DESIGN AND IMPLEMENTATION OF A DOMAIN SPECIFIC LANGUAGE FOR EVENT SEQUENCE GRAPHS

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
Mert KALECİK

July 2022
İZMİR

ACKNOWLEDGEMENTS

First, I'd like to express my heartfelt gratitude to my supervisor, Assoc. Prof. Dr. Tugkan Tuglular, for his compassion, support, and encouragement. I am thankful for his guidance with experiences and knowledge for me. I would also like to sincerely thank and appreciate Prof. Dr. Fevzi Belli for his inspiration, remarks, and advice on my research. I am extremely grateful to my other half all for her support and motivation unconditionally all of these years. Also, I could not have undertaken this journey without my family who gave me moral support and infinite motivation.

ABSTRACT

DESIGN AND IMPLEMENTATION OF A DOMAIN SPECIFIC LANGUAGE FOR EVENT SEQUENCE GRAPHS

Nowadays, large-scale software applications are being developed because of the increasing q-commerce or e-commerce conversion rate. Companies extend their service operation areas with the trend of having a super app. As the result of extended functionality brings some risks together. Therefore, software quality is one of the crucial metrics for achieving reliable and faultless software products. One way of achieving software quality is systematic testing, which is often materialized by model-based testing. An example of model-based testing approaches is Event Sequence Graphs (ESGs). Domain specific language is usually a declarative language that provides substantial gain on a restricted business domain. This thesis mainly focuses on the development of a domain specific language (DSL) for ESG building and visualization process with a modularization support for sub-ESGs and decision tables. The ESGs are augmented by decision tables visualized with a vertex and that vertex is visualized with two tables such as property table and property definition table. The use of the proposed DSL is compared with the existing ESG tool called Test Suite Designer (TSD) in areas such as measuring the cost of quality, understanding the value of quality, motivation to achieve quality, and understand how to overcome it. The comparison results obtained through a questionnaire applied to a focus group show that some improvements for both ESG DSL and TSD are necessary.

ÖZET

OLAY SIRA ÇİZGELERİ İÇİN ALANA ÖZGÜ DİL TASARIMI VE UYGULAMASI

Artan e-ticaret ve hızlı ticaret etkileşim oranlarının sonucu olarak günümüzde büyük ölçekli yazılım uygulamaları geliştirilmeye başlandı. Şirketler servis operasyon alanlarını genişleterek bir süper uygulamaya sahip olmaya yöneliyorlar. Genişletilen bu işlevselliklerin sonucu olarak yanında bazı riskler getiriyor. Bu nedenle Yazılım Kalitesi, güvenilir ve hatasız yazılım ürünleri elde etmek için önemli ölçütlerden biridir. Yazılım kalitesine ulaşmanın bir yolu, genellikle model tabanlı testlerle gerçekleştirilen sistemik testtir. Model tabanlı test yaklaşımlarına bir örnek Olay Sırası Çizgeleridir (OSÇ). Alana Özgü Dil (AÖD) genellikle sınırlandırılış bir iş alanında önemli kazanç sağlayan bildirimsel bir dildir. Bu tez esas olarak Alana Özgü Dil (AÖD) geliştirmeye, mevcut yazılım ürünlerinin yeniden kullanılabilirliğini arttırmaya, üretkenliği arttırmaya ve teknoloji altyapısı olmayan kişileri geliştirme sürecine dahil etmeyi amaçlayan yeni bir yaklaşıma odaklanır. Bu çalışma, alt Olay Sırası Çizgeleri (OSÇ) ve Karar Tabloları (KT) için modüler hale getirme desteği ile bir OSÇ görselleştirme sürecini tanıtmaktadır. KT ile arttırılmış bir OSÇ bir köşe ile gösterilir ve bu köşe özellik tablosu ve özellik detayları tablosu olarak iki tablo olarak görselleştirilir. Önerilen AÖD tasarım, uygulama yaklaşımı ve mevcut araç ile kalite maliyetini ölçmek, kalitenin değerini anlamak, kaliteye ulaşma motivasyonu ve bunun nasıl üstesinden gelineceğini anlamak gibi alanlarda karşılaştırıldı. Karşılaştırma vaka sonuçları her iki araç için, OSÇ AÖD ve Test Paketi Tasarımcısı (TPT), anket sonucunda test gruplarından alınan geri bildirimler ve iyileştirmeleri göstermektedir.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

Figure **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

In the last twenty years, the software industry growing rapidly with the improvement of internet speed that connected the world and expensed knowledge turnover within engineering. The difficulties and complexities of the problem domain differ from those of previous generations in terms of diversification of demand and size of it (Boehm, 2006). With the increasing demand and functional set extension, the complexity of the project is increased and prone to making mistakes. It becomes difficult to keep the quality of the software product at a certain level. Software quality is one of the most important metrics of the software development for faultless products. Quality comprises all characteristics and significant features of a product or an activity which relate to the satisfying of given requirements (Tomar et al., 2011).

The complicated nature of software development and working with large size projects make it hard to measure and enhance software quality. Marco said “You can’t manage what you can’t measure!” (DeMarco, 1982). Software testing is the process of ensuring that a software product has the satisfied quality for the end-users. Hence, software testing has an important role for achieving reliable and faultless software products. On 26 April 1986, the incident happened at Chernobyl nuclear power plant while the scientists made a wrong power cutdown decision to system under test. The accident caused genetic diseases in millions of people and thousands of people were dead. A large area of land closed to human access because of the environmental change. (Higley, 2006). This example shows that software testing is one of the most crucial steps in software development life cycle.

Behavior Driven Development (BDD) (Diepenbeck et al., 2014) is concerned with describing briefly defined specifications of the behaviors of the target system that are influenced by the interactions from end-users, system under test itself, and environmental changes. Clearly written Gherkin (Cucumber, n.d.) based test scenarios help test designers in writing test cases. Also, the test scenarios which are written as gherkin based can be transformed into the formal test models (Tuglular, 2021).

Model Based Testing (MBT) is a model-based design technique that represent the required behaviors of the System Under Test (SUT). Formal models are derived from the requirements in model based testing (Eeles et al., 2014).

Event Sequence Graph (ESG) is a way to represent the behaviors of the system under test. With the modularization support, makes it easy to understand in smaller component layers. The difference between ESG and Finite State Machine (FSM) (Chow, 1978) is that FSM contains states, ESG contains events of the system under test.

In this thesis, a DSL design and development steps are introduced in order of decision, analysis, design, implementation, and deployment. The software product's behaviors represented with ESGs and for the different behaviors set under a certain condition is represented by DTs. The modularization support for ESGs provided by vertex refinement and vertex augmentation by DTs. The generated ESG objects visualized by Graphviz (Ellson et al., 2003) with using DOT language. Also, Graphviz supports modular ESG visualization for the refined vertex container ESGs.

For the motivation of the project, the ESG visualization cost is high and requires rework for each ESG. Further, the existing ESG visualization tools is not user friendly, there is no error checking and highlighting support. The motivation is creating a domain specific language that is user friendly, reusable, and easy to understand.

The aim of this thesis to design and develop a ESG DSL and visualization process for the ESGs. This thesis tries to answer the following questions:

1. How to design and develop a DSL with supporting modularization and decision table for multiple input case for ESGs?
2. How can the ESG DSL reduce the complexity of the ESG visualization by using a close to nature language syntax?
3. How to visualize the ESGs, sub-ESGs (inner ESGs), and DTs with reusing the software artifacts?

This thesis is organized as the given order. Second chapter examines an overview of the literature. Third chapter introduces fundamental concepts namely, Domain Specific Languages (DSLs), Event Sequence Graphs (ESGs), vertex refinement for modularization, augmented ESGs by Decision Tables (DTs). Fourth chapter includes the detailed design and development steps of a DSL and graph visualization for ESGs. The case study, Bank ATM project, of this thesis study is given in chapter five. Finally, the conclusion and the future work are mentioned in chapter six.

CHAPTER 2

RELATED WORK

Software testing is the process of validating a service or application to prove if it meets the given requirements and all the characteristics of the software are implemented correctly (Uddin et al., 2019). Thus, software testing is a critical step for creating faultless and high-quality software products.

Behavior-driven development (BDD) is a branch of test-driven development (TDD). The methods of the system under test are passed without no failure at all, if the methods do not satisfy the required behavior for the system under test, then the system does not meet the requirements (Mishra, 2017). Hence, BDD defines the behaviors of the target system. The model-based testing (MBT) provides a technique, the required specification of the software is defined in a model which is generally a graph. The finite state machines (Chow, 1978) is extended to hold events on the each vertex of the graph. Event sequence graph (ESG) is introduced to illustrate the events of the system under test (Belli, 2001).

For decades, the people of software community have been trying to increase reusability of the software artifacts. The early example of this intention is the fourth-generation languages (4GL). 4GL languages have statement that are close to the natural language and commonly using in scripting languages like Python, and Perl. The first attempts were called micro-languages and little languages for the domain specific languages (DSLs) (Bentley, 1986). In object-oriented programming, DSL developments are injected into a subroutine library and can be implemented as a framework to the code base. DSL usually includes a general-purpose language (GPL) and enhance the abilities of the language in the domain-specific area (Deursen et al., 2000).

In this thesis, we focused on DSLs and graph visualization tools which are developed with JAVA general-purpose language. Eclipse Xtext (*Xtext*, 2006) framework is used to develop ESG DSL. Well-known companies use domain specific language that developed with Xtext (*Xtext*, 2006). Yaktor (*SciSpike/Yaktor*, 2016/2021) is event-driven, asynchronous, distributed, scalable multi-party state-machine tool.

Yaktor DSL (*SciSpike/Yaktor-Dsl-Xtext*, 2016/2017) is developed with Xtext and it creates data models and behavior for the Yaktor application. Franca (*Franca*, 2018) is a powerful IDL (Interface Definition Language) that is used for integration software components from different suppliers. Expression DSL (*Expression Language*, 2021) provides an expression language built using Xtext framework and a runtime engine to evaluate the expressions. The language can be imported in other DSLs to create composable and reusable languages using Xtext.

JGraph is a java-based framework that allows to draw graphs and runs graph algorithms. The algorithms can be run with an animation feature, which allows the end-users to see the intermediate steps as the algorithm runs (Bagga & Heinz, 2001). The tool is created by several graduate students.

Java Universal Network/Graph (JUNG) framework is an open-source project that provides a language for the modeling, analysis, and visualization of the data provided by the end-users. JUNG is a java-based tool and has a strong capability that is coming from java general-purpose language. The JUNG framework is designed to support a variety of representations of entities and relations (*JUNG Framework Tech Report*, n.d.). The framework has a support for drawing directed and undirected graphs, entities, and relations with metadata.

PlantUML (*PlantUML*, 2009) is an open-source tool that allows users to create diagrams from a plain text language. It is important to be aware of that PlantUML is more a drawing tool than a modeling tool. That means it does not help end-users with drawing inconsistent diagrams. Also, PlantUML has a support for AsciiMath, DOT, and LaTeX. The tool uses Graphviz (Ellson et al., 2003) framework to layout its diagrams.

Graphviz is an open-source graph visualization software. The representation of structural information as diagrams of abstract graphs and networks is known as graph visualization. It is usually using in software engineering, database and web design, machine learning, and bioinformatics (*Graphviz*, n.d.). Emden and Stephen introduce a tool that is manipulating the graphs and their drawings (Gansner & North, 1997). A four-pass algorithm for drawing directed graphs is described for Graphviz software. The first pass stands for the optimal rank assignment using a simplex algorithm. The second pass uses and iterative heuristic with a novel weight function and local transpositions to reduce crossing to determine the vertex order withing ranks. The third pass calculates the optimal coordinates for the vertices. Splines are used to draw edges in the final pass (Gansner et al., 1993).

CHAPTER 3

FUNDAMENTALS

This chapter introduces fundamental concepts related to thesis study. First, Domain Specific Languages (DSLs) are explained. Second, Event Sequence Graphs (ESGs) are introduced with additional features, which are Sub-Event Sequence Graphs and Event Sequence Graphs Augmented by Decision Tables.

3.1. Domain Specific Languages

Nowadays, when we mentioned about language many people think of spoken language or programming language. Most software developer think of the commonly used general purpose languages such as JAVA, C#, or C. A domain-specific language is any mechanism that has expressiveness gain as statements over the language. if that's applicable in a restricted domain then we can call it "little languages" (Gansner & North, 1997).

Over the years, different solutions have been tried for to eliminate domain complexity, increase the reusability of the software components, enhance productivity, and reduce maintain cost of the system. In the literature these solutions have been introduced:

Definition 3.1: *Subroutine libraries* contain subroutines that perform related tasks in well-defined domains like, for instance, differential equations, graphics, user-interfaces, and databases. The subroutine library is the classical method for packaging reusable domain-knowledge (Deursen et al., 2000).

Definition 3.2: *Object-oriented frameworks and component frameworks* continue the idea of subroutine libraries. Classical libraries have a flat structure, and the application invokes the library. In object-oriented frameworks it is often the case that the framework is in control, and invokes methods provided by the application-specific

code (Deursen et al., 2000).

Definition 3.3: *A domain-specific language (DSL)* is a small, usually declarative, language that offers expressive power focused on a particular problem domain. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed to hide the details of that library (Deursen et al., 2000).

In this thesis, we use Domain Specific Language (DSL) which is more convenient to use. DSL is a programming language that designed to increase abstraction level for a group of complex problems on a restricted application domain. They provide numerous advantages in ease of use and limited portion of related general-purpose language. A DSL generally provides less complex language than a general-purpose language such as Java, C#, or C.

DSL development is hard, requires deep domain knowledge and general-purpose language experience for development. DSLs are usually developed in collaboration with domain experts and senior developers. Only a limited number of people have expertise in both domain and programming knowledge with the related general-purpose language. The importance of DSL development is providing notation-based, similar to natural language, and an easy-to-use environment for non-tech people. Thus, domain experts and some business partners are included in the process without deep knowledge of the related general-purpose language.

The power of DSLs is coming from the underlying general-purpose language, DSLs can use all the functionality, such as a large set of public frameworks, useful data structures to hold parsed data from DSL models, file creating, exporting tools etc., of the general-purpose language. In this way, DSLs simplify the domain-restricted code and create an easy-to-use user interface for end-users.

Another advantage of using DSL is that it increases productivity in development once comparison between development while without DSL usage. “Their importance should not be underestimated as they are directly related to the to the productivity improvement associated with the use of DSLs” (Mernik et al., 2005). Even if the development process is hard and requires domain knowledge, DSLs increase productivity in sense of time, reusability for software components and ease-of-use for non tech people.

DSL development is investigated in 5 main steps such as decision, analysis, design, implementation, and deployment. Each main step is valuable and requires

domain expertise, development steps are examined in Chapter 4.

There are two commonly using meta-modeling frameworks, which are Eclipse XText and JetBrains Meta Programming System (MPS). Both provides powerful grammar language for DSL, parser, generator, type checker, and compiler. Main difference between XText and MPS is that XText parser-based framework and best fit for textual files, MPS projection-based framework and doesn't parse textual files.

In this thesis, Eclipse XText is preferred. Because XText framework is more flexible in sense of XText can work with textual files. Also, Eclipse XText provides Eclipse Modeling Framework (EMF) and code generation for building components. (Merks et al., 2009). Open Architecture Ware (OAW) is a modular Model Driven Architecture (MDA) or Model Driven Design (MDD) which is implemented in JAVA programming language. OAW stands for parsing models from given syntax, applying code analysis, and transform models into generated target output (Efftinge et al., 2006).

Example 3.1: A Smart Home Automation system will be constructed with DSL. Each device of Smart Home takes an identifier and n number of state that states are separated with a comma. For controlling the devices, Smart Home Automation takes rules and each rule takes a description field, when condition, and then action.

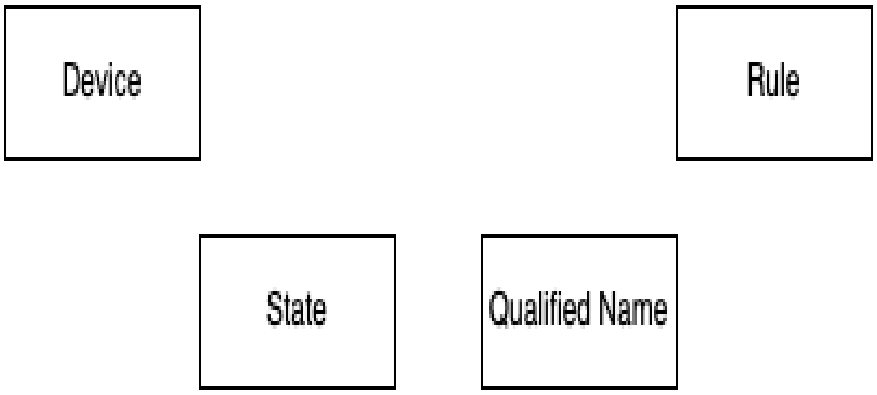

Figure 3.1. Smart Home Automation Domain Analysis Result

Definition 3.4: *An xText Grammar* consist of a set of rules (Model, Field, Comment, and Type). A rule is described using sequences of tokens. A token is either a reference to another rule or primitive tokens (INT, STRING, ID) (Efftinge et al., 2006).

In Figure 3.2, Smart Home Automation rules are declared. For one Declaration

grammar can take a device or a rule. A Device has a name and “can be” suffix after the name field and takes a list of states each state separated with a comma. The state contains only the name field. A rule contains a description, when state or qualified name, and then state or qualified name fields. A Qualified name takes an ID, and it is also can be combined with a comma.

```
Declaration:
    Device | Rule
;

Device:
    'Device' name=ID 'can' 'be'
        states += State (',' states+=State)* // Question mark makes it optional
;

State:
    name=ID
;

Rule:
    'Rule' description = STRING
        'when' when=[State|QualifiedName]
        'Then' then=[State|QualifiedName];

QualifiedName: ID("." ID)*;
```

Figure 3.2. Smart Home Automation DSL Grammar

Once the grammar development is done, the Generate XText Artifacts workflow is started. The workflow takes some time and the whole process is traceable on the console. When the workflow is complete, the DSL is ready to be launched as an Eclipse application. After the start of the application, DSL is ready to be used on the editor screen.

```
Rule 'Close Window, when heating turned on'
    when Heating.ON
    Then Window.CLOSED

Rule 'Switch off heating, when windows gets opened'
    when Window.OPEN
    Then Heating.OFF

Rule 'Switch Door, when Heating turned off'
    when Heating.OFF
    Then Door.CLOSED
```

Figure 3.3. Smart Home Automation Rule Declarations

```
Device Window can be OPEN, CLOSED
Device Heating can be ON, OFF
Device Door can be OPEN, CLOSED
```

Figure 3.4. Smart Home Automation Device Declarations

Eclipse XText editor checks the grammar syntax and warns if there is a missing part of the written grammar with red color. When all the device and rule declarations are completed, saving the editor screen triggers the file generator. The file generator generates a file from the declarations that are defined in DSL grammar. The generated file is shown in Figure 3.5.

```
public class Homeautomation {
	public static void fire(String event) {
		[com.mert.smarthome.rules.impl.StateImpl@b3c62a2 (name: OPEN), com.mert.smarthome.rules.impl.StateImpl@769b2d11 (name: CLOSED)]
		if(event.equals("OPEN")) {
			System.out.println("Window is now OPEN!");
		}
		if(event.equals("CLOSED")) {
			System.out.println("Window is now CLOSED!");
		}
		[com.mert.smarthome.rules.impl.StateImpl@2cc570d1 (name: ON), com.mert.smarthome.rules.impl.StateImpl@44042e99 (name: OFF)]
		if(event.equals("ON")) {
			System.out.println("Heating is now ON!");
		}
		if(event.equals("OFF")) {
			System.out.println("Heating is now OFF!");
		}
		[com.mert.smarthome.rules.impl.StateImpl@3da71bbb (name: OPEN), com.mert.smarthome.rules.impl.StateImpl@3ae3a726 (name: CLOSED)]
		if(event.equals("OPEN")) {
			System.out.println("Door is now OPEN!");
		}
		if(event.equals("CLOSED")) {
			System.out.println("Door is now CLOSED!");
		}
		if(event.equals("ON")) {
			fire("CLOSED");
		}
		if(event.equals("OPEN")) {
			fire("OFF");
		}
		if(event.equals("OFF")) {
			fire("CLOSED");
		}
	}
```

Figure 3.5. Smart Home Automation Generated JAVA File

Commonly known example of DSL usages that are SQL (Structured Query Language, using with relational databases), MATLAB (programming language designed for specifically scientists and engineers), HTML (Hyper Text Markup Language, is using for creating web pages). As we can see on the previous examples, the list of defined rules that are written on the user interface of designed DSL. Then all

the given rules compiled into a general-purpose language.

Example 3.2: DSL usage on Structured Query Language (SQL) is given in Figure 3.6. The related DSL syntax selects related columns from the given table name and orders the return values according to the given parameter in ascending or descending order.

```
SELECT colum1, colum2, …
FROM table_name
ORDER BY colum1, colum2, … ASC|DESC;
```

Figure 3.6. Sample DSL usage in SQL

3.2. Event Sequence Graphs

Several validation methods are proposed for testing in the software industry, such as Specification Oriented Testing, Implementation Oriented Testing. Each proposed methods identify relevant features to system under test. An Event Sequence Graph (ESG) is a way that represents behaviors of the system under test (Belli et al., 2005). The proposed methodology is an interactive system, which means that system reacts actions for user events or response triggered by the system.

The main difference between finite-state-automata (FSA) and ESG is that FSA represents states of the related system under test, but an ESG provides an abstraction layer to better understanding for event flow of the related system in external point of view. An ESG is a directed graph and each vertex of the ESG represents an event triggered by user interaction or system response. Directed edges of the ESG connects two events on the system.

Definition 3.5: An Event Sequence Graphs $ESG = (V, E)$ are directed graphs and has some rules. $V \neq \emptyset$ is a limited number of nodes and $E \subseteq V \times V$ is a finite set of arcs (edges), and $\Xi, \Gamma \subseteq V$ finite set of distinguished vertices with $\xi \in \Xi, \gamma \in \Gamma$, named as start vertices and end vertices, correspondingly, in which $\forall v \in V$ there will

be at least one vertex sequence $<\xi, v_0, \ldots, v_k, \gamma>$ by one $\xi \in \Xi$ to the next $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for i = 0, …, k-1 and $v \neq \xi, \gamma$ (Belli et al., 2006).

The start and finish vertices of the ESG are marked as applying the following convention: all $\xi \in \Xi$ are precipitated by such a pseudo vertex '*[*' $\notin V$ so all $, \gamma \in \Gamma$ are pursued by other pseudo node '*]*' $\notin V$ (Belli et al., 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 et al., 2005, 2006).

Example 3.3: For the ESG given in Figure 3.7, **V** = {*turn on heating, turn on lights, close windows, close curtains, heating turned on, lights turned on*}, **Ξ** = {*turn on heating, turn on lights*}, **Γ** = {*heating turned on, lights turned on*}, and **E** = {(*turn on heating, close windows), (close windows, heating turned on), (turn on lights, close curtains), (close curtains, lights turned on).* E does not include the edges from entry vertex (*'[')* and to exit vertex ('*]*').

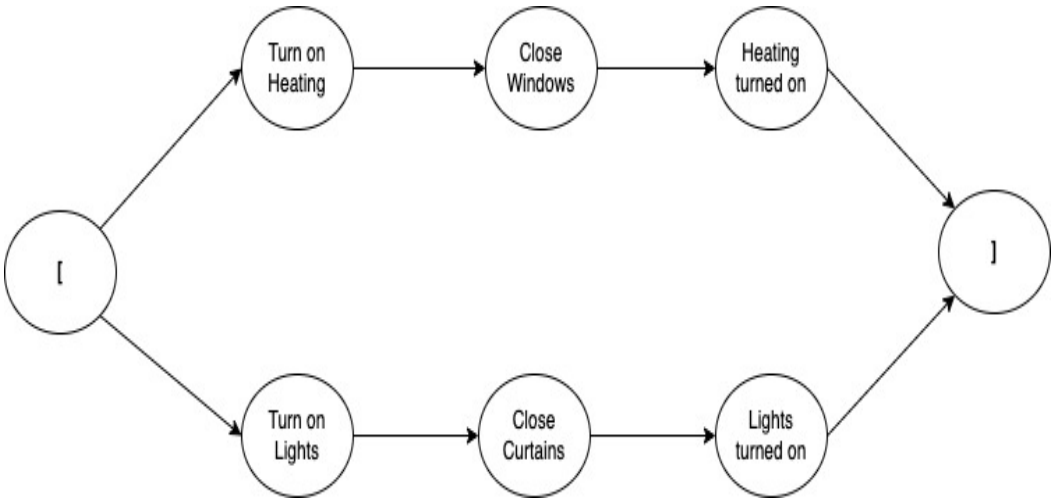

Figure 3.7. Smart Home Automation ESG

Definition 3.6: In Definition 3.5, assume that V and E are described. So any nodes sequence $< v_0, \ldots, v_k >$ is known as an ES (Event Sequence) if $(v_i, v_{i+1}) \in E$, for i = 0, …, k-1 (Belli et al., 2004, 2005, 2006).

Example 3.4: *In Figure 3.7, turn on heating - close windows – heating turned on* is an event sequence that has length of 3.

Definition 3.7: The Event Sequence is a CES (Complete Event Sequence),

where $\propto (ES) = \xi \in \Xi$ is the entry and $\omega(ES) = \gamma$ is the exit vertex (Belli et al., 2004, 2005, 2006).

Example 3.5: For the ATM deposit ESG in Figure 3.8, {*deposit money, enter amount, show error*} is an Event Sequence of length 3. Complete Event Sequences for the related graph are {(deposit money, show error, enter amount), (deposit money, enter amount, show error, enter amount). Each edge on the ESG marked as a legal Event Pair (EP), also each event pairs represented with ES with fixed length where length is 2. For Figure 3.2, Event pairs (EP) as follows:

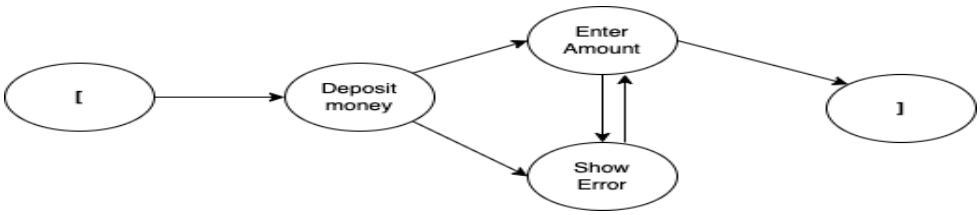

Figure 3.8. ESG of ATM deposit

EP = {EP_1 = (*deposit money, enter amount*), EP_2 = (*deposit money, show error*), EP_3 = (*enter amount, show error*), EP_4 = (*show error, enter amount*)}.

3.2.1. Sub Event Sequence Graphs (Sub ESGs)

Maintainability, readability, refactor, and new feature development processes are getting complex when dealing with large scale projects. ESG modularization comes to the stage at this point. The ESG contains vertices that are abstract, and modularized in any layer or under it. The final ESG can be refined with refinement of each sub ESG.

Definition 3.8: Given an ESG, say $ESG_1 = (V_1, E_1, \Xi_1, \Gamma_1)$ a vertex $v \in V_1$, and an ESG, say $ESG_2 = (V_2, E_2, \Xi_2, \Gamma_2)$. Then, replacing v by ESG_2 produces a refinement of ESG_1, say $ESG_3 = (V_3, E_3, \Xi_3, \Gamma_3)$ with $V_3 = V_1 \cup V_2 \setminus \{v\}$ and $E_3 = E_1 \cup E_2 \cup E_{pre} \cup E_{post} \setminus E_{1\ replaced}$ ('\': set difference operation), where in $E_{pre} = N(v) \times \Xi_2$ (connections of the predecessors of v with the entry nodes of ESG_2), $E_{post} = \Gamma_2 \times N^+(v)$ (connections of

exit nodes of ESG_2 with the successors of v), and $E_{1\ replaced} = \{(v_i, v), (v, v_k)\}$ with $v_i \in N^-(v)$ and $v_k \in N^+(v)$ (replaced arcs of ESG_1) (Belli et al., 2005, 2007).

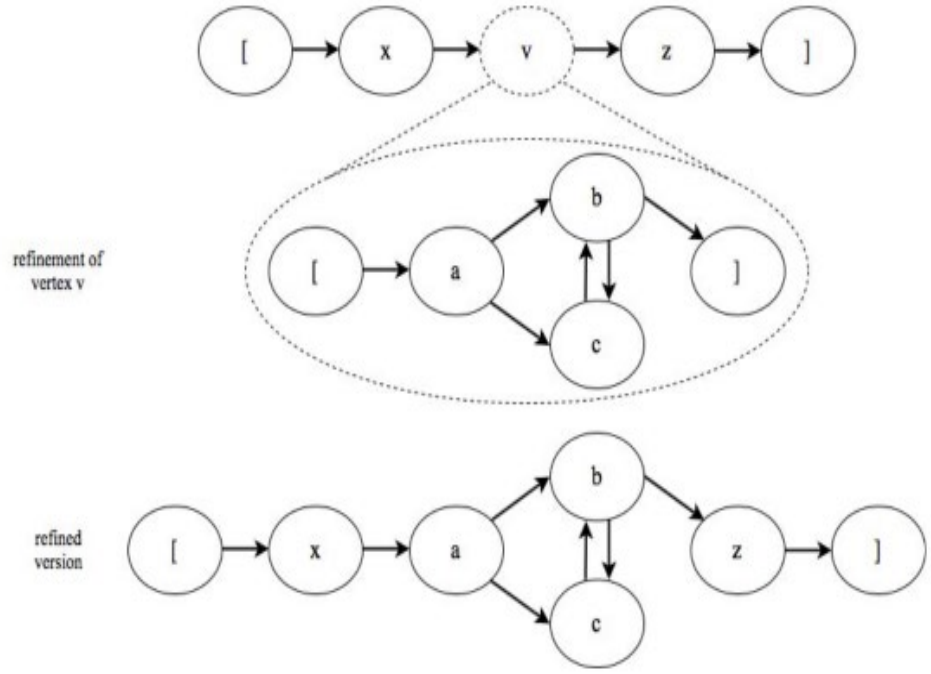

Figure 3.9. Refinement process of a Sub ESG (Tuglular, 2018)

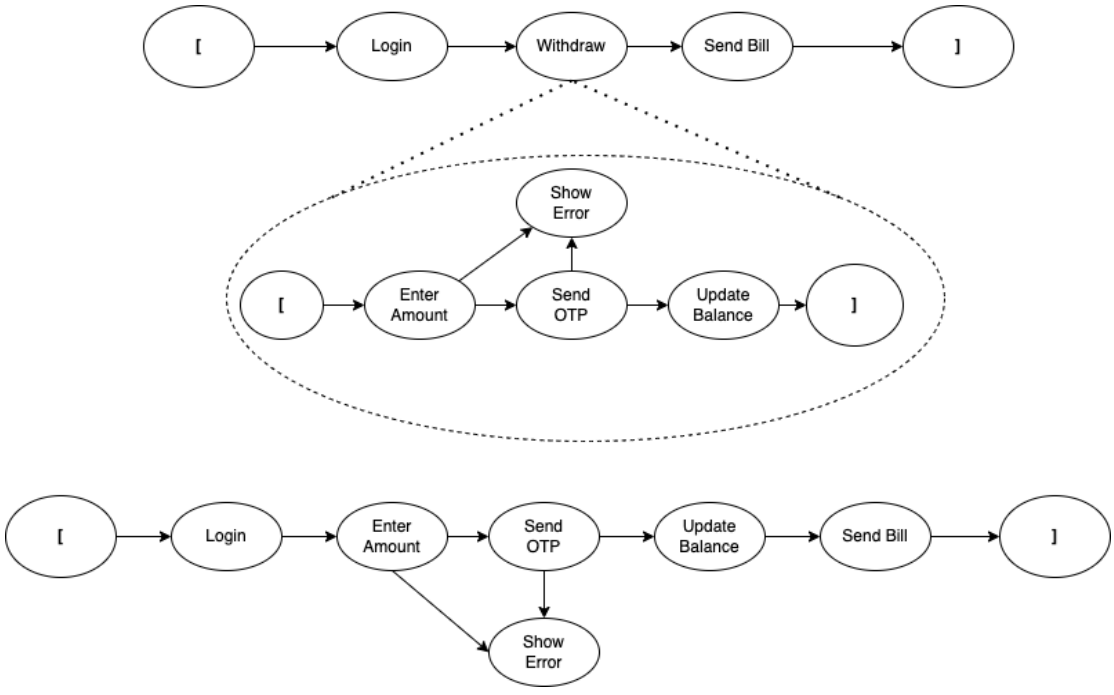

Figure 3.10. ATM Refined ESG with Sub ESG

Example 3.6: The ESG in Figure 3.10. shows the refinement operation on ESG as stated in Definition 3.4. The event modeling consists of three user story, as follows *login, withdraw, and send bill*. Withdraw vertex refined by another ESG, the Sub ESG consist of *enter amount, show error, send otp, and update balance* vertices. On the refinement process the entry and exit vertices eliminated and connected to the relevant parts of the refined ESG.

Definition 3.9: A tagged ESG is an ESG, where a node or vertex may contain a tag instead of an event (Tuglular, 2021).

Figure 3.11. Tagged ESG (Tuglular, 2021).

3.2.2. Augmented Event Sequence Graphs by Decision Tables

Decision tables are a graphical illustration for identifying the actions under given set of conditions. A decision table (DT) logically links conditions ("if") with actions ("then") that are determined by combinations of given conditions ("rules"). Decision tables are useful when the decision to create a DT during system design.

Definition 3.10: Decision Table is represented with a table of DT = (X, Y, Z) trio (Tuglular et al., 2016b). Where $X \neq \emptyset$ and $X = \{x_1, \ldots, x_n\}$ is a limited series of conditions, $Y \neq \emptyset$ and $Y = \{y_1, \ldots, y_m\}$ is a limited number of actions and $Z \neq \emptyset$ and $Z = \{z_1, \ldots, z_k\}$ is a limited number of rules, each of which perform specific actions based on a predefined number of conditions (Murnane et al., 2001).

Definition 3.11: Assume Z is declared in Definition 3.10. Then, $\forall z \in Z$ can be defined as $z = (X_{true}, X_{false}, Y_m)$, where $X_{true} \subseteq X$ is the series of requirements to be met.

$X_{false} \subseteq X$ is the number of conditions that must be false . $Y_m \subseteq Y$ *denotes the* set of actions to be taken if all $a \in X_{true}$ are settled to true and all $b \in X_{false}$ are settled to false (Murnane & Reed, 2001). Below ordinary situations: $X_{true} \cup X_{false} = X$ and $X_{true} \cap X_{false} = Ø$ (Murnane et al., 2001). If a condition is not perceived in certain scenarios, it is merely denoted as '-' (ignore) in the rule (Murnane & Reed, 2001). The real number of DT rules can be easily calculated based on the number of '-' in each rule as follows: If $m < |X|$ is the number of '-' in $z \in Z$, then the set of rules replaced by '-' is 2^m (Murnane et al., 2001).

Definition 3.12: Assume Z is defined in Definition 3.10. The highest set of rules in DT will be $2|X| = 2t$, (Tuglular et al., 2016). Complete DT is defined as DT with $|Z| = 2t$. If $|Z| > 2t$, the DT is inconsistent and should be reconstructed (Murnane et al., 2001).

Example 3.7: DT defined in Definition 3.9, Table 3.1 is an example of DT. Where C = {*printer does not print, red light is flashing, printer unrecognized*} is condition set of DT. A = {*check power cable, ensure software is installed, check for paper jam*} is action set of DT. Finally, $Z = \{Z_1, Z_2, Z_3, Z_4\}$ is rule set of DT.

Table 3.1. Printer DT

		Rules			
		R_1	R_2	R_3	R_4
Conditions	Printer does not print	T	T	T	F
	Red light is flashing	F	T	T	T
	Printer unrecognized	F	F	T	F
Actions	Check power cable	-	-	X	-
	Ensure software is installed	X	-	X	-
	Check for paper jam	X	X	-	X

For the given Definitions 3.9, 3.10, and 3.11 the maximum number of condition combination represented with $2^{|C|}$, where |C| stands for the condition numbers. This combination of conditions makes a mess on the Event Sequence Graph. To escape this

complexity, Event Sequence Graph Augmented by a refined vertex that contains the related decision table.

Example 3.8: In Figure 3.11, Login ESG contains password entered vertex refined by DT. The purple triple octagon shape is representing a DT. DT is given in Figure 3.2.2.2. For the related DT, C = {C0, C1, C2, C3}, A = {A0, A1, A2}, and R = {R0, R1}.

Figure 3.12. Login ESG augmented by DT

The vertex *password entered* given in Figure 3.12 contains a decision table. Conditions, actions, rules, and a table for the property details of the decision table is illustrated in Figure 3.13.

DT_PasswordEntered

-	R0	R1
C0	*T*	*T*
C1	*F*	*T*
C2	*F*	*T*
C3	*T*	*T*
A0	*X*	-
A1	-	*X*
A2	*X*	-

Table Properties

C0	password is type of string
C1	password length greater than or eq. 10 AND password length less than 100
C2	password should have upper case AND password should have lower case
C3	password should have special character OR password should contain number
A0	User name Entered
A1	Login Successful
A2	Show Error

Figure 3.13. Password Entered DT

CHAPTER 4

EVENT SEQUENCE GRAPH DOMAIN SPECIFIC LANGUAGE

This part of the thesis focuses on Event Sequence Graph (ESG) Domain Specific Language (DSL) development stages in detail. ESG DSL introduces a new graph drawing approach which is easy-to-use, more likely to natural language, abstracted from programming and domain expertise. Also, the related DSL provides modularization for sub-ESGs and Augmented ESGs by Decision Tables.

4.1. Decision

The decision to develop DSL is a crucial point to take into consideration because the development process takes a long time and requires deep domain knowledge. The main purpose of the related DSL is to draw Event Sequence Graphs (Belli, 2001) using a well-defined grammar. Secondly, ESG DSL target tech people and non-tech people who are business partners. With DSL development the abstraction layer is created between the tech infrastructure and non-tech people. Thirdly, using DSL increases reusability of software artifacts such as the use of software frameworks in a general-purpose language. In Figure 4.1, vertex grammar model is defined and the vertex is reusable. With the same vertex model, the “ID” and “Event” properties are signature for it, we can draw two different vertices without reimplementation of it.

```
Vertex
ID "4"
Event "Login Successful"
```

Figure 4.1. ESG DSL vertex definition

On the other hand, there are some disadvantages to giving a decision for developing a DSL. Initially, you must use your resources for development such as developers, time, and business partners. Once the development process is over, the education period is required for users of ESG DSL. Since the feature set expands, there will be a need for maintaining and developing cost on the related DSL repository.

Test Suite Designer (TSD, please see Appendix A) tool draws ESGs and generates test suites from the related ESGs. The tool is developed with JAVA general-purpose language, double click creates a vertex on the editor page, each edge is created with drag-and-drop action from source vertex to target vertex. There is no modularization support in the tool but tagged ESG (Tuglular 2021) is used as an alternative for it. The user experience is very difficult to dealing with complex project modeling with TSD tool. TSD tool also allows copy and paste operation for vertices and edges on the graph. But it is not working as expected, when you try to copy, there are some missing parts that might be occurred while copy operation. In Figure 4.2, ATM card read operation tagged ESG is given.

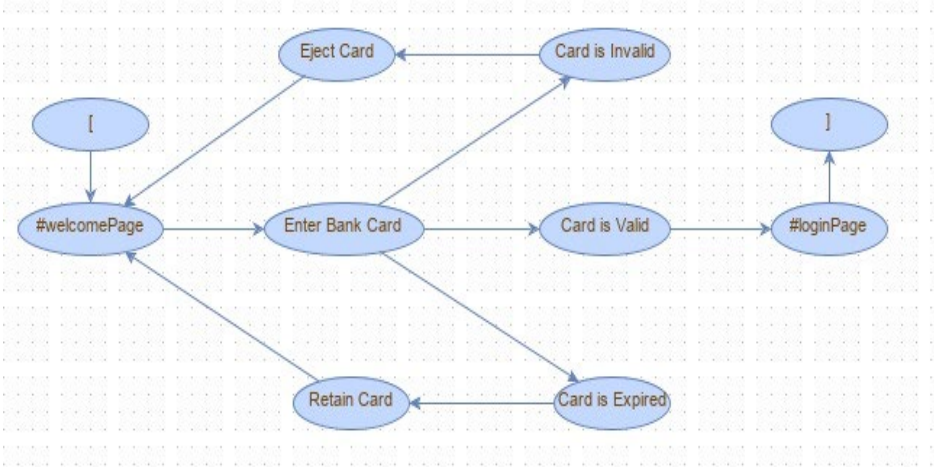

Figure 4.2. Sample ESG developed with TSD.

Finally, the decision to develop ESG DSL is made to increase the productivity of the users (Mernik et al., 2005), enhance reusability of software artifacts, design better user experience with easy-to-use grammar for the users. DSL creates an abstraction layer that provides some advantages such as develop models with less domain and tech information in a restricted domain (Fischer et al., 2004; Nardi, 1993).

4.2. Analysis and Design

During the analysis part, the problem set is examined, the problem domain is identified, and the required information extracted from sources such as analysis documents, domain experts, existing code repository. In this thesis, domain knowledge provided by the existing code repository (ESG-Engine, please see Appendix A). Designing a DSL is investigated into two dimension which are the relationship between the DSL and existing languages (for ESG DSL it is JAVA general-purpose language) and the nature of the design description (Mernik et al., 2005).

In the domain analysis of ESG DSL development, the problem domain is the development of a domain specific language which provides a graphical user interface that acts like an editor for DSL, textual parsing, syntax highlighting, error checking, model converter from syntax, and generator for the target output. Java Script Object Notation (JSON) is specified as the target output of the DSL. JSON is widely used in communication technologies such as Representational State Transfer (REST), and GraphQL (Brito et al., 2020).

The domain analysis is done as the same with domain analysis strategy in database management, Algorithm 4.1 is used for domain identification.

Algorithm 4.1. ESG DSL Domain Analysis Algorithm

Input: $List_i$ = (Domain Entity) – an ESG domain entity list
k – integer number of elements in the input list
Output: $List_j$ = (Domain Entity) reified domain entity list

```
for n=1 to k incrementing by 1 do
    val element = List_i[n]
    if (Check element matches a model in DSL)
        List_j.add(element)
endfor
```

All the candidate entities are written on a page, where C(E) stands for set of entities C(E) = {*Edge, Arrow, Direction, Graph, Vertex, Event, Identifier, Source,*

Target, Inner ESG, Abstraction, Decision Table, Rule, Action, Condition, Result, Expression, Literal, Operand, Pattern, Recognition, Recursion, Shape, Color}. C(E) is given as input to Algorithm 4.1, then the output of the domain analysis is gathered. In Figure 4.3 The domain analysis result is illustrated.

Figure 4.3. ESG DSL domain analysis result

For the next step, double validation process is applied with ESG-Engine project (Öztürk, 2020). The related project contains all the ESG models, the required fields of the entities, and the member functions. The missing entity models are finalized by merging the results from the Algorithm 4.1 output entity list and the checking missing entity models from the code repository.

As the last step of domain analysis, conceptual analysis (Compatangelo et al., 2002) is performed for showing entity relations briefly. This modeling technique provides entity identifiers, attribute list, attribute relations, cardinality of constraints. It

also shows the inheritance relationships of the ESG DSL grammar entities. The output of the conceptual analysis is given in Table 4.1. The table shows entities, identifiers, attributes, and relation of attributes.

Table 4.1. ESG DSL Conceptual Domain Analysis

entity ESG **has identifier** name as STRING **has attributes** event as 1:1 EVENT subESGs as 1:N ESG edges as 1:N EDGE	**entity** DT **has identifier** id as INT **has attributes** name as 1:1 STRING conditions as 1:N CONDITION rules as 1:N RULES actions as 1:N ACTIONS
entity VERTEX **has parents** ESG; **has identifier** ID as INT **has attributes** event as 1:1 EVENT color as 1:1COLOR dt as 1:1 DT	**entity** RULE **has identifier** ID as INT **has attributes** name as 1:1 STRING value as 1:1 STRING variables as 1:N VARIABLE actions as 1:N ACTION
entity EDGE **has attributes** source as 1:1 INT target as 1:1 INT color as 1:1 COLOR	**entity** ACTION **has identifier** ID as INT **has attributes** name as 1:1 STRING event as 1:1 INT
entity EVENT **has attributes** name as 1:1 STRING	**entity** CONDITION **has attributes** name as 1:1 INT evals as 1:1 EVALUABLE
entity CONNECTIVE **has attributes** connective as 1:1 AND | OR	**entity** EVALUABLE **has attributes** expression as 1:1 EXPRESSION or connective as 1:N CONNECTIVE
entity EXPRESSION **has attributes** left as 1:1 LITERAL operand as 1:1 OPERAND right as 1:1 LITERAL	**entity** DECLARATION **has attributes** esg as 1:1 ESG **(cont. on the next page)**

cont. of Table 4.1.

entity VARIABLE **has attributes** name as 1:1 STRING value as 1:1 LITERAL	**entity** OPERAND **has attributes** operand as 1:1 < | > | == | <= | >=
entity LITERAL **has attributes** literal as 1:1 INT | STRING **entity** COLOR **has attributes** name as 1:1 black | red | green | blue | orange	**entity** SUBESG **has parents** ESG; **has identifier** ID as INT **has attributes** event as 1:1 EVENT subESGs as 1:N ESG edges as 1:N EDGE

The design characteristics are grouped in two categories, which are the relationship with the DSL and the existing code repository, and the relationship with DSL and the nature of the design specifications. To avoid entity confusion, the same concepts and the entities are chosen from the existing code repository (ESG-Engine, please see Appendix A).

The set of entities extracted from existing code repository C(E) = {*Edge, Vertex, Event, Inner ESG (sub ESG), Decision Table (DT), Rule, Action, Condition, Result, Expression, Literal, Operand*}.

The general-purpose language design principles provided (Brooks, 1996) are applied for the rest of the entities such as Color, Declaration, Variable, etc. The provided principles include readability, simplicity, and orthogonality design criteria.

In Figure 4.4, the ESG DSL entity relation is given. At the root, there is an ESG contains elements under it. Elements must either be an ESG (sub ESG) or a vertex again. With this logic there is a recurrence relation comes into the stage. Each sub ESG is also an ESG and contains elements that must either be an ESG or a vertex again. Each ESG contains edges and one event. Vertex has the possibility to store a decision table on it. That is helpful in extracting actions when dealing with user inputs under some conditions. One decision table contains a set of rules, actions, and conditions. One condition contains f-number of expressions that are connected with connectives. The cardinality information is taken from the conceptual analysis and shown on the ESG DSL entity relation graph.

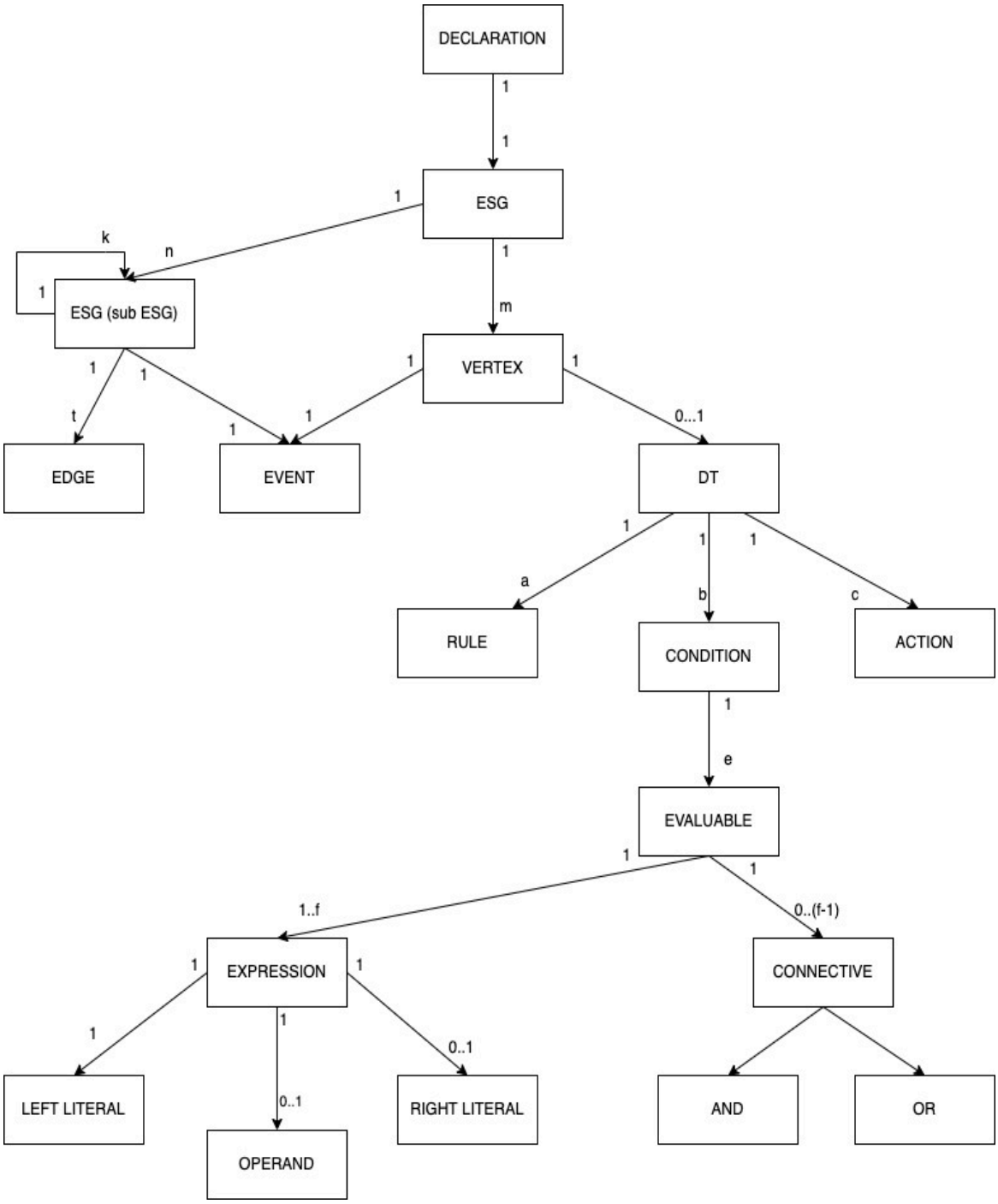

Figure 4.4. ESG DSL Entity Relation Diagram

4.3. Implementation

The DSL implementation step begins after the DSL design and domain analysis steps completed. Eclipse XText framework is used to develop ESG DSL, the related

framework provides textual parsing, error checking, syntax highlighting, code compiler, general-purpose language support, model converter from syntax, and generator for the target output.

In implementation step, there are several implementation patterns proposed. Table 4.2 Implementation Patters for Executable DSLs taken as it is (Mernik et al., 2005).

Table 4.2. Implementation Patterns for Executable DSLs

Pattern	Description
Interpreter	DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle. This approach is appropriate for languages having a dynamic character or if execution speed is not an issue. The advantages of interpretation over compilation is greater simplicity, greater control over the execution environment, and easier extension.
Compiler/application generator	DSL constructs are translated to base language constructs and library calls. A complete static analysis can be done on the DSL program/specification. DSL compilers are often called application generators.
Preprocessor	DSL constructs are translated to constructs in an existing language (the base language). Static analysis is limited to that done by the base language processor.
Embedding	DSL constructs are embedded in an existing GPL (the host language) by defining new abstract data types and operators. Application libraries are the basic form of embedding.
Extensible compiler/interpreter	A GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind.
Commercial Off-The-Self(COTS)	Existing tools and/or notations are applied to a specific domain.
Hybrid	A combination of the above approaches.

From Table 4.2, compiler/application generator pattern is selected for ESG DSL development. Our expectation from the ESG DSL is firstly model translations into Java objects, then generation of the target file (JSON file). In the ESG domain, DSL defines domain-related rules and at the end of the process generates a target file (JSON). Compilation and interpretation are important for both general-purpose languages and DSLs. Spinellis (Spinellis, 2001) examined DSL development and general-purpose language development is quite different, DSL development requires more effort and time-consuming at the development steps. But effective and timesaving for the usage on the restricted domain.

The implementation step for ESG DSL is developed with Eclipse XText framework using Eclipse IDE. The grammar structure of ESG DSL is created according to relationship of the entities from Table 4.1 and Figure 4.4. The creation process started with ESGs and inner ESGs (sub-ESGs), followed by augmented ESG with Decision Tables (DTs).

```
Event:
name = STRING
;
```

Figure 4.5. ESG DSL Event Grammar Definition

Firstly, event model created in ESG DSL grammar. Event model includes name field as type of string. Event is used with ESG, sub-ESG, and Vertex.

The color grammar model is created to color the edges and vertices while drawing graphs with DOT language on the Graphviz framework. Each edge and vertex element of the JSON contains a color property and it comes black by default.

```
COLOR:
      name = ("black"|"red"|"green"|"blue"|"orange")
;
```

Figure 4.6. ESG DSL Color Grammar Definition

```
EDGE: 'Edge' name = STRING
      'Source' source= INT
      'Target' target= INT
      ('Color' color= COLOR)?
;
```

Figure 4.7. ESG DSL Edge Grammar Definition

The edge grammar model is created to represent an edge from the source vertex to the target vertex. The source field takes the source vertex identifier as integer and the target field takes the target vertex identifier as integer. Edges have also the color field, it is an optional field, and it comes black by default. The question mark states that it is an optional field. Each edge of the ESG DSL is directed edge and the direction from source vertex to target vertex.

```
VERTEX: 'Vertex'
      'ID' ID = STRING
      'Event' event = Event
      ('Color' color= COLOR)?
      (dt = DT)?
;
```

Figure 4.8. ESG DSL Vertex Grammar Definition

The vertex grammar model is created to represent a vertex in ESG, a vertex has an identifier field, an event field that holds the vertex's event, an optional color field it comes black by default, and a decision table field which is optional.

```
ESG: 'ESG' name = STRING
     ('Event' event = Event)?
     subESGs += (VERTEX | ESG) (',' subESGs += (VERTEX | ESG))*
     edges += EDGE(',' edges += EDGE)*
;
```

Figure 4.9. ESG DSL ESG Grammar Definition

The ESG grammar model is created to represent both the root ESG and the sub-ESGs (inner ESGs). Event field is an optional field because the root ESG is not taking event field, but each sub-ESGs are taking event field to represent the event for the refined vertex's event on the main ESG. Each sub-ESG can be either ESG again or a vertex, sub-ESGs are separated by a comma and the asterisk sign means for sub-ESGs are not an optional field. Each ESG contains edges separated by a comma and the edges field is not an optional field.

```
CONDITION: "Condition" name = INT
      evals += EVALUABLE(evals += EVALUABLE)*
;


EVALUABLE:
      EXPRESSION | CONNECTIVE
;


EXPRESSION:
    "("
    left = LITERAL
    (operand = OPERAND)?
    (right= LITERAL)?
    ")"
;

CONNECTIVE:
      connective = ("AND" | "OR")
;
```

Figure 4.10. ESG DSL Condition Grammar Definition

Secondly, ESG DSL grammar is extended to support vertex augmentation by decision tables The condition grammar model is created to represent conditions for decision tables. Each condition takes name field as identifier for it and takes set of evaluable models as required field. Each evaluable grammar model must be either an expression or a connective. Expression DSL model contains left literal, an operand (<, >, <=, >=), and right literal. A Literal also grammar model and it accepts input as string or integer. Evaluable can be a connective grammar model which acts as a conjunction between two expressions.

```
RULE: "Rule" name = STRING
      "ID" ID = INT
      "value" value = STRING
      "variables" vars += VARIABLE("," vars += VARIABLE)*
      "actions" actions += [ACTION]("," actions += [ACTION])*
;

ACTION: "Action" name = ID
      "ID" ID = STRING
      "event" event = INT
;


VARIABLE: 'var' name = STRING
      'value' value = LITERAL
;
```

Figure 4.11. ESG DSL Rule Grammar Definition

The rule grammar model is created to represent decision table's rules. Each rule has a name field as string, an identifier field as integer, a value field that contains a string, every char of this string corresponds to sequential condition results. If there are three condition (C_0, C_1, C_2) and value is equal to "TFT" that means C_0 = T, C_1 = F, C_2 = T. Variable grammar model is created to hold input variables of the rules, rule grammar model has set of variables separated by commas and variables field is not optional.

The action grammar model is created to hold the rule's actions that triggers under certain circumstances. Rule grammar model has set of actions separated by commas and actions field is not optional.

```
  DT: "DT" name = STRING
       "ID" ID = INT
       conditions += CONDITION (',' conditions += CONDITION)*
       actions += ACTION (',' actions += ACTION)*
       rules += RULE (',' rules += RULE)*
  ;
```

Figure 4.12. ESG DSL Decision Table Grammar Definition

The Decision Table (DT) grammar model is created to represent an augmentation operation of a vertex. The DT has a name field as string, an identifier

field as integer, set of conditions separated by commas and the condition field is not optional, set of actions separated by commas and the actions field is not optional, set of rules separated by commas and also the rules field is not optional.

Once the DSL grammar implementation is done, then the DSL generator class implementation process is started to create the desired output file (in our thesis the desired output is JSON file). XText converts the grammar syntax from the editor pane to EMF (Eclipse Modeling Framework) models. In Figure 4.13, the base generator class, provided by the Eclipse XText framework, is given.

```
class MyDslGenerator extends AbstractGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
        fsa.generateFile('greetings.txt', 'People to greet: ' +
            resource.allContents
                .filter(Greeting)
                .map[name]
                .join(', '))
    }
}
```

Figure 4.13. Default XText Generator for DSLs

In Figure 4.14, the skeleton of the output file is given. To create a JSON file which contains id as integer, name as string, edge array, and vertices array that contains both vertices and sub-ESGs. ESG DSL Generator class generates a JSON file that is suitable with given skeleton.

```
{
        "ID": 0,
        "name": "refinedESG",
        "vertices":[
                ...
        ],
        "edges":[
                ...
        ]
}
```

Figure 4.14. ESG DSL output skeleton

```
class MkDslGenerator extends AbstractGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
        val simpleClassName = resource.getURI.trimFileExtension.lastSegment
        if(resource.contents?.head === null) {
            return;
        }
        val declarations = resource.contents.head.eContents.filter(Declaration)
        fsa.generateFile(simpleClassName + '.json', '''
        {
        «FOR graph : declarations.filter(ESG)»
        "ID": «graph.name»,
        "name":"«graph.event.name»",
        "vertices":[
            «FOR subESG : graph.subESGs SEPARATOR ","»
                «IF(subESG instanceof ESG)»
                    «runSubESGRule(subESG)»
                «ENDIF»
                «IF(subESG instanceof VERTEX)»
                    «runVertexRule(subESG)»
                «ENDIF»
            «ENDFOR»
        ],
        «runEdgeRule(graph.edges)»
        «FOR subESG : graph.subESGs»
            «IF(subESG instanceof VERTEX)»
                «IF (subESG.dt !== null) »
                ,
                «runDecisionTableRule(subESG.dt, subESG)»
                «ENDIF»
            «ENDIF»
        «ENDFOR»
        }
        «ENDFOR»
    '''
    )}
```

Figure 4.15. ESG DSL Generator Implementation

The generator inherits “doGenerate” method from AbstractGenerator which is provided by the XText framework. ESG DSL Generator filters all declarations from the model set and apply filter operation to find root ESGs in the filtered declarations. For each root ESG, the generator class generates JSON objects appropriate for the skeleton in Figure 4.14. If a vertex model has DT attribute, then it generates the DT JSON objects with vertex. For the full version of the ESG DSL generator, see implementation part in Appendix A.

4.4. Graph Visualization

Once the desired JSON file generated, the generated file is read and converted into Java objects by ESG-Engine project. The ESG-Engine project details are given in Appendix A. After the Java object conversation is done, ESG structure is designed by using DOT Language with Graphviz framework support. Graphviz provides graph visualization for tools and web applications in software engineering, knowledge representation, bioinformatics, databases, networking (Ellson et al., 2003). DOT

Language is providing a modularization layer for ESG and sub-ESGs (inner ESGs). For the main ESG is covered by digraph root element. Each of the sub-ESGs represented with a subgraph element on dot file. DOT language skeleton for ESG is given in Figure 4.16.

```
digraph ESG (name){
        subgraph clusterSubESG {
        …
        }
        vertex definitions…
        edge definitions…
}
```

Figure 4.16. ESG Visualization Skeleton

The DOT Language is defined by the following abstract grammar. Single quoted are used for literal characters. When needed, parentheses "(" and ")" indicate grouping. Optional items are enclosed in square brackets "[" and "]". Node, edge, graph, digraph, subgraph case insensitive terms. Compass point values are not keywords, this syntax stand for using as other identifiers. An edge operator is "-->" for directed graphs and "--" is for undirected graphs. For the ESG DSL, directed edges are used for all the edges.

Table 4.3. Graphviz DOT Language Syntax (Graphviz, n.d.)

Graph	[strict] (graph | digraph) [ID] '{' element list '}'
Element list	[element [';'] element list]
Element	node | edge | attr | ID | subgraph
attr	(graph | node | edge) attr list
subgraph	[subgraph [ID]] '{' element list'}'
Compass_pt	(n | ne | e | se | s | sw | w | nw | c | _)

In Graphviz, subgraphs serve three main objectives. A subgraph, for example, can be used to express graph structure by signaling that specific nodes and edges should be clustered inside to create an abstraction layer. Also, subgraphs usually used to specify semantic information about the graph components. Subgraphs can be used as handy shorthand for edges.

```
subgraph test {
        A → B
        A → C
}
```

Figure 4.17. Subgraph Definition with DOT Language

In Figure 4.17, simple subgraph syntax is given. The subgraph contains three nodes respectively A, B and C. There are two directed edges, first one from A to B and the second one from A to C. In Figure 4.18, the subgraph, given in Figure 4.17, is visualized by Graphviz framework.

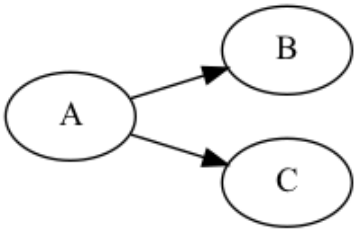

Figure 4.18. Subgraph Visualization with Graphviz

The root graph is defined as digraph, rank direction as left to right, one label for main ESG, vertex declarations, and edge declarations are included in ESG DOT file. Each vertex has a label and ellipse shape attribute as optional fields that are given inside

the square brackets. For each vertex of the graph, vertex names given with pre-tagged by ESG name. Then under score with vertex name. In figure 4.19, simple ESG with {*a, b, c, d*} vertices are given.

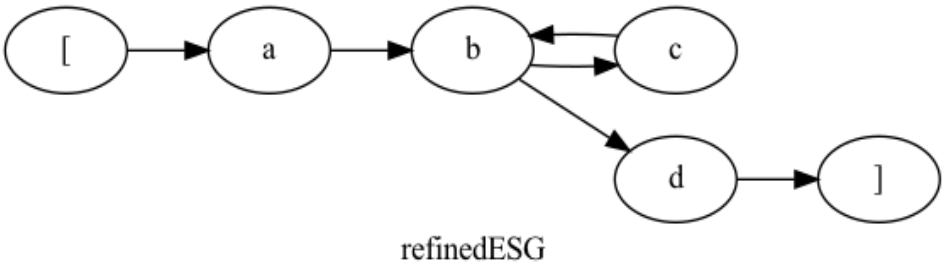

Figure 4.19. Simple ESG Visualization with Graphviz

To construct the ESG in Figure 4.19, digraph root element is used because the edges of the ESG is directed from source vertex to target vertex. Rank direction is set left to right. The vertex and edge declarations are given in Figure 4.20, the ellipse shape is used for all the simple vertices.

```
digraph G {
      rankdir = LR
      label = "refinedESG";
      esg0_vertex0 -> esg0_vertex1;
      esg0_vertex1 -> esg0_vertex2;
      esg0_vertex2 -> esg0_vertex3;
      esg0_vertex3 -> esg0_vertex2;
      esg0_vertex2 -> esg0_vertex4;
      esg0_vertex4 -> esg0_vertex5;
      esg0_vertex0 [label = "[", shape = "ellipse"]
      esg0_vertex1 [label = "a", shape = "ellipse"]
      esg0_vertex2 [label = "b", shape = "ellipse"]
      esg0_vertex3 [label = "c", shape = "ellipse"]
      esg0_vertex4 [label = "d", shape = "ellipse"]
      esg0_vertex5 [label = "]", shape = "ellipse"]
}
```

Figure 4.20. DOT Language Syntax for Simple ESG

There is an abstraction layer support with subgraph clusters for sub-ESG (inner ESG) creation with using DOT language on Graphviz framework. The sub-EGS is also another ESG, the refined node of the main ESG shaped as double circle on the graph.

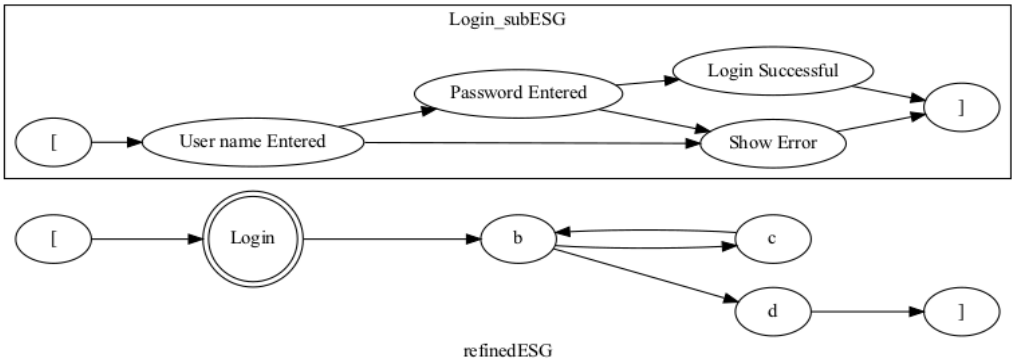

Figure 4.21. Refined ESG with Login sub-ESG

For visualization of the refined ESG which is given in Figure 4.21, the subgraph element is used to represent sub-ESG. The login cluster indicates another ESG layer just similar to the ESG DSL abstraction layer.

```
digraph G {
        rankdir = LR
        label = "refinedESG";
        esg0_vertex0 -> esg0_vertex1;
        esg0_vertex1 -> esg0_vertex2;
        esg0_vertex2 -> esg0_vertex3;
        esg0_vertex3 -> esg0_vertex2;
        esg0_vertex2 -> esg0_vertex4;
        esg0_vertex4 -> esg0_vertex5;
        esg0_vertex0 [label = "[", shape = "ellipse"]
        esg0_vertex1 [label = "Login", shape = "doublecircle"]
        esg0_vertex2 [label = "b", shape = "ellipse"]
        esg0_vertex3 [label = "c", shape = "ellipse"]
        esg0_vertex4 [label = "d", shape = "ellipse"]
        esg0_vertex5 [label = "]", shape = "ellipse"]

        subgraph clusterLogin_subESG {
                label = Login_subESG
                color = "black"
                label = "Login_subESG";
                esg1_vertex0 -> esg1_vertex1;
                esg1_vertex1 -> esg1_vertex3;
                esg1_vertex1 -> esg1_vertex2;
                esg1_vertex2 -> esg1_vertex3;
                esg1_vertex2 -> esg1_vertex4;
                esg1_vertex4 -> esg1_vertex5;
                esg1_vertex3 -> esg1_vertex5;
                esg1_vertex0 [label = "[", shape = "ellipse"]
                esg1_vertex1 [label = "User name Entered", shape = "ellipse"]
                esg1_vertex3 [label = "Show Error", shape = "ellipse"]
                esg1_vertex2 [label = "Password Entered", shape = "ellipse"]
                esg1_vertex4 [label = "Login Successful", shape = "ellipse"]
                esg1_vertex4 [label = "Login Successful", shape = "ellipse"]
                esg1_vertex5 [label = "]", shape = "ellipse"]
        }
}
```

Figure 4.22. DOT Language Syntax for Refined ESG

DT_PasswordEntered

-	R0	R1
C0	*T*	*T*
C1	*F*	*T*
C2	*F*	*T*
C3	*T*	*T*
A0	*X*	-
A1	-	*X*
A2	*X*	-

Table Properties

C0	password is type of string
C1	password length greater than or eq. 10 AND password length less than 100
C2	password should have upper case AND password should have lower case
C3	password should have special character OR password should contain number
A0	User name Entered
A1	Login Successful
A2	Show Error

Figure 4.23. Decision Table visualization with Graphviz

Decision table visualization process can be divided into three main steps. Firstly, change the decision table container vertex's shape from ellipse to triple octagon on main ESG. Secondly, decision table properties table is created to describe all the conditions and actions given as input to decision table. Thirdly, decision table is visualized in a table that contains rules in x-axis, conditions, and actions in y-axis. In Figure 4.23, the visualized decision table properties table and the decision table itself is given.

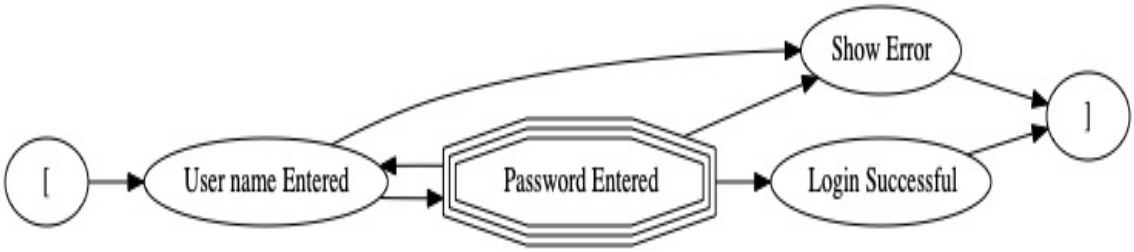

Figure 4.24. Decision Table containing vertex visualization with Graphviz

For the illustrations given in Figure 4.23 and Figure 4.24, subgraph clusters are created for the properties table and decision table itself. To implement a table structure with DOT language in Graphviz, <*table*> tag is used to create a table structure. Each row is created with a <*tr*> tag and for each column <*td*> tag is used to create a column in the row. In Figure 4.25, the table creation syntax is given. For detailed decision table implementation please see Appendix A implementation part.

```
subgraph cluster2PasswordTable {
        label = "Table Properties"
        color = "black"
        n2[shape=none label=<<table border="0" cellborder="1" cellspacing="0">
        <tr>
        <td>C0</td>
        <td>password is type of string</td>
        </tr>
        <tr>
        <td>C1</td>
        <td>password length greater than or eq. 10 AND password length less than
        100</td>
        </tr>
        <tr>
        <td>C2</td>
        <td>password should have upper case AND password should have lower
        case</td>
        </tr>
        <tr>
        <td>C3</td>
        <td>password should have special character OR password should contain
        number</td>
        </tr>
        <tr>
        <td>A0</td>
        <td>User name Entered</td>
        </tr>
        <tr>
        <td>A1</td>
        <td>Login Successful</td>
        </tr>
        <tr>
        <td>A2</td>
        <td>Show Error</td>
        </tr>
        </table>>];
}
```

Figure 4.25. DOT Language Syntax for Decision Table

CHAPTER 5

CASE STUDY

The case study examination will focus five operational flows on bank Automatic Teller Machine) ATM project. The Bank ATM is a computational device that provides digital banking operations such as deposit money, withdraw money, personal information update, money transfer, etc. The ATM devices allowing customers to perform reliable transactions and decreasing the location dependency for the bank branches.

In this thesis, we will investigate a limited portion of the bank ATM functionality. The case study will *cover login, withdraw money, deposit money, print bill*, and *logout* user scenarios. Each scenario is important to overcome some problems that will be examined in detail. The Gherkin (Gutiérrez et al., 2017) based scenarios are used to represent the flows of the ATM operations. The Gherkin based scenarios for bank ATM operations are given below.

Scenario: operation 1 – Login Successful
Given I am at #loginPage
And I entered password
And password is correct
Then Login successful and then navigate to #operationList

Scenario: operation 2 – Login Failed
Given I am at #loginPage
And I entered password
And password is wrong
And show error
And card blocked
And card retained
Then Login failed and then navigate to #loginPage

Scenario: operation 3 – Successful Withdraw Operation

Given I am at #withdrawPage

And I entered amount

And amount confirmed

And balance updated

And session refreshed

Then Withdraw successful navigate to #operationList

Scenario: operation 4 – Failed Withdraw Operation

Given I am at #withdrawPage

And I entered amount

And amount invalid

And show Error

And Enter amount again

Then Withdraw failed navigate to #operationList

Scenario: operation 5 – Successful Deposit Operation

Given I am at #depositPage

And I entered amount

And amount confirmed

And balance updated

And session refreshed

Then Deposit successful navigate to #operationList

Scenario: operation 6 – Failed Deposit Operation

Given I am at #depositPage

And I entered amount

And amount invalid

And show Error

And enter amount again

Then Withdraw failed navigate to #operationList

Scenario: operation 7 – Successful Print Bill Operation

Given I am at #resultPage

And I choose print bill

And bill printed

And SMS sent

Or e-mail sent

Then Print Bill successful navigate to #homePage

Scenario: operation 8 – Failed Print Bill Operation

Given I am at #resultPage

And I choose print bill

And show error

Then Print Bill failed navigate to #homePage

Scenario: operation 9 – Logout Operation

Given I am at #homePage and session is active

And I request to logout

And session cleared

And card ejected

And show error

Then Logout successful navigate to #welcomePage

The given Gherkin based scenarios will be implemented in both Test Suite Designer (TSD) tool and also ESG Domain Specific Language (DSL). In this case study, comparison will be given in functional suitability, usability, reliability, maintainability, productivity, compatibility, and expressiveness. Following figures illustrate the visualized ESGs in both TSD and ESG DSL. The implementation source code is included with the abstraction implementation for sub-ESGs in Appendix A. Also, the output JSON file for the visualized graph is included in Appendix A.

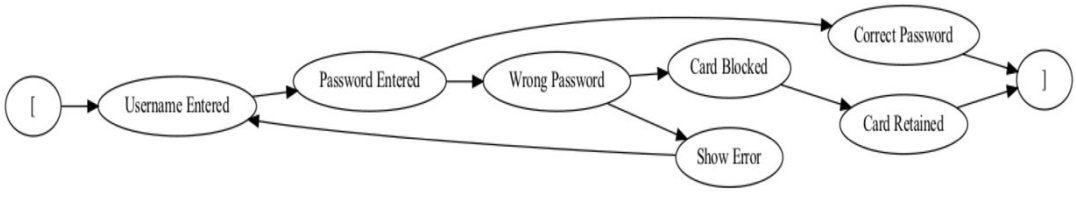

Figure 5.1. Login sub-ESG visualized by ESG DSL

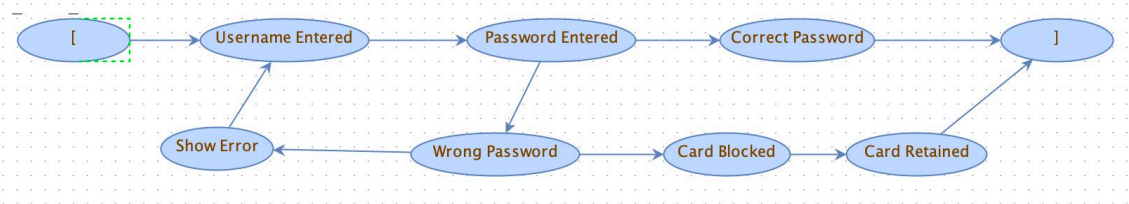

Figure 5.2. Login sub-ESG visualized by TSD

For Login sub-ESG, the vertex set C(V) = {*Username Entered, Password Entered, Correct Password, Wrong Password, Show Error, Card Blocked, Card Retained*} is visualized with both TSD and ESG DSL.

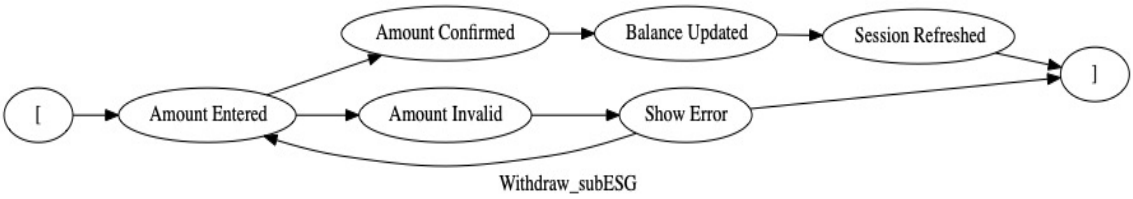

Figure 5.3. Withdraw sub-ESG visualized by ESG DSL

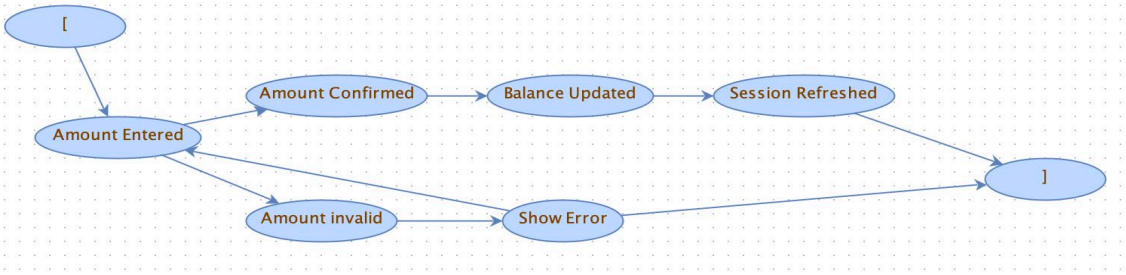

Figure 5.4. Withdraw sub-ESG visualized by TSD

For Withdraw sub-ESG, the vertex set C(V) = {*Amount Entered, Amount Confirmed, Balance Updated, Session Refreshed, Amount Invalid, Show Error*} is visualized with both TSD and ESG DSL. Withdraw and deposit operations have the same vertex set and the visualization process differs in sense of time. The reusability of the ESG DSL is high when we compare it with TSD. ESG DSL increases the reuse of

the software artifacts because there is copy and paste support on the textual models. It offers copy and paste support not only for edges and vertices, but for the entire ESG.

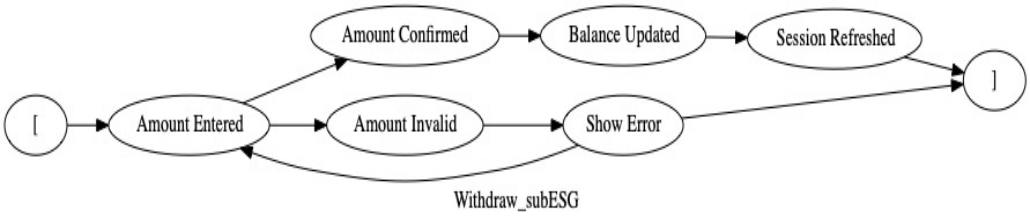

Figure 5.5. Deposit sub-ESG visualized by ESG DSL

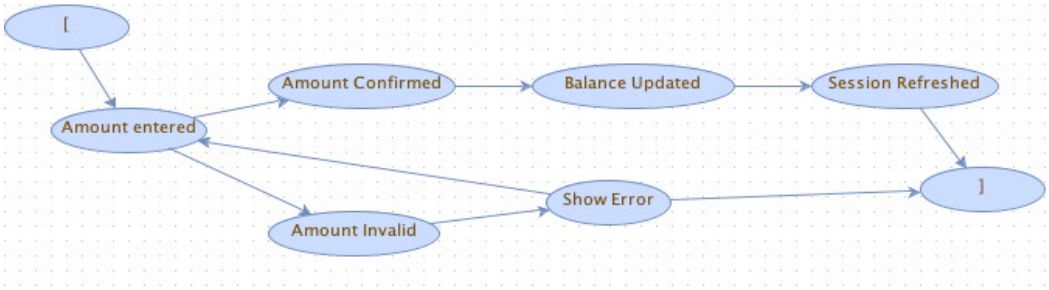

Figure 5.6. Deposit sub-ESG visualized by TSD

For the Withdraw and Deposit sub-ESGs, the vertices and the edges are completely the same. The scenarios are given on purpose and the time required to visualize similar ESGs for the ESG DSL and TSD is observed.

In Figure 5.7 and Figure 5.8 print bill sub-ESG ,the vertex set C(V) = {*Bill Requested, Bill Printed, Email sent, SMS sent, Show Error*}, is visualized with both TSD and ESG DSL.

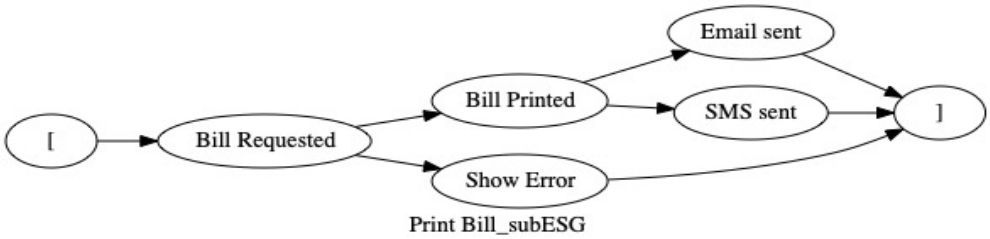

Figure 5.7. Print Bill sub-ESG visualized by ESG DSL

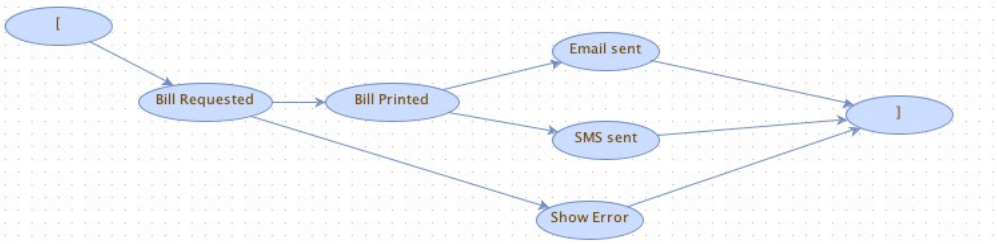

Figure 5.8. Print Bill sub-ESG visualized by TSD

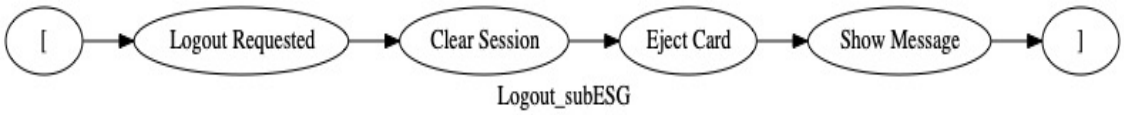

Figure 5.9. Logout sub-ESG visualized by ESG DSL

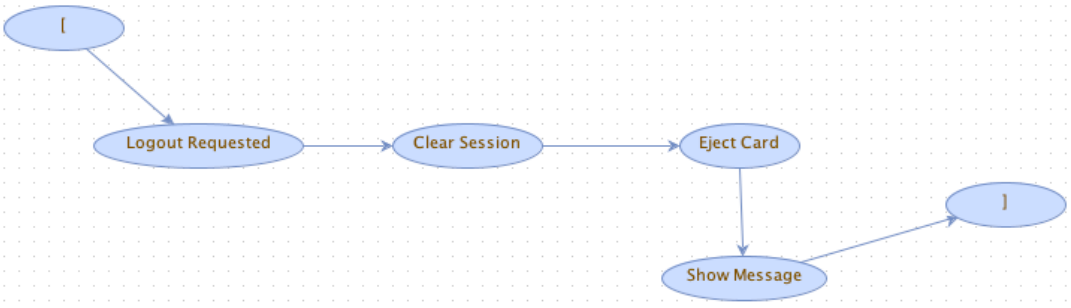

Figure 5.10. Logout sub-ESG visualized by TSD

In Figure 5.9 and Figure 5.10, logout sub-ESG, the vertex set C(V) = {Logout Requested, Clear Session, Eject Card, Show Message}, is visualized with both TSD and ESG DSL.

This case study is carried out with the participation of software developers, software testers, managers, and analysts from three different software companies, each having over 100 employees. The participants divided into two groups, these groups used TSD and ESG DSL tools respectively. The presentation, which lasted 15 minutes, is given for each group. The presentation content contains functional and structural oriented testing difference, definition for ESG, event pairs, complete event sequencies, vertex refinement. Finally, TSD tool usage is shown to the participants for the first

group and ESG DSL definition, usage, advantages and disadvantages is presented for the second group.

The evaluation questionnaire has three parts:

1) personal information gathering from participant
2) scoring ESG DSL/TSD to a set of DSL/tool characteristics
3) open-ended questions

We used open-ended questions to take feedback from the end-users about functionality of the ESG DSL and suggestions for future development of it.

The Framework for Qualitative assessment of Domain-specific Languages (FQAD) (Kahraman et al., 2015) is customized to adoption for ESG DSL. Each characteristic of the questionnaire is scored between one to five where 1 stands for "Very Bad" and 5 stands for "Very Good".

The first section of the questionnaire consists of five questions that are related to gathering information about the participant such as name, surname, graduated department, academic degree, work experience, and role in their companies. Participants distributed equally across the two platforms ESG DSL and TSD. Both of the group has the same distribution percentage on the basis of graduated departments. All the participants have Bachelor of Science academic degree.

The second section of the questionnaire consists of seven subsections such as *functional suitability, usability, reliability, maintainability, productivity, compatibility, and expressiveness*. In total, these sections consist of 18 quality characteristic questions for both the TSD tool and ESG DSL. In Figure 5.11, the average scores for each quality characteristic collected from the questionnaire evaluators are given. The illustrated graph includes both ESG DSL and TSD evaluation scores, orange color shows TSD scores, and blue shows ESG DSL scores over quality metrics. The bar graph evaluated by quality metrics between "Very Bad" and "Very Good" which is mentioned above.

In the final part of the questionnaire, the following open-ended questions were asked to the participants to get future development plan and feedbacks:

1) Does ESG DSL/TSD make graph visualization easier?
2) Do you find ESG DSL/TSD useful for graph visualization process?
3) Do you think that ESG DSL/TSD is covered the whole domain models?

4) Please write your suggestions and other comments for improving ESG DSL/TSD.

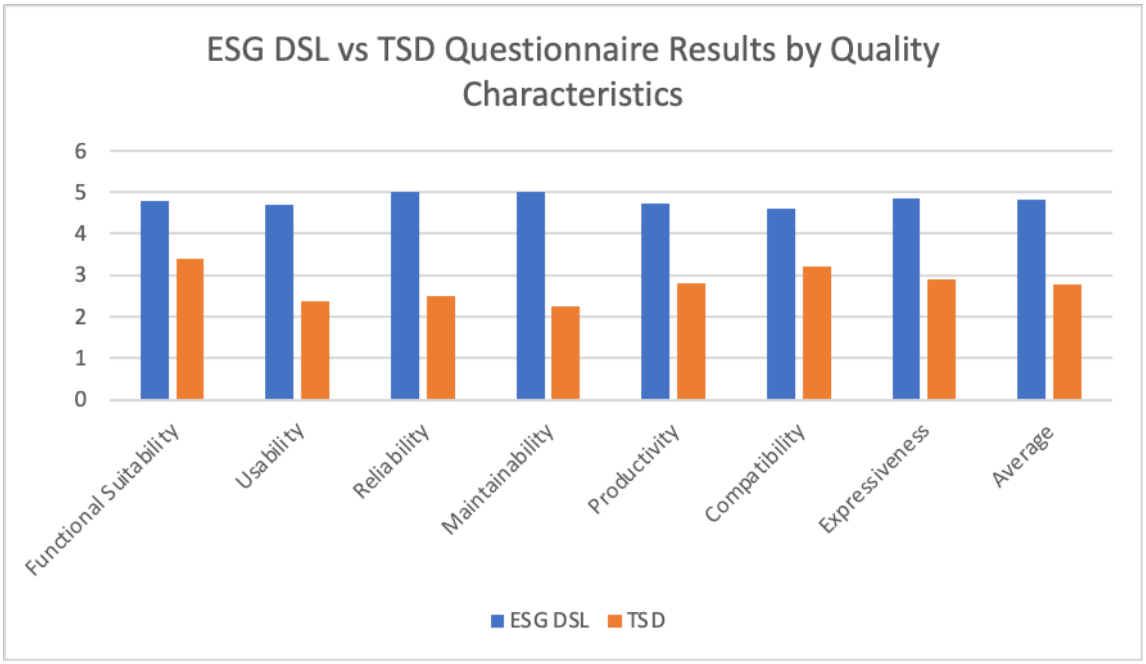

Figure 5.11. ESG DSL vs TSD Questionnaire Results

In the functional suitability aspect, ESG and sub-ESG visualization process can be done with both ESG DSL and TSD tool. On the other hand, ESG DSL provides visualization support for the decision table (DT) container vertices. Also, the DT's properties table and DT's itself visualized with ESG DSL. The DT visualization is not supported with TSD tool.

In the usability aspect, there is a gap between ESG DSL and TSD. TSD tool is hard to understand how it is working and it does not have user friendly user interface. ESG DSL provides coloring support for edges and vertices. It has user friendly user interface for the ESG DSL user. Conversely, ESG DSL provides a natural language-based syntax for the ESG DSL users. That is easier to understand and also the mapping domain entities to syntax models provides integrity for the domain.

In the reliability aspect, there is error checking mechanism, syntax highlighting and error visualization support for ESG DSL. TSD tool does not provide an error prevention mechanism for ESG visualization process.

In the maintainability aspect, ESG DSL can be combined with any platform because ESG DSL produces a JSON file output to communicate with the other platforms. It is harder to maintain TSD, further development must be implemented with JAVA general-purpose language.

In the productivity aspect, both ESG DSL and TSD tool enhance the productivity of ESG visualization when comparing with general-purpose implementation of the process visualization. In Figure 5.12 and Figure 5.13, the time, in minutes, bar graph is given to better explanation.

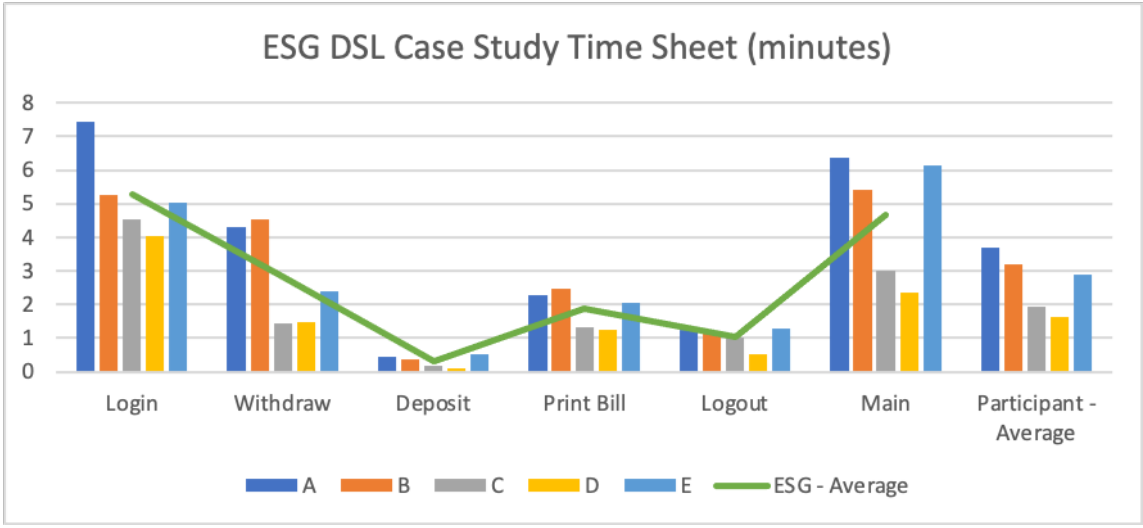

Figure 5.12. ESG DSL Case Study Time Sheet

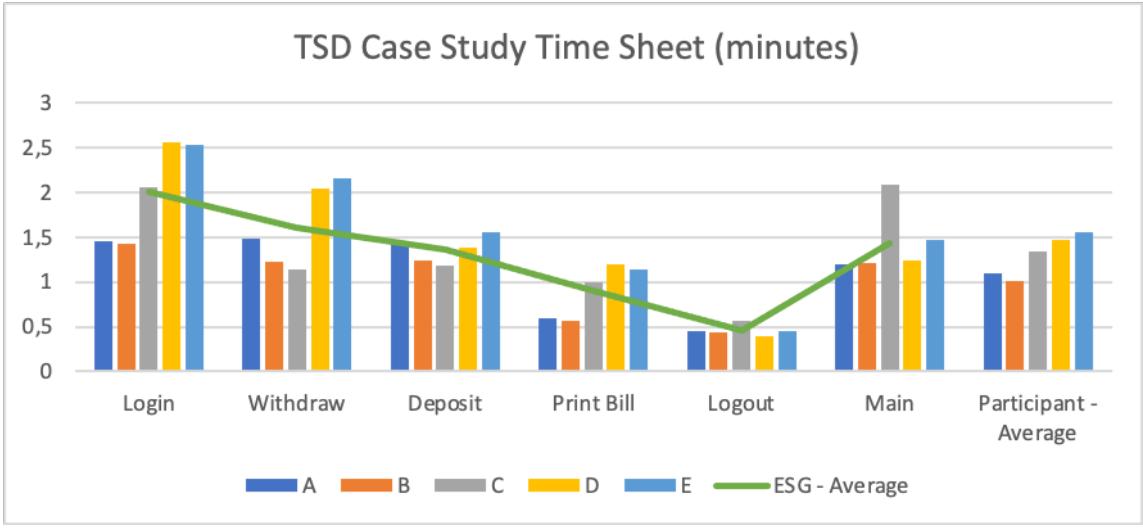

Figure 5.13. TSD Case Study Time Sheet

By having the graph visualization order of *main, login, withdraw, deposit, print bill, and logout* the Figure 5.12 and Figure 5.13 shows that TSD ESG visualization process requires less time than ESG DSL. The letters from “A” to “E” show the participants of the case study. Also, The repeating ESG visualization process, withdraw sub-ESG and deposit sub-ESG visualization has the same number of vertices and edges, is consuming much less time with ESG DSL than TSD tool. The withdraw and deposit ESGs similar to each other, the related ESGs are given in Figure 5.4 and Figure 5.5.

For the open-ended questions, we got feedback from the participants. They gave mostly “yes” responses to the question “Does ESG DSL make graph visualization easier?”. They replied the second question “Do you find ESG DSL useful for the graph visualization process?” as “yes” and said that ESG DSL makes the visualization process easier when they need to visualize a graph when compared with the existing tools. All of the participants answered yes to “Do you think that ESG DSL is covered the whole domain models?” because all the scenarios were implemented without any development on ESG DSL. The last question for the open-ended is suggestions for the future of the ESG DSL.

Finally, for the conclusion , ESG DSL provides a suitable graph visualization environment, that helps to visualize the graphs with support of error checking, syntax highlighting. ESG DSL consuming more time than TSD graph visualization process.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, a Domain Specific Language (DSL) called Event Sequence Graph (ESG) DSL has been introduced. The study gives brief explanation about ESG visualization, decision to develop a DSL, domain analysis, corner case discovery for DSL development, design, implementation, and deployment process. Moreover, this study presents a nested modularization technique for sub-ESGs (inner ESGs) and the ESGs augmented by decision tables (DTs). Each abstraction layer might have a container vertex that contains a sub-ESG or a combination of sub-ESGs along their layers. The DTs are visualized by two different tables, first table illustrates the DT itself and the second one illustrates table properties and definitions of the properties.

The study presents an editor that is closer to the natural language. Enhance the logical entity relation between domain models and DSL EMF models. The editor provides error checking mechanism, colorful syntax highlighting, and error indicator in the pane. With this development environment, ESG DSL offers an easy-to-understand editor for non-tech person such as business partners, managers, etc. In this way, it increases the number of people who can get involved the event-bases modeling without tech background.

The tool developed in this thesis provides a platform independent output file that can be read and process with any other platform. Also, the produced file crates a contract with other software languages. ESG DSL developed with JAVA general-purpose language and the output file of the program can be read by other languages such as kotlin, swift, etc.

ESG DSL increases productivity when it is focused on a restricted domain of the project. The case study shows that after the main ESG and the first sub-ESG visualization, there are serious gains in sense of time because it is an enabler of the reuse of software artifacts. Reusability enhances the outcome of the people and increases the project output indirectly. Also, the colorful syntax is easy to remember and copy and paste operations increase the productivity with a small effort.

For future work, the design and implementation of a pipeline for the communication between ESG DSL and the graph visualization project are required. The implementation can be deployed as a cloud application to increase the accessibility of the ESG DSL. Also, the decision table composition and improvements for defining an easy and useful syntax will be done. The simplification operation over the DSL models will decrease the time that is spent visualizing ESGs.

REFERENCES

Bagga, J. S., & Heinz, A. (2001). JGraph—A Java Based System for Drawing Graphs and Running Graph Algorithms. *Undefined*. https://www.semanticscholar.org/paper/JGraph-A-Java-Based-System-for-Drawing-Graphs-and-Bagga-Heinz/4df05ecf4e2b9ca94f591fc9d90b2bdb5279d1c0

Belli, F. (2001). Finite state testing and analysis of graphical user interfaces. In *Proceedings of the International Symposium on Software Reliability Engineering, ISSRE* (p. 43). https://doi.org/10.1109/ISSRE.2001.989456

Belli, F., Budnik, C., & . (2004). *Minimal Spanning Set for Coverage Testing of Interactive Systems* (Vol. 3407, p. 234). https://doi.org/10.1007/978-3-540-31862-0_17

Belli, F., Budnik, C., & . (2005). *Towards Minimization of Test Sets for Coverage Testing of Interactive Systems.* (p. 90).

Belli, F., Budnik, C. J., & . (2007). Test minimization for human-computer interaction. *Applied Intelligence*, 161–174. https://doi.org/10.1007/s10489-006-0008-0

Belli, F., Budnik, C., & White, L. (2006). Event-based modelling, analysis and testing of user interactions: Approach and case study. *Softw. Test., Verif. Reliab.*, *16*, 3–32. https://doi.org/10.1002/stvr.335

Bentley, J. (1986). Programming pearls: Little languages. *Communications of the ACM*, *29*(8), 711–721. https://doi.org/10.1145/6424.315691

Boehm, B. (2006). A view of 20th and 21st century software engineering. In *Proceedings—International Conference on Software Engineering* (Vol. 2006, p. 29). https://doi.org/10.1145/1134285.1134288

Brito, G., Valente, M. T., & . (2020). REST vs GraphQL: A controlled experiment. *Proceedings - IEEE 17th International Conference on Software Architecture, ICSA 2020*, 81–91. https://doi.org/10.1109/ICSA47634.2020.00016

Brooks, F. P. (1996). Keynote address: Language design as design. In *History of programming languages—II* (pp. 4–16). Association for Computing Machinery. https://doi.org/10.1145/234286.1057806

Chow, T. S. (1978). Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, *SE-4*(3), 178–187. https://doi.org/10.1109/TSE.1978.231496

Compatangelo, E., Meisel, H., & . (2002). *Conceptual Analysis of EER Schemas and Ontologies*. https://www.semanticscholar.org/paper/Conceptual-Analysis-of-EER-Schemas-and-Ontologies-Compatangelo-Meisel/47d57c45ab8b52643f7a250bd3db8fd3e58a25f6

Cucumber, G. (n.d.). *Gherkin Reference—Cucumber Documentation*. Retrieved June 3, 2022, from https://cucumber.io/docs/gherkin/reference/

DeMarco, T. (1982). *Controlling software projects: Management, measurement & estimation*. New York, NY : Yourdon Press. http://archive.org/details/controllingsoftw0000dema

Deursen, A., Klint, P., & Visser, J. (2000). Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, *35*, 26–36.

Diepenbeck, M., Kühne, U., Soeken, M., & Drechsler, R. (2014). Behaviour Driven Development for Tests and Verification. In M. Seidl & N. Tillmann (Eds.), *Tests and Proofs* (pp. 61–77). Springer International Publishing. https://doi.org/10.1007/978-3-319-09099-3_5

Eeles, P., Sam, H. B., Mistrík, I., Roshandel, R., & Stal, M. (2014). *Relating System Quality and Software Architecture: Foundations and Approaches*. https://doi.org/10.1016/B978-0-12-417009-4.00001-6

Efftinge, S., Völter, M., & . (2006). oAW xText: A framework for textual DSLs. *Workshop on Modeling Symposium at Eclipse Summit*, *32*.

Ellson, J., Gansner, E. R., Koutsofios, E., North, S. C., & Woodhull, G. (2003). Graphviz and dynagraph – static and dynamic graph drawing tools. *Graph Drawing Software*, 127–148.

Expression Language. (2021). https://github.com/intuit/common-xtext-expression-language/commits/develop?after=44a9ce7bc5d7a43d5252c453c622d8a3cc013420+34&branch=develop&qualified_name=refs%2Fheads%2Fdevelop

Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A., & Mehandjiev, N. (2004). Meta-Design: A manifesto for End-User Development. *Commun. ACM*, *47*, 33–37. https://doi.org/10.1145/1015864.1015884

Franca. (2018). GitHub. https://github.com/franca/franca

Gansner, E., Koutsofios, E., North, S., & Vo, K. (1993). A Technique for Drawing Directed Graphs. *Software Engineering, IEEE Transactions On*, *19*, 214–230. https://doi.org/10.1109/32.221135

Gansner, E., & North, S. (1997). An Open Graph Visualization System and Its Applications to Software Engineering. *Software - Practice and Experience - SPE*, *30*. https://doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.CO;2-N

Graphviz. (n.d.). Graphviz. Retrieved June 5, 2022, from https://graphviz.org/

Gutiérrez, J. J., Ramos, I., Mejías, M., Arévalo, C., Sánchez-Begines, J. M., & Lizcano, D. (2017). Modelling gherkin scenarios using uml. *Information Systems*

Development: Advances in Methods, Tools and Management - Proceedings of the 26th International Conference on Information Systems Development, ISD 2017, undefined-undefined. https://www.mendeley.com/catalogue/25265db9-9e9e-3e19-91a2-2b665a941dd3/?utm_source=desktop&utm_medium=1.19.8&utm_campaign=open_catalog&userDocumentId=%7Be777957f-a095-3145-be8c-f71f593f0c73%7D

Higley, K. A. (2006). Environmental consequences of the chernobyl accident and their remediation: Twenty years of experience. Report of the chernobyl forum expert group 'environment.' *Radiation Protection Dosimetry*, *121*(4), 476–477. https://doi.org/10.1093/rpd/ncl163

JUNG Framework Tech Report. (n.d.). Retrieved June 5, 2022, from http://www.datalab.uci.edu/papers/JUNG_tech_report.html#related

Kahraman, G., Bilgen, S., & .. (2015). A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, *14*(4), 1505–1526. https://doi.org/10.1007/s10270-013-0387-8

Merks, E., Paternostro, M., Budinsky, F., & Steinberg, D. (2009). *EMF: Eclipse Modeling Framework 2nd edition* (2nd edition).

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, *37*(4), 316–344. https://doi.org/10.1145/1118890.1118892

Mishra, A. (2017). *Introduction to Behavior-Driven Development* (pp. 317–327). https://doi.org/10.1007/978-1-4842-2689-6_10

Murnane, T., & Reed, K. (2001). On the effectiveness of mutation analysis as a black box testing technique. *Proceedings 2001 Australian Software Engineering Conference*, 12–20. https://doi.org/10.1109/ASWEC.2001.948492

Murnane, T., Reed, K., & .. (2001). *On the Effectiveness of Mutation Analysis as a Black Box Testing Technique.* (p. 20).

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press.

Öztürk, D. (2020). *A model-based test generation approach for agile software product lines*. https://gcris.iyte.edu.tr/handle/11147/10970

PlantUML. (2009). PlantUML.Com. https://plantuml.com/

SciSpike/yaktor. (2021). [JavaScript]. scispike. https://github.com/SciSpike/yaktor (Original work published 2016)

SciSpike/yaktor-dsl-xtext. (2017). [Java]. scispike. https://github.com/SciSpike/yaktor-dsl-xtext (Original work published 2016)

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of Systems and Software*, *56*(1), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3

Tomar, A., Vilas, D., & Thakare, V. M. (2011). A Systematic Study Of Software Quality Models. *International Journal of Software Engineering & Applications*, *2*. https://doi.org/10.5121/ijsea.2011.2406

Tuglular, T. (2018). Event Sequence Graph-Based Feature-Oriented Testing: A Preliminary Study. *Proceedings - 2018 IEEE 18th International Conference on Software Quality, Reliability, and Security Companion, QRS-C 2018*, 580–584. https://doi.org/10.1109/QRS-C.2018.00102

Tuglular, T. (2021). *On the Composability of Behavior Driven Acceptance Tests* (IARIA SOFTENG).

Tuglular, T., Belli, F., & Linschulte, M. (2016a). Input Contract Testing of Graphical User Interfaces. *International Journal of Software Engineering and Knowledge Engineering*, *26*(2), 183–215. https://doi.org/10.1142/S0218194016500091

Tuglular, T., Belli, F., & Linschulte, M. (2016b). 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

Uddin, A., Anand, A., & .. (2019). *Importance of Software Testing in the Process of Software Development*. 2321–0613.

Xtext. (2006). https://www.eclipse.org/Xtext/

APPENDIX A

ESG DSL SOFTWARE

Java SE

Java Standard Edition (Java SE) is a desktop and server computing platform that environment helps for developing and deploying portable code. Java SE defines a variety of general purpose and open-source APIs for the Java Class Library. ESG DSL development is done with JAVA general-purpose language. The project runs with the java SE 11 or newer versions.
(https://www.oracle.com/tr/java/technologies/javase/jdk11-archive-downloads.html)

Eclipse

Eclipse is a programming Integrated Development Environment (IDE). It comes with a standard workspace and a plug-in framework for configurating the environment. It is the second-most used IDE for Java development, and it was the most popular until 2016. Eclipse is developed mostly in Java and its primary use is for developing Java applications. (https://www.eclipse.org/downloads/)

XText Framework

XText is a programming language and domain-specific language development framework. With XText, you may use a robust grammar language to define your domain specific language. As a result, you receive a complete infrastructure for Eclipse, including a parser, linker, type-checker, compiler, and editing support.
(https://www.eclipse.org/Xtext/)

Graphviz

Graphviz is a graph visualization program that is free and open source. Graph visualization is a method of displaying structural data in the form of diagrams of abstract graphs and networks. Networking, bioinformatics, software engineering, web design, and visual interfaces for other technical disciplines all benefit from it. Graphviz has many useful features such as concrete diagrams, support options for colors, fonts, tabular layouts, line styles and custom shapes. (https://graphviz.org/download/)

Test Suite Designer

TSD is a scientific software tool that is non-commercial and freely available to the software analysis and testing research community. TSD relies on Event Sequence Graphs and generation of test sequences from CES and FCES.
(http://download.ivknet.de/)

Installation

ESG DSL installation instructions are given in git-hub repository. Also, the sample ESG DSL grammars, output files are included in the repository. (https://github.com/esg4aspl/esg-dsl)

ESG-Engine

ESG structure, its features and positive/negative test generation are implemented under this project. (https://github.com/ esg4aspl/esg-engine)

ESG DSL Case Study

ESG DSL “Bank ATM Project” case study grammar syntax and the visualized graph is given in the git-hub repository provided at the end of the paragraph. The DSL

syntax file named as “bank_atm_mert.mkdsl” and visualized graph is named as “bank_atm_mert.dot” (https://github.com/esg4aspl/esg-dsl).