a formal model from requirements is derived and a test case set based on that model are generated which leads us to model-based testing. A high potential to exploit reuse opportunities for SPL testing is provided by MBT (Olimpiew 2008; Tuglular, Beyazıt, and Öztürk 2019). Various model MBT techniques has been proposed for SPLs which are explained in
. Nevertheless, most of the current SPL testing approaches cannot remedy the following two deficiencies, potentially (Lochau et al. 2016). Firstly, for some approaches, one overlaying specification with all possible variants of the software product line is required and because of computational overhead, this becomes unwieldy for large-scale software product lines (Tuglular, Beyazıt, and Öztürk 2019; Lochau et al. 2016; Czarnecki and Antkiewicz 2005). Secondly, instead of considering behavioral impact of variations, the emphasis is on structural and syntactical variability (Apel and Hutchins 2010). Therefore, there is no systematic propagation of behavioral properties from one product form (variant) to another (Lochau et al. 2016).

To decrease the test cost of SPLs with the aid of automatic MBT techniques, this thesis which proposes a test generation approach for agile SPLs and remedies the potential deficiencies of model-based techniques is presented.

1.2. Major Contributions of the Thesis

This thesis aims to develop a model-based test generation approach for agile software product lines and addresses the following questions:

1. How to build variable testing models in order to explicitly represent the variability of behaviors in SPLs?
2. How can the exertion of test generation and thus the test cost be decreased while the number of product variants and their complexity continuously increasing?
3. How can the existing test cases be reused in extension of a product with new features?

To answer the first question, this thesis utilizes Featured Event Sequence Graphs (FESGs) as variable testing models which are used to explicitly represent the variability

of behaviors in SPLs. The core of the SPL and each feature are modelled as partial or full ESGs which are independent and named as c-ESG and f-ESG, respectively. Then, the combination of core ESG and selected feature ESGs represent the behavior of the corresponding product.

To answer the second question, this thesis employs a *full test sequence composition approach* which exploits the FESGs. The implementation of this approach resembles as divide-and-conquer strategy in the sense that the core of the SPL and existing features of each SPL variant are modelled separately. Prepared test models together constitute the FESG of a product. Comparing to the single ESG model of each product variant which is called as full-ESG model, the FESG constituents are quite simple models in terms of number of vertices and edges. This because, a full-ESG model includes both the core's and selected features' behaviors and it is a more complex model. Therefore, test generation is faster and traceability and maintainability of these models are easier.

One can simply prepare test models of the core and all the features within the SPL and then combine these models for different products' test generation. New features could be added easily, or existing features could be updated without interfering other features. In the test sequence composition approach, existing test models are reused and each time a new variant is obtained their test sequence are composed from scratch. Even though the test composition from scratch, this approach is more efficient than the traditional test generation approach of full-ESG models.

To answer the third question, this thesis utilizes another approach called *incremental test sequence composition approach.* This approach is an enhanced version of test sequence composition approach and it also exploits the FESGs. The aim of this approach is reusing both the test models and the existing test cases. Thus, each time a new feature (or features) is added to an existing variant, the test model of the new feature is prepared, and its corresponding test sequences are generated. The existing test cases of the product and the new test sequences are composed in order to generate the test cases of a new product variant. Therefore, the test models and their corresponding test cases are reusable and configurable for different variants within the SPL.

Additionally, another contribution of this thesis is a directed graph which is called Feature-Based Incremental Product Graph. The Feature-based Incremental Product Graph holds FESGs in its vertices and feature sets in its edges. The structure of the Feature-based Incremental Product Graph allows to automatically run the

incremental test sequence composition approach very easily and automatically. The Feature-based Incremental Product Graph is used to experiment on SPL product variants by using incremental test sequence composition approach. This graph is used to validate the configurations of these product variants. In order to achieve these operations two different algorithms are proposed in which the graph is traversed via Breadth First Traversal.

1.3. Outline of Thesis

This thesis is organized according to the following. The next chapter provides an overview of the literature. CHAPTER 3 provides context on software product lines, feature modeling, event sequence graphs and related algorithms. CHAPTER 4 includes the detailed explanation of the model-based full test sequence composition approach. In CHAPTER 5, the incremental test sequence composition approach is proposed which is an improved version of test sequence composition approach. The feature-based incremental product graph is introduced in CHAPTER 6. Case study of this thesis work is presented in CHAPTER 7. Finally, CHAPTER 8 provides final comments and future work.

CHAPTER 2

RELATED WORK

In model-based testing (MBT), the specifications of the software to be tested are defined by a model in accordance with the specification. These models are usually graph-based. Examples of these models can be given as finite state machines (Chow 1978; Fujiwara et al. 1991), petri nets (Xu 2011) and event sequence graphs (Belli 2001). An algorithm for test generation that takes this model as input creates a test set using a test selection criterion (Belli 2001).

Whittaker (Whittaker 1997) suggested that models used in MBT could be decomposed or combined, and showed that test cases can be generated from partial models or model parts and also from the combined large model, and then compared the results. El-Far and Whittaker (El-Far and Whittaker 2002) examined the issue of test generation from hierarchical models. They showed how the main finite state machine can be expanded by replacing a state with a finite state machine. They made the definition of a hierarchical finite state machine and discussed test generation from hierarchical finite state machines. Belli et al. (Belli, Guler, and Linschulte 2011) proposed a method for test generation from hierarchical models that use event sequence graphs. However, these ideas have not been applied to MBT of SPLs.

Scenario based TEst case Derivations (ScenTED) was one of the initially proposed approaches in model based testing of software product lines that exploits reuse of the core properties and components in order to reuse of the test cases (Reuys et al. 2005). Customizable Activity diagrams, Decision tables and Test specifications (CADeT) method was also another significant research on model based testing of software product lines (Olimpiew 2008) which generates feature-based test suites by employing UML use case and activity diagrams. In order to model variability and generate test cases, decision tables are used. A method named as 150% finite state machines, which employs an overlaying model for the software product line under consideration and includes a coverage-driven method for SPL testing was proposed by Cichos et al. (Cichos et al. 2011). Another model-based testing method for software

product lines is model-checking. Kishi and Noda (Kishi and Noda 2006) suggested modelling the design as a finite machine and checking if the product has the determined features indeed, by using model checking. In order to apply model checking to SPLs several approaches has been suggested (Gruler, Leucker, and Scheidemann 2008; Classen 2011; Classen et al. 2011) .

Additionally, Olimpiew and Gomaa (Olimpiew and Gomaa 2005) suggested the Product Line UML based Software engineering (PLUS) method that maps the software product line models to functional test cases to generate and select the functional tests automatically for the corresponding SPL applications. PLUS has outlined that how to build specifications, analyzes and the design for an SPL. A feature model and a use case model are included in the requirement models; a class, state chart and object interaction model are included by an analysis models; component-based software architecture models are included by design models. All of the requirement, analysis and design models are based on UML 2.0 notation. Another approach for testing in SPLs was proposed by Lamancha et al. (Lamancha, Usaola, and de Guzman 2009) which is based on familiar standards for instance UML 2.0, the UML testing profile and QVT Language in which the traceability is preserved between different levels of abstraction, as well as between levels of domain and product engineering.

Moreover, Geppert et al. (Geppert 2004) proposed a method for SPL testing that employs a decision model to guide feature selection during derivation of an application, test selection and customization. In this method, test parameters are contained by generic test templates and corresponds to points of variation or groups of feature in an SPL. To assign a value to the test parameter, a feature of the SPL is selected. Gebizli and Sözer (Gebizli and Sözer 2016) suggested a method for product family testing which models the system functionality by using Markov chains in which the behavioral variability is depicted via a feature model. The Markov chains are employed to capture usage scenarios for products within the SPL and testing is performed on an industrial case study which shows that testing even a small number of products redeem the cost of SPL engineering adoption.

Furthermore, Oster et al. (Oster et al. 2011) proposed a tool chain that realizes Model-based Software Product Line Testing (MoSo-PoLiTe) concept which combines model-based and combinatorial testing of SPLs. Variant management tool pure::variants for Rational Rhapsody and the Rational Rhapsody Plugin ATG underlies the tool chain

for test case generation and a plugin for pure::variants is implemented that realizes the pairwise algorithm of the MoSo-PoLiTe concept.

To reduce redundancies in SPL testing, regression-based and subset selection heuristics are two research directions (Lochau et al. 2016). Since this thesis is on regression-based testing of SPLs, the related literature is covered. Uzuncaova et al. (Uzuncaova, Khurshid, and Batory 2010) suggested a method using SAT-based analysis to produce test inputs for each software product line variant. Incremental enhancement of test cases for a specific variant is enabled by their approach. Neto et al. (Neto et al. 2010) suggested a method that decreases the effort of testing by reusing test suites which requires exploiting similarities in the architecture of the SPL. Additionally, an approach for reuse of test artifacts between product variants was proposed by Lochau et al. (Lochau et al. 2012). All these studies are based on finite state machines (FSMs), without explicit mapping of features with FSMs. In other words, how states, transitions, etc. represent a single feature and how states and transitions representing a single feature are connected to a product's FSM are not depicted. In practical terms, these representations are significant for the techniques to be used by industry for traceability reasons, all features are represented distinctly and the way how to connect a single feature to a product is stated clearly. Another novelty of the approaches proposed in this thesis is that starting with a base product and coming up to other incrementally are not necessary.

CHAPTER 3

FUNDAMENTALS

This chapter explains the fundamental notions and algorithms that construct the background of this study. Firstly, the software product line paradigm and feature modeling are introduced. Secondly, the Event Sequence Graphs (ESGs), which are the building blocks of this study are presented. The ESG test generation algorithm is explained in depth. Finally, the ESG transformation algorithm is introduced.

3.1. Software Product Line and Feature Modeling

A set of products that have common features with varying additional features which are related to each other in a specific domain constitute a software product line (SPL) (Withey 1996). Software product line paradigm enables systematic software asset reusing therefore, it promises faster development, automatic testing and increased product quality (Devroey et al. 2012).

In software product lines, there are different variations of a software equipped with different features to appeal to different target audiences. For example, a product set of standard, professional and enterprise versions of a software represent a software product line. The number of elements of the product set that represents the software product line can be quite high. Every new feature added to the mutual features at the core/base of the software increases the complexity of the software and thereby reinforces its predisposition to failure.

The features in a software product line shape the products according to the selection of the stakeholders and enable us to distinguish one product from another (Czarnecki, Krysztof and Eisenecker, Ulrich 2000). The feature selection is done through feature diagrams. A feature diagram is given in Figure 3.1 which is used as a running example in this study and it belongs to a bank account software product line.

This diagram enables us to develop related bank account products such as one allowing cancellation of deposit and/or withdraw operations, one allowing extra money, one allowing daily limit etc.

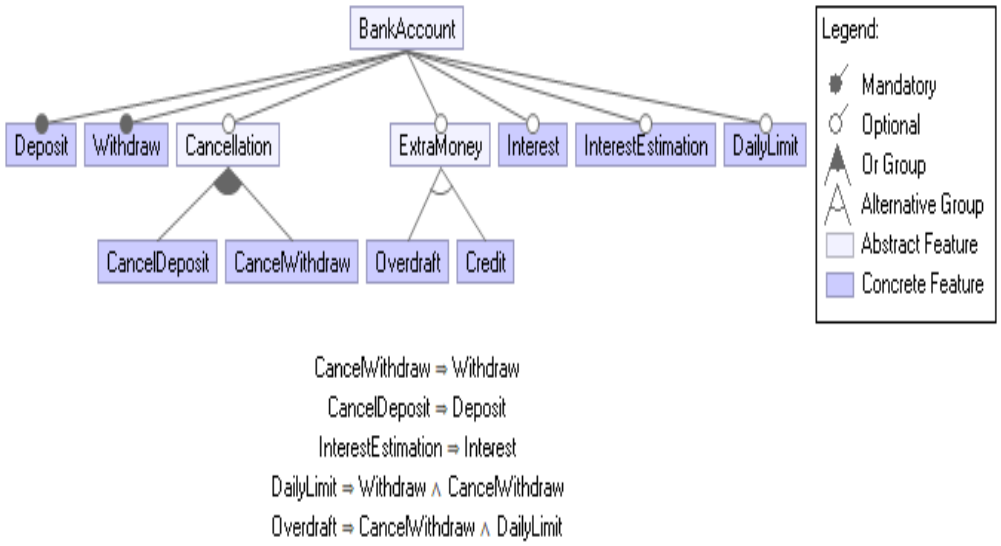

Figure 3.1. Bank account SPL feature diagram

The configuration options and the dependencies are represented by feature diagrams(Tuglular, Beyazıt, and Öztürk 2019), which are originally proposed by Kang et al (Kang n.d.). In feature diagrams, the SPL is represented by the root and the features of the SPL are represented by the leaves. The features which are either mandatory or optional, could be combined with XOR or OR relationships (Tuglular, Beyazıt, and Öztürk 2019). The features that are in OR relationship could be included in a product in different combinations (Tuglular, Beyazıt, and Öztürk 2019). However, the ones that are in XOR relationship could be included exclusively, i.e., only one of them could be included in a certain product (Tuglular, Beyazıt, and Öztürk 2019). The *require* relationship in feature diagrams, denotes the implication between two features (Tuglular, Beyazıt, and Öztürk 2019). The *exclude* relationship denotes the exclusion between two features (Tuglular, Beyazıt, and Öztürk 2019). In feature diagrams, there are also abstract features which are employed to group features and concrete features which are actually corresponding real features in an SPL.

The feature diagram shown in Figure 3.1, a bank account SPL is shown with mandatory features which are Deposit and Withdraw. In this diagram, the abstract feature Cancellation is grouping CancelDeposit and CancelWithdraw features with OR relationship and the abstract feature ExtraMoney is grouping Overdraft and Credit features with XOR relationship. The features Cancellation, ExtraMoney, Interest, InterestEstimation and DailyLimit are optional for this SPL. Furthermore, the implications that are written below the feature diagram correspond to require relationship where InterestEstimation feature requires Interest feature, DailyLimit feature requires Withdraw and CancelWithdraw features, and, Overdraft feature requires CancelWithdraw and DailyLimit features in a product configuration. The bank account SPL example is modified from the online software product line catalog SPL2go ("SPL2go" n.d.), in which the source code and the feature model of the running example is publicly available.

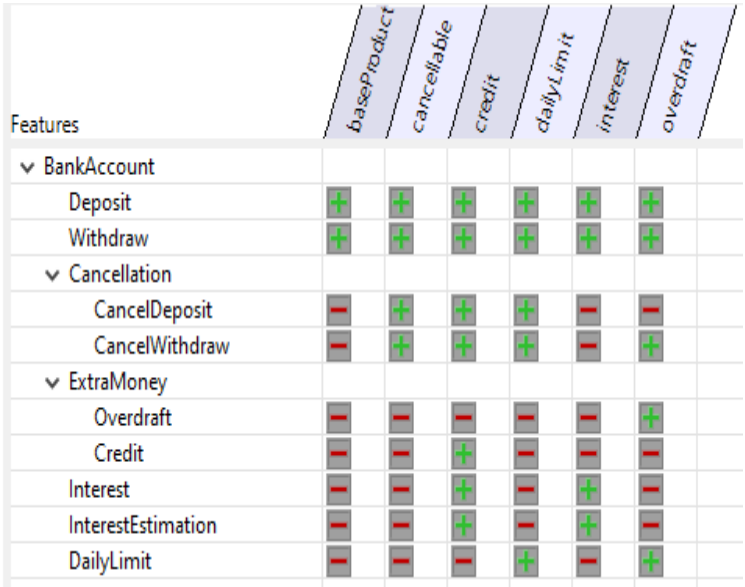

Figure 3.2. Product matrix of the bank account SPL

Formal presentations of feature diagrams which are feature models are generally user-centric (Tuglular, Beyazıt, and Öztürk 2019). The definitions of the feature model and the product configuration are given in following.

Definition 3.1: Let *B* indicates the Boolean values domain by *B = {false, true}*. Let *F* be a finite Boolean variables (features) set. A *feature model (FM) fm:*

(F→B)→B is represented as a propositional formula over the set *F* (Lochau et al. 2016).

Definition 3.2: An assignment of Boolean values to features such that *fm(p)=true* holds is a *product configuration (PC) p: F→B* is (Lochau et al. 2016; Kang et al. 1990).

Product diagrams are visual representations of product configurations. Product matrix of a SPL shows available products for that SPL. Product matrix of the bank account SPL is given in Figure 3.2.

3.2. Event Sequence Graph

An Event Sequence Graph which is abbreviated as ESG is a technique that is used for behavioral-modelling of systems (Belli et al. 2005). Both the expected (i.e. correct) and un expected (i.e. exceptional) behavior of the system could be represented by using ESGs from the system user's point of view (Belli et al. 2005; Belli and Budnik 2005). ESGs focus on the externally observable behavior of computer-based systems by means of discrete event-based models (Belli et al. 2005).

The interactions between the user events, the environmental actions, and the system responses are modelled by exploiting the event-based structure of ESGs (Belli et al. 2005). The complete set of interactions is obtained in terms of an ESG set, where each ESG stands for a possibly infinite set of event sequences (Belli et al. 2005). This event sequences set is used in order to test both the desired and the undesired behavior of a computer-based system. The following event sequence graph's definitions are used throughout this study. An example event sequence graph is depicted in Figure 3.3.

Definition 3.3: An Event Sequence Graph $ESG = (V, E)$ is a directed graph where $V \neq \emptyset$ is a finite set of nodes (vertices) and $E \subseteq V \times V$ is a finite set of arcs (edges), and $\Xi, \Gamma \subseteq V$ finite sets of distinguished vertices with $\xi \in \Xi, \gamma \in \Gamma$, called entry nodes and exit vertices, respectively, wherein $\forall\, v \in V$ there is at least one sequence of vertices $< \xi, v_0, \ldots, v_k, \gamma >$ from each $\xi \in \Xi$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for i = 0, …, k-1 and $v \neq \xi, \gamma$ (Belli, Budnik, and White 2006).

The start (entry) and finish (exit) vertices of an ESG are marked by applying the following convention: all $\xi \in \Xi$ are preceded by a pseudo vertex '$[$' $\notin V$ and all

$, \gamma \in \Gamma$ are followed by another pseudo vertex '$]$' $\notin V$ (Belli, Budnik, and White 2006). The start (entry) and finish (exit) vertices which are demonstrated by '$[$' and '$]$' respectively, are called pseudo vertices and they are not included in V (Belli and Budnik 2005; Belli, Budnik, and White 2006). The pseudo vertices are not included also in event sequences.

Example 3.1: For the ESG given in Figure 3.3, $\boldsymbol{V}$ = {*get balance*, *select deposit*, *enter deposit amount*, *put money*, *select withdraw*, *enter withdraw amount*, *take money*}, Ξ={*get balance*, *select deposit*, *select withdraw*}, Γ={*get balance*, *put money*, *take money*} and $\boldsymbol{E}$ ={(*get balance*, *select deposit*), (*select deposit*, *enter deposit amount*), (*enter deposit amount*, *put money*), (*get balance*, *select withdraw*), (*select withdraw*, *enter withdraw amount*), (*enter withdraw amount*, *take money*), (*put money*, *get balance*), (*take money*, *get balance*)}. $\boldsymbol{E}$ does not contain the edges from pseudo start vertex '[', and to pseudo finish vertex ']'.

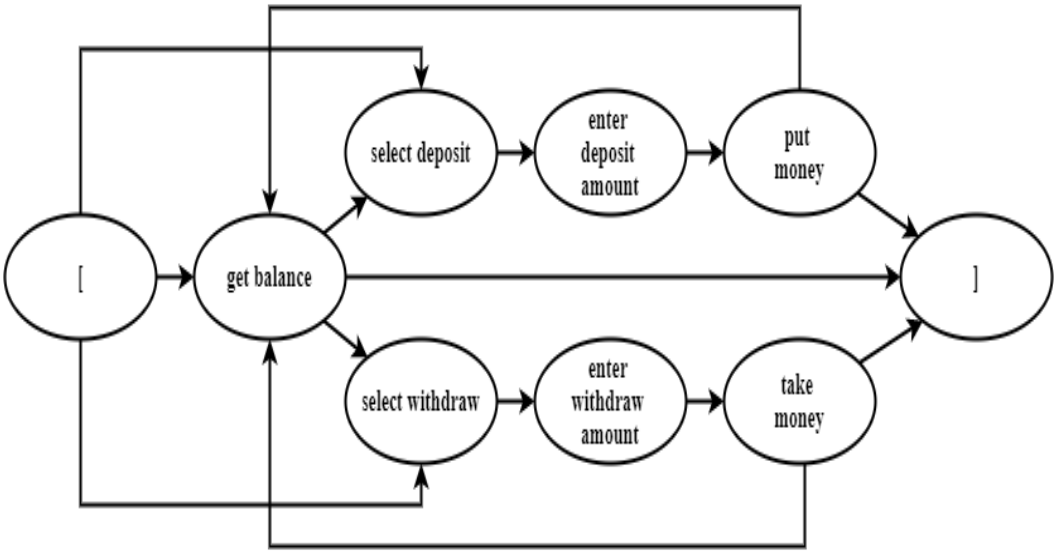

Figure 3.3. ESG of bank account SPL-base product

Definition 3.4: Let *V,E* be defined as in Definition 3.3. Any sequence of vertices $< \boldsymbol{v_0}, \ \ldots \ , \ \boldsymbol{v_k} >$ is called an event sequence (ES) if $(\boldsymbol{v_i}, \boldsymbol{v_{i+1}}) \in \boldsymbol{E}$, for i=0, …, k-1 (Belli and Budnik 2005; Belli, Budnik, and White 2006; Belli and Budnik 2004).

Example 3.2: *select deposit - enter deposit amount - put money* is an ES of length 3 of the ESG in which given in Figure 3.3.

In order to specify the start vertex and finish vertex of an ES $\propto (initial)$ and $\omega(end)$ are functions are used, i.e., $\propto (ES) = v_0$, $\omega(ES) = v_k$ (Belli and Budnik 2005; Belli, Budnik, and White 2006; Belli and Budnik 2004). The *successors* set of $\forall\, v \in V$ is denoted by $N^+(v)$ and the *predecessors* set of $\forall\, v \in V$ is denoted by $N^-(v)$ (Belli and Budnik 2005; Belli, Budnik, and White 2006; Belli and Budnik 2004). The number of vertices of an ES is determined by the function $l(length)$ (Belli and Budnik 2005; Belli, Budnik, and White 2006; Belli and Budnik 2004). If $l(ES) = 1$ then $ES = < v_i >$ is an ES of length one (1) (Belli and Budnik 2005; Belli, Budnik, and White 2006; Belli and Budnik 2004). Each edge of ESG or an $ES = < v_i, v_k >$ of length two (2) represent an ***event pair (EP)*** (Belli and Budnik 2005; Belli, Budnik, and White 2006; Belli and Budnik 2004)**.**

Definition 3.5: An ES is called a complete ES (Complete Event Sequence, CES), if $\propto (\boldsymbol{ES}) = \boldsymbol{\xi} \in \boldsymbol{\Xi}$ is the entry and $\boldsymbol{\omega(ES)} = \boldsymbol{\gamma}$ is the exit (Belli and Budnik 2005; Belli, Budnik, and White 2006; Belli and Budnik 2004).

Example 3.3: The ESG demonstrated in Figure 3.3 has a CES *get balance - select withdraw - enter withdraw amount -take money* which describes a walk from the start of the ESG to its finish.

A *test sequence*, i.e., test case, CES, of the ESG is of the shape "(initial) user inputs → (interim) system responses → … → (final) system response"(Belli and Budnik 2007).

A *legal walk* is represented by a CES which traverses the ESG from its start to its finish (Belli et al. 2005). A *complete legal walk* or an *entire walk* contains each event pair (EP) in the corresponding ESG at least once (Belli et al. 2005; Belli and Budnik 2005). A complete legal walk or a legal entire walk is *minimal* if its length cannot be decreased without changing it into an incomplete legal walk (Belli et al. 2005). If a minimal legal walk contains every EP exactly once then it is considered ideal (Belli et al. 2005). CESs are considered as legal walks of ESGs (Belli et al. 2005). However, constructing an entire or an ideal walk is not always feasible (Belli et al. 2005).

3.3. Test Generation from Event Sequence Graphs

Test generation consists of extracting the complete event sequences (CESs) from ESGs. In order to extract the CESs of an ESG, one is required to solve the Chinese Postman Problem, which is abbreviated as CPP (Belli et al. 2005). Solving CPP means finding the Euler cycles on the graph, i.e., starting from and returning back to the same vertex by visiting each edge exactly once (Belli, Guler, and Linschulte 2011; Tuglular, Belli, and Linschulte 2016).

The solution to the CPP on ESGs is the set of CESs of the corresponding ESG(Belli, Guler, and Linschulte 2011). The generation of CESs is anticipated to have a lower degree complexity comparing to solution of CPP, since the edges of ESG are not weighted (Belli et al. 2005).

In order to derive the solution of CPP, the given ESG is required to be an Eulerian graph (Belli, Guler, and Linschulte 2011). If a graph is *strongly connected* and *balanced*, i.e., $\forall\, v \in V$ has equal *in degree* and *out degree* (Belli, Guler, and Linschulte 2011). An Eulerian graph has a cycle which goes exactly once across each edge and returns to the starting vertex (Belli, Guler, and Linschulte 2011).

Figure 3.4 illustrates the transformation of an ESG given in Figure 3.3 into a strongly connected graph. Strongly connected means that there is a path between each vertex pair (Belli et al. 2005). A backward edge symbolized as a dashed arrow is added from exit vertex to entry vertex in order to transform the given ESG into a strongly connected ESG(Belli et al. 2005). All ESGs have a pseudo start vertex which reaches to each vertex in V by a path. Also, each vertex in V reaches to a pseudo end vertex by a path in all ESGs. Therefore, adding an edge from pseudo finish vertex to pseudo start vertex makes ESGs strongly connected.

Definition 3.6: The number of edges going into a vertex v is the in degree 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 (Belli, Guler, and Linschulte 2011).

The vertex labels of Figure 3.5 indicates the balance values of the vertices of ESG given in Figure 3.4. The number of additional edges that are used to balance each vertex will be determined by exploiting these balance values (Belli et al. 2005). Definition 3.6 concludes that a directed graph is Eulerian if each of its vertices are balanced, i.e., $\delta(v) = 0 \in V$ [20].

In order to balance the ESG, first, a positive degree vertex partition which is shown below as A and a negative degree vertex partition which is shown below as B must be determined (Belli, Guler, and Linschulte 2011). Then, the vertices of these vertex partitions must be matched by taking the path lengths between them into consideration (Belli, Guler, and Linschulte 2011).

$$A = \{ v_i \mid i \in \{1, \dots, \delta(v)\} \wedge \delta(v) > 0 \}$$

$$B = \{ v_i \mid i \in \{1, \dots, -\delta(v)\} \wedge \delta(v) < 0 \}$$

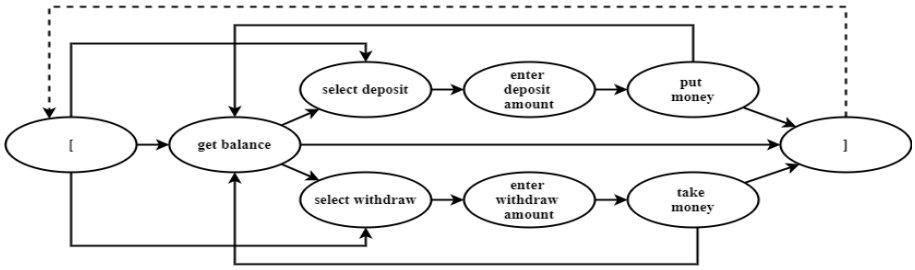

Figure 3.4. Strongly connected ESG of bank account SPL - *base product* (derived from Figure 3.3)

Figure 3.5, which is based on Figure 3.4, demonstrates the degree belonging to the each vertex in Figure 3.4 with set $A = \{],], select\ deposit, select\ withdraw \}$ and set $B = \{ [, [, \ put\ money, take\ money \}$ (Belli, Guler, and Linschulte 2011). $'['$ and $']'$ occur twice in A and B, because their degree is -2 and, $+2$, respectively.

To balance the given graph, each element of set A is strictly assigned to one element of set B until no unassigned element in either set is left and no other assignment with a lower number of edges to be added up to the assignment is left (Belli, Guler, and Linschulte 2011). This leads to assignment problem which answers the question of how *n* items (agents) are assigned to *n* other items (tasks) with varying costs, relying on the agent-task assignment (Belli, Guler, and Linschulte 2011). It is necessary to perform all

tasks by assigning exactly one agent to each task so that the total cost of the assignment is minimum (Belli, Guler, and Linschulte 2011).

In this study, The Hungarian Matching Algorithm (Burkard, Dell'Amico, and Martello 2012) so-called Kuhn-Munkres Algorithm, which solves assignment problem is applied to match the two partitions A and B's vertices and the ESG is made balanced by using this matching. The Hungarian Matching Algorithm is one of the fastest methods for solving the assignment problem and it provides a solution in $O(n^3)$ time (Belli, Guler, and Linschulte 2011).

The balanced and strongly connected ESG is demonstrated in Figure 3.6 based on the ESG shown in Figure 3.3. The resulting paths of the matchings $'] \rightarrow [', '] \rightarrow [', select\ deposit \rightarrow put\ money'$ and $'select\ withdraw \rightarrow take\ money'$ are added to the corresponding ESG and the additional paths are shown with dashes arrows.

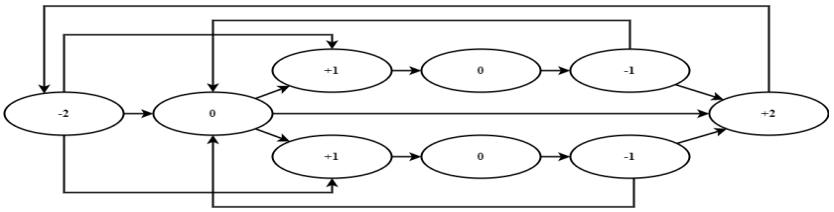

Figure 3.5. Vertex degrees of the bank account SPL - *base product* (derived from Figure 3.4)

Since the ESG becomes an Eulerian graph, the problem is transformed into the construction of an Euler cycle from this graph (Belli et al. 2005). Each separate test case is identified by each occurrence of the ES = *] [* in the Euler cycle (Belli and Budnik 2005). The number of walks or CESs are indicated by the number of backward edge which is contained by the Euler cycle (Belli et al. 2005).

This study solves CPP by finding an Euler cycle on an ESG by using Hierholzer Algorithm (Hierholzer and Wiener 1873) and dividing the cycle by the occurrence of ES = *] [*. Each divided part corresponds to a CES and represents one of the test cases.

On the basis of the Eulerian graph given in Figure 3.6, the resulting Eulerian cycle is given as follows:

enter withdraw amount – take money – get balance][select withdraw – enter withdraw amount – take money][select deposit – enter deposit amount – put money – get balance – select deposit – enter deposit amount – put money][get balance – select withdraw – enter withdraw amount

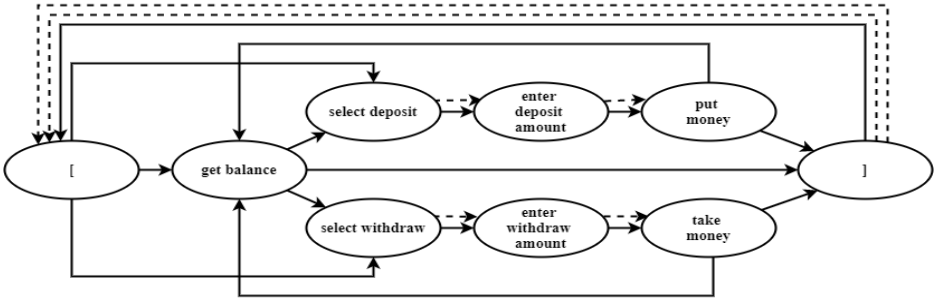

Figure 3.6. Strongly connected and balanced ESG of bank account SPL *base product*

The last vertex "*enter withdraw amount*" of the output cycle does not play a role in the desired test sequences, therefore it could be deleted (Linschulte 2013). Whenever this cycle is split up between each occurrence of "*][*", the resulting CESs are as follows:

CES1: *select withdraw – enter withdraw amount – take money*

CES2: *select deposit – enter deposit amount – put money – get balance – select deposit – enter deposit amount – put money*

CES3: *get balance – select withdraw – enter withdraw amount – take money – get balance*

Algorithm 3.1 achieves test sequence generation from an ESG.

Algorithm 3.1 Sequence Generation

Input: G = (V, E, Ξ, Γ) – an ESG

k – integer coverage parameter for k-sequence coverage

Output: T – a set of complete sequences for G achieving k-sequence coverage

G_k = transformESG(G) // See Algorithm 3.2

$G_{k\text{-stronglyConnected}}$ = add an edge from '] ' to '['

$G_{k\text{-stronglyConnected-balanced}}$ = add necessary paths untill degree of each $V \in G_{k\text{-stronglyConnected}}$ is 0.

T = generate Euler cycles from $G_{k\text{-stronglyConnected-balanced}}$

To conclude, the test generation algorithm is required to solve the Chinese Postman Problem on the ESGs. The solution set of CPP contains each CES of the corresponding ESG and it covers each event pair (EP) of the ESG.

3.4. ESG Transformation

ESG transformation algorithm is proposed in Algorithm 3.2 to cover event triples, quadruples etc., i.e. test sequences of length k where k could be 3,4 etc. When an ESG is transformed, it becomes to have event sequences in each of its nodes. The length of the ES, determines the value of k. If it is not transformed, only the event pairs could be covered, however k-length event sequences could help to connect the testing process thoroughly and reveal possible extra faults.
Suppose an ESG has V= {x, a, b, c, z}, E= {(x, a), (a, b), (a, c), (b, c), (c, b), (b, z)} before transformation. When it is transformed once, each node becomes to have an ES of length 2 in it. After transformation, V= {xa, ab, ac, bc, cb, bz} and E= {(xa, ab), (xa, ac), (ab, bc), (ab, bz), (ac, cb), (bc, cb), (cb, bc), (cb, bz)}. When the edges of transformed ESG are covered, event triples which are xab, xac, abc, abz, acb, bcb, cbc and cbz are covered in test sequences. We cover k events by covering a single edge, when the length of the ES in a vertex is k-1. For example, the transformed ESG has a length-2 ES in each of its nodes, event triples are covered, i.e. k is 3.

Algorithm 3.2 convert the input ESG to a sequence ESG, i.e., an ESG that contains vertices of a special type called as sequence vertex by calling "convertToOneESG" procedure. Each sequence vertex includes event sequences (ESs) of length k of the Algorithm 3.2**.** Note that, the ESs of sequence vertices are acted as an event but they are not actual events, indeed. In "transform" procedure of Algorithm 3.2**,** the sequence ESG version of input ESG is transformed with itself k times until all the edges to cover ESs of length k are added.

Example 3.4: Consider the ESG that is given in Figure 3.3. *V = {get balance, select deposit, enter deposit amount, put money, select withdraw, enter withdraw amount, take money}* before transformation. When it is transformed once, each node becomes to have an ES of length 2 in it. After transformation, the vertex set becomes *V*

= {get balance: select deposit, select deposit: enter deposit amount, enter deposit amount: put money, put money: get balance, get balance: select withdraw, select withdraw: enter withdraw amount, enter withdraw amount: take money, take money: get balance}. Therefore, an event triple could be obtained by covering a single edge that belongs to the one-time transformed ESG. For example, when the edge (get balance*:* select deposit, select deposit*:* enter deposit amount*)* is covered the event triple *get balance: select deposit: enter deposit amount* is obtained.

Algorithm 3.2 Transformation of ESG

Input: G = (V, E, Ξ, Γ) – an ESG to be transformed
k – integer transformation parameter
Output: G_k – transformed ESG
G_{oneESG} = convertToOneESG(G)
$G_k = G_{oneESG}$
for n=1 to k incrementing by 1 **do**
G_k = transform (G_k , G_{oneESG})
endfor

In this chapter, the notions software product line, the feature modelling and the event sequence graphs are explained. Also, the algorithms test generation of EGSs and the transformation of EGSs are given. These notions and algorithms provide a basis for the next chapter, which will present a model-based test generation approaches for SPLs.

CHAPTER 4

FULL TEST SEQUENCE COMPOSITION FROM FEATURED EVENT SEQUENCE GRAPHS

The full test sequence composition approach is explained in this chapter. The full test sequence composition approach is a model-based approach which introduces Featured Event Sequence Graphs (FESG). FESGs are variable testing models which are used to express the variability of SPL products' behavior explicitly (Tuglular, Beyazıt, and Öztürk 2019). In this approach, the core feature and each separate feature are modelled as ESGs into a FESG. Afterwards, the behavior of a specific product is obtained from the FESG that results from the combination of the core ESG and the feature ESGs. The purpose of this approach is to reuse the existing test models and also the test cases through composition.

4.1. Featured Event Sequence Graph

The notion Featured Event Sequence Graph is proposed in order to fulfill the need of associating a new feature to an existing product configuration so that the corresponding test model can be updated accordingly (Tuglular, Beyazıt, and Öztürk 2019). Featured Event Sequence Graphs are abbreviated as FESGs and FESG is an extension of ESG(Tuglular, Beyazıt, and Öztürk 2019).

Definition 4.1: A featured event sequence graph (FESG) is (F, c, Ξ, Γ) where F = $\{f_1, f_2, \ldots, f_N\} \neq \emptyset$ is a finite set of EGSs called feature ESGs (f-ESGs) with each f_i = (V_i, E_i, Ξ_i, Γ_i). c ∈ F is a special f-ESG called core ESG (c-ESG). Ξ, Γ ⊆ $\bigcup_{i=1}^{N} V_i$ are finite sets of excided vertices called entry nodes (start events) and exit nodes (finish events), respectively.

Definition 4.1 suggest that an FESG is a set of feature-ESGs (f-ESGs) one of which is designated as core-ESG (c-ESG). These f-ESGs forms a behavioral model for a product/system when considered together. An f-ESG except for the c-ESG contains one or more nodes of other f-ESGs, which signifies that it is connected to these other f-ESGs. A c-ESG, however, does not contain any node of another f-ESG. Also, two f-ESGs cannot contain nodes from each other.

Definition 4.2: A core-ESG (c-ESG) contains the events that represent the core behavior of the SPL These events are called core events and they are not necessarily connected to the c-ESG.

Figure 4.1. c-ESG of bank account SPL

The c-ESG of the running example is demonstrated in Figure 4.1. The core behavior of the SPL is represented as c-ESG (Tuglular, Beyazıt, and Öztürk 2019). Selected features' behaviors are represented as feature ESGs which are defined in Definition 4.3. f-ESGs are combined with c-ESG in order to obtain a specific product's behavior within SPL (Tuglular, Beyazıt, and Öztürk 2019).

Definition 4.3: A particular feature in the feature model is represented by a feature-ESG (f-ESG). Unlike ordinary ESG vertices and c-ESG vertices, a f-ESG contains vertices associated with the points of variability, which are named as connection events. Connection events are in fact occurrences in other c-ESGs or f-ESGs. They are named as (ESG, Event) pairs (Tuglular, Beyazıt, and Öztürk 2019).

The constraints are defined in feature diagram (Figure 3.1.) due to the existence of connection events. For example, the *daily limit* f-ESG requires both *withdraw* and *cancelWithdraw* features because of its connections to these f-ESGs. Note that, whenever an f-ESG A is connected to another f-ESG B, there should be a constraint that is *A requires B*, so that, A and B features are added to the product configurations together. Note that, even though the *withdraw* and *deposit* features are mandatory, the constraints that implies these features are added because if these features become

optional, the *cancelDeposit* and *cancelWithraw* will still be requiring them.

The bank account SPL running example is constituted from the c-ESG (Figure 4.1) and nine f-ESGs. The behavior of *withdraw* feature is shown in Figure 4.2. In the given f-ESG, the connection points are placed right before the pseudo entry and the pseudo exit vertices in which "[,], get balance" of c-ESG are events to be connected.

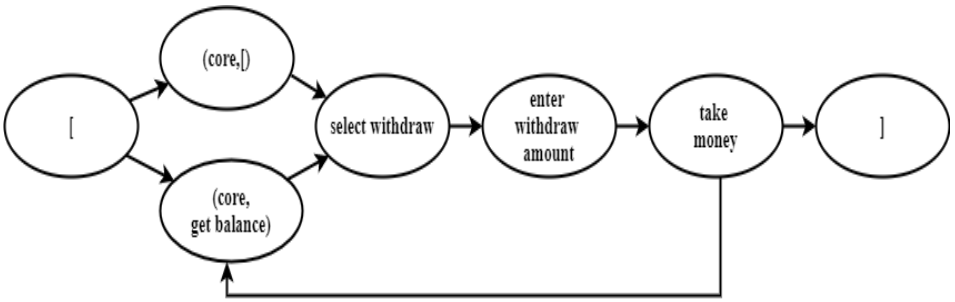

Figure 4.2. *withdraw* f-ESG of bank account SPL

Connection events are added in one-way direction from bottom to top which means that a c-ESG does not have connection event to its feature events (Tuglular, Beyazıt, and Öztürk 2019). This is the selected convention for this study in order to take advantage of reuse (Tuglular, Beyazıt, and Öztürk 2019). Nevertheless, each f-ESG contains variability points, i.e., the information of connection events (Tuglular, Beyazıt, and Öztürk 2019). In order to constitute a FESG for a specific product configuration, a c-ESG and a f-ESG set are composed (Tuglular, Beyazıt, and Öztürk 2019).

4.2. Discussion

In this thesis work, one novelty is Featured Event Sequence Graphs. Since these FESGs are extensions of EGS models, the reasons behind the preference of ESGs are explained in this section. Afterwards, the necessity for extending ESGs and introducing FESGs are mentioned.

The first reason is that the ESGs are event-based models. As it is written before in CHAPTER 2, Chow (Chow 1978) described test models based on finite state

machines in 1978. The finite state machine-based test models and the Petri nets (Xu 2011) which are variation of finite state machines are all state-based models. Using ESGs as test models provide a better reflection of the interactive human-machine systems' behavior. In event-based modelling of the ESGs, both the desirable and the undesirable behavior of the reactive systems are modelled, and this brings a complementary view on the system which enables modelling potential user faults.

The second reason is that ESGs are graphs and for graphs, there exist algorithms to get minimal set of test sequences such as Chinese Postman Problem or Travelling Salesman Problem. The efficiency is neglected by generating large test sets including redundant and unnecessary test sequences and, these large test sets do not always result in a better test coverage (Linschulte 2013).

The third reason is that model refinement of ESG by hierarchical structures exists. These hierarchical refined models help to apply the principle of "divide and conquer" and allows modularization. Furthermore, there is no other approach to use hierarchical structures to generate optimized test sets (Linschulte 2013).

ESGs are preferred in model-based testing because of the advantages outlined above and extended for the following reasons. First of all, Featured Event Sequence Graphs are also event-based models and they are preferable for interactive human-machine systems. Second, FESGs contains c-ESG and f-ESG models, which are modelled as ESG, therefore, they exploit the optimal test sequence generation approaches.

Furthermore, Featured Event Sequence Graphs follows the "divide and conquer" principle in a more traceable way. In the hierarchical ESG models, models that are low-level in hierarchy are hidden under certain vertices. This makes the traceability of these models harder comparing to the FESG models, since in FESG models the hierarchy is shown explicitly via connection points. Also, c-ESG and f-ESGs of the FESG model could be handled separately from each other. Additionally, the hierarchical models of EGSs are not used before in SPL testing however the c-ESG and f-ESG model usage directly fits into the SPL testing since each feature is modelled individually and this allows tailored product configurations.

The traceability of FESG models contributes updating both the features and the product configurations easily. Comparing the large ESG models, adding, removing and updating vertices are quite easier. Also, updating the vertices of one f-ESG does not affect other f-ESGs except for the vertices that will be connected by other f-ESGs.

Consequently, ESGs are extended to FESGs to fit the SPL paradigm and to create more traceable and easily updatable small models.

4.3. Full Test Sequence Composition

The Feature Model (FM) with product configurations corresponding to feature diagram, Featured Event Sequence Graphs to model behavior of the SPL and a mapping between features and FESGs are required for test sequence composition technique (Tuglular, Beyazıt, and Öztürk 2019). The Feature Model and Featured Event Sequence Graphs are explained in previous sections. A product FESG tree is obtained by using the mapping of features with FESGs by using the selected features for a particular product configuration (Tuglular, Beyazıt, and Öztürk 2019). The root stores a link to the SPL's c-ESG and the leaves store links to the selected features' corresponding f-ESG in a product FESG tree (Tuglular, Beyazıt, and Öztürk 2019).

f-ESGs could include events that are not in the product configuration (Tuglular, Beyazıt, and Öztürk 2019). First of all, these events should be removed (Tuglular, Beyazıt, and Öztürk 2019). A product FESG lattice is constructed using the product FESG tree, where all connection relationships are ordered (Tuglular, Beyazıt, and Öztürk 2019). The Algorithm 4.1 is proposed in order to construct the product FESG lattice (Tuglular, Beyazıt, and Öztürk 2019). In order to notate either a c-ESG or an f-ESG, x-ESG is used (Tuglular, Beyazıt, and Öztürk 2019).

Algorithm 4.1 Construction of product FESG lattice

1. for each f-ESG
2. if it contains events, which are not in the product configuration, remove them
3. for each f-ESG (f)
4. if it has connection point(s) to c-ESG (c), build **f → c**
5. if it has connection point(s) to other one or more f-ESGs (g, h, etc.), build **f → g , f → h , etc.**
6. loop back

The output of the Algorithm 4.1 for bank account SPL's *daily limit product* is the hierarchical product FESG lattice and given in Figure 4.3. In this figure, c-ESG is at the top level (level 0) and the f-ESGs with connection points are children of the root

(level 1, level 2 and level 3). Note that, the number of leaves could be increased according to the product configuration's number of features.

The feature ESG of *daily limit* feature is demonstrated in Figure 4.4. Consider the *daily limit bank account product* that includes this feature which is given in Figure 4.5. Since the *daily limit* feature requires *withdraw* and *cancel withdraw* features, this product includes also these features and the feature *cancel deposit.* The configured features of this product do not contain any event that does not belong to *daily limit bank account product.* Therefore, the step 2 of Algorithm 4.1 is omitted for this product configuration.

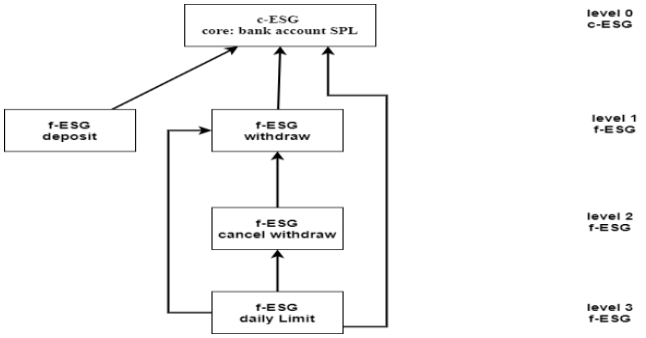

Figure 4.3. Product FESG lattice for *daily limit product*

The ESs of length 2 of each different path in the product FESG lattice for product *daily limit* that are covered by the partial test cases are as in the following:

PTS1: *get balance, ..., select deposit, enter deposit amount, put money*

PTS2: *get balance, ..., select withdraw, enter withdraw amount, take money*

PTS3: *...,* select *withdraw, cancel withdraw*

PTS4: *..., enter withdraw amount, confirm daily limit excess, enter withdraw amount,...*

PTS5: *... confirm daily limit excess, cancel withdraw*

PTS6: *..., enter daily withdraw limit*

Figure 4.4. *daily limit* f-ESG of bank account SPL

PTS1 is obtained from the path **deposit→core**, PTS2 is obtained from the path **withdraw→core**, PTS3 is obtained from **cancelWithdraw→core**, PTS4 is obtained from **dailyLimit→withdraw**, PTS5 is obtained from **dailyLimit→cancelWithdraw** and PTS5 is obtained from **dailyLimit→core.**

In order to compose the test sequences, the PTs should be connected in all possible ways (Tuglular, Beyazıt, and Öztürk 2019). In order to achieve this purpose, the Algorithm 4.2 is proposed (Tuglular, Beyazıt, and Öztürk 2019).

Algorithm 4.2. Composition of product test sequences

1. find connection points in partial test sequences
2. order them w.r.t their levels in product FESG lattice, c-ESG having the highest order
3. starting from lowest order for each connection point
4. find PTSs from PTS list and classify them as preceding and succeeding sequences with respect to this connection point
5. combine with all preceding sequences and add to PTS list
6. combine with all succeeding sequences and add to PTS list

The running example's partial test sequences have connection points which are the *[,], get balance, select withdraw, enter withdraw amount* and *cancel withdraw* events. Whenever they are composed, the following test sequences or CESs in other words, are obtained:

CES1: *get balance*

CES2: *select deposit, enter deposit amount, put money, get balance, select deposit, enter deposit amount, put money*

CES3: *select withdraw, enter withdraw amount, take money, get balance, select withdraw, enter withdraw amount, take money*

CES4: *enter daily withdraw limit, enter daily withdraw limit*

CES5: *select deposit, cancel deposit*

CES6: *select withdraw, cancel withdraw*

CES7: *select withdraw, enter withdraw amount, confirm daily limit excess*

CES8: *select withdraw enter withdraw amount, confirm daily limit excess, enter withdraw amount, confirm daily limit excess, cancel withdraw*

The composed test sequences cover the same ESs of length 2 with the test sequences generated from the ESG given in Figure 4.5.

Test sequence composition from FESGs allows us to obtain the test sequences that are equivalent with the complete product model's test sequences in terms of

coverage of length 2-ESs (Tuglular, Beyazıt, and Öztürk 2019). The product FESG enables reusing of generated partial test sequences for new-coming product configurations, which makes this approach efficient[5].

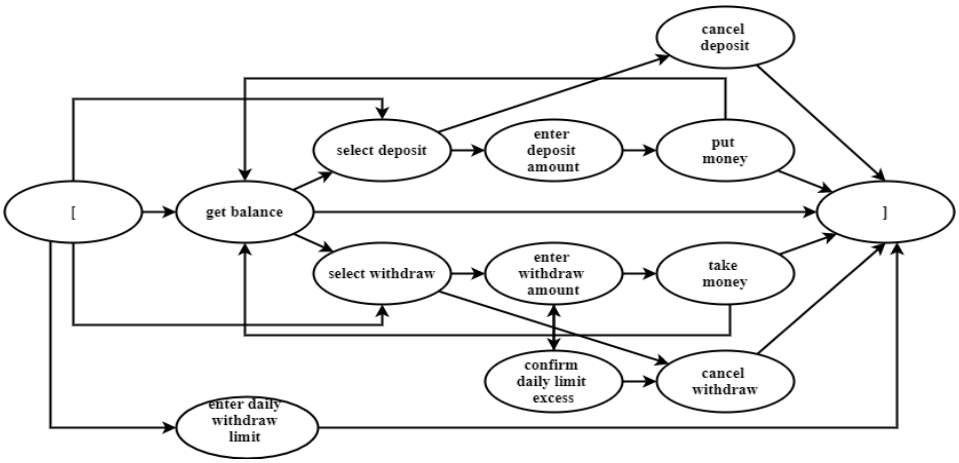

Figure 4.5. ESG of bank account SPL - *daily limit product*

The next chapter will introduce the incremental test sequence composition approach which is established on the base of the test sequence composition approach.

CHAPTER 5

INCREMENTAL TEST SEQUENCE COMPOSITION FROM FEATURED EVENT SEQUENCE GRAPHS

The incremental test sequence composition approach is explained in this chapter. The approach is model based; that is, it is based on the use of featured ESGs (FESGs). It exploits the fact that test cases for certain products already exist and they can be employed to obtain test cases of new products which are related to the existing products as implied by the corresponding feature diagram on a software product line. Therefore, the purpose of incremental test sequence composition is to reuse the FESG model and the test cases of an existing product in order to obtain test cases of a new product which is constructed by including new features to the existing product.

5.1. Incremental Test Sequence Composition

In the incremental test generation approach, the employed models are based on the definition of an ESG (Definition 3.3) and definition of an FESG (Definition 4.1). The following example is given in order to explain the components of FESGs on the *base product* of bank account SPL.

Example 5.1: The *base product* of bank account SPL ESG which is given in Figure 3.3 includes *deposit* and *withdraw* features, the FESG (F, c, Ξ, Γ) of this product is given in the following:

- F = {f_1, f_2}
- f_1 = *deposit* f-ESG (Figure 5.1)

V_1 = {*(core, [), (core,get balance), select deposit, enter deposit amount, put money, (core,])*}, E_1 = {(*(core,[), select deposit*),(*(core, get balance), select deposit*), (*select deposit, enter deposit amount*), (*enter deposit amount, put money*), (*put money, (core,])*) },

Ξ_1 = {*select deposit*}

Γ_1 = {put money}

- f_2 = *withdraw* f-ESG (Figure 4.2)

V_2 = {*(core, [), (core, get balance), select withdraw, enter withdraw amount, take money, (core,])*}

E_2 = {(*(core, [), select withdraw*), (*(core, get balance), select withdraw*), (*select withdraw, enter withdraw amount*), (*enter withdraw amount, take money*), (*take money, (core,])*)}

Ξ_2 = {*select withdraw*}

Γ_2 = {*take money*}

- c = c-ESG of bank account with V_{core} = {*get balance*}, E_{core} = ∅, Ξ_{core} = Γ_{core} = {*get balance*}
- $\Xi = \Xi_1 \cup \Xi_2 \cup \Xi_{core}$ therefore, Ξ = { *select deposit, select withdraw, get balance* }
- $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_{core}$ therefore, Γ = { put money, *take money, get balance*}

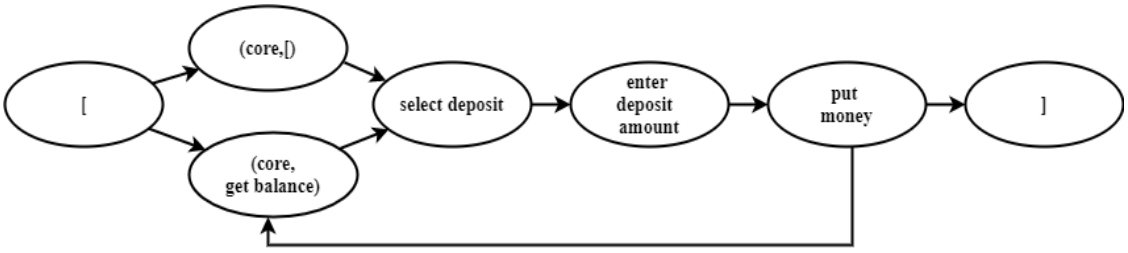

Figure 5.1. *deposit* f-ESG of bank account SPL

The incremental test sequence composition approach is more efficient than the regular test sequence composition approach, which is mentioned in CHAPTER 4, in terms of composition time, since the test sequences are not composed from scratch each time a new feature is added to the product configuration. The existing test sequences of the product are used, and test sequences for the new features are composed with the existing ones. Algorithm 5.1 demonstrates the main steps of incremental sequence generation.

Algorithm 5.1 assumes that T, i.e. a set of complete sequences for G achieving k-sequence coverage, achieves k-sequence coverage for the existing FESG. Therefore, it only transforms the new f-ESGs (incremental transformation), generates test sequences (partial sequences) from the transformed f-ESGs and compose these sequences with the existing ones in order to obtain a set of sequences achieving k-sequence coverage for the new FESG that is obtained by adding the new f-ESGs to the existing FESG properly.

Algorithm 5.1. Incremental Sequence Generation

Input: $G = (F, c, \Xi, \Gamma)$ – an FESG
T – a set of complete sequences for G achieving k-sequence coverage
$(F_{new}, \Xi_{new}, \Gamma_{new})$ – sets of new f-ESGs, and start and finish vertices to be added to G
k – integer coverage parameter for k-sequence coverage
Output: T' – a set of complete sequences for the new FESG $G' = (F \cup F_{new}, c, \Xi \cup \Xi_{new}, \Gamma \cup \Gamma_{new})$ achieving k-sequence coverage
F_{new_k} = transformIncremental(F_{new}, F, k) // See Algorithm 5.2
T_{new} = generateSequences(F_{new_k}, k) // See Algorithm 5.3
T' = composeSequences(T, T_{new}, $\Xi \cup \Xi_{new}$, $\Gamma \cup \Gamma_{new}$) // See Algorithm 5.4

Example 5.2: The Algorithm 5.1 is explained using the input G = *base product*-FESG exemplified in Example 5.1. k is chosen as 3 which means that the event triples will be covered. T = {CES1, CES2, CES3, CES4, CES5} which is the set of complete test sequences for G where CESs are given in the following:

CES1: *select deposit, enter deposit amount, put money, get balance, select deposit, enter deposit amount, put money*

CES2: *select deposit, enter deposit amount, put money, get balance, select withdraw, enter withdraw amount, take money, get balance, select withdraw, enter withdraw amount, take money*

CES3: *select withdraw, enter withdraw amount, take money, get balance, select deposit, enter deposit amount, put money*

CES4: *get balance, select deposit, enter deposit amount, put money*

CES5: *select deposit, enter deposit amount, put money*

G is incrementally updated to the *daily limit product*-FESG of the bank account SPL by addition of *cancelDeposit*, *cancelWithdraw* and *dailyLimit* (Figure 4.4) features. Therefore, F_{new} = {*cancelDeposit* f-ESG, *cancelWithdraw* f-EGS, *dailyLimit* f-ESG}, Ξ_{new} = {*enter daily withdraw limit*} and Γ_{new} = {*cancelDeposit, cancelWithdraw, enter daily withdraw limit*}. Note that, the elements of Ξ and Γ are event names, not the feature names.

The output of the Algorithm 5.1 for T'= {CES1, CES2, CES3, CES4, …., CES15} is given as follows:

CES1: *select deposit, enter deposit amount, put money, get balance, select deposit, enter deposit amount, put money*

CES2: *enter daily withdraw limit, enter daily withdraw limit, enter daily withdraw limit*

CES3: *enter daily withdraw limit*

CES4: *select deposit, enter deposit amount, put money, get balance, select withdraw, enter withdraw amount, take money, get balance, select withdraw, enter withdraw amount, take money*

CES5: *select deposit, cancel deposit*

CES6: *select deposit, enter deposit amount, put money, get balance, select deposit, cancel deposit*

CES7: *select withdraw, cancel withdraw*

CES8: *select deposit, enter deposit amount, put money, get balance, select withdraw, cancel withdraw*

CES9: *select withdraw, enter withdraw amount, confirm daily limit excess, enter withdraw amount, take money*

CES10: *select withdraw, enter withdraw amount, confirm daily limit excess, cancel withdraw*

CES11: *select withdraw, enter withdraw amount, confirm daily limit excess, enter withdraw amount, confirm daily limit excess, cancel withdraw*

CES12: *select withdraw, enter withdraw amount, take money, get balance, select deposit, enter deposit amount, put money*

CES13: *get balance, select deposit, enter deposit amount, put money*

CES14: *select deposit, enter deposit amount, put money*

CES15: *select withdraw, enter withdraw amount, take money*

The "transformIncremental", "generateSequences", and "composeSequences" procedures are explained in Algorithm 5.2, Algorithm 5.3, and Algorithm 5.4, respectively.

Algorithm 5.2 demonstrates how f-ESGs of the new features are incrementally transformed before generating sequences using them. Note that elements of the sets are accessed by using index values and it is assumed that insertion order is preserved in sets. This is done to obtain an f-ESG and its corresponding transformed form with less effort.

Algorithm 5.2 transforms each new f-ESG, i.e. f, k-2 times by using

1. f itself,
2. all the other f-ESGs which come before f and
3. all the base f-ESGs.

Remember that the length of the ES, determines the value of k. In order to cover event triples, for example, the ESG is transformed once so that it could have ESs of

length 2 in its vertices. In the case that event triples are covered, k is 3 and the number of transformations is 1. Therefore, Algorithm 5.2 transforms each new f-ESG k-2 times.

Algorithm 5.2. Incremental Transformation of f-ESGs

Input: F – a set of new f-ESGs to be transformed
B – a set of base f-ESGs to be used to trasform the new f-ESGs
k – integer transformation parameter
Output: F_k – a set of transformed features

```
Fk = F
for n=2 to k-1 incrementing by 1 do
    H = {}
    for i=1 to |F| incrementing by 1 do
        h = create an empty f-ESG to store the transformed f
        f = get the i^th f-ESG in Fk
        g = get the i^th f-ESG in F which is the f-ESG corresponding to Fk
        transformfESG(f, g, h) // Transform f using g updating h
        for j=i-1 to 1 decrementing by 1 do
            g = get the j^th f-ESG in F
            transformfESG(f, g, h) // Transform f using g updating h
        endfor
        for j=|B| to 1 decrementing by 1 do
            g = get the j^th f-ESG in B
            transformfESG(f, g, h) // Transform f using g updating h
        endfor
        H = H ∪ {h}
    endfor
    Fk = H
endfor
```

Each time f is transformed, "transformfESG" procedure is carried out. A separate algorithm is not given for this procedure because it is extended from the one given in Algorithm 3.2 by the following points.

1. "transformfESG" procedure used in Algorithm 5.2 also adds edges into the transformed f-ESG h to generate sequences with length <k in case such sequences cannot be included in sequences of length k.

2. Intermediate sequences which appear during "transformfESG" procedure are extended at both head and tail ends whereas those that appear in Algorithm 3.2 are extended at only tail end. This stems from the fact that, when an f-ESG f is transformed using an f-ESG g, g is not transformed using f to avoid repeated/redundant sequences and increase efficiency by reducing the number for transformations.

Algorithm 5.2 is explained in the following example. Here, the f-ESGs to be transformed are those given in Example 5.2.

Example 5.3: F = {*cancelDeposit* f-ESG, *cancelWithdraw* f-EGS, *dailyLimit* f-ESG} and B = {*deposit* f-ESG, *withdraw* f-ESG} are inputs of Algorithm 5.2. The integer transformation parameter is 1 since k = 3 in Example 5.2 and the number of transformations ,i.e., integer transformation parameter, is k - 2. Therefore, the f-ESGs are transformed once. In the execution of this algorithm, the f-ESG elements of F are transformed by exploiting the pre-transformed f-ESG elements of B. The one-time transformed version of *daily limit* f-ESG is given in Figure 5.2. Also, this algorithm allows us to generate sequences that have length smaller than k. For instance, CES3, CES5 and CES7 which are given in Example 5.2 have length 1 or 2 when k=3.

Algorithm 5.3. Sequence Generation of Transformed f-ESGs

Input: F – a set of transformed f-ESGs
k – integer transformation parameter
Output: T – a set of partial sequences for f-ESGs in F achieving k-sequence coverage

```
T = {}
for each f ∈ F do
    Tf = generateSequences(f)
    Tf = removeRepetitions(Tf, k)
    T = T ∪ Tf
endfor
```

After the new f-ESGs are transformed, Algorithm 5.3 can be used to generate partial sequences from the transformed f-ESGs to achieve k-sequence coverage.

In Algorithm 5.3 for each transformed f-ESG, sequences are generated from the f-ESG by covering all its edges (achieving edge coverage) using "generateSequences" procedure. After sequences are generated, they need to be processed in order to remove

repetitions due to using transformed models, which is done by "removeRepetitions" procedure considering the value of k. Following this step, obtained sequences are inserted into a single set.

The following example explains “generateSequences” procedure of Algorithm 5.1, which is Algorithm 5.3. The input *F* of Algorithm 5.3 is the one which is given in Example 5.3.

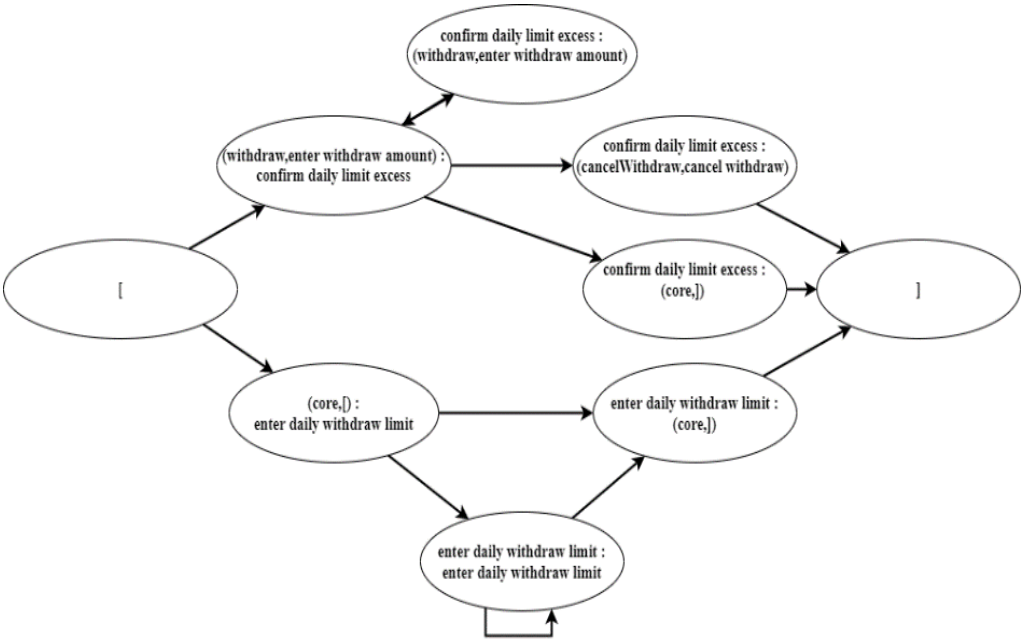

Figure 5.2. The one-time transformed *daily limit* f-ESG

Example 5.4: The test sequences of elements of the set *F* are generated in Algorithm 5.3. The output of Algorithm 5.3 is T_{new} = { $T_{transformedDailyLimit} \cup T_{transformedWithdraw} \cup T_{transformedCancelWithdraw}$ }. The sequence generation $T_{transformedDailyLimit}$ = {CES1, CES2,...,CES8} of one-time transformed *daily limit* f-ESGs (Figure 5.2) is given as in the following:

CES1: *(withdraw,enter withdraw amount), confirm daily limit excess, (withdraw,enter withdraw amount), (withdraw,take money)*

CES2: *enter daily withdraw limit, enter daily withdraw limit, enter daily withdraw limit*

CES3: *(withdraw,enter withdraw amount), confirm daily limit excess, (cancelWithdraw,cancel withdraw)*

CES4: *(withdraw,select withdraw)*

CES5: *(withdraw,enter withdraw amount), (withdraw,take money)*

CES6: *(withdraw,select withdraw), (withdraw,enter withdraw amount), confirm daily limit excess*

CES7: *enter daily withdraw limit*

CES8: *(withdraw,enter withdraw amount), confirm daily limit excess, (withdraw,enter withdraw amount), confirm daily limit excess, (cancelWithdraw,cancel withdraw)*

After partial sequence achieving k-sequences coverage are generated for the new f-ESGs, can be used to compose them with the existing complete sequences which are generated for the FESG model.

Algorithm 5.4. Sequence Composition

Input: T – a set of existing complete sequences
T_{new} – a set of new partial sequences
Ξ – a set of start vertices
Γ – a set of finish vertices

Output: CS – a set of composed complete sequences

```
SS = initializeStartSequences(T, T_new, Ξ)
notfinished = true
while notfinished is true do
        notfinished = updateStartSequences(T_new, SS)
endwhile
CS = initializeCompleteSequences(SS, Γ)
notfinished = true
while notfinished is true do
        notfinished = updateCompleteSequences(SS, CS)
endwhile
```

Algorithm 5.4 uses four different procedures in order to perform sequence composition: "initializeStartSequences", "updateStartSequences", "initializeCompleteSequences" and "updateCompleteSequences".

- "initializeStartSequences" constructs a set of start sequences SS. For each sequence $s \in T \cup T_{new}$, if s is a start sequence, it is removed from the set that contains it. s is included in SS if it increases the coverage.

Note that all sequences in T are complete sequences; whereas T_{new} may or may not contain start sequences.

- "updateStartSequences" goes through the remaining partial sequences in T_{new}. For each partial sequence $s \in T_{new}$, it tries to find a start sequence $seq \in SS$ such that s and seq can be composed to obtain a new sequence seq_new; that is, s can be completed to a start sequence by using a prefix of seq. If this is possible, s is removed from T_{new} and, if seq_new increases coverage, it is included in SS. Furthermore, if seq is a prefix of seq_new, seq is removed from SS. "updateStartSequences" is repetitively called until no more sequences remain in T_{new}.
- "initializeCompleteSequences" constructs a set of complete sequences CS. For each sequence $s \in SS$, if s is also a finish sequence, it is removed from SS and included in CS.
- "updateCompleteSequences" goes through the remaining start sequences in SS. For each start sequence $s \in SS$, it tries to find a complete sequence $seq \in SS$ such that s and seq can be composed to obtain a new sequence seq_new; that is, s can be completed to a finish sequence by using a suffix of seq. If this is possible, s is removed from SS and seq_new is included in CS. " updateCompleteSequences " is repetitively called until no more sequences remain in SS.

The following example explains “composeSequences” procedure of Algorithm 5.1, that is Algorithm 5.4.

Example 5.5: In this example, the input T of Algorithm 5.4 is the one that is given in Example 5.2. Also, the input T_{new} of Algorithm 5.4 is the one that is given in Example 5.4. All four procedures are explained with a simple instance of T, $T_{new,}$ or $T \cup T_{new}$.

initializeStartSequences procedure : There is $SS = \{ s_1, s_2, s_3, s_4, s_5\}$ for the given FESG in Example 5.2 with F_{new} ,Ξ_{new} and Γ_{new} , where,

s_1: *get balance*

s_2: *select deposit, enter deposit amount, put money, get balance, select deposit, enter deposit amount, put money*

s_3: *select withdraw, enter withdraw amount, take money, get balance, select deposit*

s_4: *enter daily withdraw limit, enter daily withdraw limit, enter daily withdraw limit*

s_5: *enter daily withdraw limit*

These sequences are added to SS since they all start with an element of $\Xi \cup \Xi_{new}$.

updateStartSequences procedure: For this procedure, a sequence of *daily limit* f-ESG is chosen in order to explain the procedure in a simpler way.

$S \in (T_{transformedDailyLimit} \subseteq T_{new})$ = *(withdraw,enter withdraw amount), confirm daily limit excess, (withdraw,enter withdraw amount), (withdraw,take money)*

seq = $s_3 \in SS$ = *select withdraw, enter withdraw amount, take money, get balance, select deposit*

seq_new = *select withdraw, enter withdraw amount, confirm daily limit excess, enter withdraw amount, take money*

seq_new increases coverage, therefore, it is included in SS = { s_1, s_2, s_3, s_4, s_5, seq_new}. Furthermore, seq is not a prefix of seq_new, therefore, it is not removed from SS.

initializeCompleteSequence procedure: For each sequence $s \in SS$ = { s_1, s_2, s_3, s_4, s_5, seq_new}, each sequence is investigated in terms start and finish events. If start event is an element of Ξ and finish event is an element of Γ, s is included in CS. CS is given in the following:

CS = { s_1, s_2, seq_new}

updateCompleteSequences procedure: This procedure is called repetitively called until no more sequences remain in SS.

$s \in SS$ = *select deposit, enter deposit amount, put money, get balance, select deposit, enter deposit amount, put money* which is a CES, is composed with the following:

seq = $s_3 \in SS$ = *select withdraw, enter withdraw amount, take money, get balance, select deposit*

And yet, seq_new = select withdraw, enter withdraw amount, take money, get balance, select deposit, enter deposit amount, put money is obtained.

In this chapter, the incremental test sequence composition approach that is based on the FESG usage is explained in detail. This approach exploits the fact that the existing FESGs and their corresponding test sequences are reusable. Therefore, in order

to build a new product with new features from an existing product, this approach is advantageous and efficient. In the next chapter, a helper graph to apply the incremental test sequence generation is introduced.

CHAPTER 6

FEATURE-BASED INCREMENTAL PRODUCT GRAPH

The feature-based incremental product graph is introduced in this chapter. Also, the ‘base product’ notion for SPLs is introduced in this chapter. A feature-based incremental product graph is a graph that helps to generate the test sequences of products within an SPL incrementally and automatically.

6.1. Feature-Based Incremental Product Graph

A feature-based incremental product graph, short for IPG is a directed graph that holds the FESGs of products within the SPL. It holds f-ESGs in its edges. By traversing this graph, the CESs of corresponding products are obtained automatically. Furthermore, by traversing the edges of this graph, a product configuration could be validated.

The novelty of the feature-based incremental product graph is producing the complete event sequences of each product variant within an SPL and validating these variants automatically by holding each product variant’s configuration in its vertices. Also, it could be traversed via Breadth First Traversal by using the features which are held by the feature-based incremental product graph’s edges.

In this thesis work, the usage of the IPG makes contribution in experimenting of incremental test sequence composition approach. Thanks to this graph, for four different case studies which will be mentioned in the next chapter, different experimenting scenarios are built. These scenarios include, generating the CESs of, for example, the *credit product*, from reusing the *base product* FESG model or from reusing the *interest product* FESG model. Therefore, the CESs of different product variants’ configurations are obtained via different scenarios. The CESs of each product variant are obtained at one run by employing the incremental test sequence composition approach so,

experimenting and getting results becomes quite easier. Also, each product variant's configuration is validated at one run by feature-based incremental product graph.

An IPG necessarily holds product FESGs in its vertices, i.e. each vertex corresponds to a product within the SPL which the generated CES could be executed on. Also, it has a start vertex as a basis product to others which could be any product within SPL.

In IPGs, there is no limitation to choose a product as a start vertex's product but of course, the selected product should make sense as a basis product. Assume that, the running example bank account could have only six products which are depicted in Figure 3.2. Since no more product could be configured, the given ones should be used between each other to generate CESs of all products. In this scenario, using for example *overdraft product* is not restricted in theory, but in practice it does not make the test generation process advantageous. This because, the incremental test sequence composition approach does not support feature removal yet and the only product that has the *overdraft* feature is the *overdraft product of Bank Account SPL*. The feature-based incremental product graph of running example is given in Figure 6.1.

In order to build a feature-based incremental product graph, the feature model of the SPL under consideration and the configuration file of each product variant within the SPL which could be created depending on the feature model and its constraints are parsed. The feature model and its corresponding configuration files are built by using a tool which is called FeatureIDE ("FeatureIDE" n.d.).

Three characteristics of features in the feature model are parsed: the name of the feature, being mandatory/optional and being abstract/concrete. Also, the constraints between features, which are implication, conjunction and disjunction, are parsed. The parsed information is used while building the incremental product graph. For instance, if a feature implies another feature this means the vertex that includes the implying feature should include the implied feature as well; or, if there is a disjunction constraint between two features, this means that these features should not be included in the same vertex or there should be no edge that directs the vertices so that these vertices include both of the features. Furthermore, each product variant's configuration file is parsed. By this way, each product's configuration is obtained and the FESGs of these configurations could be created.

Each vertex of the graph that is given in Figure 6.1 is traversed by the Breadth-First Traversal starting from the *base product* and the CESs of each product are

generated automatically by exploiting incremental test sequence composition approach. Here the start vertex includes the ‘base product’ FESG of the SPL, which is named as *base product.* The ‘base product’ notion is defined in the following to prevent intangibility.

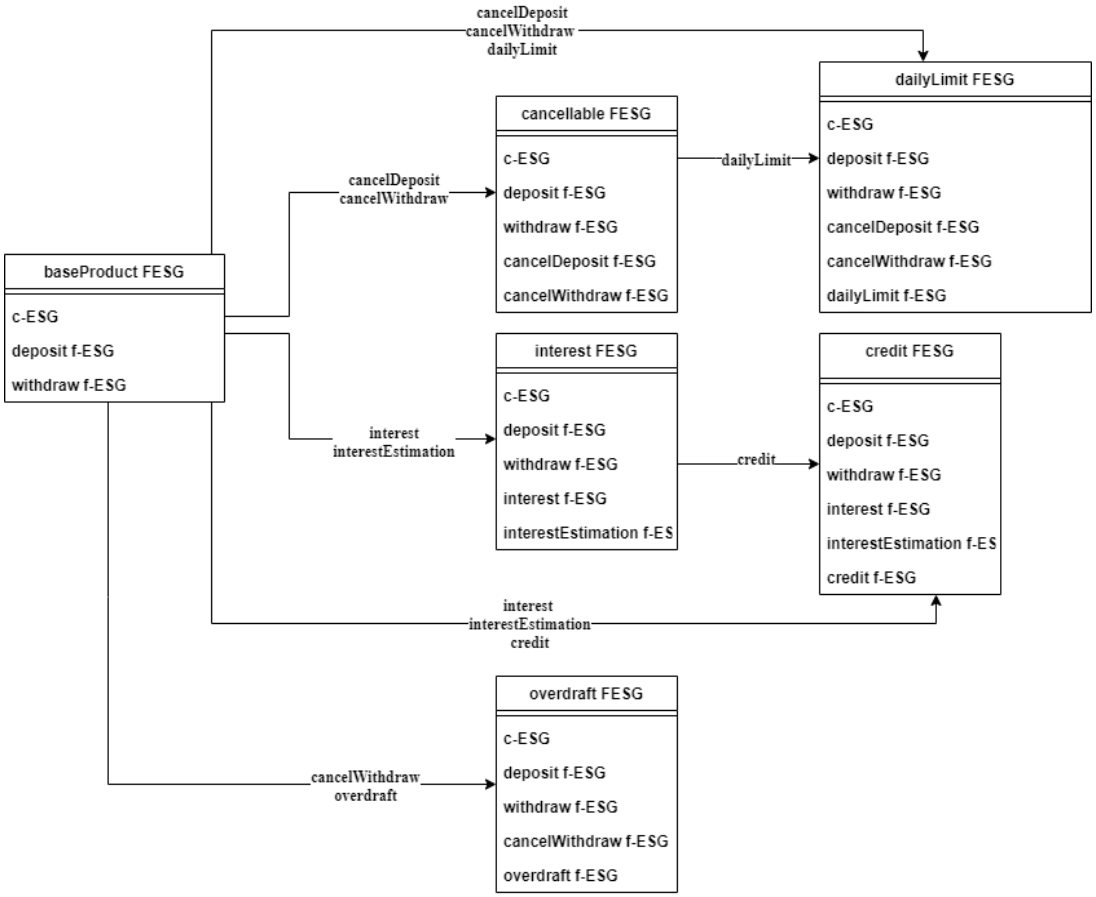

Figure 6.1. Feature-Based Incremental Product Graph of Bank Account SPL

Definition 6.1: A base product is a product that its configuration is up to the SPL domain and up to the developer who models the products within SPL. A product could be a base product in one of the following cases:

1. If the c-ESG of the SPL resembles a product within the domain (see Figure B.1 of Email SPL)
2. The product that is constituted from the c-ESG and non-excluding mandatory features (see Figure 3.3 of Bank Account SPL).

   Furthermore, an SPL could not have a base product, either. If mandatory featu-

res are related by XOR, i.e., only one of them should be selected within a product configuration, the base product could not be constituted.

Example 6.1: Consider the feature model of the Student Attendance System case study (see Figure 7.26). All of the grouping mandatory features of this feature model are mandatory, however, two of them group alternative features via XOR relation (see *SubmitAttendanceMethod* and *Notification* features). Since selecting one of these alternative features omit the rest of the features, a base product could not be decided for this SPL.

For the example above, the products could be incremented from 8 different basis products due the existence of *SubmitAttendanceMethod* which has 4 alternative sub-features and *Notification* which has 2 alternative sub-features that could be combined into 8 different basis products. Therefore, in order to generate test sequences of all the product variants within Student Attendance System SPL, there needs to be at least 8 different IPG with 8 different start vertices.

In order to traverse the graph, the product FESG of the start vertex should be decided and by using the start vertex with other vertices' product FESGs and the edges, the IPG should be build. The definition of the IPG is given in the following.

Definition 6.2: A Feature-Based Incremental Product Graph IPG(V(FESG),E(f-ESG), $\boldsymbol{v_0}$) is a directed graph where V ≠ ∅ is a finite set of nodes (vertices) that each node contains a Featured ESG and E ⊆ V ×V is a finite set of arcs (edges) that each edge contains a set of feature ESG (f-ESG). $\boldsymbol{v_0} \in$ V is the start vertex of the graph and it holds the product FESG that is decided as a basis product to generate the CESs of products within the SPL incrementally.

6.2. The Connectivity of Feature-Based Incremental Product Graph

Let $F = \{ c - ESG \cup \{f - ESG\ set\}\}$ is the set of features within the SPL and is denoted as $F = \{c, f_1, f_2, f_3, \dots, f_n\}$ with the cardinality $n + 1$. Remember that the c-ESG is a special f-ESG. The power set of F which is

$P(F) = \{ \emptyset, \{c\}, \{f_1\}, \{f_2\}, \{f_3\}, \dots, \{f_n\}, \{c, f_1\}, \{c, f_2\}, \{c, f_3\}, \dots, \{c, f_n\}, \dots, \{c, f_1, f_2, f_3, \dots, f_n\}\}$

which has the cardinality $2^{n+1} + 1$. In an $IPG(V(FESG), E(f - ESG), v_0)$,each vertex V

corresponds to one of the elements of $\mathrm{P(X)} = \mathrm{P}(F) - \{\emptyset, \{f_1\}, \{f_2\}, \{f_3\}, \ldots, \{f_n\}, \{f_i, f_j\}\}$ where f_i and f_j are excluding features and $V = \{v_0, v_1, \ldots, v_m\}$ where $m < 2^{n+1} + 1$. The edge set $E = \{(v_0, v_1), (v_0, v_2), (v_1, v_3), \ldots, (v_{m-1}, v_m)\}$. This means that each vertex corresponds to one of the elements of $\mathrm{P(X)}$ and E contains the element pairs of $\mathrm{P(X)}$. Therefore, a feature-based incremental product graph forms a partial order over subset relation $\subseteq$ on power set of F. This means that it is an inf-semilattice. The related definitions and examples to explain the details of being an inf-semilattice are given below.

Definition 6.3: A partial order is a relation with certain properties (Garg 2015).

The properties of a partial order could be Reflexivity, Irreflexivity, Symmetry, Antisymmetry, Asymmetry and Transitivity (Garg 2015; "Readings of Mathematics for Computer Science MIT OpenCourseWare - Chapter 7" n.d.). Also, a partially ordered set is called a poset (Garg 2015; "Readings of Mathematics for Computer Science MIT OpenCourseWare - Chapter 7" n.d.).

Definition 6.4: A reflexive partial order which is also called a non-strict partial order is a relation which is reflexive, antisymmetric and transitive (Garg 2015). It is symbolized as (X,≤).

Definition 6.5: An irreflexive partial order which is also called a strict partial order is a relation which is irreflexive, antisymmetric and transitive (Garg 2015). It is symbolized as (X, <).

Throughout this section, a poset means a set X with either a reflexive partial order or an irreflexive partial order.

Two operations are defined on the set X's subsets which are meet or infimum (inf) and join or supremum(sup) (Garg 2015).

Definition 6.6: To define meet operator (⋂) let Y⊆X, where (X,≤) is a poset. For any m ∈X,we say that m=inf Y iff ∀y ∈Y:m ≤y and ∀m'∈X: (∀y ∈ Y: m^'≤y) ⟹m' ≤ m. This means that m is the greatest lower bound of Y (Garg 2015).

Definition 6.7: To define join operator (⋃) let Y⊆X, where (X,≤) is a poset .For any s ∈X, s=sup Y iff ∀y ∈Y:y ≤s and ∀s^'∈X:(∀y ∈ Y: y≤s') ⟹ s≤s'. This means that s is the lowest upper bound of Y.

Definition 6.8: A lattice is a poset in which any two elements have an *inf* and a *sup* ("Readings of Mathematics for Computer Science MIT OpenCourseWare - Chapter 7" n.d.). Also, if $\forall x, y \in Y : x \bigcup y$ exists, then it is called sup-semilattice. If $\forall x, y \in Y : x \bigcap y$ exists, then it is called inf-semilattice (Garg 2015).

Since an incremental product graph has the v_0, which is used as the basis vertex to reach other vertices, and SPLs generally has excluding features, which prevent to have a product configuration, an incremental product graph is an inf-semilattice, where ∀ x,y ∈ Y:x⋂y exists. The inf-semilattice form of the IPG is demonstrated in Figure 6.2.

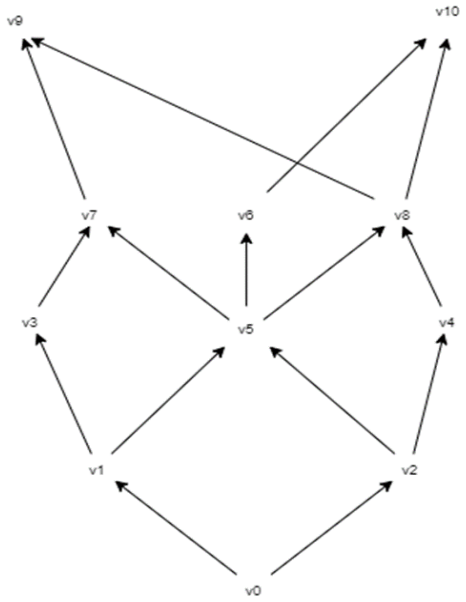

Figure 6.2. inf-semilattice from of the IPG

Example 6.2: F = {core, withdraw, deposit, cancelWithdraw, cancelDeposit, interest, interestEstimation, dailyLimit, overdraf, credit} is the feature set of Bank Account SPL where the credit and overdraft features are excluding. Therefore, P(X) = **{**{core, withdraw, deposit}, {core, withdraw, deposit, cancelDeposit, cancelWithdraw}, {core, withdraw, deposit, cancelDeposit, cancelWithdraw, credit, interest, interestEstimation}, {core, withdraw, deposit, cancelDeposit, cancelWithdraw, dailyLimit}, {core, withdraw, deposit, interest, interestEstimation}, {core, withdraw, deposit ,cancelWithdraw, dailyLimit, overdraft}**}**

$V = \{ v_0, v_1, \ldots, v_6 \}$

$v_0 = \{\text{core}, \text{withdraw}, \text{deposit}\}$

$v_1 = \{\text{core}, \text{withdraw}, \text{deposit}, \text{cancelDeposit}, \text{cancelWithdraw}\}$

$v_2 = \{\text{core}, \text{withdraw}, \text{deposit}, \text{cancelDeposit}, \text{cancelWithdraw}, \text{credit}, \text{interest}, \text{interestEstimation}\}$

$v_3 = \{\text{core}, \text{withdraw}, \text{deposit}, \text{cancelDeposit}, \text{cancelWithdraw}, \text{dailyLimit}\}$

$v_4 = \{\text{core}, \text{withdraw}, \text{deposit}, \text{cancelWithdraw}, \text{dailyLimit}, \text{overdraft}\}$

$v_5 = \{\text{core}, \text{withdraw}, \text{deposit}, \text{interest}, \text{interestEstimation}\}$

$E = \{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_0, v_4), (v_0, v_5), (v_1, v_3), (v_5, v_2)\}$

6.3. Incremental Test Generation from Feature-Based IPG

The runs of incremental test sequence composition approach in case study of this thesis are executed by using feature-based incremental product graphs. The Algorithm 6.1 is introduced in order to summarize the test sequence generation of FESGs in IPG vertices incrementally. The following example explains this algorithm on feature based incremental product graph of running example which is demonstrated in Figure 6.1.

Example 6.3: The vertex set and the edge set of the IPG of running example are given as V = {v_0(*base product* FESG), v_1(*cancellable* product FESG), v_2(*interest* product FESG), v_3(*daily limit* product FESG), v_4(*credit* product FESG), v_5(*overdraft* product FESG)}, E = {[*dailyLimit*], [*credit*], [*cancelDeposit*], [*cancelWithdraw*], [*interest*, *interest Estimation*], [*cancelWithdraw*, *overdraft*], [*cancelDeposit*, *cancelWithdraw*, *dailyLimit*], [*interest*, *interestEstimation*, *credit*]

After building the Feature-Based Incremental Product Graph, the incremental test sequence composition approach is experimented on this graph at a click.

Algorithm 6.1. Incremental Test Generation from IPG via Breadth First Traversal

Input: IPG(V, E, v_0) – a feature-based incremental product graph

Output: T – a set of test sequences for each FESG ∈ V

```
T = {}
Q ← create an empty queue
B ← create a Boolean array that is indexed to V's vertices
enqueue v0 to Q
B[v0] ← true
while Q is not empty do // perform Breadth First Traversal to traverse each vertex
    e ← deque Q
    if e = v0 do
        T0 ← full-test-sequence-composition(FESG ∈ v0)
    endif
    for (e,x) ∈ E do
        enqueue x to Q, B[x] ← true
        F ← get the set of f-ESGs on edge (e,x)
        Te ← incremental-test-sequence-composition(FESG ∈ e, F)
        assign Te of FESG ∈ e
        T = T ∪ Te
    endfor
endwhile
```

6.4. Validation of Product Configurations Using Feature-Based Incremental Product Graphs

As it is shown in Example 6.3, the edges of this graph contains one-element such as '*dailyLimit'*, two-element such as '*cancelWithdraw, overdraft*' and three-element such as '*interest, interestEstimation, credit*' feature sets. These different-size feature sets prove that new features could be added to an existing product configuration both as a chain or a bulk, incrementally. Also, the test sequences of, for example *credit product*, can be obtained from following either one of the paths below:

1. *base product + 'interest f-ESG, interestEstimation f-ESG' → interest product + 'credit f-ESG' → credit product*

2. *base product + 'interest f-ESG, interestEstimation f-ESG, credit f-ESG' → credit product*

Algorithm 6.2. Product Configuration Validator

Input: IPG(V, E, v_0) – a feature-based incremental product graph

Output: true – if product configurations in IPG vertices valid, false – otherwise

```
Q ← create an empty queue
B ← create a Boolean array that is indexed to V's vertices
enqueue v0 to Q
B[v0] ← true
v ← true
while Q is not empty do // perform Breadth First Traversal to traverse each vertex
        e ← deque Q
        for (e,x) ∈ E do
            enqueue x to Q, B[x] ← true
            F ← get the set of f-ESGs on edge (e,x)
            Fe ← get the f-ESG set of e
            Sx ← create an empty set to add f-ESGs
            Sx ← addAll(F + Fe)
            Fx ← get the f-ESG set of x //product configuration
            v ← v  AND (Sx = Fx)
        endfor
endwhile
return v
```

In order to validate product configurations in each vertex, Algorithm 6.2 is introduced. Traversing each vertex, by following different paths and comparing the set of features that come along the path and the destination vertex FESG's feature set, validates the destination vertex's product configuration.

Example 6.4: Assume that we want to validate *daily limit product*'s configuration which has '*c-ESG, deposit* f-ESG, *withdraw* f-ESG, *cancelDeposit* f-ESG, *cancelWithdraw* f-ESG, *dailyLimit* f-ESG'. By following the path below:

base product + 'cancelDeposit f-ESG, cancelWithdraw f-ESG, dailyLimit f-ESG' → daily limit product

The set of features that come along the path starting from *base product* becomes '*c-ESG, deposit* f-ESG, *withdraw* f-ESG, *cancelDeposit* f-ESG, *cancelWithdraw* f-ESG,

dailyLimit f-ESG'. Since the product configuration, i.e., the f-ESG set of *daily limit product* is the same, the configuration is determined as valid. Traversing each vertex by this algorithm, determines the validation of product configurations within the IPG.

In this chapter, the notions feature-based incremental product graph and base product are introduced. A feature-based incremental product graph is used to perform test sequence composition incrementally with one click for any coverage length. Also, it helps to validate the product configurations of products within the SPL. Example case studies are given elaborately in the next chapter, in order to show the results of proposed approaches.

CHAPTER 7

CASE STUDY

The test sequence composition approach and incremental test sequence composition approach are performed on several SPL examples which are Soda Vending Machine SPL, Email SPL, Bank Account SPL and Student Attendance System SPL. The results are depicted in this chapter, as well.

7.1. Soda Vending Machine SPL

Soda Vending Machine SPL (Tuglular, Beyazıt, and Öztürk 2019), or short for SVM SPL, is a small demonstrative SPL example. It has six features and one of them is mandatory. The feature model of the corresponding SPL is given in Figure 7.1. Related product for example one serving soda in USD, one serving free tea and one serving just soda in EUR can be developed using this diagram.

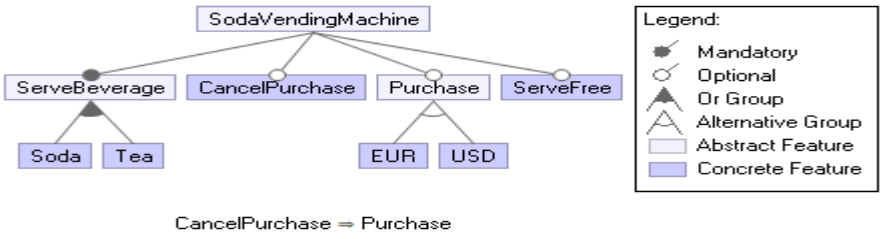

Figure 7.1. Soda Vending Machine SPL feature diagram

As demonstrated in the feature diagram, the EUR pay and USD pay are alternative features, i.e. they cannot be included by the same product. The possible product variants are demonstrated in the product matrix given in Figure 7.2.

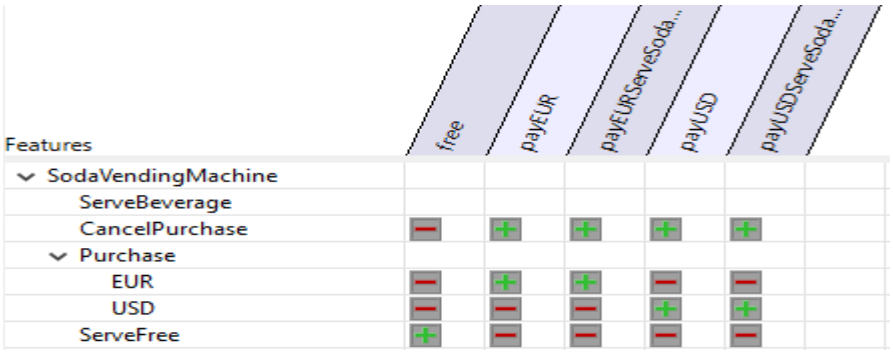

Figure 7.2. Product matrix of the SVM SPL

7.1.1. Soda Vending Machine Models

The SVM models are demonstrated and explained in this section. The core ESG of the SVM SPL is given in Figure 7.3. Note that, the core ESG is not necessarily a connected graph, i.e. some of the vertices could be disconnected. In this c-ESG, the *select* event is disconnected to be reusable and as the core is incremented by features it will become connected.

Figure 7.3. c-ESG of the SVM SPL

In Figure 7.4, the *pay EUR* feature is shown. The *pay EUR* event of this f-ESG, is connected between the *prompt* and *select* events.

Figure 7.4. *pay EUR* f-ESG of SVM SPL

Additionally, the *serve soda* feature of SVM SPL is demonstrated in Figure 7.5. This feature changes the course of events by starting from *select* event to pseudo finish event and it makes *select* event connected. Remember that, if an event is connected to pseudo finish event it is a finish event. Therefore, connecting the pseudo finish event of the c-ESG makes an event a finish event.

Figure 7.5. *serve soda* f-ESG of SVM SPL

In Figure 7.6, the *pay EUR serve soda* product of SVM SPL is depicted. As it is shown in the figure, the *payEUR* event is connected to the *prompt* and *select* events; the *serveSoda* event is connected to the *select* and pseudo finish events. Also, the *cancel purchase* feature which is shown in Figure 7.7, is also added to the product configuration.

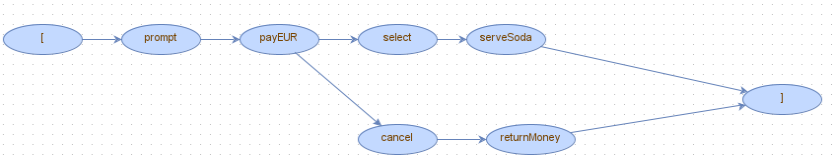

Figure 7.6. ESG of SVM SPL – *pay EUR-serve soda product*

All of the other f-ESGs and product ESGs are given in APPENDIX A.

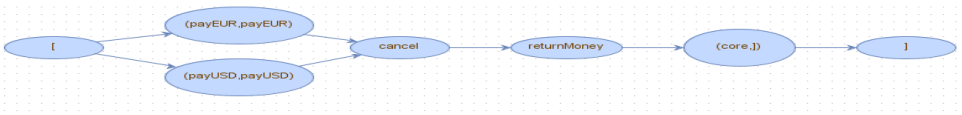

Figure 7.7. *cancel purchase* f-ESG of SVM SPL

7.1.2. Soda Vending Machine Results

The SVM results are depicted in this section. FESG components which contains the c-ESG and the f-ESGs are shown in Table 7.1. Since the c-ESG has two vertices which are disconnected except for pseudo vertices, the number of edges is zero due to the fact that the edges that connect vertices to one of the pseudo vertices are not counted as real edges. Also, the number of vertices and edges are shown in this table. The total number of vertices is twenty-two and the total number of edges is fourteen of the components in this SPL as it is shown in Table 7.1.

Table 7.1. Soda Vending Machine SPL FESG Components

SPL FESG Components	**Number of Vertices**	**Number of Edges**
c-ESG	2	0
cancel f-ESG	5	4
free f-ESG	3	2
payEUR f-ESG	3	2
payUSD f-ESG	3	2
serveSoda f-ESG	3	2
serveTea f-ESG	3	2
TOTAL	22	14

The products of the SVM SPL are demonstrated in Table 7.2 in order to present the complexity of the test models as well as their differences. The *free product* ESG has the least number of vertices and edges since it has the least number of features. Also, the vertex number and the edge number of *payEURServeSoda* and *payUSDServeTea* products are equal just like *payEUR* and *payUSD* products due to the fact that they have the features which are equivalent in terms of vertex number and edge number.

“Isolated” means that the existing test generation technique is executed on the product ESGs of Table 7.2. The FESG models and isolated products are compared in terms of total number of vertices and total number of edges in Table 7.3. In terms of test artefacts, the approaches in this thesis agree with the statement that SPL-based

development approach is more convenient if the software products are alike. Table 7.3 depicts this agreement.

Table 7.2. Isolated Product ESGs of SVM SPL

Isolated Product ESGs of SVM SPL	**Number of Vertices**	**Number of Edges**
Free product ESG	4	3
payEURServeSoda product ESG	6	5
payEUR product ESG	7	6
payUSDServeTea product ESG	6	5
payUSD product ESG	7	6
TOTAL	30	25

Test sequences of SVM products are generated by using the existing ESG test generation approach, the full test sequence composition approach and the incremental test sequence composition approach. The proposed full test sequence composition and incremental test sequence composition approaches are executed on FESGs while the existing test generation approach is executed on the isolated product ESG models.

Table 7.3. Model Comparison of SPL FESGs and Isolated Products

	Number of Vertices	**Number of Edges**
FESGs of SVM SPL	22	14
Isolated Product ESGs of SVM SPL (5 ESGs)	30	25

The execution time to obtain complete event sequences (CESs) of these approaches are given in Table 7.4, Table 7.6 and Table 7.8 respectively, for the existing ESG test generation approach, full test sequence composition approach and incremental test sequence composition approach.

The results are obtained for 10 runs which are performed on a PC having Intel 2.60 GHz CPU and 12 GB RAM with 64-bit Windows 10 Enterprise operating system. Also, the results are obtained by setting coverage length to 2 for SVM SPL. This means

that the CESs that cover event pairs are generated for SVM SPL

Table 7.4. Test Generation Time of Isolated Product ESGs of SVM SPL

	Coverage Length 2		
Test Generation Time (ms)	**Min**	**Max**	**Avg**
free ESG	34.44	36.66	35.684
payEURServeSoda ESG	36.63	38.3	37.459
payEUR ESG	37.16	39.18	37.982
payUSDServeTea ESG	36.22	37.53	36.728
payUSD ESG	36.84	41.83	38.387

The number of CESs and number of events are shown in Table 7.5 and Table 7.7 for ESG test generation and for full test sequence composition, respectively. The SVM SPL case study shows that the full test sequence composition approach provides CESs which are approximately 50% longer than the CESs generated by traditional test generation approach. This holds for *payEUR* and *payUSD* products which have the most number of features for this SPL. The rest of the products have the same length and the same number of CESs for both approaches.

Table 7.5. Complete Event Sequences of Isolated Product ESGs of SVM SPL

	Coverage Length 2	
Test Generation CESs	**# of CESs**	**# of Events**
free ESG	1	4
payEURServeSoda ESG	2	8
payEUR ESG	3	12
payUSDServeTea ESG	2	8
payUSD ESG	3	12

Furthermore, the full test sequence composition approach takes nearly 9% to 13% more time than existing test generation approach for SVM SPL. Therefore, both the size of the test sets and the execution time of existing approach are less than the full test sequence composition approach for SVM SPL when the coverage length is 2.

Table 7.6. Full Test Sequence Composition Time of FESGs

	Coverage Length 2		
Test Sequence Composition Time (ms)	**Min**	**Max**	**Avg**
free FESG	39.39	41.25	40.322
payEURServeSoda FESG	40.89	45.55	42.067
payEUR FESG	40.98	44.24	42.05
payUSDServeTea FESG	41.05	42.54	41.784
payUSD FESG	40.51	42.88	41.754

Comparing the incremental test sequence composition approach with the other mentioned approaches, Table 7.8 indicates that the execution time is drastically smaller than both of the approaches and the test set size is greater than the ESG test generation approach and equal to the full test sequence composition approach. This because incremental test sequence composition approach reuses both the existing test sequence of previously configured products and the test models where the full test sequence composition approach reuses only the test models and compose the test sequences of each from scratch.

Table 7.7. Complete Event Sequences of Full Test Sequence Composition FESGs

	Coverage Length 2	
Test Sequence Composition CESs	**# of CESs**	**# of Events**
free FESG	1	4
payEURServeSoda FESG	2	8
payEUR FESG	3	12
payUSDServeTea FESG	2	8
payUSD FESG	3	12

For this and the rest of the case studies, the incremental test sequence composition approach cannot be applied to all products within the domain. This because it requires a basis product that is previously configured and tested, also, the basis product is required to have the common features with the one which is will be tested.

Table 7.8. Time and CESs of Incremental Test Sequence Composition FESGs

Incremental Test Sequence Composition (ms)			**Coverage Length 2**				
FESG to be reused	**f-ESG(s) to be added**	**Obtained FESG**	**Min**	**Max**	**Avg**	**# of CESs**	**# of Events**
payEURServeSoda FESG	serveTea f-ESG	payEUR FESG	2.44	2.68	2.55	3	12
payUSDServeTea FESG	serveSoda f-ESG	payUSD FESG	2.53	2.87	2.64	3	12

This relation is similar to subset relation but it is stricter. For instance, although the *payEURServeSoda product* and *free product* has *serveSoda* feature in common, the *free product* cannot be the basis product to be reused since it also has the *free* feature. The feature-based incremental product graphs of two experiments in Table 7.8 are given in Figure 7.8. There are two IPGs in this figure, since there are two basis products.

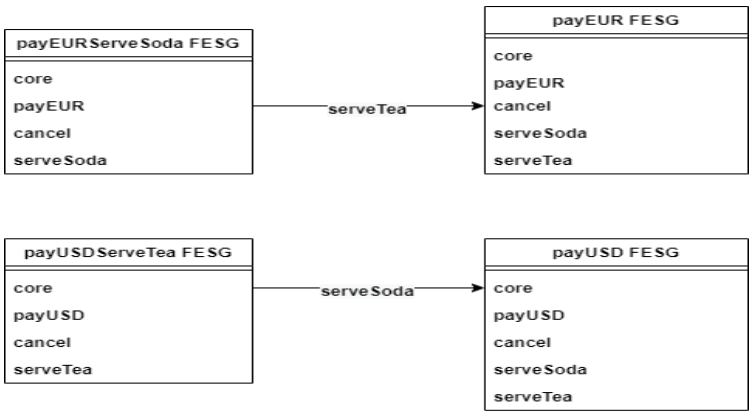

Figure 7.8. Feature-Based Incremental Product Graph of SVM SPL

7.2. Email SPL

The product within the Email SPL, enables the users to compose a new email, send an email and to read an incoming email, basically. It also has six features which extend the core of the SPL in order to build an address book, auto respond incoming emails, forward emails, encrypt emails, get a public key of receiver and sign emails.

The feature diagram of Email SPL is given in Figure 7.9. Also, the product diagram which shows the product configurations within this SPL is given in Figure 7.10.

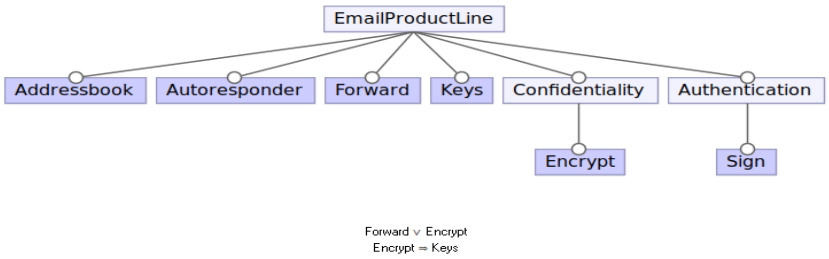

Figure 7.9. Email SPL feature Diagram

All six features of this SPL are optional. The *encrypt* feature requires *keys* feature. Also, *forward* feature excludes *encrypt* and, vice versa. The exclusion could be noticed from Figure 7.10 where these two features omit each other in product configurations.

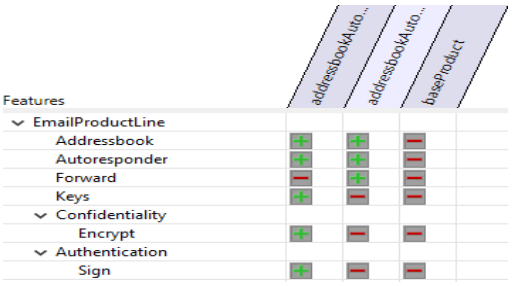

Figure 7.10. Product matrix of Email SPL

7.2.1. Email SPL Models

The core of the SPL is shown in Figure 7.11. The core itself is a product which represents the basic functionality of the SPL such as composing an email, sending an email and reading an email.

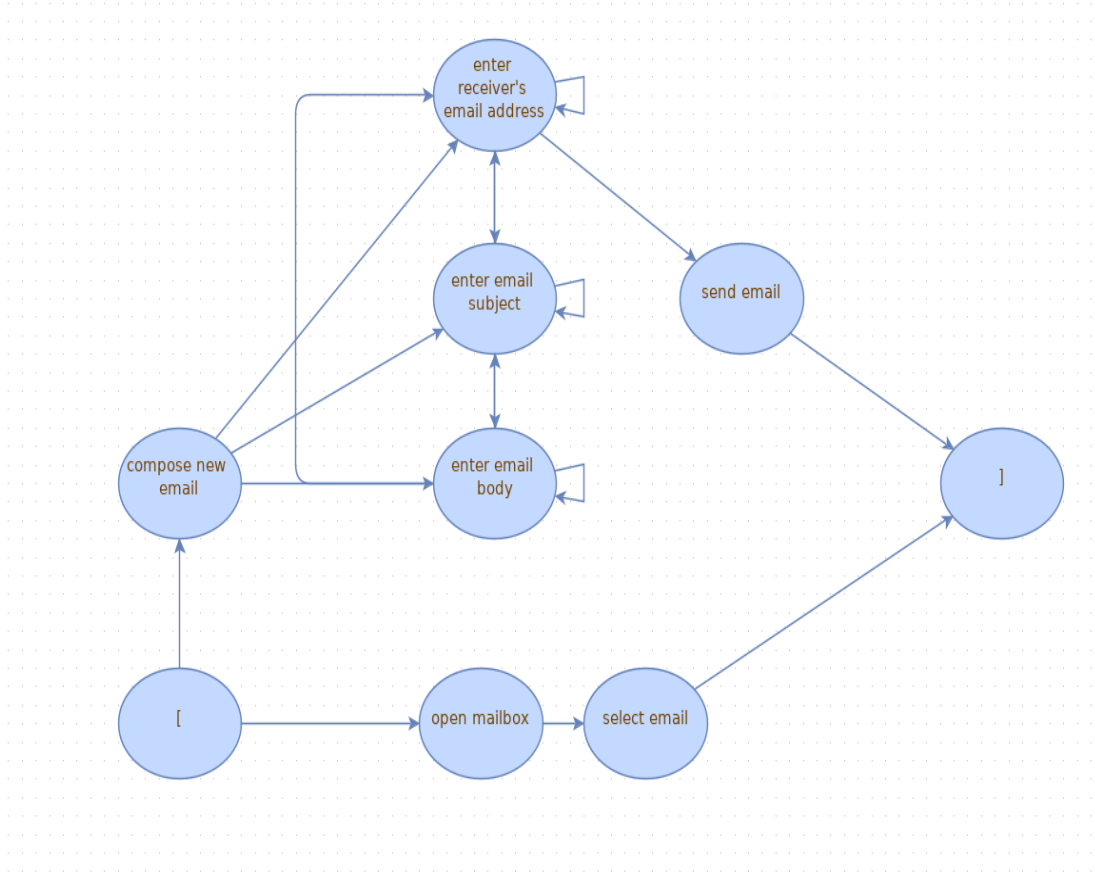

Figure 7.11. c-ESG of Email SPL

The *addressbook* feature allows creating an address book for a contact who has more than one email address and sending an email to all of these addresses. The f-ESG of this feature is shown in Figure 7.12.

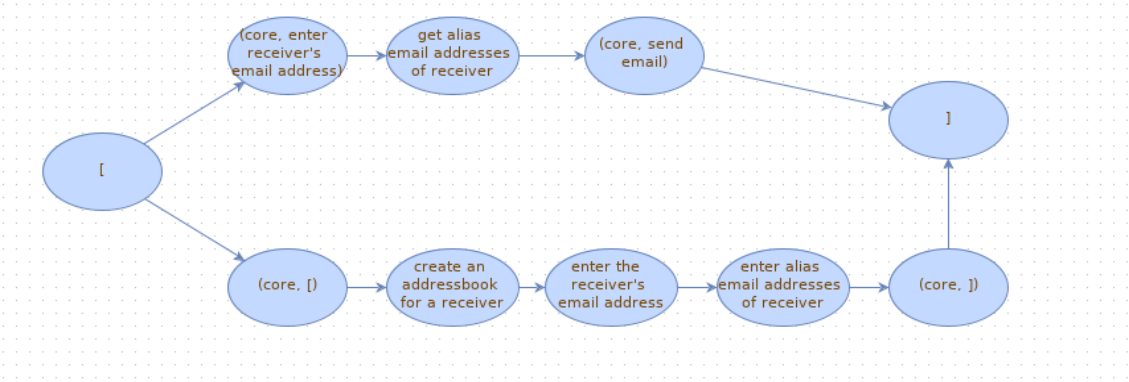

Figure 7.12. *addressbook* f-ESG of Email SPL

The *autoresponder* feature which is given in Figure 7.13, enables composing an email body as an auto response and setting the date interval of occurrence.

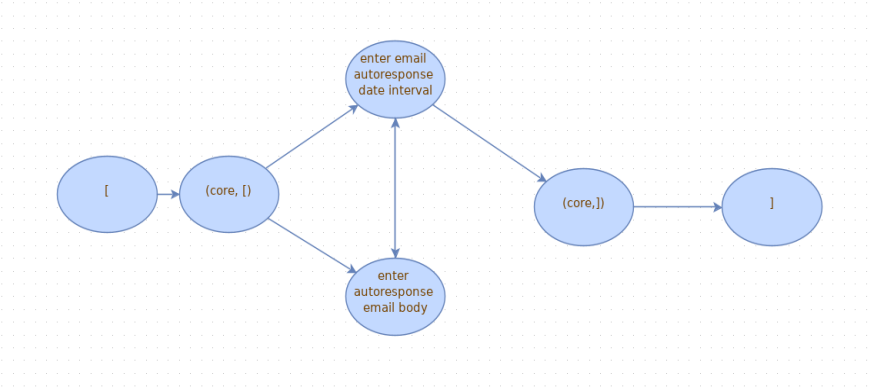

Figure 7.13. *autoresponder* f-ESG of Email SPL

The *forward* feature allows the user to forward an incoming email. It excludes the *encrypt* feature, i.e., encrypted emails could not be forwarded. The f-ESG of the *forward* feature is given in the following.

Figure 7.14. *forward* f-ESG of Email SPL

The *keys* feature enables the user to determine the receiver's public key. This public key is used to encrypt an email; therefore, this feature is required by *encrypt* feature. The f-ESG of the *keys* feature is given in the following.

Figure 7.15. *keys* f-ESG of Email SPL

The *encrypt* feature activates the encryption of emails. Since it has a connection point to *keys* f-ESG and the encryption process needs the receivers public key by definition, it requires the *keys* feature. The f-ESG of the *keys* feature is given in the following.

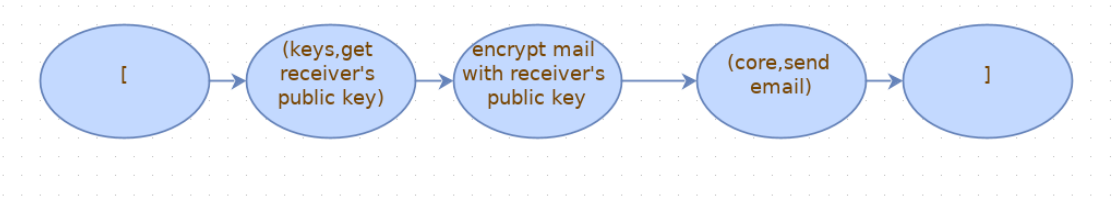

Figure 7.16. *encrypt* f-ESG of Email SPL

The sender user's signature is added by the *sign* feature. Its f-ESG is given in the following.

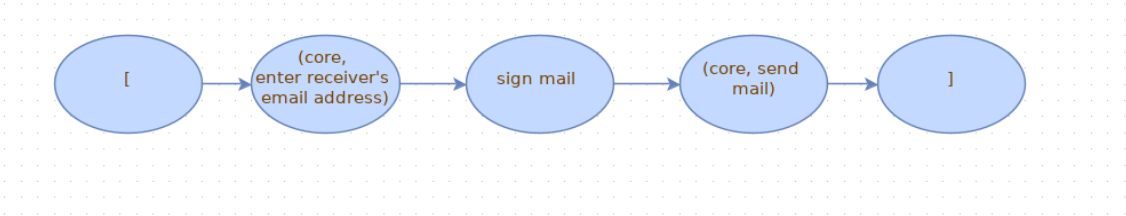

Figure 7.17. *sign* f-ESG of Email SPL

7.2.2. Email SPL Results

The Email Software Product Line test generation results are given in this section. The c-ESG and the f-ESGs of Email SPL are given in Table 7.9. In this case study, the c-ESG corresponds to an existing product namely *base product* since it models the behavior of a basic email product. The total number of vertices which is thirty-one and the total number of edges which is thirty-two are given in Table 7.9.

Table 7.9. Email SPL FESG Components

SPL FESG Components	**Number of Vertices**	**Number of Edges**
c-ESG	7	13
addressbook f-ESG	8	6
autoresponder f-ESG	4	5
forward f-ESG	3	2
keys f-ESG	3	2
encrypt f-ESG	3	2
sign f-ESG	3	2
TOTAL	31	32

The sample Email SPL products are shown in Table 7.10. The product number of this SPL could be increased, however in order to make the example simple, three of them are given in Table 7.10.

Table 7.10. Isolated Product ESGs of Email SPL

Isolated Product ESGs of SVM SPL	**Number of Vertices**	**Number of Edges**
baseProduct ESG	11	17
addressbookAutoresponderEncryptSign ESG	16	24
addressbookAutoresponderForward ESG	14	21
TOTAL	41	62

The FESG models and isolated products are compared in terms of total number of vertices and total number of edges in Table 7.11. This table proves the importance of modelling an SPL in small features instead of modelling it for each different product. The results show that the number of vertices and edges are greater than total of FESG components even for three products. Since modelling each product in isolation restrains reusability, the vertices and the edges repeat in different products and it increases their number.

Product test sequences are obtained by executing ESG test generation approach on isolated product ESGs and, the full test sequence composition approach and the incremental test sequence composition approach on FESGs. Table 7.12, Table 7.14 and Table 7.15 demonstrates the execution time to generate CESs from ESG test generation

approach, the full test sequence composition approach and the incremental test sequence composition approach, respectively.

Table 7.11. Model Comparison of SPL FESGs and Isolated Products

	Number of Vertices	**Number of Edges**
FESGs of Email SPL	31	32
Isolated Product ESGs of Email SPL (4 ESGs)	41	62

Similar to SVM SPL, not only the execution time, but also the number of CESs and the number events of ESG test generation of isolated products are less than the ones obtained by full test sequence composition approach. Since the difference between average execution time is not more than 2.48 milliseconds, also for this SPL, the full test sequence composition approach is considerably good for test generation of products since it produces a larger test set in a reasonable time.

The number of CESs and their total number of events are given in Table 7.13.

Table 7.12. Test Generation Time of Isolated Product ESGs of Email SPL

	Coverage Length 2		
Test Generation Time (ms)	**Min**	**Max**	**Avg**
baseProduct ESG	37.24	39.35	38.502
addressbookAutoresponderEncryptSign ESG	42.36	47.62	44.695
addressbookAutoresponderForward ESG	42.41	43.9	43.094

Table 7.13. Complete Event Sequences of Isolated Product ESGs of Email SPL

	Coverage Length 2	
Test Generation CESs	**# of CESs**	**# of Events**
baseProduct ESG	4	20
addressbookAutoresponderEncryptSign ESG	8	35
addressbookAutoresponderForward ESG	8	33

Here, the *addressbook-autoresponder-encrypt-sign product* cannot be incremented from *addressbook-autoresponder-forward product* due to the existence of *encrypt* and *forward* features which are excluding. Even though these features are not excluding, the *addressbook-autoresponder-encrypt-sign product* still, cannot be incremented from *addressbook-autoresponder-forward product*, since the current version of the incremental test sequence composition approach does not support feature removal and for this case, the *forward* feature should be removed from *addressbook-autoresponder-forward product.*

Table 7.14. Full Test Sequence Composition Time of FESGs

	Coverage Length 2		
Test Sequence Composition Time (ms)	**Min**	**Max**	**Avg**
baseProduct FESG	38.94	47.63	40.982
addressbookAutoresponderEncryptSign FESG	44.37	47.2	45.576
addressbookAutoresponderForward FESG	43.02	54.44	45.063

When we examine the results of incremental test sequence composition approach which are given in Table 7.15, we see that the resulting CESs are longer and more in number than both of the previous approaches. Also, it is seen from the results that incremental test sequence composition approach performs ~%87 to ~%94 better in average execution time.

Table 7.15. Time and CESs of Incremental Test Sequence Composition Approach

Incremental Test Sequence Composition (ms)			**Coverage Length 2**				
FESG to be reused	**f-ESG(s) to be added**	**Obtained FESG**	**Min**	**Max**	**Avg**	**# of CESs**	**# of Events**
baseProduct FESG	autoresponder f-ESG forward f-ESG	addressbookAutoresponder-Forward FESG	2.51	3.08	2.69	9	38
baseProduct FESG	autoresponder f-ESG encrypt f-ESG sign f-ESG	addressbookAutoresponder-EncryptSign FESG	5.54	6.2	5.9	11	53

7.3. Bank Account SPL

Bank Account SPL which is given as a running example throughout this thesis has nine features in which the two of them are mandatory. The feature diagram and the product matrix of this software product line are given in Figure 3.1 and Figure 3.2, respectively.

7.3.1. Bank Account SPL Models

The core behavior of this SPL which is given in the c-ESG in Figure 4.1 represents the core behavior of a bank account product which is getting the balance of the account. The mandatory features of this SPL are *deposit* and *withdraw*. The f-ESG of *deposit* feature is given in the following. This feature represents the behavior of putting money into the account. The *withdraw* feature is demonstrated in Figure 4.2 and it represents the behavior of taking money from the account.

The optional features *cancelDeposit* and *cancelWithdraw* represent the cancel operations of the depositing and withdrawing. These features make one bank account product cancellable. The f-ESG of *cancelDeposit* is depicted in Figure 7.19, and the f-ESG of *cancelWithdraw* is given in Figure 7.20.

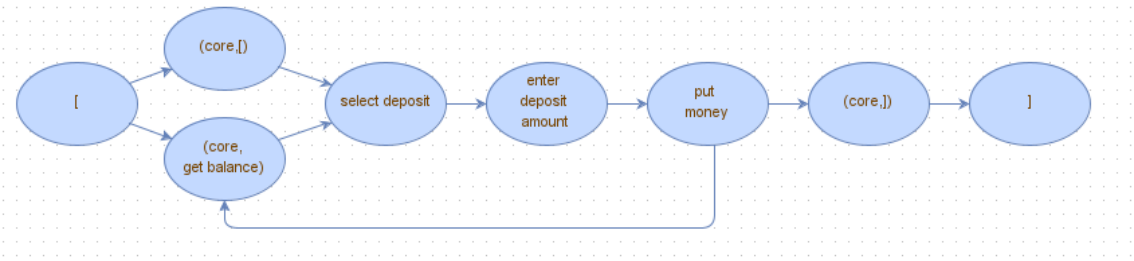

Figure 7.18. *deposit* f-ESG of Bank Account SPL

The *"cancel deposit"* event is connected to "*select deposit*" event of *deposit*

f-ESG and "*]*" event of the c-ESG. The connection to the pseudo finish event of c-ESG, makes the *"cancel deposit"* event a finish event. Note that, a f-ESG could contain connection points to c-ESG and more than one f-ESGs.

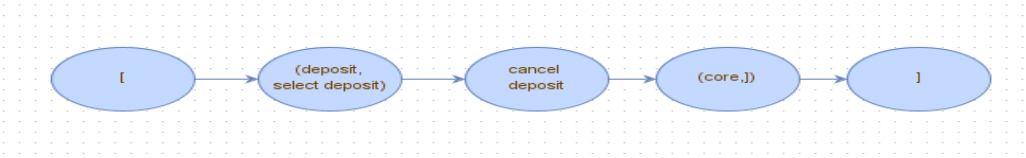

Figure 7.19. *cancelDeposit* f-ESG of Bank Account SPL

Since, the *cancelDeposit* and *cancelWithdraw* f-ESG has connection points to *deposit* and *withdraw* f-ESGs, respectively, there is a constraint the feature diagram (Figure 3.1) for each of these f-ESGs. These constraints force the product configurations to have, for example,the *deposit* feature if the PC has the *cancelDeposit* feature. Even though the *deposit* and *withdraw* are mandatory features, these constraints are added because of the existence of connection points.

The *overdraft* and *credit* features which are grouped under *extraMoney* abstract feature in the feature diagram (Figure 3.1), are alternative features and allow the bank account user to take extra money from the account. The user can define a limit to exceed the account balance if the bank account product has the *overdraft* feature. The f-ESG of *overdraft* is given in .The *overdraft* feature requires both the *cancelWithdraw* and *dailyLimit* (Figure 4.4) features due to the connection points.

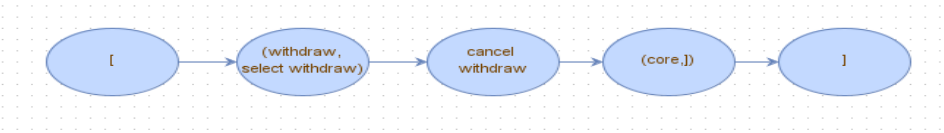

Figure 7.20. *cancelWithdraw* f-ESG of Bank Account SPL

The *credit* feature which is given in Figure 7.22 allows the bank account users to take extra money as a debt. It has connection points to core, therefore it doesn't imply any other features' existence.

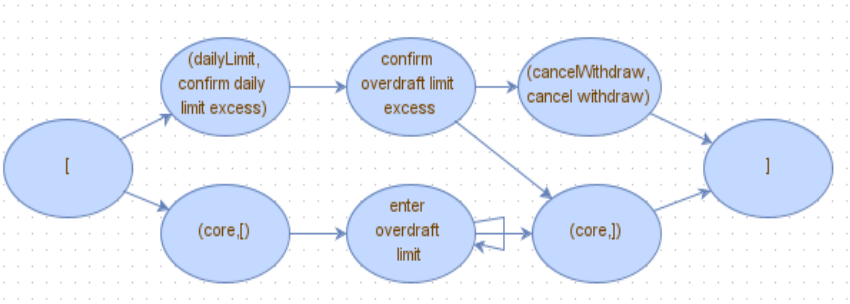

Figure 7.21. *overdraft* f-ESG of Bank Account SPL

The *interest* and *interestEstimation* features enable the users of a bank account product to request an interest rate and to determine total gain of interest for particular days, respectively. Additionally, *interestEstimation* requires the *interest* feature due to the connection point to *interest* f-ESG.

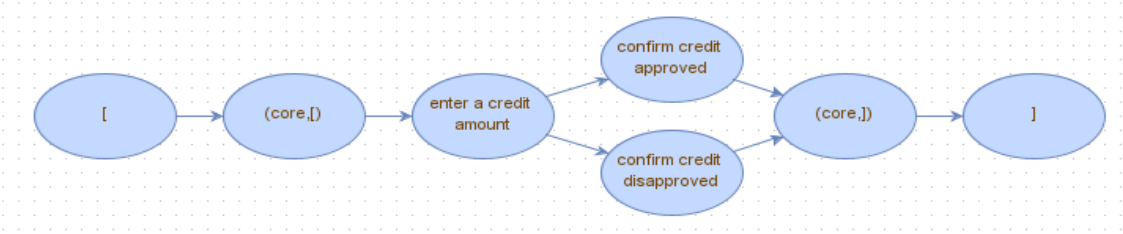

Figure 7.22. *credit* f-ESG of Bank Account SPL

The *dailyLimit* feature (Figure 4.4) helps users of a bank account product to put a limitation on the amount of money that could be taken from the account in daily basis. It requires the *withdraw* and *cancelWithdraw* features in the product configuration.

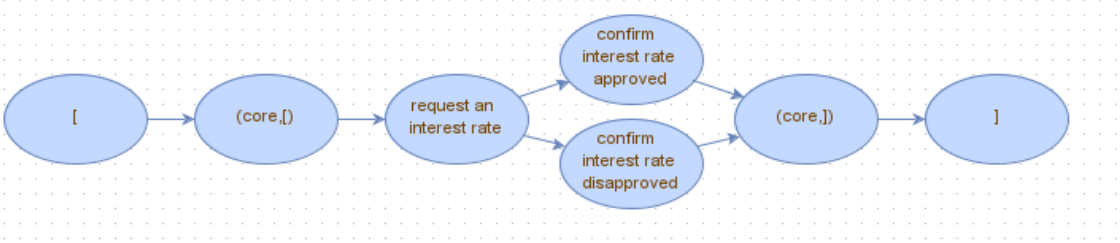

Figure 7.23. *interest* f-ESG of Bank Account SPL

The bank account SPL forty-two possible product configurations. These could be configured by employing FESGs easily, however the number of complete product ESGs is forty-two. Therefore, six of the complete product ESGs are modelled due to the space limitations.

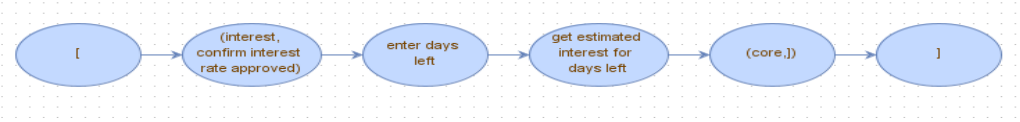

Figure 7.24. *interestEstimation* f-ESG of Bank Account SPL

The *cancellable product* is given as an example in Figure 7.25. Also, the *daily limit product* is given in Figure 4.5. The rest of the product ESGs are given in APPENDIX C.

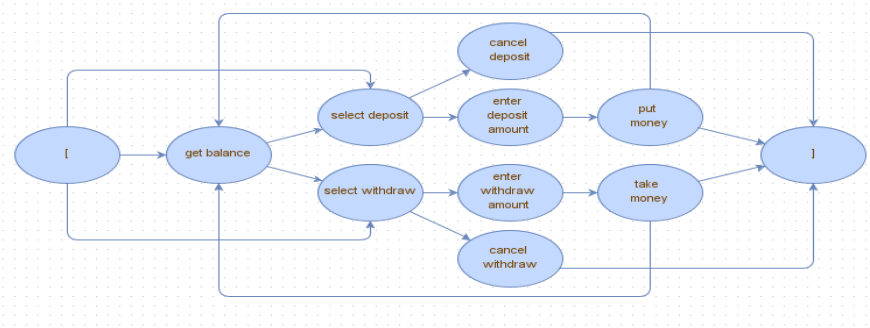

Figure 7.25. ESG of bank account SPL – *cancellable product*

7.3.2. Bank Account SPL Results

The Bank Account SPL results are given in this section. FESG components which contains the c-ESG and the f-ESGs and their corresponding number of vertices and edges are shown in Table 7.16. Since the c-ESG has only one vertex except for pseudo vertices, the number of edges are zero. The total number of vertices is forty-five

and the total number of edges is forty-two of the components in this SPL as it is shown in Table 7.16.

Table 7.16. Bank Account SPL FESG Components

SPL FESG Components	**Number of Vertices**	**Number of Edges**
c-ESG	1	0
deposit f-ESG	6	6
withdraw f-ESG	6	6
cancelDeposit f-ESG	3	2
cancelWithdraw f-ESG	3	2
overdraft f-ESG	6	6
credit f-ESG f-ESG	5	5
dailyLimit f-ESG	6	7
interest f-ESG	5	5
interestEstimation f-ESG	4	3
TOTAL	45	42

The six products of the Bank Account SPL are demonstrated in Table 7.17 in order to present their complexity. Since the base product has the least number of features, it has the least number of vertices and edges. The total number of vertices is sixty-eight and the total number of edges is seventy-six of the isolated products in this SPL as it is shown in Table 7.17. This result shows that one has to deal with many more vertices and edges in isolated product EGSs which makes updating and maintaining the models more difficult.

Table 7.17. Isolated Product ESGs of Bank Account SPL

Isolated Product ESGs of Bank Account SPL	**Number of Vertices**	**Number of Edges**
baseProdcuct ESG	7	8
cancellable ESG	9	10
credit ESG	17	16
dailyLimit ESG	11	14
interest ESG	12	12
overdraft ESG	12	16
TOTAL	68	76

The FESG models and isolated products are compared in terms of total number of vertices and total number of edges in Table 7.18. Similar to other case studies, the total number of vertices and edges of FESGs are less than the isolated products since there are no repeating vertices and edges in FESGs.

Table 7.18. Model Comparison of Bank Account SPL and Isolated Products

	Number of Vertices	**Number of Edges**
FESGs of Bank Account SPL	45	42
Isolated Product ESGs of Bank Account SPL (6 Product ESGs)	68	76

The CESs of the six products of Bank Account SPL are obtained by using ESG test generation approach, full test sequence composition approach and incremental test sequence composition approach for event pairs, triples and quadruples. The results are shown in terms of execution time in milliseconds, number of CESs and the total number of events.

Table 7.19. Test Generation Time of Isolated Product ESGs of Bank Account SPL

Test Generation Time (ms)	**Coverage Length 2**			**Coverage Length 3**			**Coverage Length 4**		
	Min	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
baseProduct ESG	**39**	**41.79**	**40.25**	42.58	48.02	44.43	**48.48**	**56.54**	**50.59**
cancellable ESG	38.89	40.77	39.85	**45.53**	**48.42**	**46.84**	46.07	48.97	47.92
credit ESG	**44.18**	**49.95**	**45.64**	47.22	54.04	49.41	**47.34**	**50.24**	**48.78**
dailyLimit ESG	41.26	48.13	43.31	**46.67**	**49.14**	**47.96**	49.5	54.38	51.27
interest ESG	**41.92**	**48.41**	**43.79**	44.71	47.82	46.62	**43.8**	**48.7**	**45.64**
overdraft ESG	44.17	48.5	45.26	**47.59**	**53.16**	**49.38**	50.9	56.43	52.81

In Table 7.19, the test generation time of isolated product ESGs are demonstrated for coverage length 2, 3 and 4. Note that, for coverage length 2 event pairs, for coverage length 3 event triples, and, for coverage length 4 event quadruples are covered in CESs. It could be resulted from the Table 7.19 that, as the coverage length increases, the test generation time also increases. This stems from the case that as the coverage length increases, the test sets become larger in general. This deduction

could be obtained from Figure 7.20 which shows us as the coverage length increases, the number of events increases except for *base product* and *interest product*.

Table 7.20. Full Test Sequence Composition Time of FESGs

Test Sequence Composition Time (ms)	**Coverage Length 2**			**Coverage Length 3**			**Coverage Length 4**		
	Min	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
baseProduct FESG	**40.44**	**44.4**	**42.38**	48.48	56.54	50.59	**48.48**	**56.54**	**50.59**
cancellable FESG	42.18	47.81	44.29	**51.46**	**56.53**	**53.35**	59.2	63.82	60.82
credit FESG	**44.6**	**46.68**	**45.63**	57.3	58.91	58.09	**76.07**	**84.28**	**79.72**
dailyLimit FESG	45.68	47.85	46.55	**56.47**	**59.01**	**57.71**	80.4	88.17	83.49
interest FESG	**42.42**	**43.67**	**43.12**	53.32	55.96	54.32	**58.14**	**61.38**	**60.04**
overdraft FESG	46.2	50	47.75	**59.31**	**61.84**	**60.73**	90.1	97.24	94.13

Full test sequence composition time of FESGs are given in Table 7.20 for coverage length 2,3 and 4. Similar to the ESG test generation approach, as the coverage length increases the test sequence composition time also increases with the test set size except for *base product* and *interest product*.

Table 7.21. Complete Event Sequences of Full Test Sequence Composition FESGs

Test Sequence Composition CESs	**Coverage Length 2**		**Coverage Length 3**		**Coverage Length 4**	
	# of CESs	**# of Events**	**# of CESs**	**# of Events**	**# of CESs**	**# of Events**
baseProduct FESG	3	15	6	39	5	32
cancellable FESG	5	19	12	61	14	69
credit FESG	9	29	17	69	19	86
dailyLimit FESG	8	30	17	76	23	113
interest FESG	5	21	17	76	9	48
overdraft FESG	8	36	18	81	24	116

When we compare the results of ESG test generation with full test sequence composition approach, we see that full test sequence composition approach performs similar to the previous case studies in which the resulting test sets are larger and the execution time of algorithms are much more.

Incremental test sequence composition has the best performance comparing with the other two approaches, similar to the previous case studies. The results show that, the CES set is larger than full test sequence composition approach test sets, approximately %42 when coverage length is 2, %155 when coverage length is 3 and %134 when coverage length is 4. Therefore, with minimal execution times, severely greater test sets could be obtained by exploiting incremental test sequence composition approach.

Table 7.22. Incremental Test Sequence Composition Time of FESGs

Incremental Test Sequence Composition Time (ms)			**Coverage Length 2**			**Coverage Length 3**			**Coverage Length 4**		
FESG to be reused	**f-ESG(s) to be added**	**Obtained FESG**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
baseProduct FESG	cancelDeposit cancelWithdraw	cancellable FESG	**2.87**	**3.56**	**3.24**	9.59	11	10.3	**17.3**	**20.7**	**19**
baseProduct FESG	interest interestEstimation	interest FESG	1.57	5.27	2.3	**6.61**	**8.22**	**7.62**	10.74	15.1	12.5
baseProduct FESG	cancelDeposit cancelWithdraw dailyLimit	dailyLimit FESG	**3.77**	**5.96**	**4.65**	11.41	16.56	14.3	**25.6**	**56.7**	**31.6**
baseProduct FESG	cancelWithdraw overdraft dailyLimit	overdraft FESG	2.87	4.27	3.42	**14**	**21.7**	**17.6**	33.32	43.42	38.1
baseProduct FESG	cancelDeposit cancelWithdraw credit interest interestEstimation	credit FESG	**3.46**	**4.75**	**3.89**	18.48	23.27	20.9	**39.1**	**54.6**	**47.2**
cancelable FESG	dailyLimit	dailyLimit FESG	4.25	4.67	4.4	**5.8**	**8.64**	**6.56**	15.17	20.08	17.7
cancelable FESG	credit interest interestEstimation	credit FESG	**4.46**	**4.95**	**4.7**	14.96	16.87	16	**30.5**	**36.6**	**32**
interest FESG	credit	credit FESG	2.96	3.53	3.21	**7.08**	**7.6**	**7.28**	10.64	14.92	12.1

Table 7.23. Incremental Test Sequence Composition CESs

Incremental Test Sequence Composition CESs			**Coverage Length 2**		**Coverage Length 3**		**Coverage Length 4**	
FESG to be reused	**f-ESG(s) to be added**	**Obtained FESG**	**# of CESs**	**# of Events**	**# of CESs**	**# of Events**	**# of CESs**	**# of Events**
baseProduct FESG	cancelDeposit f-ESG cancelWithdraw f-ESG	cancellable FESG	5	19	11	63	13	79
baseProduct FESG	interest f-ESG interestEstimation f-ESG	interest FESG	5	21	12	61	11	54
baseProduct FESG	cancelDeposit f-ESG cancelWithdraw f-ESG dailyLimit f-ESG	dailyLimit FESG	8	30	18	104	21	134
baseProduct FESG	cancelWithdraw f-ESG overdraft f-ESG dailyLimit f-ESG	overdraft FESG	9	36	19	114	20	121
baseProduct FESG	cancelDeposit f-ESG cancelWithdraw f-ESG credit f-ESG interest f-ESG interestEstimation f-ESG	credit FESG	9	29	20	86	19	95
cancelable FESG	dailyLimit f-ESG	dailyLimit FESG	8	30	19	102	28	157
cancelable FESG	credit f-ESG interest f-ESG interestEstimation f-ESG	credit FESG	9	29	22	91	25	110
interest FESG	credit f-ESG	credit FESG	7	25	14	64	15	66

7.4. Student Attendance System SPL

The Student Attendance System, short for SAS, is a software product line which twenty-four concrete features. The Student Attendance System allows the users to submit attendance, to manage class details and class schedule, to monitor and update attendance records and to receive notifications. The feature diagram of this SPL is given in Figure 7.26 and it has six abstract mandatory features. These abstract features group other features as XOR or OR groups. Making these abstract features mandatory guarantees that at least one of their sub-features will be included in each product configuration within the SPL.

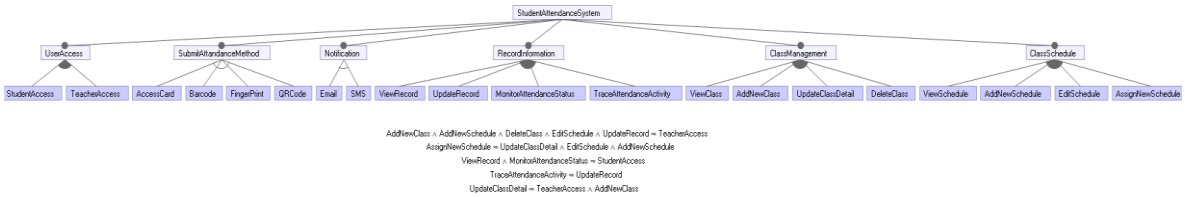

Figure 7.26. Student Attendance System SPL feature diagram

The number of possible product variations is two hundred forty-eight whenever at least one sub-feature of mandatory features is added. The product matrix which is given in Figure 7.27 demonstrates only the eleven of these configurations.

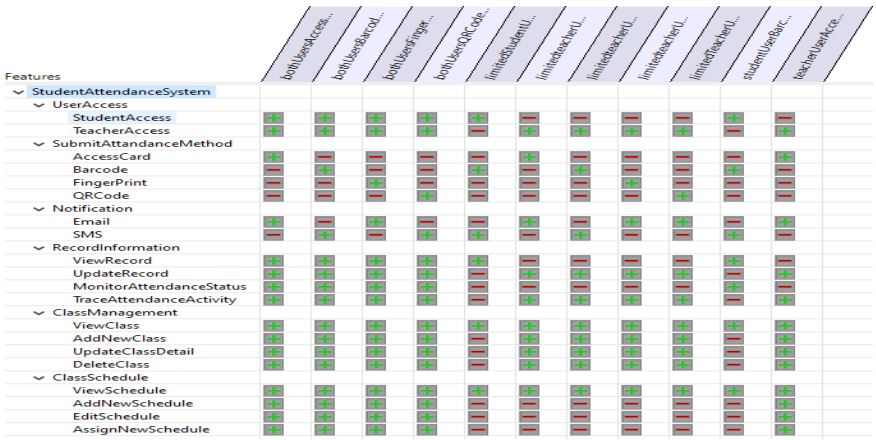

Figure 7.27. Product Matrix of the Student Attendance System SPL

7.4.1. Student Attendance System SPL Models

The core of the SAS is given in Figure 7.28. The c-ESG of this is SPL is not a connected ESG and the events within this c-ESG will become connected via feature connections. The "*submit attendance*", "*confirm your identity*", "*log in*", "*open notification settings*" and "*confirm notification settings*" are reusable core events in the models of this SPL.

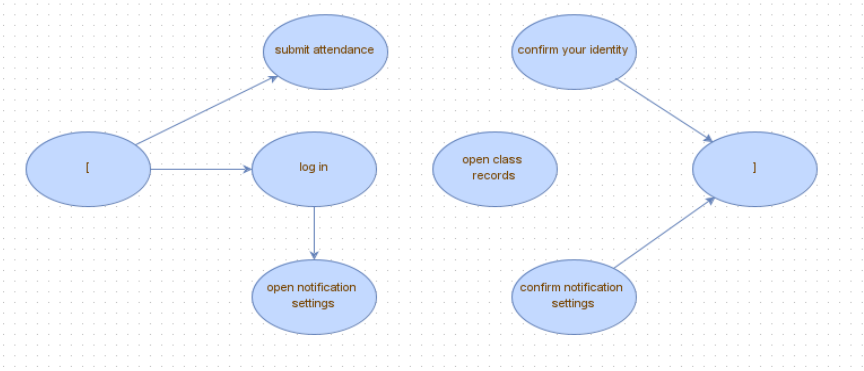

Figure 7.28. c-ESG of SAS SPL

In the Student Attendance System, there are two types of users which are student and teacher. The products within this SPL could be configured for the student user, the teacher user or both of the users. The *userAccess* abstract feature groups *studentAccess* and *teacherAccess* features which represent user credentials. The *teacherAccess* f-ESG is given in Figure 7.29 and it gives a user who is a teacher the permission to edit the data stored in the system such as editing the class details, class schedule or student attendance records.

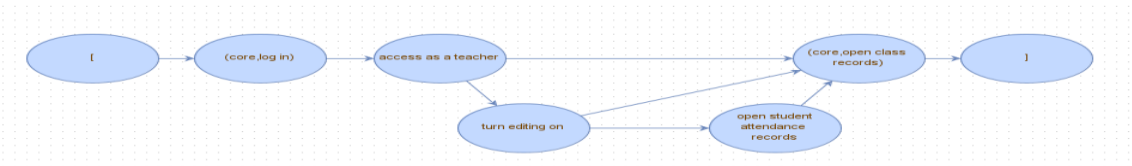

Figure 7.29. *teacherAccess* f-ESG of SAS SPL

The users who are students in this system, could monitor class, schedule and attendance records. However, they do not have the authorization to update classes, class schedules and attendance records. The f-ESG of *studentAccess* feature is given in Figure 7.30.

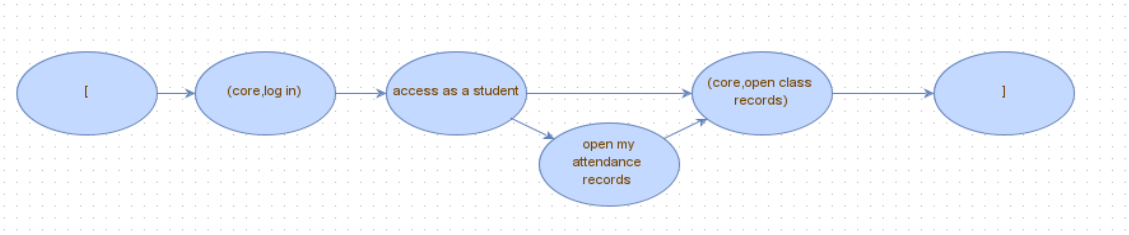

Figure 7.30. *studentAccess* f-ESG of SAS SPL

The attendance could be submitted to the system via an access card, a barcode, a fingerprint or a QR code. Therefore, the attendance methods are gathered under the abstract mandatory feature *SubmitAttendanceMethod* in an XOR relation. This means that only one submission could be selected in the product configurations of this system. The f-ESGs of these submission methods contain connection points only to c-ESG and they demonstrated in APPENDIX D

The users of this system could receive notifications via email or SMS. Hence, the *email* and *SMS* features are alternative features under *Notification* abstract mandatory features. Likewise to attendance submission methods, the f-ESGs of these features have connection points only to c-ESG and are given in APPENDIX D.

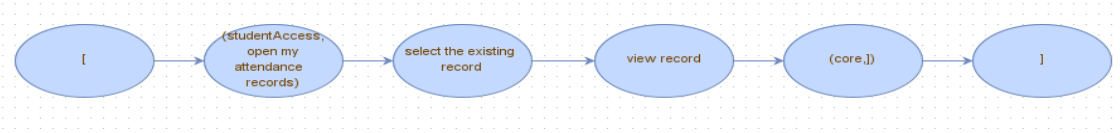

Figure 7.31. *viewRecord* f-ESG of SAS SPL

The *viewRecord*, *updateRecord*, *monitorAttendanceStatus* and *traceAttendanceActivity* are gathered under *RecordInformation* abstract mandatory

feature. The *viewRecord* feature which is depicted in Figure 7.31 and *monitorAttendanceStatus* feature which is given in Figure 7.32 requires *studentAccess* feature due to the connection points. These features help the students to view attendance records and monitor their current attendance status.

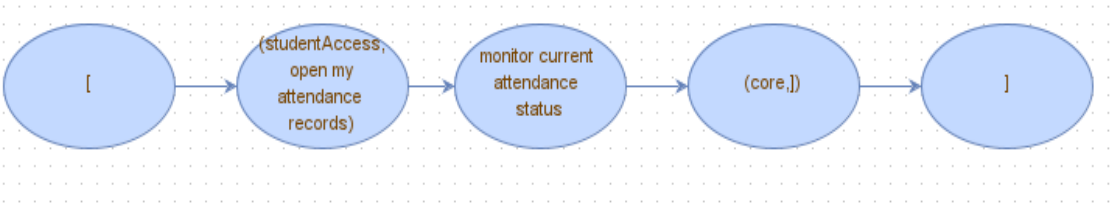

Figure 7.32. *monitorAttendanceStatus* f-ESG of SAS SPL

The *updateRecord* feature implies *teacherAccess* feature and the *traceAttendanceActivity* feature implies *updateRecord* feature. Therefore, only the teachers could update a student's or a class of students' attendance records. The f-ESG of *updateRecord* is shown in Figure 7.33. Also, the *traceAttendanceActivity* which is given in Figure 7.34 feature allow a teacher to trace the attendance activity of a class of students.

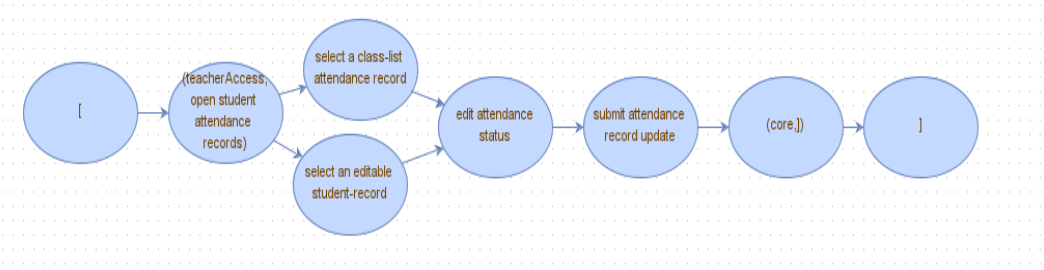

Figure 7.33. *updateRecord* f-ESG of SAS SPL

The *ClassManagement* abstract mandatory feature groups *viewClass, addNewClass, updateClassDetail,* and, *deleteClass* features and the *ClassSchedule* abstract mandatory feature groups *viewSchedule, addNewSchedule, editSchedule* and

assignNewSchedule in which the features *viewClass* and *viewSchedule* have no user credentials and no requirements to other features which are shown in APPENDIX D.

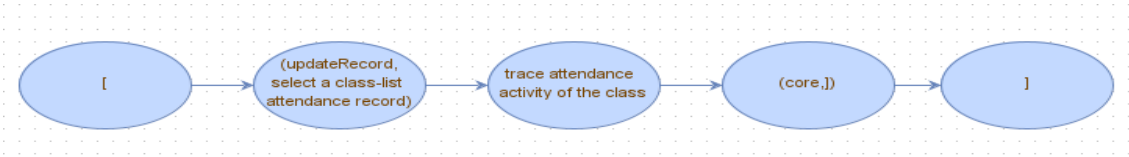

Figure 7.34. *traceAttendanceActivity f*-ESG of SAS SPL

The rest of the features except *viewClass* and *viewSchedule*, are accessible only by teachers. Since their names are self-explanatory, e.g., *addNewClass* feature enable adding a new class behavior of the system, *addNewClass* feature enable adding a new schedule behavior of the system, and so on, they are also given in APPENDIX D except for *assignNewSchedule* feature.

The *assignNewSchedule* feature allows a teacher to assign schedule to an editable class by selecting an existing schedule or by adding new schedule to it. Hence, it requires *updateClassDetail, addNewSchedule* and *editSchedule* features. The f-ESG of the *assignNewSchedule* feature is shown in Figure 7.35.

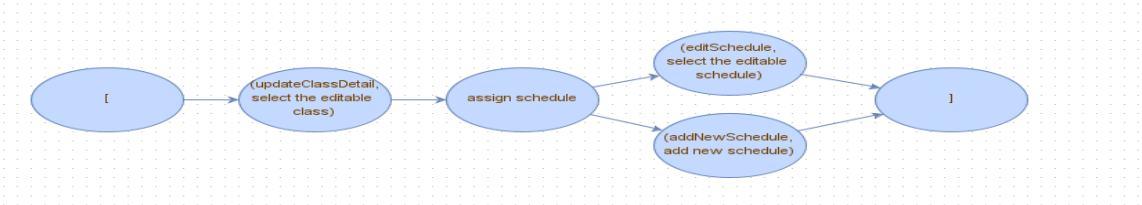

Figure 7.35. *assignNewSchedule* f-ESG of SAS SPL

The complete product ESG of one of the product configurations in product matrix (Figure 7.27) is given in Figure 7.36 which enables teacher access, access card attendance submission and email notification. The rest of the product ESGs are given in APPENDIX D.

Additionally, the complete product ESG in Figure 7.36 has only fourteen features including core. The Figure 7.36 may help the readers to imagine how modelling and maintaining the complete product ESGs become difficult as the number of features increases and the product configurations gets more complex. As a consequence, employing FESGs in SPL testing that the products have great number of features is an advantage.

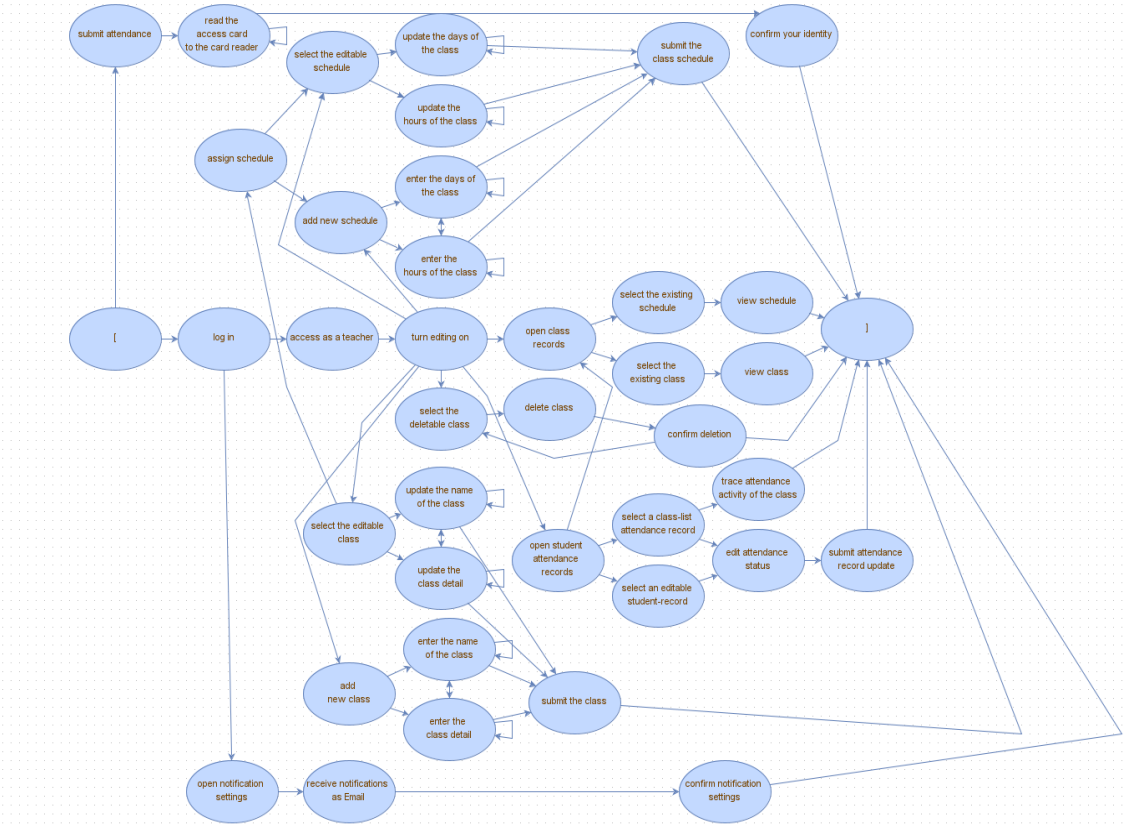

Figure 7.36. ESG of SAS SPL – *teacher user-access card-email product*

7.4.2. Student Attendance System SPL Results

The Student Attendance System results are depicted in this section. Table 7.24 shows the FESG components which contains the c-ESG and the f-ESGs and their corresponding number of vertices and edges. The c-ESG is a disconnected ESG except for two vertices, therefore it has only one edge. The total number of vertices is eighty-

seven and the total number of edges is ninety-two of the components in this SPL as it is shown in Table 7.24. Since this SPL has twenty features including c-ESG, it is a much larger example.

Table 7.24. SAS SPL FESG Components

SPL FESG Components	**Number of Vertices**	**Number of Edges**
c-ESG	8	1
barcode f-ESG	4	6
fingerPrint f-ESG	3	3
accessCard f-ESG	3	3
QRCode f-ESG	3	3
studentAccess f-ESG	4	4
teacherAccess f-ESG	5	6
viewRecord f-ESG	4	3
updateRecord f-ESG	6	6
monitorAttendanceStatus f-ESG	3	2
traceAttendanceActivity f-ESG	3	2
email f-ESG	3	2
SMS f-ESG	3	2
addNewClass f-ESG	6	10
updateClassDetail f-ESG	5	9
deleteClass f-ESG	5	5
viewSchedule f-ESG	4	3
addNewSchedule f-ESG	6	10
editSchedule f-ESG	5	9
assignNewSchedule f-ESG	4	3
TOTAL	87	92

Ten of the product variants of SAS SPL are given in Table 7.25. Since these are large products with at least six features, the total number of vertices and edges in isolated products are a lot bigger than the previous case study. The FESG models and isolated products are compared in terms of total number of vertices and total number of edges in Table 7.26. Similar to previous case studies, the total number of vertices and edges of FESGs are less than the isolated products since there are no repeating vertices and edges in FESGs.

Table 7.25. Isolated Product ESGs of SAS SPL

Isolated Product ESGs of SVM SPL	**Number of Vertices**	**Number of Edges**
studentUserBarcodeSMS ESG	18	20
teacherUserAccessCardEmail ESG	38	62
limitedStudentUserBarcodeSMS ESG	17	19
limitedTeacherUserAccessCardEmail ESG	30	43
limitedTeacherUserFingerprintEmail ESG	30	43
limitedTeacherUserQRCodeSMS ESG	30	43
bothUsersAccessCardEmail ESG	43	68
bothUsersBarcodeSMS ESG	44	71
bothUsersFingerPrintEmail ESG	43	68
bothUsersQRCodeSMS ESG	43	68
TOTAL	336	505

Table 7.26. Model Comparison of SAS SPL FESGs and Isolated Products

	Number of Vertices	**Number of Edges**
FESGs of SAS SPL	87	92
Isolated Product ESGs of SAS SPL (10 ESGs)	336	505

The CESs of the product variants of SAS SPL are obtained by using ESG test generation approach, full test sequence composition approach and incremental test sequence composition approach for event pairs, triples and quadruples similar to the other case studies in this thesis. Also, the results are demonstrated in terms of execution time in milliseconds, number of CESs and the total number of events.

Table 7.27. Test Generation Time of Isolated Product ESGs of SAS SPL

Test Generation Time (ms)	**Coverage Length 2**			**Coverage Length 3**			**Coverage Length 4**		
	Min	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
studentUserBarcodeSMS ESG	**42.19**	**44.98**	**43.16**	46.02	49.43	47.4	**48.39**	50.38	**49.18**
teacherUserAccessCardEmail ESG	52.27	59.29	55.01	**78.11**	**88.32**	**82.21**	117.3	153.8	133
limitedStudentUserBarcodeSMS ESG	**44.44**	**47.54**	**46.13**	48.06	66.07	54.3	**50.69**	**62.6**	**53.83**
limitedTeacherUserAccessCardEmail ESG	53.03	58.23	54.63	**64.67**	**70.93**	**67.42**	88.22	96.11	91.29
limitedTeacherUserFingerprintEmail	**49.24**	**54.42**	**52.67**	43	54.42	49.83	**87.63**	**104.7**	**93.76**
limitedTeacherUserQRCodeSMS ESG	49.54	66.59	53.14	**64.23**	**72.06**	**67.64**	84.75	112.5	93.88
bothUsersAccessCardEmail ESG	**59.47**	**67.48**	**61.87**	82.56	105.1	94.65	**130.1**	**171**	**141.9**
bothUsersBarcodeSMS ESG	63.82	70.19	65.86	**88.11**	**105.2**	**96.8**	133.3	154.5	141
bothUsersFingerPrintEmail ESG	**61.73**	**67.17**	**64.09**	83.92	99.98	93.34	**125.8**	**145**	**134.3**
bothUsersQRCodeSMS ESG	58.45	63.22	60.12	**87.54**	**106.7**	**96.22**	122.3	148.1	135.9

Table 7.28. Complete Event Sequences of Isolated Product ESGs of SAS SPL

Test Generation CESs	Coverage Length 2		Coverage Length 3		Coverage Length 4	
	# of CESs	# of Events	# of CESs	# of Events	# of CESs	# of Events
studentUserBarcodeSMS ESG	8	35	12	55	14	66
teacherUserAccessCardEmail ESG	16	116	33	247	64	520
limitedStudentUserBarcodeSMS ESG	7	31	11	51	13	62
limitedTeacherUserAccessCardEmail ESG	12	82	23	163	40	312
limitedTeacherUserFingerprintEmail	12	82	23	163	40	312
limitedTeacherUserQRCodeSMS ESG	12	82	23	163	40	312
bothUsersAccessCardEmail ESG	19	130	37	266	68	569
bothUsersBarcodeSMS ESG	21	137	40	278	72	557
bothUsersFingerPrintEmail ESG	19	130	37	266	68	569
bothUsersQRCodeSMS ESG	19	130	37	266	68	569

In Table 7.27, the test generation time of isolated product ESGs are demonstrated for coverage length 2, 3 and 4. The test generation time increases, as the coverage length increases. For SAS SPL, the test sets become larger as the coverage length increases without any exceptions, therefore, the test generation time increases accordingly.

Table 7.29. Full Test Sequence Composition Time of FESGs of SAS SPL

Test Sequence Composition Time (ms)	Coverage Length 2			Coverage Length 3			Coverage Length 4		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
studentUserBarcodeSMS FESG	**50.35**	**56.41**	**52.9**	55.95	58.17	56.96	**73.27**	**87.99**	**79.59**
teacherUserAccessCardEmail FESG	51.49	57.78	54.39	**92.66**	**100.8**	**97.1**	176	189.1	182.5
limitedStudentUserBarcodeSMS FESG	**48.55**	**51.42**	**49.99**	55.51	59.4	57.12	**65.75**	**73.37**	**69.49**
limitedTeacherUserAccessCardEmail	48.61	49.93	49.38	**74.65**	**79.93**	**78.1**	132.1	144.9	137.3
limitedTeacherUserFingerprintEmail FESG	**48.21**	**52.58**	**49.62**	76.85	86.48	81.09	**130.9**	**146.5**	**137.4**
limitedTeacherUserQRCodeSMS FESG	48.56	51.52	50.13	**74.37**	**84.47**	**81.12**	121.4	145	133.4
bothUsersAccessCardEmail FESG	**52.23**	**54.82**	**53.57**	92.82	102.4	99.68	**179.4**	**196.8**	**187.8**
bothUsersBarcodeSMS FESG	52.77	62.76	58.39	**101.4**	**114.7**	**109.2**	178.1	196.9	189.2
bothUsersFingerPrintEmail FESG	**52.21**	**56.5**	**53.87**	96.64	107.8	103.2	**178.7**	**196.5**	**190**
bothUsersQRCodeSMS FESG	52.96	56.51	54.72	**98.59**	**111.8**	**105.1**	52.96	111.8	79.96

Table 7.29 presents the results of full test sequence composition time of FESGs for coverage length 2,3 and 4. For coverage length 2, full test sequence composition performs better than the ESG test generation approach in terms of both execution time

and test set size, unlike the previous case studies. This could stem from the number of features and bigger size products of SAS SPL. Since the full test sequence composition approach generates each f-ESG's test sequences from scratch and composes them, for small SPLs like previous examples, generating each f-ESG's test sequences from scratch could not make it faster comparing the other examples. However, as the number of features increases and the isolated product ESGs get more complex, the exact speed performance of full test sequence composition could be understood.

Table 7.30. Complete Event Sequences of Full Test Sequence Composition FESGs

Test Sequence Composition CESs	Coverage Length 2		Coverage Length 3		Coverage Length 4	
	# of CESs	# of Events	# of CESs	# of Events	# of CESs	# of Events
studentUserBarcodeSMS FESG	8	35	14	66	17	81
teacherUserAccessCardEmail FESG	19	139	44	319	89	692
limitedStudentUserBarcodeSMS FESG	7	31	13	62	16	77
limitedTeacherUserAccessCardEmail	13	87	28	191	46	344
limitedTeacherUserFingerprintEmail FESG	13	87	28	191	46	344
limitedTeacherUserQRCodeSMS FESG	13	87	28	191	46	344
bothUsersAccessCardEmail FESG	23	159	53	366	95	745
bothUsersBarcodeSMS FESG	25	166	56	378	99	769
bothUsersFingerPrintEmail FESG	23	159	53	366	95	745
bothUsersQRCodeSMS FESG	23	159	53	366	95	745

When comparing full test sequence composition approach with ESG test generation approach, for coverage length 3 and 4, full test sequence composition performs similar to the previous case studies, as the coverage length increases the test sequence composition time also increases with the test set size.

Incremental test sequence composition has the best performance comparing with the other two approaches, similar to the previous case studies. The results show that, the CES set is larger than full test sequence composition approach test sets, nearly %16 when coverage length is 2, %30 when coverage length is 3 and %27 when coverage length is 4. Thus, greater test sets could be generated in drastically smaller times in incremental test sequence composition approach, comparing to other two approaches.

Table 7.31. Incremental Test Sequence Composition Time of FESGs

Incremental Test Sequence Composition Time (ms)			Coverage Length 2			Coverage Length 3			Coverage Length 4		
FESG to be reused	**f-ESG(s) to be added**	**Obtained FESG**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**
limitedStudentUserBarcodeSMS FESG	monitorAttendanceStatus f-ESG	studentUserBarcodeSMS FESG	2.73	7.76	3.761	5.85	6.44	6.033	11.03	16.07	13.15
limitedStudentUserBarcodeSMS FESG	teacherAcccess f-ESG updateRecord f-ESG traceAttendanceActivity f-ESG addNewClass f-ESG updateClassDetail f-ESG deleteClass f-ESG addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG monitorAttendanceStatus f-ESG	bothUsersBarcodeSMS FESG	7.98	14.84	10.15	74.68	99.23	85.36	194.5	213.2	205.6
studentUserBarcodeSMS FESG	teacherAcccess f-ESG updateRecord f-ESG traceAttendanceActivity f-ESG addNewClass f-ESG updateClassDetail f-ESG deleteClass f-ESG addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	bothUsersBarcodeSMS FESG	12.03	14.49	12.42	69.05	85.13	79.42	211.3	219.3	214.6
limitedTeacherUserAccessCardEmail FESG	addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	teacherUserAccessCardEmail FESG	4.77	5.38	5.026	28.7	37.85	31.59	85.19	98.3	91.59
limitedTeacherUserAccessCardEmail FESG	studentAccess viewRecord monitorAttendanceStatus addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	bothUsersAccessCardEmail FESG	4.18	6.98	4.612	26.88	33.78	30.29	85.65	116.3	102.2
teacherUserAccessCardEmail	studentAccess f-ESG viewRecord monitorAttendanceStatus	bothUsersAccessCardEmail	4.08	4.93	4.29	14.99	18.45	17.19	31.89	41.71	36.23
limitedTeacherUserFingerprintEmail FESG	studentAccess viewRecord monitorAttendanceStatus addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	bothUsersFingerprintEmail FESG	6.24	8.54	7.871	38.21	50.42	42.3	11.7	116.9	103.4
limitedTeacherUserQRCodeSMS FESG	studentAccess viewRecord monitorAttendanceStatus addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	bothUsersQRCodeSMS FESG	6.15	8.48	7.91	39.3	46.4	41.95	110.3	118.8	114

Table 7.32. Incremental Test Sequence Composition CESs of FESGs

Incremental Test Sequence Composition CESs			Coverage Length 2		Coverage Length 3		Coverage Length 4	
FESG to be reused	**f-ESG(s) to be added**	**Obtained FESG**	**# of CESs**	**# of Events**	**# of CESs**	**# of Events**	**# of CESs**	**# of Events**
limitedStudentUserBarcodeSMS FESG	monitorAttendanceStatus f-ESG	studentUserBarcodeSMS FESG	8	35	16	77	22	109
limitedStudentUserBarcodeSMS FESG	teacherAcccess f-ESG updateRecord f-ESG traceAttendanceActivity f-ESG addNewClass f-ESG updateClassDetail f-ESG deleteClass f-ESG addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG monitorAttendanceStatus f-ESG	bothUsersBarcodeSMS FESG	25	166	60	409	117	865
studentUserBarcodeSMS FESG	teacherAcccess f-ESG updateRecord f-ESG traceAttendanceActivity f-ESG addNewClass f-ESG updateClassDetail f-ESG deleteClass f-ESG addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	bothUsersBarcodeSMS FESG	8	35	58	398	115	855
limitedTeacherUserAccessCardEmail FESG	addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	teacherUserAccessCardEmail FESG	19	139	52	379	104	825
limitedTeacherUserAccessCardEmail FESG	studentAccess viewRecord monitorAttendanceStatus addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	bothUsersAccessCardEmail FESG	23	159	61	426	115	883
teacherUserAccessCardEmail	studentAccess f-ESG viewRecord monitorAttendanceStatus	bothUsersAccessCardEmail FESG	23	159	53	366	103	797
limitedTeacherUserFingerprintEmail FESG	studentAccess viewRecord monitorAttendanceStatus addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	bothUsersFingerprintEmail FESG	23	159	61	425	116	879
limitedTeacherUserQRCodeSMS FESG	studentAccess viewRecord monitorAttendanceStatus addNewSchedule f-ESG editSchedule f-ESG assignNewSchedule f-ESG	bothUsersQRCodeSMS FESG	23	159	61	425	116	879

In this chapter, four case studies are presented and experimented under ESG-based test generation approach, FESG-based full test sequence composition approach and FESG-based incremental test sequence composition approach. Also, the results are demonstrated in several tables. In the next chapter, the conclusion and future work of this thesis are presented.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this chapter, the thesis work is concluded, and the future work which are planned as feature interactions, alternative course of events in features and feature removal are discussed.

8.1. Conclusion

In this thesis, Featured Event Sequence Graphs (FESGs) are introduced as variable testing models which are used to explicitly represent the variability of behaviors in SPLs. A FESG is constituted from the core ESG (c-ESG) which models the core behavior of the SPL and feature ESGs (f-ESGs) which model selectable features' behaviors and enables variability in SPL. In order to express a particular product within the SPL, the c-ESG and selected f-ESGs are combined into a FESG. Moreover, this thesis presents two different model-based test sequence generation approach for software product lines which are test sequence composition and incremental test sequence composition. Both of the approaches exploit FESGs as reusable test models. The incremental test sequence composition approach reuses the existing test sequences as well.

The test sequence composition approach introduces Algorithm 4.1 *Construction of product FESG lattice* to construct a product FESG lattice and Algorithm 4.2. *Composition of product test sequences* to compose partial test sequences of f-ESGs through the usage of product FESG lattice. This approach resembles divide-and-conquer strategy in the sense that each simple test models' partial test sequences are generated and composed from scratch in order to obtain a product's test sequences. This approach is more efficient than the traditional test generation of full-ESG model since the full-ESG model

is more complex in terms of number of vertices and edges. Also, traceability and maintainability of FESG models are easier.

The incremental test sequence composition approach is considered as an enhanced version of aforementioned test sequence composition approach. This approach introduces Algorithm 5.1. *Incremental Sequence Generation* which compose the test sequences of additional features with the existing test sequences of product variant. Algorithm 5.2. *Incremental Transformation of f-ESGs*, Algorithm 5.3. *Sequence Generation of Transformed f-ESGs* and Algorithm 5.4. *Sequence Composition* are introduced in the approach since they formalize incremental transformation of f-ESGs, sequence generation of transformed f-ESGs and sequence composition of generated sequences, respectively. The difference of this approach is that not only the test models but also the existing test cases are reusable. Therefore, whenever a new feature (or features) is used to increment an existing variant, the generated test sequences of variant and the new feature's test cases are composed. Since the existing test sequences are reused, the composition is not realized from scratch. Thus, incremental test sequence composition approach is more efficient than the compositional approach.

8.2. Future Work

The test sequence composition and incremental test sequence composition are based on considering a feature as an increment in functionality of the product and each product as a peerless feature-combination. One of the major problems in feature-based development is consideration of feature interactions where features could replace existent features (Uzuncaova, Khurshid, and Batory 2010). A feature not only extends but also replaces the existent constraint, whenever a feature interaction occurs (Uzuncaova, Khurshid, and Batory 2010). This means that the behavior of one or both of the features which are integrated in a product would be modified by each other. Two forms of featuıre interactions are introduced: intentional and unintentional feature interactions (Benz 2009). A thesis work which focuses on test generation for feature interactions and takes automative information system of a 2009 model BMW vehicle which combines a multimedia player, navigation system, telephony, telematic services,

and internet access as an example. This work gives the situation when the CD playback is interfered by an incoming phone call as an intentional feature interaction example(Benz 2009). Also, the situation when the increased bus load delays messages between graphical user interface (GUI) of the automative information system (AIS) and the navigation system of the AIS is given as an unintentional feature interaction between the feature seat adjustment and feature navigation (Benz 2009).

Since unintentional feature interactions are foundations of a fault model and are more difficult to predict and/or detect (Benz 2009), especially unintentional feature interactions could be an interesting future research because features transform a characteristic of a system for example an existing constraint is replaced with another and for this reason, it disrupts the monotonic increment in feature composition. In such cases, our full test sequence composition approach and incremental test sequence composition approach cannot be applied directly because of the existence of the feature requirement constraints that originates from connection points. As a consequence, feature interactions could be an intriguing topic for our future work.

In addition to the feature interactions, depending on the product domain, whenever a feature is connected between two events of core (or another feature), it sometimes brings an alternative course of events between corresponding two events; and sometimes, it becomes the only course of events between the two by removing the edge between them. Since this property comes from the domain knowledge and depends on the model, it could be interesting to try to solve this issue on the modelling level.

Finally, the feature removal could be suggested as a future work for this study. In feature removal, the features that have connection points only to the pseudo start and pseudo finish events of the c-EGS, could be removed directly because the CESs of the feature to be removed does not affect any other sequence of the product. The *interest* feature of Bank Account SPL that is given in Figure 7.23 is a good example to the direct feature removal. However, the features that have connection points to the features that could be removed directly, should also be removed with these features such as *interestEstimation* feature which has a connection point to *interest* feature and is given in Figure 7.24. For other feature removal cases such that the connection points to the core events and/or other features' events, the CESs of the feature to be removed should be decided and then removed. At this point, it is important that if the removed feature's CESs are in-between the remaining events and there is no alternative edge therefore, no CESs between these remaining events, the CES should be added to the test suite after

removal. For example, let's focus on the *daily limit product* of Bank Account SPL which is demonstrated in Figure 4.5. When we remove the *daily limit* feature from this product, the "*enter withdraw amount, confirm daily limit excess, enter withdraw amount, confirm daily limit excess*" event sequence of the CES "*select withdraw, enter withdraw amount, confirm daily limit excess, enter withdraw amount, confirm daily limit excess, cancel withdraw*" is removed and the "*select withdraw, cancel withdraw*" CES is decided to be added to the test suite of the remaining product. However, since, there is an edge between *select withdraw* and *cancel withdraw* events, this CES is already on the test suite and a duplication occurs and the decision is not applied. Now, assume that there is no edge between *select withdraw* and *cancel withdraw* events. The "*select withdraw, cancel withdraw*" CES should be added to the test suite of the remaining product. Consequently, the feature removal operation could be defined as future work and could be examined, researched and experimented elaborately.

REFERENCES

Apel, Sven, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Science & Business Media.

Apel, Sven, and Delesley Hutchins. 2010. “A Calculus for Uniform Feature Composition | ACM Transactions on Programming Languages and Systems.” *ACM Transactions on Programming Languages and Systems* 32 (5): 19.

Belli, Fevzi. 2001. “Finite State Testing and Analysis of Graphical User Interfaces.” In *Proceedings 12th International Symposium on Software Reliability Engineering*, 34–43. Washington, DC: IEEE Comput. Soc. https://doi.org/10.1109/ISSRE.2001.989456.

Belli, Fevzi, and Christof J. Budnik. 2004. “Minimal Spanning Set for Coverage Testing of Interactive Systems.” In *Theoretical Aspects of Computing - ICTAC 2004*, edited by Zhiming Liu and Keijiro Araki, 220–234. Guiyang, China: Springer Berlin Heidelberg.

Belli, Fevzi, and Christof J Budnik. 2005. “Test Cost Reduction for Interactive Systems.” In *Sicherheit 2005*, 12. Regensburg, Germany.

Belli, Fevzi, and Christof J. Budnik. 2007. “Test Minimization for Human-Computer Interaction.” *Applied Intelligence* 26 (2): 161–74. https://doi.org/10.1007/s10489-006-0008-0.

Belli, Fevzi, Christof J. Budnik, and Lee White. 2006. “Event-Based Modelling, Analysis and Testing of User Interactions: Approach and Case Study.” *Software Testing, Verification and Reliability* 16 (1): 3–32. https://doi.org/10.1002/stvr.335.

Belli, Fevzi, Nevin Guler, and Michael Linschulte. 2011. “Does ‘Depth’ Really Matter? On the Role of Model Refinement for Testing and Reliability.” In *2011 IEEE 35th Annual Computer Software and Applications Conference*, 630–39. Munich, Germany: IEEE. https://doi.org/10.1109/COMPSAC.2011.17.

Belli, Fevzi, Nimal Nissanke, Christof J Budnik, and Aditya Mathur. 2005. “Test Generation Using Event Sequence Graphs.” University of Paderborn, Institute for Electrical Engineering and Information Technology.

Benz, Sebastian. 2009. “Generating Tests for Feature Interaction.” Technischen Universität München. https://mediatum.ub.tum.de/doc/805656/805656.pdf.

Burkard, Rainer, Mauro Dell’Amico, and Silvano Martello. 2012. *Assignment Problems*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611972238.

Chow, T.S. 1978. “Testing Software Design Modeled by Finite-State Machines.” *IEEE Transactions on Software Engineering* SE-4 (3): 178–87. https://doi.org/10.1109/TSE.1978.231496.

Cichos, Harald, Sebastian Oster, Malte Lochau, and Andy Schürr. 2011. “Model-Based Coverage-Driven Test Suite Generation for Software Product Lines.” In *Model Driven Engineering Languages and Systems*, edited by Jon Whittle, Tony Clark, and Thomas Kühne, 425–39. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-24485-8_31.

Classen, Andreas. 2011. “Modelling and Model Checking Variability-Intensive Systems.” Ph.D. Dissertation.

Classen, Andreas, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. “Symbolic Model Checking of Software Product Lines.” In *Proceedings of the 33rd International Conference on Software Engineering*, 321–330. ICSE ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery. https://doi.org/10.1145/1985793.1985838.

Czarnecki, Krysztof, and Eisenecker, Ulrich. 2000. *Generative Programming: Methods, Tools, and Applications*. Addision-Wesley.

Czarnecki, Krzysztof, and Michał Antkiewicz. 2005. “Mapping Features to Models: A Template Approach Based on Superimposed Variants.” In *Generative Programming and Component Engineering*, edited by Robert Glück and Michael Lowry, 422–37. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/11561347_28.

Devroey, Xavier, Maxime Cordy, Gilles Perrouin, Eun-Young Kang, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Benoit Baudry. 2012. “A Vision for Behavioural Model-Driven Validation of Software Product Lines.” In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, edited by Tiziana Margaria and Bernhard Steffen, 208–22. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-34026-0_16.

El-Far, Ibrahim K., and James A. Whittaker. 2002. “Model-Based Software Testing.” In *Encyclopedia of Software Engineering*, edited by John J. Marciniak, sof207. Hoboken, NJ, USA: John Wiley & Sons, Inc. https://doi.org/10.1002/0471028959.sof207.

“FeatureIDE.” n.d. Accessed August 8, 2020. http://www.featureide.com/.

Fujiwara, S., G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. 1991. “Test Selection Based on Finite State Models.” *IEEE Transactions on Software Engineering* 17 (6): 591–603. https://doi.org/10.1109/32.87284.

Garg, Vijay K. 2015. *Introduction to Lattice Theory with Computer Science Applications*. 1st ed. Wiley Publishing.

Gebizli, Ceren Sahin, and Hasan Sözer. 2016. “Model-Based Software Product Line Testing by Coupling Feature Models with Hierarchical Markov Chain Usage Models.” In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 278–83. https://doi.org/10.1109/QRS-C.2016.42.

Geppert, Birgit. 2004. “Towards Generating Acceptance Tests for Product Lines.,” 35–48. https://doi.org/10.1007/978-3-540-27799-6_4.

Gruler, Alexander, Martin Leucker, and Kathrin Scheidemann. 2008. “Modeling and Model Checking Software Product Lines.” In *Formal Methods for Open Object-Based Distributed Systems*, edited by Gilles Barthe and Frank S. de Boer, 113–31. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-68863-1_8.

Hierholzer, Carl, and Chr Wiener. 1873. “Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren.” *Mathematische Annalen* 6 (1): 30–32. https://doi.org/10.1007/BF01442866.

Jones, Capers. 1991. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill.

Kang, Kyo C. n.d. “Feature-Oriented Domain Analysis (FODA) Feasibility Study.” Accessed May 25, 2020. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=11231.

Kang, Kyo C., Sholom G. Cohen, James A. Hess, William E. Novak, and A. S. Peterson. 1990. “Feature-Oriented Domain Analysis (FODA) Feasibility Study:” Fort Belvoir, VA: Defense Technical Information Center. https://doi.org/10.21236/ADA235785.

Kang, Kyo C., Vijayan Sugumaran, and Sooyong Park. 2009. *Applied Software Product Line Engineering*. CRC Press.

Kishi, Tomoji, and Natsuko Noda. 2006. “Formal Verification and Software Product Lines.” *Communications of the ACM* 49 (12): 73–77. https://doi.org/10.1145/1183236.1183270.

Lamancha, Beatriz Perez, Macario Polo Usaola, and Ignacio Garcia Rodriguez de Guzman. 2009. “Model-Driven Testing in Software Product Lines.” In *2009 IEEE International Conference on Software Maintenance*, 511–14. https://doi.org/10.1109/ICSM.2009.5306324.

Linschulte, Michael. 2013. “On the Role of Test Sequence Length, Model Refinement, and Test Coverage for Reliability.” In 1–212. https://nbn-resolving.org/urn:nbn:de:hbz:466:2-11890.

Lochau, Malte, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. 2016. “Incremental Model Checking of Delta-Oriented Software Product Lines.” *Journal of Logical and Algebraic Methods in Programming*, Formal Methods for Software Product Line Engineering, 85 (1, Part 2): 245–67. https://doi.org/10.1016/j.jlamp.2015.09.004.

Lochau, Malte, Ina Schaefer, Jochen Kamischke, and Sascha Lity. 2012. “Incremental Model-Based Testing of Delta-Oriented Software Product Lines.” In *13th International Conference, TAP 2019*, edited by Achim D. Brucker and Jacques Julliand, 7305:67–82. Lecture Notes in Computer Science. Porto, Portugal: Lecture Notes in Computer Science Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-30473-6_7.

Memon, A.M., M.E. Pollack, and M.L. Soffa. 2001. “Hierarchical GUI Test Case Generation Using Automated Planning.” *IEEE Transactions on Software Engineering* 27 (2): 144–55. https://doi.org/10.1109/32.908959.

Mistrik, Ivan, Rami Bahsoon, Peter Eeles, Roshanak Roshandel, and Michael Stal. 2014. *Relating System Quality and Software Architecture*. Morgan Kaufmann.

Neto, Paulo Anselmo da Mota Silveira, Ivan do Carmo Machado, Yguarata Cerqueira Cavalcanti, Eduardo Santana de Almeida, Vinicius Cardoso Garcia, and Silvio Romero de Lemos Meira. 2010. “A Regression Testing Approach for Software Product Lines Architectures.” In *2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse*, 41–50. https://doi.org/10.1109/SBCARS.2010.14.

Nolan, Andy J., Silvia Abrahao, Paul Clements, John D. McGregor, and Sholom Cohen. 2011. “Towards the Integration of Quality Attributes into a Software Product Line Cost Model.” In *2011 15th International Software Product Line Conference*, 203–12. https://doi.org/10.1109/SPLC.2011.44.

Olimpiew, Erika Mir. 2008. “Model-Based Testing for Software Product Lines A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy at George Mason University | Semantic Scholar.” George Mason University.

Olimpiew, Erika Mir, and Hassan Gomaa. 2005. “Model-Based Testing for Applications Derived from Software Product Lines.” *ACM SIGSOFT Software Engineering Notes* 30 (4): 1–7. https://doi.org/10.1145/1082983.1083279.

Oster, Sebastian, Ivan Zorcic, Florian Markert, and Malte Lochau. 2011. “MoSo-PoLiTe: Tool Support for Pairwise and Model-Based Software Product Line Testing.” In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, 79–82. VaMoS ’11. Namur, Belgium: Association for Computing Machinery. https://doi.org/10.1145/1944892.1944901.

“Readings of Mathematics for Computer Science MIT OpenCourseWare - Chapter 7.” n.d. Accessed August 8, 2020. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-fall-2010/readings/.

Reuys, Andreas, Erik Kamsties, Klaus Pohl, and Sacha Reis. 2005. “Model-Based System Testing of Software Product Families.” In *Advanced Information Systems Engineering*, edited by Oscar Pastor and João Falcão e Cunha, 519–34. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/11431855_36.

“SPL2go.” n.d. Accessed June 1, 2020. http://spl2go.cs.ovgu.de/projects/54.

Tuglular, Tugkan, Fevzi Belli, and Michael Linschulte. 2016. “Input Contract Testing of Graphical User Interfaces.” *International Journal of Software Engineering and Knowledge Engineering* 26 (02): 183–215. https://doi.org/10.1142/S0218194016500091.

Tuglular, Tugkan, Mutlu Beyazıt, and Dilek Öztürk. 2019. “Featured Event Sequence Graphs for Model-Based Incremental Testing of Software Product Lines.” In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 1:197–202. https://doi.org/10.1109/COMPSAC.2019.00035.

Uzuncaova, Engin, Sarfraz Khurshid, and Don Batory. 2010. “Incremental Test Generation for Software Product Lines.” *IEEE Transactions on Software Engineering* 36 (3): 309–22.

Whittaker, James A. 1997. “Stochastic Software Testing.” *Annals of Software Engineering* 4 (1): 115. https://doi.org/10.1023/A:1018975029705.

Withey, James. 1996. “Investment Analysis of Software Assets for Product Lines.” Pittsburgh, Pennsylvania: Carnegie Mellon University, Software Engineering Institute.

Xu, Dianxiang. 2011. “A Tool for Automated Test Code Generation from High-Level Petri Nets.” In *International Conference on Application and Theory of Petri Nets and Concurrency*, edited by Lars M. Kristensen and Laure Petrucci, 308–17. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-21834-7_17.

APPENDIX A

SODA VENDING MACHINE SOFTWARE PRODUCT LINE

Figure A.1. *free* f-ESG of SVM SPL

Figure A.2. *payUSD* f-ESG of SVM SPL

Figure A.3. serveTea f-ESG of SVM SPL

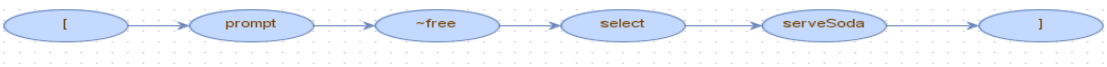

Figure A.4. ESG of SVM SPL – *serve soda-free product*

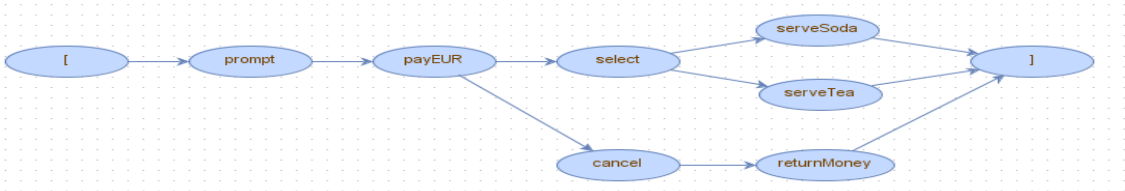

Figure A.5. ESG of SVM SPL –*pay EUR product*

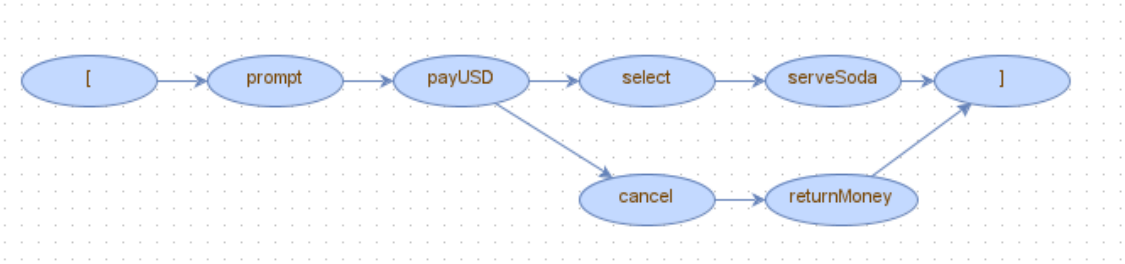

Figure A.6. ESG of SVM SPL – *pay USD-serve soda product*

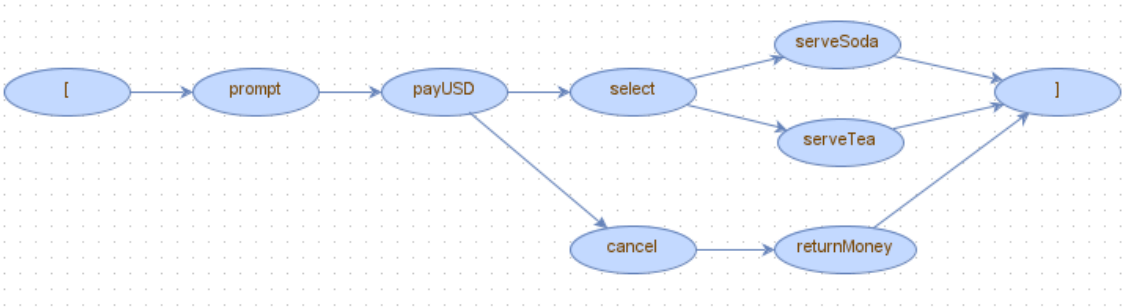

Figure A.7. ESG of SVM SPL – *pay USD product*

APPENDIX B

EMAIL SOFTWARE PRODUCT LINE

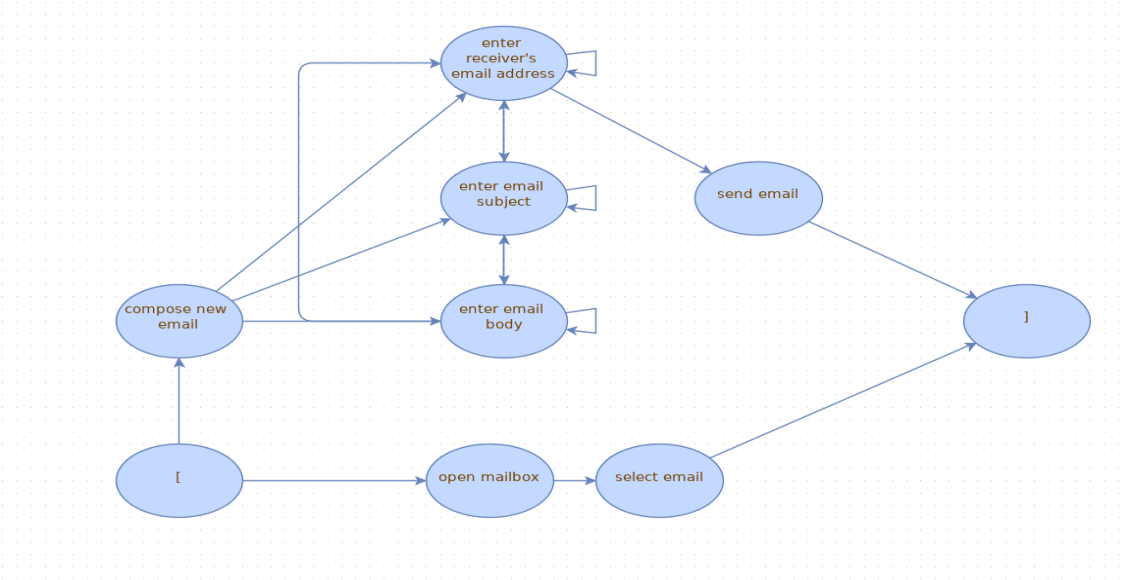

Figure B.1. ESG of Email SPL – *base product*

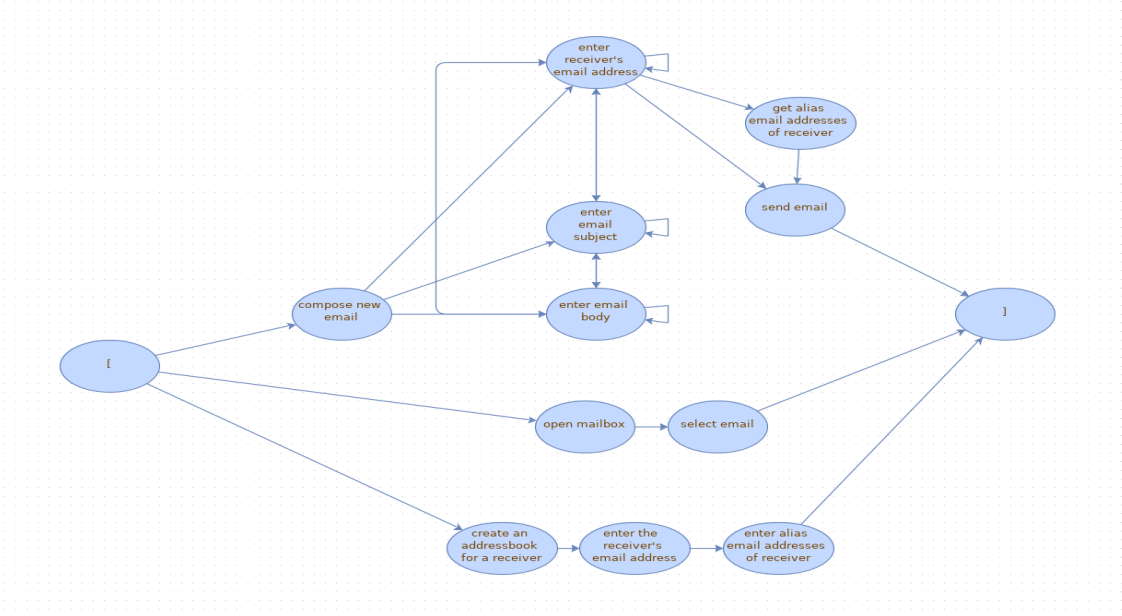

Figure B.2. ESG of Email SPL – *address book*

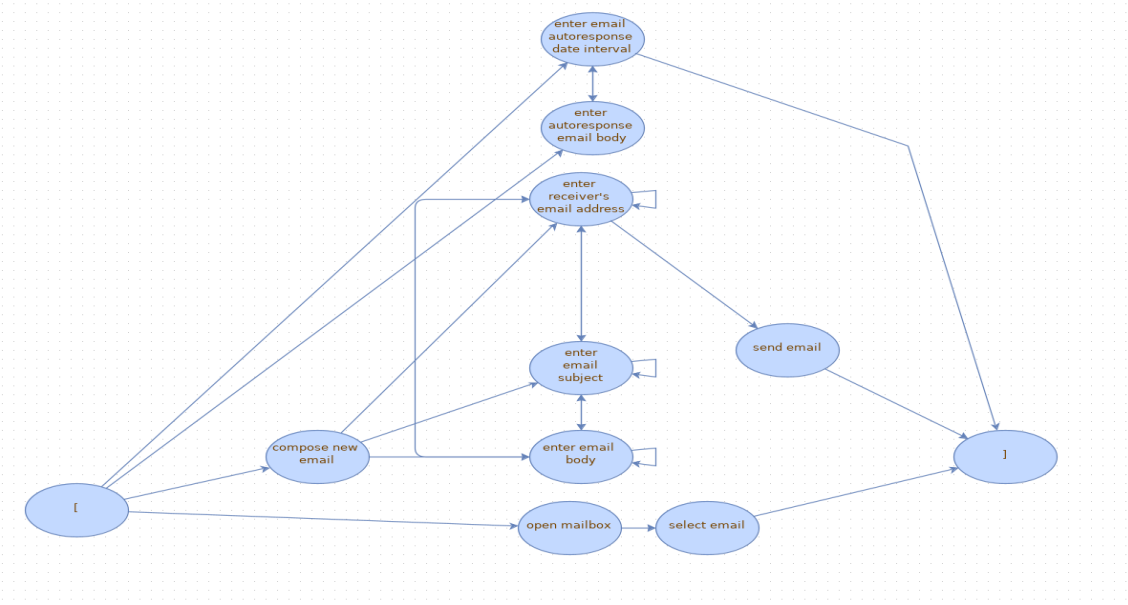

Figure B.3. ESG of Email SPL – *autoresponder*

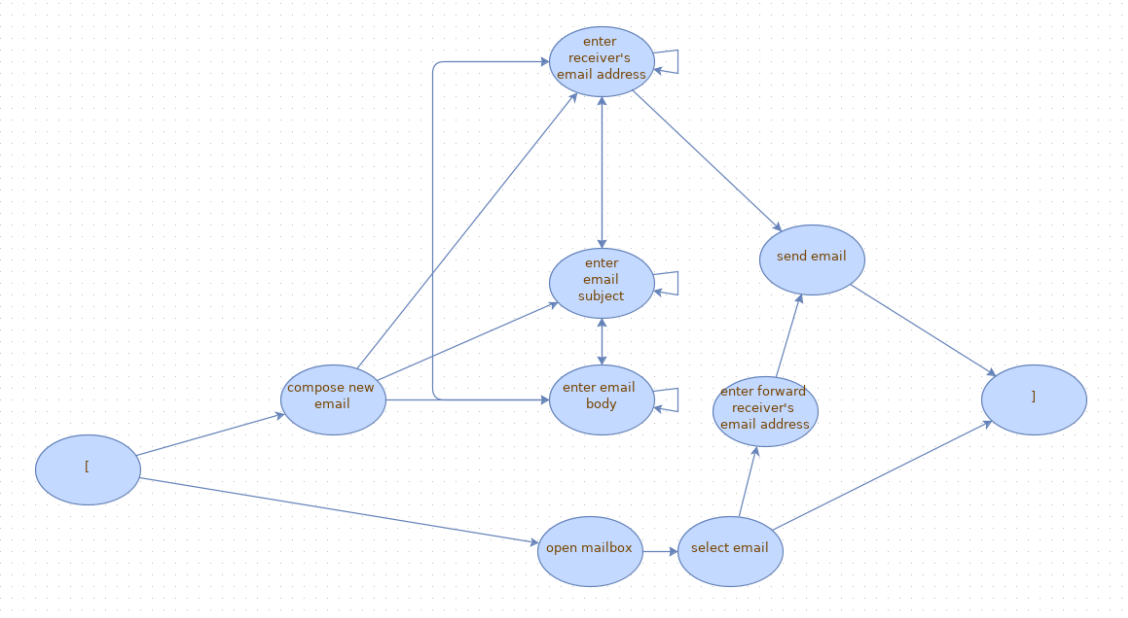

Figure B.4. ESG of Email SPL – *forward*

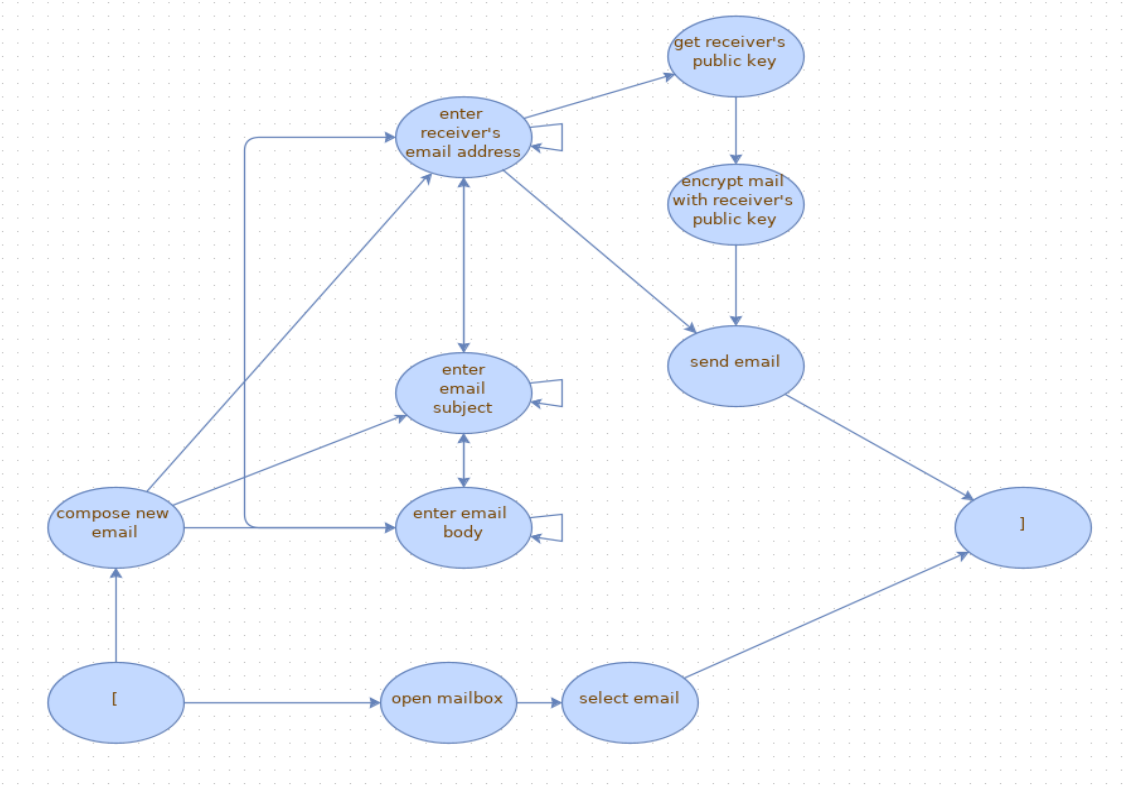

Figure B.5. ESG of Email SPL – *encrypt*

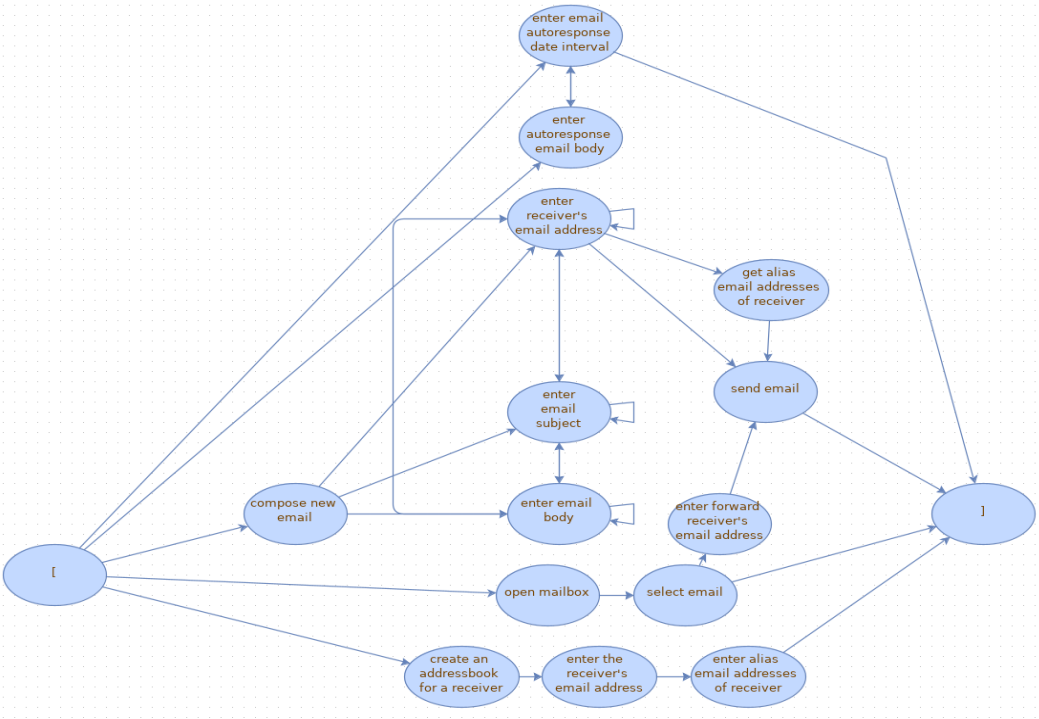

Figure B.6. ESG of Email SPL – *address book-autoresponder-forward*

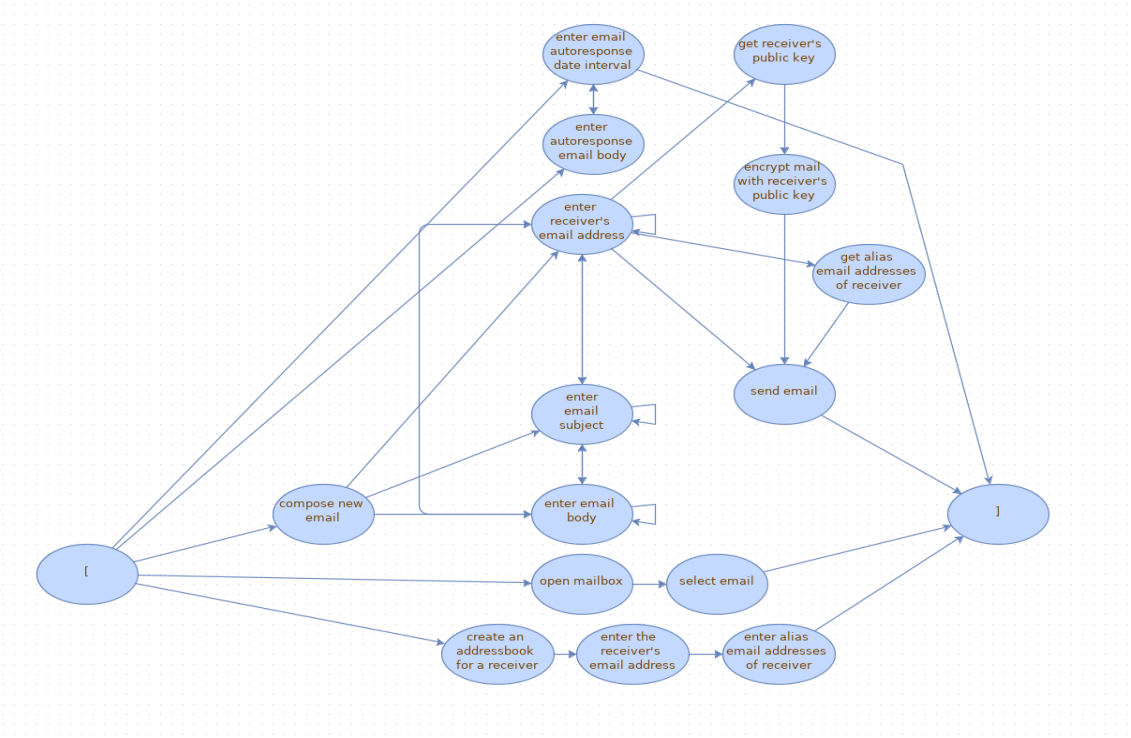

Figure B.7. ESG of Email SPL – *address book-autoresponder-encrypt*

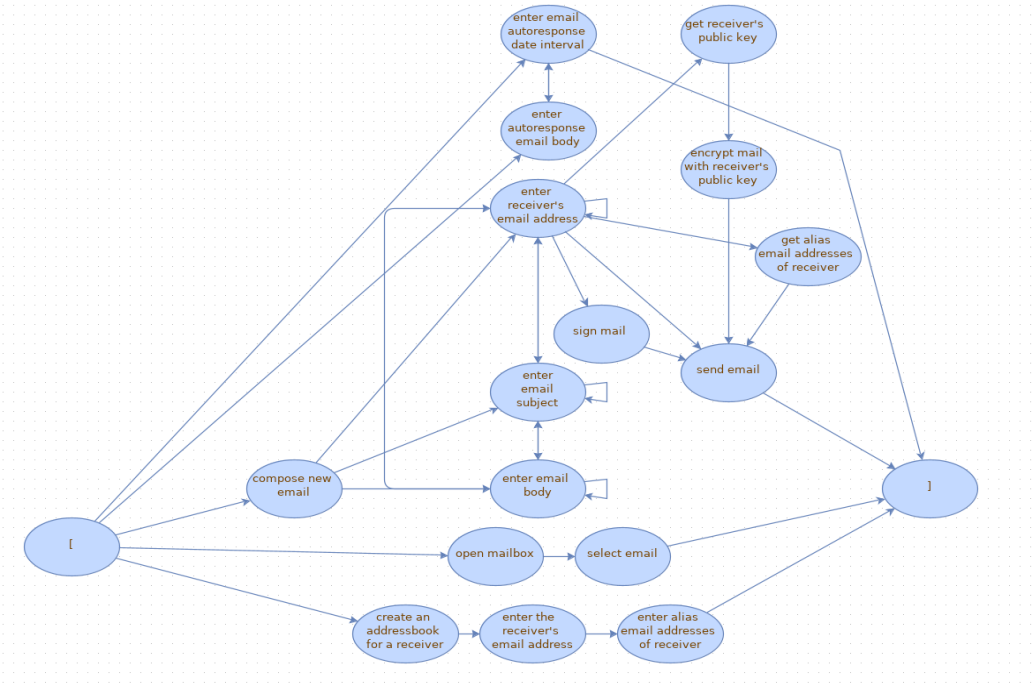

Figure B.8. ESG of Email SPL – *address book-autoresponder-encrypt-sign product*

APPENDIX C

BANK ACCOUNT SOFTWARE PRODUCT LINE

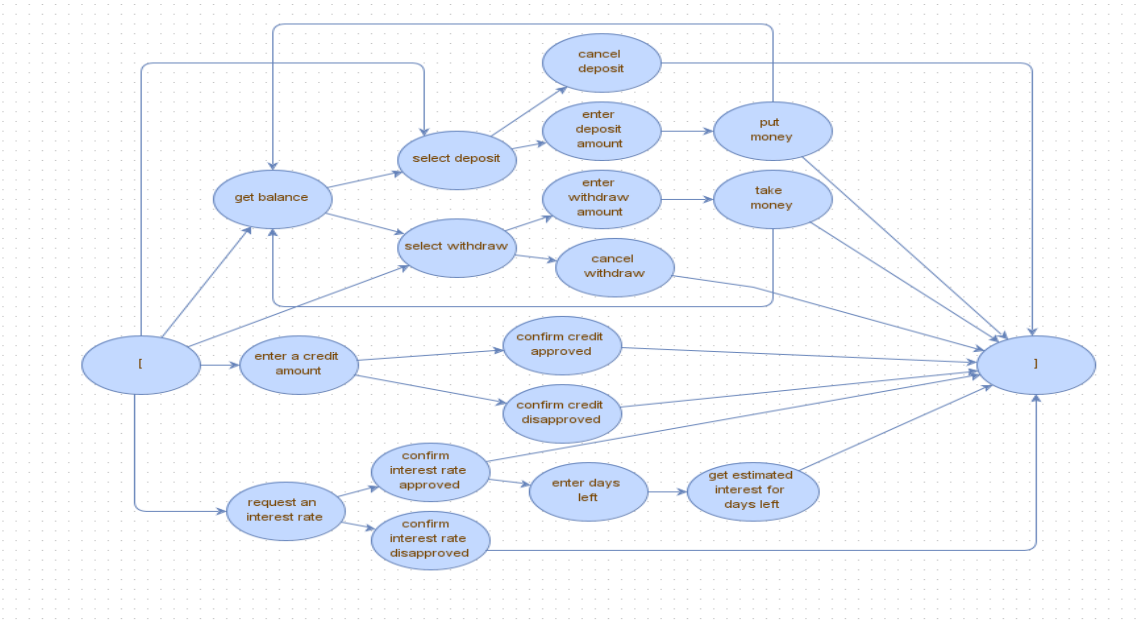

Figure C.1. ESG of bank account SPL – *credit* product

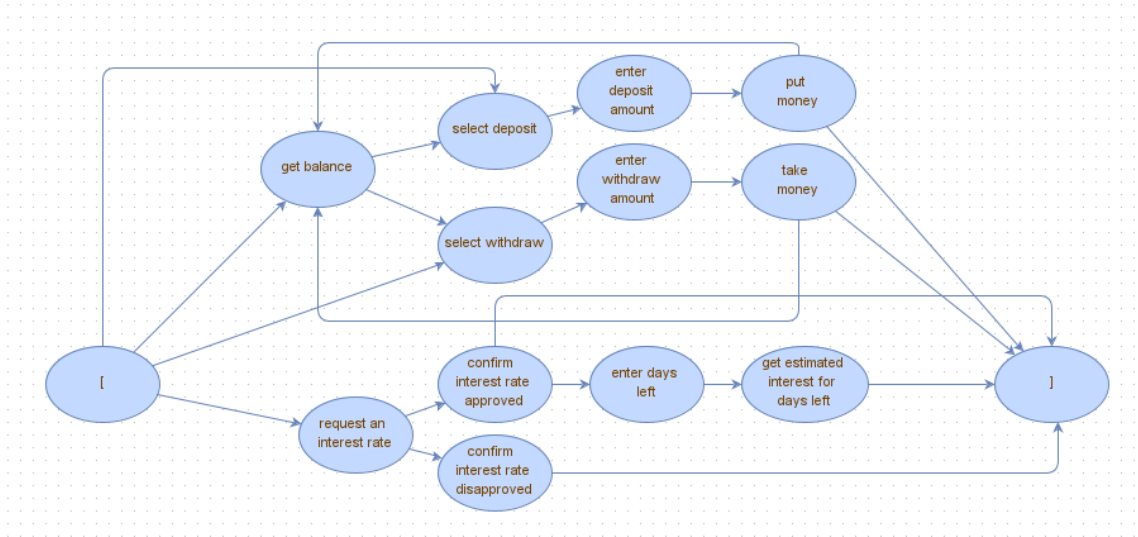

Figure C.2. ESG of bank account SPL – *interest* product

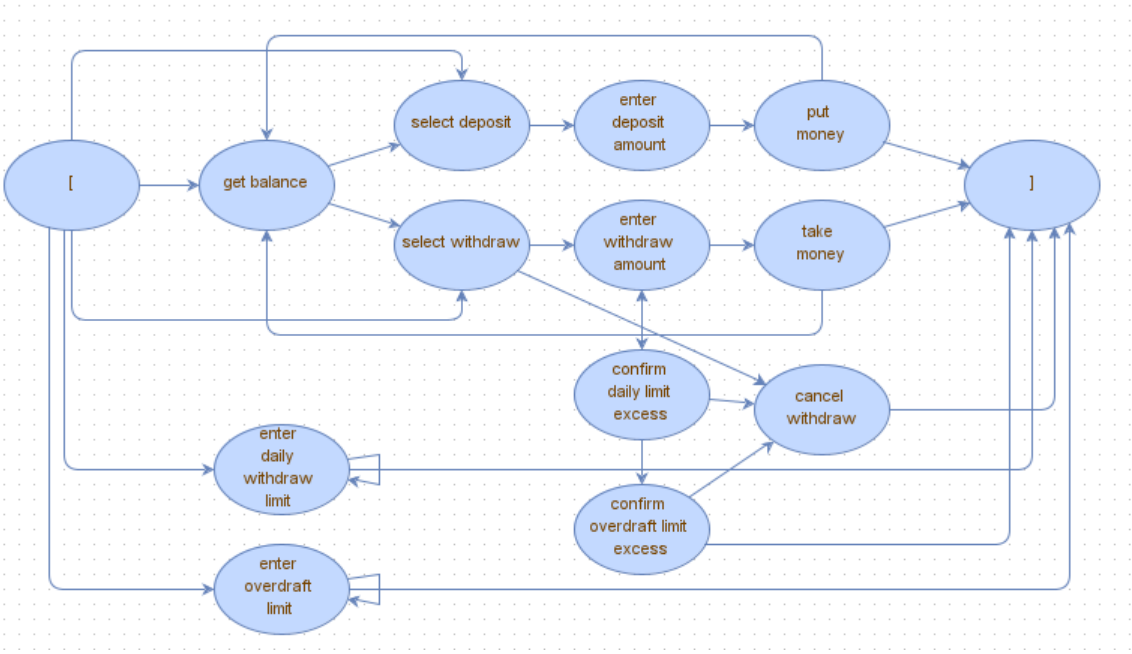

Figure C.3. ESG of bank account SPL – *overdraft* product

APPENDIX D

STUDENT ATTENDANCE SYSTEM SOFTWARE PRODUCT LINE

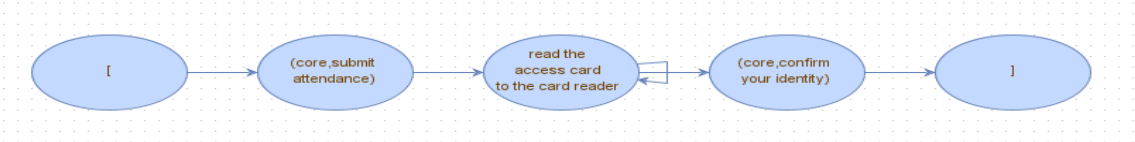

Figure D.1. *accessCard* f-ESG of SAS SPL

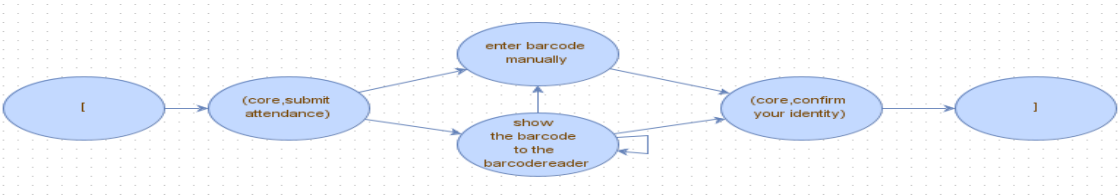

Figure D.2. *barcode* f-ESG of SAS SPL

Figure D.3. *fingerprint* f-ESG of SAS SPL

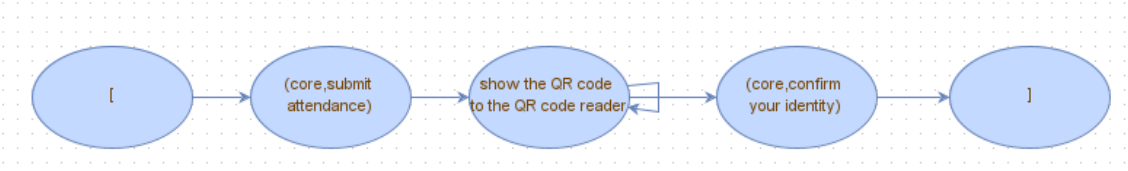

Figure D.4. *QRCode* f-ESG of SAS SPL

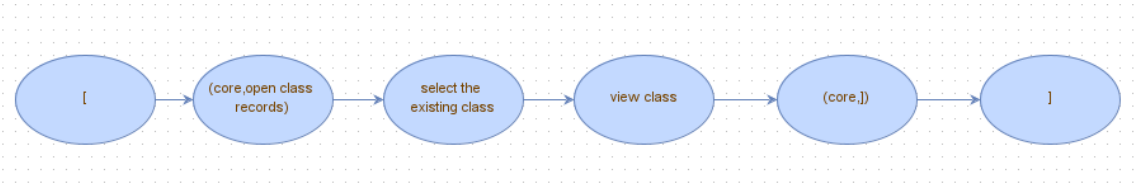

Figure D.5. *viewClass* f-ESG of SAS SPL

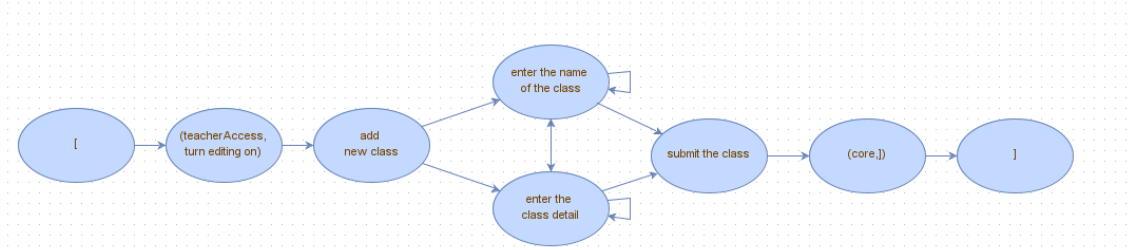

Figure D.6. *addNewClass* f-ESG of SAS SPL

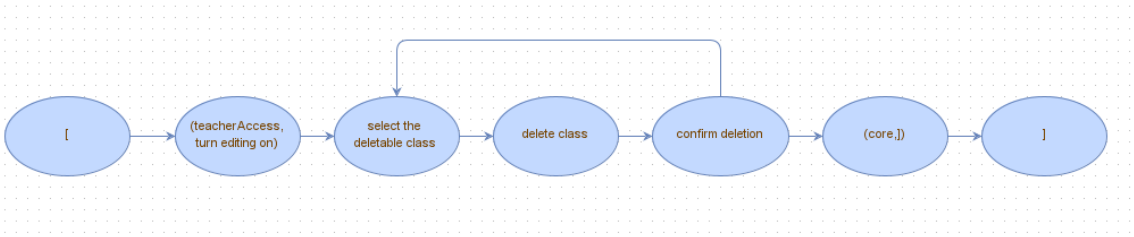

Figure D.7. *deleteClass* f-ESG of SAS SPL

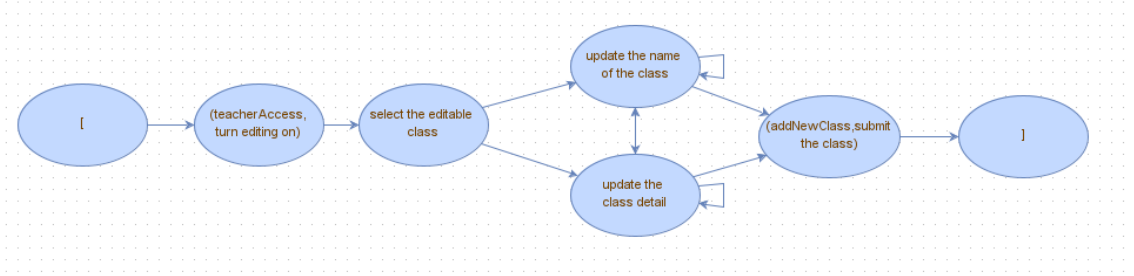

Figure D.8. *updateClassDetails* f-ESG of SAS SPL

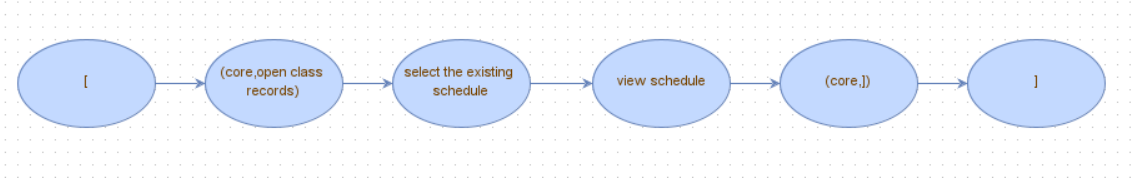

Figure D.9. *viewSchedule* f-ESG of SAS SPL

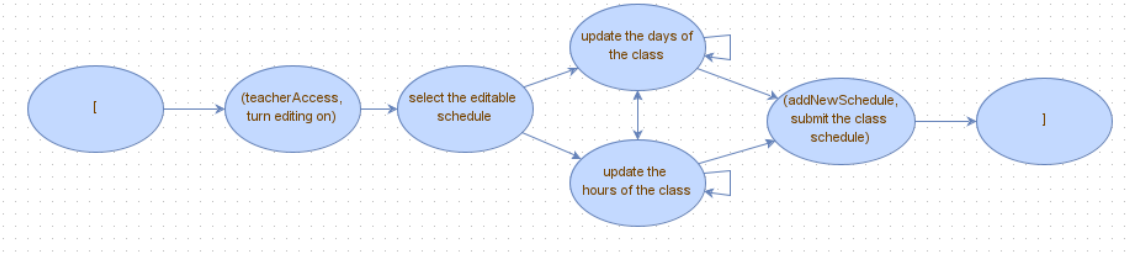

Figure D.10. *editSchedule* f-ESG of SAS SPL

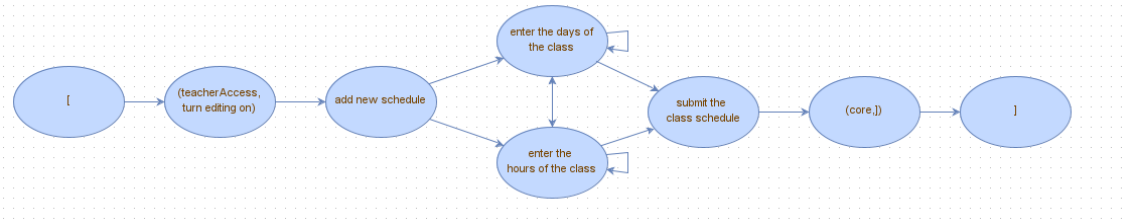

Figure D.11. *addNewSchedule* f-ESG of SAS SPL

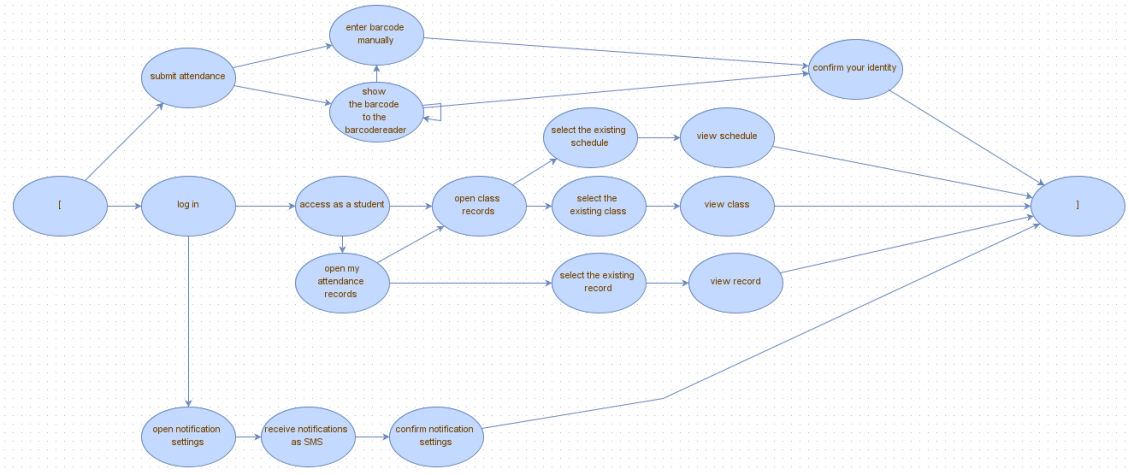

Figure D.12. ESG of SAS SPL – *limited student user-barcode-SMS*

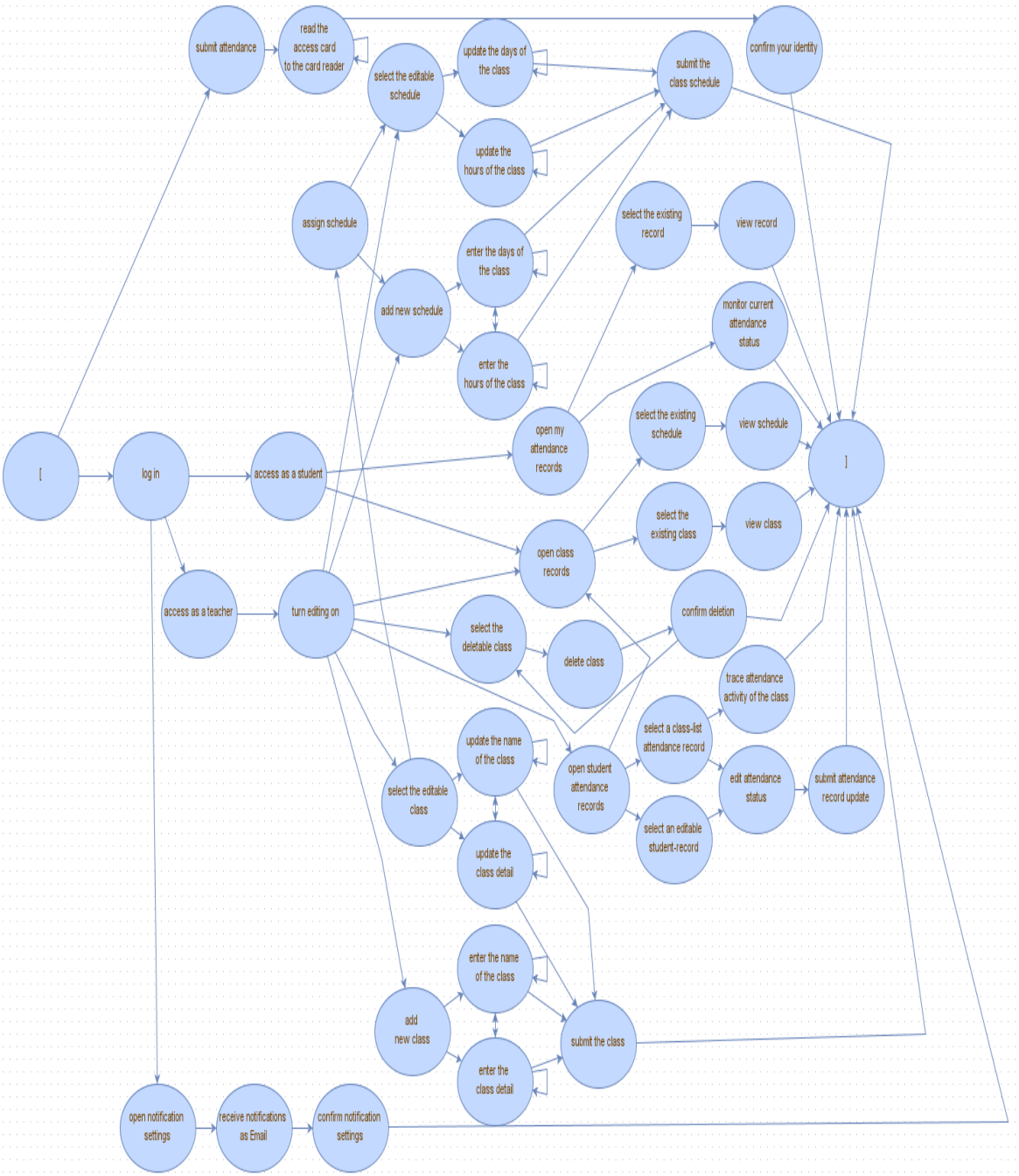

Figure D.13. ESG of SAS SPL – *both users-access card email*

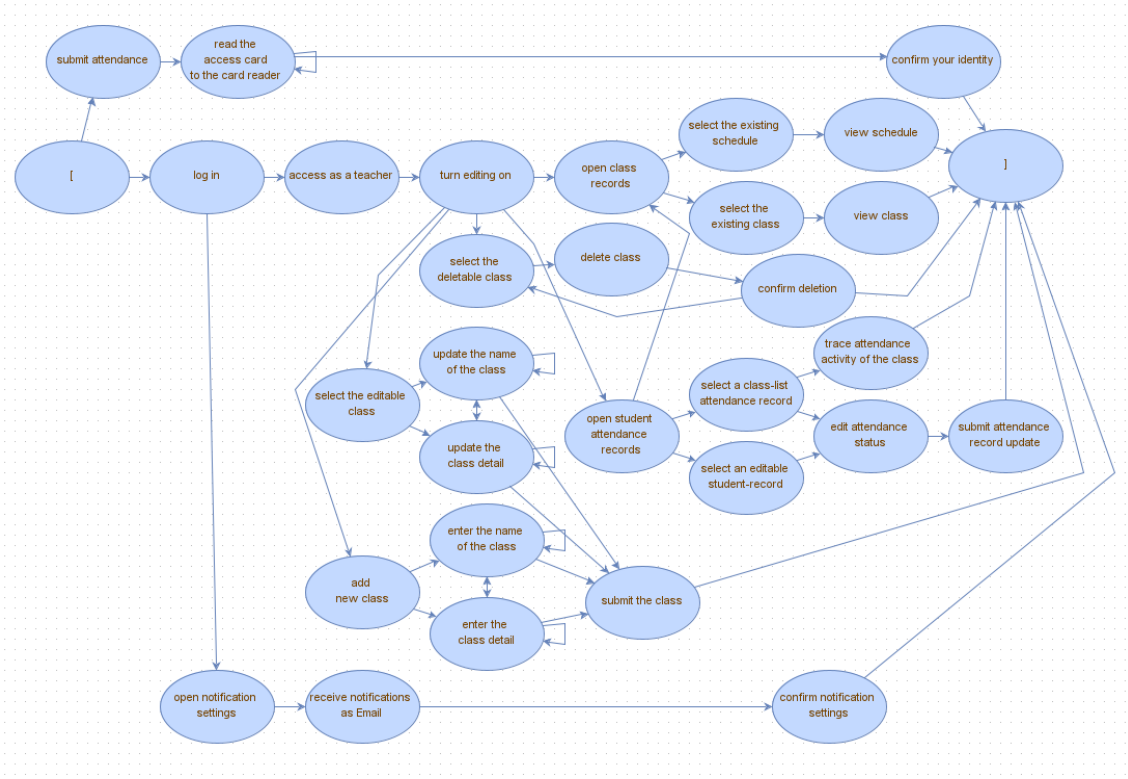

Figure D.14. ESG of SAS SPL – *limited teacher user-access card*