CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

Formalization of a Use Case Model to Kripke Structure and LTL Formulas

by

Qamar uz Zaman

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Computing
Department of Computer Science

2022

Formalization of a Use Case Model to Kripke Structure and LTL Formulas

By

Qamar uz Zaman

(PC133001)

Dr. Muhammad Taimoor Khan, Senior Lecturer
University of Greenwich, London, United Kingdom
(Foreign Evaluator 1)

Dr. Wasif Afzal, Professor
Mälardalen University, Västerås, Sweden
(Foreign Evaluator 2)

Dr. Aamer Nadeem
(Thesis Supervisor)

Dr. Nayyer Masood
(Head, Department of Computer Science)

Dr. Muhammad Abdul Qadir
(Dean, Faculty of Computing)

DEPARTMENT OF COMPUTER SCIENCE
CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY
ISLAMABAD

2022

Dedication

الحمد لله

CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY ISLAMABAD

Expressway, Kahuta Road, Zone-V, Islamabad
Phone:+92-51-111-555-666 Fax: +92-51-4486705
Email: info@cust.edu.pk Website: https://www.cust.edu.pk

CERTIFICATE OF APPROVAL

This is to certify that the research work presented in the thesis, entitled "**Formalization of Use Case Model to Kripke Structure and LTL Formulas**" was conducted under the supervision of **Dr. Aamer Nadeem**. No part of this thesis has been submitted anywhere else for any other degree. This thesis is submitted to the **Department of Computer Science, Capital University of Science and Technology** in partial fulfillment of the requirements for the degree of Doctor in Philosophy in the field of **Computer Science**. The open defence of the thesis was conducted on **April 11, 2022**.

Student Name : Qamar uz Zaman (PC133001)

The Examining Committee unanimously agrees to award PhD degree in the mentioned field.

Examination Committee :

(a) External Examiner 1: Dr. Farooque Azam
Professor
CE&ME, NUST, Rawalpindi

(b) External Examiner 2: Dr. Naveed Ikram
Professor
Riphah Int. University, Islamabad

(c) Internal Examiner : Dr. Nayyer Masood
Professor
CUST, Islamabad

Supervisor Name : Dr. Aamer Nadeem
Professor
CUST, Islamabad

Name of HoD : Dr. Nayyer Masood
Professor
CUST, Islamabad

Name of Dean : Dr. Muhammad Abdul Qadir
Professor
CUST, Islamabad

AUTHOR'S DECLARATION

I, **Qamar uz Zaman (Registration No. PC-133001)**, hereby state that my PhD thesis entitled, **'Formalization of Use Case Model to Kripke Structure and LTL Formulas'** is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/ world.

At any time, if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my PhD Degree.

(Qamar uz Zaman)

Dated: April, 2022

Registration No : PC133001

PLAGIARISM UNDERTAKING

I solemnly declare that research work presented in the thesis titled "**Formalization of Use Case Model to Kripke Structure and LTL Formulas**" is solely my research work with no significant contribution from any other person. Small contribution/ help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero-tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/ cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of PhD Degree, the University reserves the right to withdraw/ revoke my PhD degree and that HEC and the University have the right to publish my name on the HEC/ University Website on which names of students are placed who submitted plagiarized thesis.

(Qamar uz Zaman)

Dated: April, 2022

Registration No : PC133001

List of Publications

It is certified that following publication(s) have been made out of the research work that has been carried out for this thesis:-

1. **Zaman, Qamar uz**, Aamer Nadeem, and Muddassar Azam Sindhu. "Formalizing the use case model: A model-based approach." Plos one Vol. 15 Issue 4 pp 1-29 (2020).

2. **Qamar uz Zaman**, Muddassar A. Sindhu, and Aamer Nadeem, "Formalizing a Use Case to Kripke Structure." *Proceedings of the IASTED International Symposium Software Engineering and Applications Volume 829 Issue 17 pp 232-239 (SEA 2015)*, 2015.

Qamar uz Zaman
(PC133001)

Acknowledgement

PhD is an academic as well as a personal activity. Without the supervision and backing, the concepts and working is never complete. In this regard, I would like to thank first and the foremost my honourable supervisor **Professor Dr. Aamer Nadeem** who made my work an easy endeavour when it seemed to me an uphill task. **Dr. Aamer Nadeem** proved to be not only an academician but also a very humble and motivating personality for me. Not only did he guide me but also introduced new discipline in my research work. His technical support for me opened a new era in the field for me. I would like to express my special gratitude to **Dr. Muddassar Azam Sindhu**. I always found him ready for discussion, guidance and review for my work. The reviewers guidance is specially acknowledged for providing their valuable suggestions. My special and humble thanks to my **Computer Science Department of Capital University of Science and Technology** which proved a catalyst for me to boost my knowledge. My best wishes in this regard are for **my family** who provided me a very conducive and disturbance free environment that helped me pursue my research.

Abstract

Software reliability can be ensured by using software verification techniques including model checking. A model checker takes a software model along with formal specifications and verifies either the provided model satisfies the input formal specifications or not. The cost and effort required to generate a software model and formal specifications may not be feasible in all software development. This opens the door for the approaches that can transform informal software requirements to formal specifications and model. There are a number of approaches to transform informal software requirements into semiformal or formal software specifications. However, the existing approaches require informal software requirements in a controlled natural language. Some approaches, require additional skills like understanding of object-oriented paradigm and domain knowledge. A number of theses approaches generate semiformal models which may not be suitable to be directly used by software verification techniques. Some of these approaches generate formal model as an output. But formal specifications, against which the model is to be verified, are still required. There is a need for an approach that generates a model along with formal specifications from the informal software requirements. The generated model, however, is a primitive model of a software, along with the generated formal specifications can be used for model checking. This facilitates the software verification process and can help to develop a reliable software.

This work proposes an approach that generates a Kripke structure and Linear Temporal Logic (LTL) formulas from a use case model. A use case model is a de facto industry standard to specify software requirements. The presented approach is a metamodel-based approach. It performs model-to-model transformation. In addition to it, a GUI tool is also developed to support this approach. The user of this approach has to specify the input use case in a proposed template and the tool automatically generates the resultant Kripke structure and LTL formulas. This approach does not require any additional software development artefacts for the transformation. Two pedagogical and two industrial case studies are used to generate the resultant formal artefacts. The proposed approach is also compared

with the existing approaches on the basis of the user efforts required to specify informal software requirements, ability to handle use case relationships and usefulness of the generated artefacts. It is found that four existing approaches meet this criteria. Only two out of these approaches are able to handle all use cases of these case studies and examples. Whereas, other two are unable to handle 41 out of 50 input use cases, because these approaches do not handle use case relationships. In addition to it, the existing approaches generate only the behavioural model of the software. None of these approaches generate formal software specifications of the software. However, the proposed approach transforms all input use cases. This approach generates a behavioural model along with the formal software specification of a software.

Contents

List of Figures

List of Tables

Abbreviations

Acronym	**W**hat (it) **S**tands **F**or
ATM	Automated Teller Machine
ConOps	Concept of Operations Document
CTL	Computation Tree Logic
DFD	Data Flow Diagram
EBNF	Extended Backus-Naur Form
HOL	Higher Order Logic
Larch	Larch: Languages and Tools for Formal Specification
LTL	Linear Temporal Logic
NuSMV	New Symbolic Model Verifier
OBJ	OBJ First-order Functional Language
PVS	Prototype Verification System
SAL	Symbolic Analysis Laboratory
SIM	Subscriber Identity Module
SPIN	Simple Promela Interpreter

Chapter 1

Introduction

The Software requirements document provides the foundation for the software development process. Documentation of software requirements makes it easy to refer and revisit the software requirements throughout the software development process by the different stakeholders of the software. A Software requirements document is required to be understandable, unambiguous, complete, consistent, verifiable, traceable and testable [1–3]. It also provides the basis for a reliable software. Software requirements can be documented as informal, semiformal or formal specifications [4, 5]. The choice to document is dependent on its user, i.e., technical or non-technical. In addition to this, the nature of the document is also dependent on the software development phase, where it is intended to use. Formal specifications are unambiguous. These can also be verified for consistency and completeness [6–10]. Formal specifications are also used to verify the design artefacts and software implementation. Formal specifications also facilitate the automated test cases generation using specification-based testing [11–13]. The use of formal specification can improve the software quality up to 92% [14].

Formal methods represent a software as mathematical entities. It is supported by a variety of tools to model, analyse and synthesize these mathematical entities. The mathematical entities are of different types including history-based specifications, transition-based specifications, state-based specifications, functional specifications and operational specifications [15, 16]. The tools allow syntactic, model and proof checking. A model checker takes a model and formal properties as input. It verifies either the input model satisfies the formal properties or not. If the input model does not satisfy the formal properties, it generates counter examples [17]. Formal specifications also facilitate the conformance testing [18, 19] and learning-based testing [20–22]. Formal methods are applicable to the different software development phases including requirements analysis, software design and implementation for the verification purposes. The later these are applied in the software development process, the larger the cost of correction is added to the software cost [23].

1.1 Software Requirements Documentation

Software requirements documented in unrestricted natural language are informal requirements [24]. It is easy to document such requirements. But the inherent features of natural language may make them inconsistent and ambiguous [25]. It is also difficult to trace and verify informal requirements. Examples of informal software requirements include user stories [26] and Software Requirements Specification (SRS) document [27]. These requirements can easily be understood by non-technical stakeholders. But ambiguity, inconsistency and incompleteness in them can cause failure of a software project [28]. Moreover, in the case of informal software requirements, it is difficult to apprehend the software verification and validation activities [29].

Semiformal software requirements are structured as diagrams or tables [30]. Examples include use case model [31], decision table [32] and flowchart [33]. The structured presentation makes them clear and understandable. It is also easy to trace and verify them in comparison to the informal software requirements [34]. Technical and non-technical users can understand informal software requirements. Semiformal software requirements are useful for software verification and validation activities. But a verification engineer requires an in-depth knowledge of the domain to develop and use semiformal software requirements as an input in the verification and validation activities. In addition to it, the informal segment of these makes it difficult to comprehend the verification and validation process. [35].

Software requirements documented using precise mathematical notations are called formal requirement. These mathematical notations include logic and algebraic notations [36]. There are well-defined syntax, semantic and manipulation rules for the formal software specifications [37]. The formal software specifications are verifiable. It is also possible to test formal software specifications for completeness and correctness [38]. Clear, consistent and complete nature of formal software requirements make them useful for verification and validation activities [39]. Examples of formal software specifications include Z-language [40], Linear Temporal Logic (LTL) [41] and Computational Tree Logic (CTL) [42]. However, formal software requirements are expensive in terms of time and cost [43–45]. This expense may not be feasible in all software development projects [46]. There are a number of approaches that formalize informal or semiformal software requirements into formal software specifications to overcome this cost.

1.2 Formalization of Informal Software Requirements

Formalization of software requirements, transforms informal or semiformal software requirements into formal software specifications [47]. This facilitates to overcome the issues of informal software requirements. This transformation makes informal software requirements to the corresponding formal software specifications [48]. Clear, consistent and complete software requirements are required for verification and validation activities [49].

The formalization can be performed either at model level or at meta model level [50]. In model-based formalization, rules are defined to transform an input model to an output model [51]. Whereas, in meta model formalization, meta models for input model and output model are defined along with the transformation rules.

The transformation rules are also defined at meta model level. This allows to perform transformation for all possible instances, definable, by the meta model [52]. The defined meta model represents the features of a model and it can be refined over the time. Additionally, definition of meta model is not dependent on a platform and implementation. This allows to implement a meta model over multiple platforms. The transformation rules are subject to a transformation process only. Moreover, a meta model approach also supports the forward and backward transformation and the transformation process can also manage single or multiple models definable by input meta model(s) to be transformed in single or multiple models of definable output meta model(s). Whereas a model-based approach has to manage both the definition and the transformation process side by side. The defined transformation rules can only transform a particular defined model to some target model. The model definition and the transformation rules are not independent of each other [53].

There are multiple approaches that formalize informal software requirements into formal software requirements. These are discussed in Chapter 3. It is important to note that the use case model is widely used in the software industry to document software requirements [3]. There are several challenges in the formalization of a use case model. These challenges include lack of a standard use case template for the specification of a use case description. Moreover, the existing use case templates do not distinctly enlist the input and output for the software functionality being specified. In addition to these, there is no standard meta model for a use case description though UML specifies a meta model for a use case diagram. The existing formalization approaches propose a use case template and/or use controlled natural language to document the input use case [54–56]. The definition of a use case template and the usage of controlled natural language smooth the transformation process. Some of these approaches produce semiformal artefacts [54, 55, 57, 58]. The generated semiformal artefacts are suitable for software design and testing. The generated formal artefacts, by some transformation approaches, are non-deterministic in nature [59–63]. It is also observed that some of these transformation approaches are domain specific [64–66]. Besides all of these, the transformation approaches require additional artefacts other than the use case model for the transformation activity [59, 60, 67]. Some of these approaches are also not supported by a tool [59, 60, 63, 67].

1.3 Motivation

Formalization of a use case model makes it possible to generate formal notations from a semiformal artefact. This formalization reduces the cost and time required to document formal software specification manually. The generation of formal notations introduces the usage of formal methods in the software development process. The use of formal methods allows to trace and track the development of the right software at this early stage of requirements engineering. The generated formal artefacts also allow to analyse and correct, if required, the software behaviour at this early stage. The cost of correction at this early stage is less than the cost to correct the errors at the later stages of the software development process. The generated formal artefacts are also usable to verify the software design and implementation artefacts generated at the later stages of the software development process. Formalization of a use case model at meta model allows to perform transformation at meta model instead of at model level. Meta model transformation is capable to transform all definable models of a meta model.

1.4 Research Methodology

The design science methodology [68] guides the development and evaluation of a research project. A customized design science methodology for this work is shown in Figure 1.1. The activities are divided into three phases. There are a number of levels and tasks associated to each phase. There are three phases, i.e., *Phase 1: What to Research?*, *Phase 2: Research Planning* and *Phase 3: Implementation*. These phases are implemented in their ascending order. Each phase along with its associated levels and tasks are discussed in the following subsections.

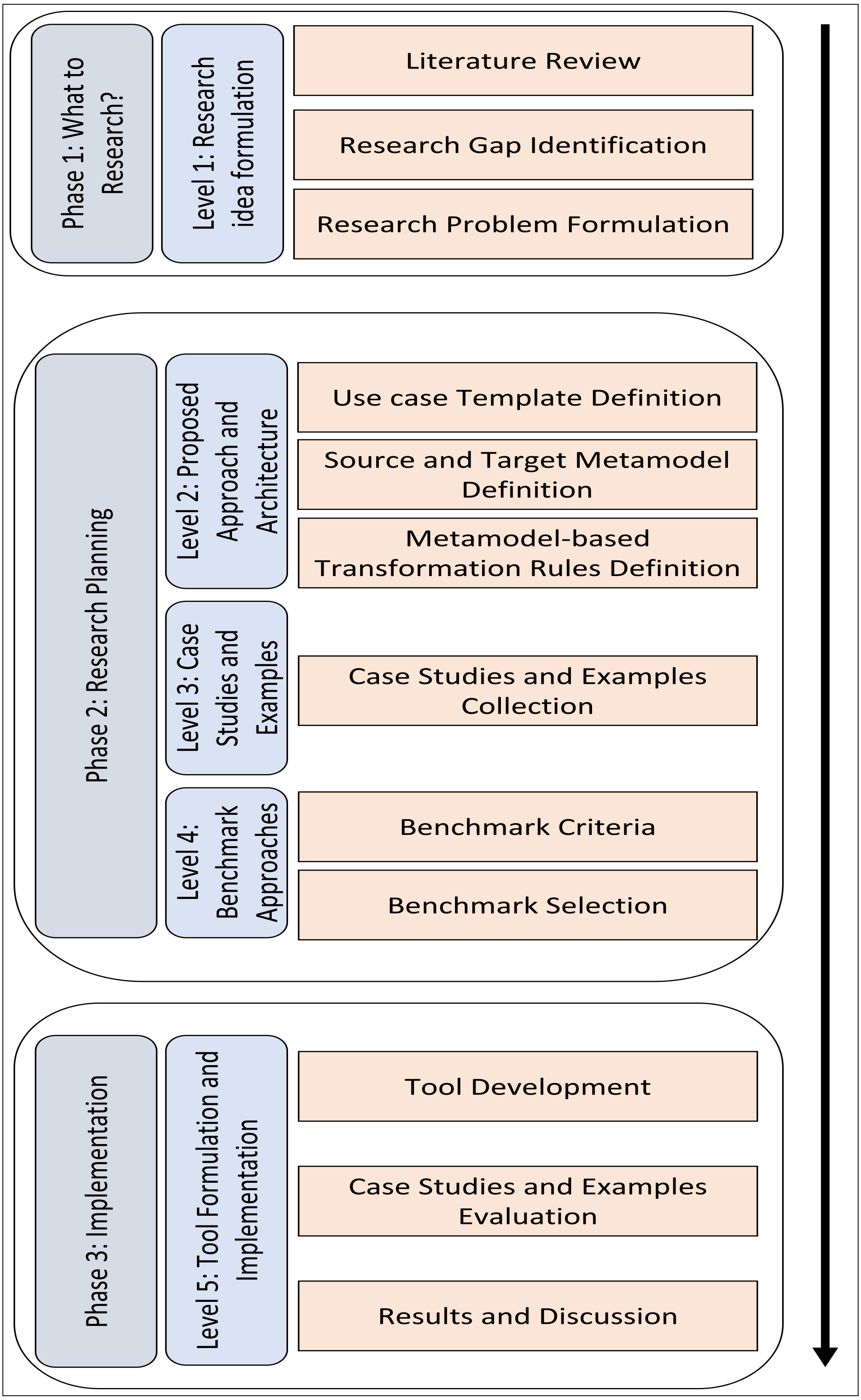


Figure 1.1: Customized Design Science Methodology.

1.4.1 Phase 1: What to Research?

This phase has one level, i.e., *Level 1: Research Idea Formulation*. This level is discussed in the following paragraph. There are three tasks has to be accomplished in this level. These include: *Literature Review*, *Research Gap Identification* and *Research Problem Formulation*.

1.4.2 Phase 2: Research Planning

This phase has 3 levels. These levels include: *Level 2: Proposed Approach and Architecture*, *Level 3: Case Studies and Examples* and *Level 4: Benchmark Approaches*. *Level 2: Proposed Approach and Architecture* has three tasks associated to it. These tasks are *Use case Template Definition*, *Source and Target Metamodel Definition* and *Metamodel-based Transformation Rules Definition*. *Level 3: Case Studies and Examples* has one associated task, i.e, *Case Studies and Examples Collection*. *Level 4: Benchmark Approaches* has two tasks and those are *Benchmark Criteria* and *Benchmark Selection*.

1.4.3 Phase 3: Implementation

This phase has only one level associated to it and that level is *Level 5: Tool Formulation and Implementation*. This level further has three tasks including *Tool Development*, *Case Studies and Examples Evaluation* and the last one is *Results and Discussion*.

1.5 Problem Statement

There are a number of approaches that transform informal or semiformal software requirements to formal software specifications. These approaches either generate behavioural model or formal software specification. Verification tools, like model checker, require both behaviour model and formal software specifications for software verification. The generated artefacts of the existing approaches can not be directly used for software verification.

1.6 Research Questions

- RQ1: What are the limitations of the existing formalization approaches?
- RQ2: How can a metamodel-based approach be proposed that generate a behaviour model as well as formal software specification?
- RQ3: How can the generated artefacts be used for software verification?

RQ1 requires the study of the existing approaches, the required input and the nature of the output along with the formalization process. *RQ2* requires the definition of an automated approach that can transform a use case model into a deterministic formal model along with the formal software specifications by handling the use case relationships. *RQ3* requires the generation of output artefacts in a format that can directly be used for software verification by using an automated tool like a model checker.

This work introduces an approach that transforms a use case model into a Kripke structure model and LTL formal specifications. The meta models for the use case model, Kripke structure model and LTL formal specification are also defined. The proposed approach is also supported by a platform independent tool. The generated artefacts can be provided as input to model checker like NuSMV, SPIN and SAL. The generated LTL formal specifications are also useful for verifications of software artefacts generated at the later stages of the software development process. The generated formal artefacts are also useful for specification-based and learning-based test case generation. Two pedagogical examples and two industrial case studies are used to demonstrate the working of the approach. It is observed that NuSMV model checker does not generate any counter example for the generated Kripke structure model and LTL formulas for the input use case model.

This validates the generated LTL formulas and Kripke structure, generated by two independent processes of the proposed approach.

1.7 Thesis Layout

This thesis is organized by providing some basic definitions in Chapter 2, the state-of-the-art approaches are discussed in Chapter 3, the proposed approach is presented in Chapter 4, the developed tool is presented in Chapter 5, examples along with case studies are presented in Chapter 6 and the output Kripke structure models as well as the generated LTL formulas for an example are provided as appendix.

Chapter 2

Background

This discussion is aimed to provide the essential concepts required for the understanding of the target problem of this work: formalization of informal software requirements to formal specifications.

Software requirements provide foundation for agreement between different stake holders of a software, basically the client and the development team. These requirements also provide foundation for different activities like software design, implementation and testing. There are multiple users of software requirements including the people with technical and non-technical knowledge. Software requirements are documented using formal or informal formats depending on the need of the intended users [69].

2.1 Informal Software Requirements

Informal software requirements are documented using unrestricted natural language. Informal software requirements can be ambiguous, incomplete and inconsistent due to the inherent features of natural language. It is difficult to verify, trace and test informal software requirements. There are multiple formats available to informally specify software requirements. These formats include user story, concept of operations document and requirements statement. These formats are discussed in the following subsections.

2.1.1 User Story

A user story [70] describes a software feature from end-user perspective. A user story is written in natural language. It describes, what feature a user requires and why the user requires this feature. A user story can be written by a user, manager or a member of development team.

2.1.2 Concept of Operations Document (ConOps)

ConOps [71] is used to document software requirements in a natural language. This document describes a software functionality from user perspective. This document is established by defence organizations for specifying their software requirements.

2.1.3 Requirements Statement

Software requirements are documented using a natural language. These requirements are numbered. These kinds of requirements are presented in software requirements specification document [72] is a detailed document. This document contains purpose, scope, plan, intended users, software requirements specification including functional, non-functional and user interface requirements.

2.2 Semiformal Software Requirements

Semiformal software requirements are structured in the form of tables and diagrams to document a software requirements. These approaches include: flowchart, data flow diagram, decision table and use case model. These approaches are discussed in the following subsections.

2.2.1 Flowchart

Software features are described in the form of a flowchart[33]. A flowchart consist of a number of graphical symbols. The semantics of these graphical symbols are defined.

2.2.2 Data Flow Diagram (DFD)

A DFD [73] describes software's functionality by describing how data flows in the proposed system by specifying input and output of a software. It also describes how data get transformed and stored in the software. This description is documented by using distinct symbols for data flow, process, data store, and external entities including data originator and data receiver.

2.2.3 Decision Tables

A decision table [32] is used to represent software requirements using logical relation between condition and actions along with their possible values. The relation among conditions and actions are represented in a tabular format consisting of conditions, condition entries, actions and action entries.

2.2.4 Use Case Model

Ivar Jacobson et al. [74] propose use case model to specify software requirements using a natural language. A use case model consists of a use case diagram and use case description(s). Unified Modelling Language (UML) [75] propose syntax of a use case diagram. A use case diagram consists of actor, use case and relationship symbols. A plain line is used to represent an actor's interaction to a use case. The relationship symbols are used to represent use case relationships including *include* and *extend* relationship. An *include* relationship is used when a use case includes the functionality of other use case. Whereas, an *extend* relationship is used to represent the extended functionality of some other use case based on some criteria. These use case relationships allow to specify a software behaviour in a more realistic way. A user software interaction and use case relationships are specified as use case description using natural language. A typical use case diagram is shown in Figure 2.1.

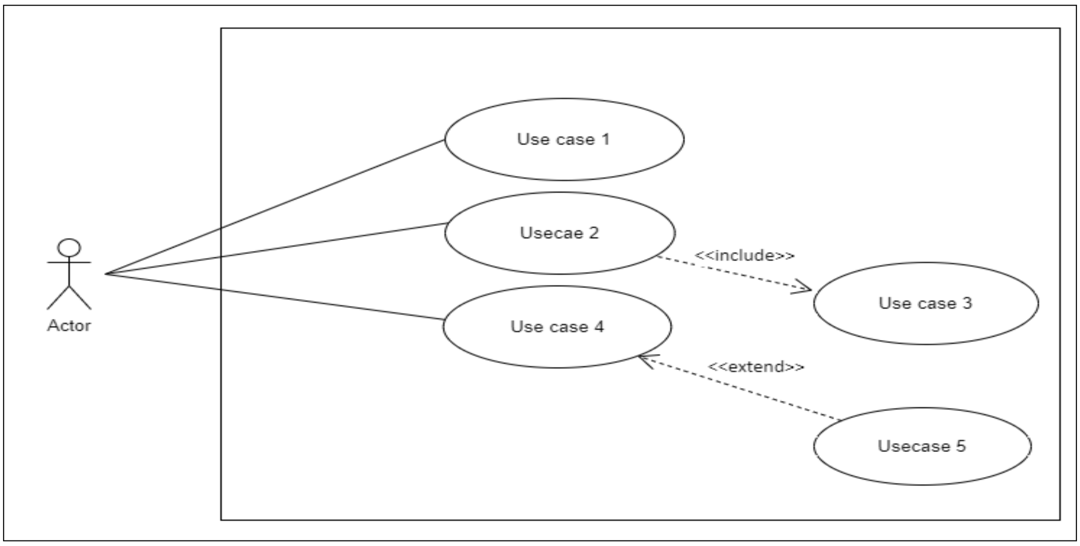


FIGURE 2.1: A Typical Use Case Diagram.

This figure shows an actor's interacting with *Use case 1*, *Use case 2* and *Use case 4*. There is an *include* relationship between *Use case 2* and *Use case 3* and an *extend* relationship between *Use case 4* and *Use case 5*.

A use case description for a distinct use case is required to be specified in a natural language. A use case description is often specified, using a template. To the best of our knowledge, UML does not specify a standard template for use case description. There are a number of use case templates being used in the software industry. These templates include: Jacobson [76], Duran et al. [77], Cockburn [78] and RUP [79]. There are some common constructs that are used in all templates. These constructs include *use case name*, *actor*, *normal scenario*, *alternate scenario* and *use case relationships*.

The distinct listing of these constructs can not overcome the problem of ambiguity in software requirements specification due to inherent issues of a natural language. Moreover, these requirements can not be verified and validated.

2.3 Formal Software Specifications

Formal software requirements are specified using set theory, propositional logic, or algebraic notations. There are defined set of rules for syntax, semantics and manipulation. Such requirements are unambiguous in nature due to defined vocabulary, syntax and semantic rules. A background knowledge is required to specify and understand requirements in this format. The additional cost and time are also required to specify requirements formally. Formal software requirements can be verified and validated. Formal software specifications can be history-based, transition-based or state-based specifications [80].

2.3.1 History-based Specification

Software behaviour is defined over time. The time can be specified as linear or branching. Examples of these are *Linear Temporal Logic (LTL)* and *Computational Tree Logic (CTL)*.

2.3.1.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [41] formulas are used to build LTL formal specifications. A LTL formula consists of atomic propositions, logical and temporal operators. An atomic proposition is a statement that can either be true or false. The logical operators include !, &, $|$, $=$, $<$, $>$, and $->$. Whereas, temporal operators include: Next operator $\bigcirc$ or **X**, Eventually operator $\Diamond$ or **F**, Global operator $\Box$ or **G** and Until operator **U**.

2.3.1.2 Computational Tree Logic

Computational Tree Logic (CTL) [42] formulas are used to specify a software behaviour. The possible functionalities in a software is considered as a tree. LTL formulas along with $\exists$ and $\forall$ operators specify software behaviour on possible paths of the tree build from the possible combinations of software behaviour. The $\exists$ operator is used to state that at least one path satisfy the formal specification. Whereas, $\forall$ operator states that all paths satisfy the formal specification.

2.3.2 Transition Based Representation

Software behaviour is represented in the form of a model consisting of state(s) and transition(s). There are multiple model formats available including *Moore model*, *Mealy model*, *Kripke structure* and *labelled transition system*. The selection of these model for software behaviour representation is dependent to the nature of software being developed.

2.3.2.1 Moore Model

A Moore model [81] is finite state machine. In Moore model, the output is determined by the current state of the model. It consists of a 6-tuple consists of $\langle Q, q_0, \Sigma, \wedge, \delta, \lambda \rangle$. Each of these tuple is described as:

- Q: set of finite states.
- $q_0 \in Q$: an initial state.
- Σ: set of input symbols.
- $\wedge$: set of output symbols.
- δ:$Q \times \Sigma \rightarrow Q$ is transition function.
- λ:$Q \rightarrow \wedge$ maps output symbols to states.

2.3.2.2 Mealy Model

A Mealy model [82] is finite state machine. In Mealy model, the output is determined by the current state and the value of input. It is a 6-tuple consists of $\langle Q, q_0, \Sigma, \wedge, \delta, \lambda \rangle$. Each of these tuple is described as:

- Q: set of finite states.
- $q_0 \in Q$: an initial state.
- Σ: set of input symbols.
- $\wedge$: set of output symbols.
- δ:$Q \times \Sigma \rightarrow Q$ is transition function.
- λ:$Q \times \Sigma \rightarrow \wedge$ maps input and output symbols to states.

2.3.2.3 Kripke Structure

A Kripke structure [83] models a software behaviour. It is used in model checking. Its nodes represent the reachable states of a software and its edges represent state transition. It is a 5-tuple consists of $\langle\ Q,\ \Sigma,\ \delta,\ q_0,\ \lambda\ \rangle$ where

- Q: set of states.
- Σ: set of input symbols.
- δ: $Q \times \Sigma \rightarrow Q$, a transition function.
- $q_0 \in Q$: an initial state.
- λ: $Q \rightarrow 2^{AP}$, a labelling function.

The atomic propositions denoted as AP are used to represent the software properties on a state.

2.3.2.4 Labelled Transition System

A labelled transition system [84] is a 6-tuple consisting of $\langle\ S,\ Act,\ \rightarrow,\ I,\ AP,\ L\ \rangle$. Each of these are described as:

- S: set of states.

- *Act*: set of actions.
- $\rightarrow \subseteq S \times AP \times S$, a transition relation.
- $I \subseteq S$: set of initial states
- AP: set of atomic propositions.
- L: $S \rightarrow 2^{AP}$, a labelling function.

2.3.3 State based Specifications

Software behaviour is represented in the form of mathematical notation using defined set of symbols. There are multiple specification formats including *Z- language* and *X-machine*. State of the system is expressed as schema or abstract data X. A system state has pre- and post-assertions specified as pre- and post-conditions.

2.3.3.1 Z Specification Language

Z specification language [40] is used to specify a software. It is based on the notations and operators of set theory and first order predicate logic. The software is specified in the form of multiple schemas. A software can have only one *state schema* and an *initial schema*. There could be multiple operational schema. Each operational schema is responsible to update the state of the software. However, there could be such schemas that do not change the sate of the software. An extension of Z for object oriented paradigm is also proposed and is known as Obj-Z.

2.3.3.2 X-machine

X-machine [85] is used to specify a software. A software is specified in the form of fundamental data type or abstract data type. This is called **X** data type. A set of operations on this data can be defined. These operations are called relations. The values of this abstract data type is manipulated as a result of execution of these defined relations. The union of these relations defines the behaviour of a software. An X-machine is capable to define finite computation and encoding functions.

2.3.4 Functional Specification

A software is specified as a collection of mathematical functions. Mathematical representation allows to verify the specified requirements for consistency and completeness. There are two approaches including *algebraic specification* and *higher-order functions*. Languages to specify algebraic functional specification include OBJ [86] and Larch [87] languages. Whereas HOL [88] and PVS [89] languages are used for higher order functions.

2.3.4.1 OBJ

OBJ [86] is a first-order functional language. It supports a declarative style to represent software requirements using equational logic and parametrized programming. A software's requirements written in OBJ are verified using theorem provers.

2.3.4.2 Larch

Larch [87] allows to document software requirements formally using abstract data types and associated operations. The software requirements represented in Larch are verified using theorem provers.

2.3.4.3 HOL

HOL [88] offers an interactive environment for documentation and verification of software requirements. Specified requirements are proved with the help automated theorem prover. Software requirements are written as meta concepts and operations on these meta concepts are also defined.

2.3.4.4 PVS

PVS [89] allows to write software requirements using higher order logic. It allows a user to write software requirements using built in data types, user defined data types and operations in the form of parametrized theories.

2.3.5 Operational Specification

In operational specification, a software is represented as a collection of processes. Languages including Gist [90] and Petri nets [91] are used for this form of specification.

2.3.5.1 Gist

Gist [90] allows to build a software prototype. A user writes a software requirements and functionalities as a collection of processes. The written processes are, then, transformed into a prototype.

2.3.5.2 Petri Nets

A Petri net [91] is a place/transition model, used to represent a distributed system. A Petri net is modelled by a defined set of notations for places, arcs and transitions.

2.4 Informal to Formal Specification Transformation

There are number of approaches that transform informal software specification to formal and semiformal software notations. A number of these approaches are listed in Chapter 3. Some of these approaches are manual and some are semi automated.

The transformation can be performed from an instance of a model to other instance or it can be performed at model level. The instance transformation can transform a particular instance only. Whereas, mode level transformation allows the transformation of all definable models of a metamodel to corresponding model of the target metamodel.It is also known as metamodel based transformation.

2.4.1 Metamodel-based Transformation

Informal software requirements can be transformed into corresponding formal software notations. Model Driven Development (MDD) [92] allows to design models for different software artefacts. This allows to build complex and big software applications by integrating different models. A model must conform to its meta model. A meta model defines the rules that a model of it must comply with.

MDD also supports model transformation of one or more models into some output model(s). A simple transformation process is capable to transform only a single instance into corresponding resultant instance. Whereas, a meta model transformation is capable to define transformation process that is able to transform all possible definable instances of a meta model to the corresponding possible instances of resultant meta model.

There are multiple languages and technologies that allow to define meta model, model and transformation rules. The examples of meta model transformation languages include ATLAS Transformation Language (ATL) [93], and Epsilon Transformation Language (ETL) [94]. ATL supports both imperative and declarative rules for the transformation. These rules are executed by ATL transformation engine. QVT consists of QVT- relational, operative and core. QVT- relational and operative are declarative in nature. Moreover, QVT-core is declarative and imperative in nature. Whereas, ETL is a part of Epsilon model management framework. This framework consists of Epsilon Object Language (EOL) [95], Epsilon Validation Language (EVL) [96], Epsilon Transformation Language (ETL) [94], Epsilon

Wizard Language (EWL), Epsilon Generation Language (EGL) [97], Epsilon Comparison Language (ECL) and Epsilon Merging Language (EMG). EOL provides model management activities. EVL allows model validation activities. ETL offers model transformation activities. EWL allows construction and refactoring of models. EGL supports model to text transformation. ECL allows comparison of different models. EMG offers model merging activities.

2.5 Summary

This chapter lists the concepts of informal, formal and semiformal software specification and their different types. Concept of meta model transformation is also discussed. These concepts are useful to understand the problem focused in this work.

Chapter 3

Literature Review

There exist a number of approaches that transform informal software requirements to corresponding formal or semiformal notations. The informal software requirements can be documented in the form of a paragraph, a user story or as a use case description using a natural language.

A natural language lacks sound semantics. This lack of sound semantics adds complexity in the formalization of a use case [61]. The formalization approaches handle this complexity in a number of ways. Some of these approaches use Object Constraint Language (OCL) to incorporate semantics in UML artefacts [98]. Some other approaches use formal language like B for the formalization [99]. The approaches enlisted in this chapter take informal software requirements as an input and generate formal or semiformal software notations. The informal software requirements can be documented by using a plain natural language or controlled natural language.

A controlled natural language constraints its use with a set of keywords and restriction rules. In addition to these, temporal annotation tags or boilerplate tuples are also used to write informal software requirements. Some of these approaches do not handle use case relationships. A number of listed approaches are domain specific and perform transformation at model level. Moreover, a number of these approaches require additional artefacts along with the informal software requirements. These approaches are summarized in the following section.

3.1 State of the Art Approaches

There are a number of approaches that take informal software requirements and generate semiformal and formal artefacts. These approaches are classified on the basis of the nature of the generated artefacts into the approaches that generate semiformal artefacts and the approaches that generate formal artefacts. These approaches on the basis of their classification are discussed in the following subsections.

3.1.1 Approaches Generating Semiformal Artefacts

There are a number of approaches that take informal software requirements as input and generate corresponding semiformal artefacts.

Harel et al., 2003

Harel et al. [58] transform informal software requirements into the corresponding Live Sequence Charts (LSC). A LSC describes mandatory, forbidden and optional scenarios among the software's modules. They developed a tool, named Play-Go, for the user to practice this approach. This tool takes informal software requirements from a user and generates corresponding LSC(s). While writing the informal software requirements, the user has to label the written nouns and verbs as data members, function members or as class instances.

This approach requires in-depth knowledge of software's domain and object oriented paradigm for the annotation of the written nouns and verbs in informal software requirements. The generated LSC(s) is a semiformal artefact. To the best of our knowledge, there are no such tools that can process LSC(s) directly for the software verification and validation activities.

Kaindl et al., 2009

Kaindl et al. [55] propose to transform a use case description to primitive sequence diagram of a software. This approach requires to specify a use case description in Requirement Specification Language (RSL). RSL consists of defined rules to specify literals and expression operators along with a defined set of keywords. The domain concepts definition using RSL requires the enlisting of distinct components, operations and equations. The specification of operations requires the distinct definition of input and output. In addition to it pre- and post-conditions for the operations are required to be specified using predicate logic.

The user of this approach requires the knowledge and skill to use predicate logic, expression operators and keywords of RSL. In addition to it, the generated sequence diagram is a semiformal notation. To the best of our knowledge, there does not exist a tool that can use a sequence diagram for automated verification and validation procedure.

Yue et al., 2010

Yue et al. [54] propose to specify a use case description, according to the template and rules of Restricted Use Case Modelling Methodology (RUCM). This specified use case is then transformed into an activity diagram of the software. RUCM requires the use of a number of keywords including *Name*, *Brief description*, *Pre-condition*, *Primary actor*, *Secondary actors*, *Dependency*, *Generalization*, *Basic flow*, *Post condition*, *Specific alternate flow*, *Global alternate* and *Bounded alternate flow*. The additional keywords to specify actions in a use case description include *IF-THEN-ELSE-ELSEIF-ENDIF*, *MEANWHILE*, *VALIDATE THAT*, *DO-UNTIL*, *ABORT* and *RESUME STEP*. In addition to these keywords, the user is constrained with a set of rules and the use of use case template to specify a use case using RUCM. They also develop a framework named aToucan [100] to practice this approach. The user of this approach has to specify a use case description with the defined constraints of RUCM. Some of the defined keywords are not in common practice. Moreover, the generated activity diagram is a semiformal notation and it can not be used for verification and validation activities.

Smialek et al., 2012

Smialek et al. [62] propose an approach that takes a use case description as an input and generates a corresponding sequence diagram of a software. This approach also requires to write the input use case description using RUCM. The features of RUCM are already discussed in the previous subsection. The required skills to specify a use case description using RUCM require the understanding of keywords and rules of RUCM. In addition to it, the generated sequence diagram is a semiformal diagram. This generated sequence diagram can not be used for verification and validation activities like model checking.

Arora et al., 2019

Arora et al. [57] propose an approach that takes a use case description, specified in a natural language, as an input and generates a corresponding domain model of a software. They define a set of rules based on natural language processing techniques to identify software's objects and relations among the software objects from the input use case description.

The generated domain model is a semiformal artefact. The user of this approach is constrained to use domain concepts while specifying the use case specifications. These approaches are evaluated against the required input, ability to handle use case relationships, usage of controlled input language, domain specific, manual efforts required, generated artefact, nature of the generated artefact, tool support and ability to perform metamodel- based transformation. The evaluation criteria and the approach features against the criteria are listed in Table 3.1.

Table 3.1: **Approaches Generating Semiformal Artefacts**

Year	**Author**	**Input Artefacts**	**Use case Relationships Handled**	**Usage of Controlled Input Language**	**Domain Specific**	**Manual Required Effort**	**Generated Artefacts**	**Nature of Generated Artefact**	**Tool Support**	**Metamodel-based Transformation**
2003	Harel et al.	Informal software requirements	No	No	No	A user of this approach has to manually identify the specified nouns and verbs in the requirements. This identification facilitate the transformation process.	Live sequence chart	Semiformal notation	Yes	No
2009	Kaindl et al.	Use case Description	No	Requirements Specification Language(RSL)	No	Distinct enlisting of input, output and operations as use case statements. Definition of pre- and post-conditions for each operations as predicates.	Sequence Diagram	Semiformal notation	Yes	No

continued ...

. . . continued

Year	Author	Input Artefacts	Use case Relationships Handled	Usage of Controlled Input Language	Domain Specific	Manual Required Effort	Generated Artefacts	Nature of Generated Artefact	Tool Support	Metamodel-based Transformation
2010	Yue et al.	Use case Description	No	RUCM	No	A user has to define precondition and postconditions of a use case while specifying a use case. The user of this approach also has to enlist use case name, actor and use case scenarios distinctly. The user of this approach is also required to specify scenario statements of use case by following the defined rules and constraints of RUCM language.	Activity Diagram	Semiformal notation	Yes	No

continued . . .

. . . continued

Year	Author	Input Artefacts	Use case Relationships Handled	Usage of Controlled Input Language	Domain Specific	Manual Required Effort	Generated Artefacts	Nature of Generated Artefact	Tool Support	Metamodel-based Transformation
2012	Smialek et al.	Use case Description	No	RUCM	No	A user of this approach has to define precondition and postconditions of a use case while specifying a use case. Distinct listing of use case name, actor, normal scenario and alternate scenario is also required. Specification of use case statements as per the defined constraint and rules in RUCM.	Sequence Diagram	Semiformal notation	Yes	No
2019	Arora et al.	Use case description	No	Natural Language	No	No	Domain Model	Semiformal notation	Yes	No

3.1.2 Approaches Generating Formal Artefacts

Informal software requirement can also be transformed directly to formal software specifications. There exist a number of approaches that take informal software requirements input and generate formal notations as an output. In this subsection, such approaches are discussed with their strengths and weaknesses.

Dranidis et al., 2003

Dranidis et al. [59] transform a use case description into a corresponding X-machine with the help of System Sequence Diagrams (SSD) and the domain model of a software. The user of this approach has to map the constructs of input artefacts to the constructs of an X-machine manually. The states and transitions of the generated X-machine are defined from the success and alternate scenario(s) of the provided use case description. Transitions are labelled with the functions of the provided SSD. Whereas, the memories of the generated X-machine are defined from the provided domain model. System level test case are defined from the generated X-machine by using W-method [101] for the occurrence of possible faults in the software.

A X-machine is an abstract data type and represents the state of a software being specified. The possible operations to manipulate this state are also required to be defined. The user of this approach has to manually map input artefacts to the generated X-machined. The generated X-machine is a nondeterministic formal notation. A domain model of a software gets evolved with the maturity and evolution of the software development process. Eleftherakis et al. [102] transform a X-machine into corresponding Computational Tree Logic (CTL) formulas. The generated CTL formulas can not be directly usable for system verification. A model checker requires CTL formulas along with a behavioural model for software verification. Moreover, the required SSD diagram(s) are produced at later stage of the requirements specification stage. The user of this approach has to wait till the development of a domain model and a SSD before the usage of this approach.

Somé, 2003

Somé [60] transforms a use case description into a corresponding state transition graph. This approach requires a domain model of the software for this transformation. The user of this approach requires to specify the input use case description by using Definite Clause Grammar (DCG). DCG requires to specify the operations in the form of active sentences. An action must be specified as a verb along with the module effected by the action. In addition to these, this grammar also requires to specify condition in the form of predicates.

The constraints of DCG requires the understanding and usage skill to write conditions as predicates. In addition to it, the required domain model is also get evolved with the progress of the software development process. The user of this approach has to wait till the maturity of the domain model before practising this approach. Moreover, the generated state transition graph is a nondeterministic notation. A nondeterministic notation specifis the abstract behaviour of a software with incomplete requirements specification. This abstract behaviour is not suitable for verification and validation activities.

Somé, 2010

Somé [61] proposes an approach to transform a use case into a corresponding Petri net [91]. This approach require to specify a use case description by using a controlled natural language. The controled natural language requires to write a use case pre- and post-conditions in the form of predicates. The operations and conditions of these operations are defined by using IF-then keyword along with a predicate. A use case follows list distinctly require the use of sync and unsync keywords. This approach also consider use case relationships during the transformation process.

The user of this approach requires the additional skill to apply predicate logic and constructs of controlled natural language. These constructs are not in common use. The generated Petri net is a nondeterministic formal notation. Model checkers like

ROMEO [103] and TAPAAL [104] take a Petri net as input for model checking. But, for model checking a model checker also requires formal specifications of a software. The user of this approach still requires to specify formal specification before the model checking.

Simko et al., 2012

Simko et al. [63] propose a Formal Verification of Annotated use case Models (FOAM) framework. This framework takes a use case description as an input and generates a corresponding Labelled Transition System (LTS). This framework uses a use case template proposed by Cockburn [78]. The user of this framework is required to manually annotate the specified use case by using annotations tags. The annotation tags can be defined by using the keywords create, use, goto, include, abort, mark and guard. In addition to these the services of an expert in the field of temporal logic and an domain engineer is required to defined temporal annotation tags. These domain experts specify formal notations of the software in the form of CTL formulas.

This approach requires the services of the domain experts in the field of temporal logic and in the domain of the software being developed. These experts define the annotation tags and formal specification of the software. In addition to these, the user of this approach also requires the skill to use the defined annotation tags in a use case description prior to the transformation process. The annotation tags evolve with the evolution of the domain. Moreover, the generated LTS is a nondeterministic formal notation. Model checker like NuSMV can use a LTS as an input model. But the formal specifications of the software is also required for model checking.

Couto et al., 2014

Couto et al. [64] transform a use case description into an ontology instance. They have also proposed a Restricted Use case Statement (RUS) language to specify a use case description in their proposed template. This language allows to define and label individuals, type, relationship and data properties specified in a use case using a set of defined markup tags. These markup tags include <I> to specify an individual, <R> to specify a relationship between two individuals, <D> for specifying data properties and <T> to define a class instance. The labelled use case description is then transformed into an ontology instance. This generated ontology instance is used to check the correctness of instances, relations and properties specified in the input use case description using the SPARQL Protocol and RDF Query Language (SPARQL).

This approach requires the services of a domain expert to describe the domain concepts and relations among these instances in the required format. The services of a domain expert is also required to write the SPARQL queries to be used for the completeness and correctness. The user of this approach requires the additional skill to annotate the input use case with the defined labels. The generated ontology instance is required to be transformed into a Promela model and then can be provided as a model input to SPIN model checker. Along with the transformation, the model checking still requires the formal specification in the form of LTL or CTL formulas.

Selway et al., 2015

Selway et al. [65] transform informal software requirements, written in a controlled natural language, into Semantic of Business Vocabulary and Business Rules (SBVR) model. SBVR model is proposed by Object Management Group (OMG) for the specification of requirements of a business software. The controlled natural language requires to adhere the defined fonts for the specification of term, name, verb and keywords. The set of keywords includes each, at least one, at least n, at most one, at most n and at least n and at most m. The allowable logical operators include and, or, if then, if and only if.

This approach requires the services of a domain expert to define business concepts and their relations. In addition to it, the user of this approach also requires the additional skill to label and specify the informal requirements using the constructs of the controlled natural language. Moreover, the generated SBVR model is required to be transformed into Business Process Modelling Notation (BPMN) model. This generated BPMN model is then represented as a Promela model. A Promela model can be used as an input to SPIN model checker. Moreover, the model checking activity still requires LTL or CTL formal specifications.

Singh et al., 2016

Singh et al. [67] propose an approach to produce Z specifications from the UML diagrams including use case, class and sequence diagrams of a software. The generated Z specification include static and dynamic views of a software. A software's static view is defined by parsing the information presented in the use case and the class diagrams provided as input. The sequence diagram of a software is used to generate the dynamic view of a software. The authors of this approach uses the Z/Eves tool [105] for the analysis of the generated Z specifications.

This approach does not consider an actor's role while generating the static and dynamic views. The generated Z specifications can be used as input to Z2SAL [106] tool. Z2SAL tool transforms Z soecifications into a model usuable for the SAL model checker. In addition to it, SAL model checker still requires LTL formal specifications to perform model checking.

Chu et al., 2017

Chu et al. [107] propose an approach that takes a use case description and a class diagram as input. This approach generates a Labelled Transition System (LTS). This approach also requires to specify the input use case description using a Use case Specification Language (USL). The USL constructs include actor-input, actor-request, system-display, system-input, system-state, system-output, system-request, system-include and system-extend. The use case specifications are specified as pre- and post- conditions along with the operation statements. The use case description is used to define operations on the objects of the class diagram.

The user of this approach requires additional skill to specify a use case description using USL allowable constructs. The generated LTS is a nondeterministic formal notation. It can be used as an input model to a model checker like NuSMV. But for model verification, a user still requires the software formal specification as LTL formulas.

Yang et al., 2019

Yang et al. [66] propose an approach to transform a use case specification into a corresponding ontology instance. The input use case is required to be specified by using a set of defined boilerplate. A boilerplate is expressed as a tuple consisting of an object and its attributes. The allowable constructs to be used in the boilerplates include system, function, precondition, action, quantity and state.

The user of this approach requires additional skill to describe and use boilerplate constructs along with the values to specify a use case description. Model checker like BLAST [108] uses an ontology instance as an input model. However, for model checking procedure, a user is still required to provide formal specification in the form of LTL formulas.

The approaches discussed in this subsection are also evaluated on the same criteria listed in Subsection 3.1.1, i.e., the required input artefacts, ability to handle use case relationships, usage of controlled input language, domain specific, manual efforts required, generated artefact, nature of the generated artefact, tool support and ability to perform metamodel based transformation. The evaluation criteria and the approach features against the criteria is listed in Table 3.2.

TABLE 3.2: **Approaches Generating Formal Artefacts**

Year	Author	Input Artefacts	Use case Relationships Handled	Usage of Controlled Input Language	Domain Specific	Manual Required Effort	Generated Artefacts	Nature of Generated Artefact	Tool Support	Metamodel-based Transformation
2003	Dranidis et al.	Use case Description, System Sequence Diagram and Domain Model	No	No	No	Understanding of X-machine memory and operations and ability to manual define these in the generated X-machine	X-machine	Non-deterministic formal notation	No	No
2003	Somé	Use case Description and Domain Model	No	Definite Clause Grammar (DCG)	No	define pre- and post conditions as predicates and specify a use case description using constructs of DCG	State Transition Graph	Non-deterministic formal notation	No	No
2010	Somé	Use case Description	Yes	Controlled Natural Language (CNL)	No	Describe pre- and post- conditions of a use case as predicate and specify a use case description using constructs of CNL	Petri nets	Non-deterministic formal notation	Yes	No
2012	Simko et al.	Use case description	Yes	Annotation tags	No	Definition and usage of the annotation tags for the specification of the input use case description	Labelled Transition System	Non-deterministic formal notation	No	No
2014	Cuoto et al.	Use case description	No	Restricted Use case Statement (RUS)	Yes	Define individuals, relations, terms and data properties tags of the domain and also specify the input use case description using RUS	Ontology instance	Formal notation	Yes	No

continued ...

. . . continued

Year	Author	Input Artefacts	Use case Relationships Handled	Usage of Controlled Input Language	Domain Specific	Manual Required Effort	Generated Artefacts	Nature of Generated Artefact	Tool Support	Metamodel-based Transformation
2015	Selway et al.	Informal specification	No	Controlled Natural Language (CNL)	Yes	Specify a use case description using CNL	SBVR model	Formal notation	Yes	No
2016	Singh et al.	Use case, Class and Sequence Diagram	No	No	No	Understanding of UML diagrams	Z specifications	Formal notation	No	No
2017	Chu et al.	Use case description and Class Diagram	Yes	Use case Specification Language (USL)	No	Describe pre- and post- conditions and action statements for the specification of a use case description using USL	Labeled Transition System	Non-deterministic formal notation	Yes	No
2019	Yang et al.	Use case Description	No	Boilerplates	Yes	Definition of Boilerplates and specification of a use case description using boilerplates	Ontology instance	Formal notation	Yes	No

3.2 Analysis

The above discussed approaches are analysed on the basis of required input, usage of controlled input language, domain specific, ability to handle use case relationships, manual efforts required, generated artefact, nature of the generated artefact, tool support and ability to perform metamodel-based transformation. Some of these approaches require a use case model, whereas the others require a use case model along with some other artefact(s). This analysis in listed in Table 3.1 and Table 3.2.

It is observed that only approach proposed by Simko et al. [63] uses a use case description template proposed by Cockburn and all of the remaining approaches define a template of their own to specify a use case. The approach of Singh et al. [67] also uses a use case diagram along with the input use case descriptions for the transformation process. It is also observed that the keywords used to specify a use case descriptions in the approaches of Simko et al. [63] and Singh et al. [67] are not common to non-technical stakeholders.

A number of the listed approaches use a variant of controlled natural language to specify a use case description. These approaches include Somé [63], Kaindl et al. [55], Yue et al. [54], Smialek et al. [62], Cuoto et al. [64], Selway et al. [65] and Chu et al. [107]. A controlled natural language constrains the usage of defined keywords and rules to structure the use case description sentences. This makes it difficult to structure the sentences in comparison to structure the sentences in a natural language. Moreover, the work of Simko et al. [63] and Yang et al. [66] also require the services of a domain expert to define the annotation tags and boilerplates prior to the specification of a use case description.

It is also observed that a number of the existing approaches require additional artefacts along with a use case description for the transformation process. The required additional artefacts include system sequence diagram, domain model, class diagram and sequence diagram. The system sequence diagram and domain model of a software is evolved with the evolution of software development process. The user of such approaches has to wait for comprehensive version of these artefacts for the transformation process. In addition to it, the class and sequence diagrams are developed at the design stage of the development process. The users of such approaches can perform transformation after the design phase of software development process.

In addition to the above, it is also observed that most of these approaches require the user efforts to identify noun as an object or a data member and verb as a function member. The users of some approaches are required to annotate a use case description using annotation tags or with boiler plates.

It is also found that some of these approaches are domain specific and can not be generalized for software development. These approaches also require the services of a domain expert to define domain concepts that are used to specify the input use case description.

On the other hand, the generated artefacts are semiformal or formal in nature. The generated semiformal artefacts include LSC, sequence diagram, activity diagram and domain model. To the best of our knowledge, these artefacts can not be used for model checking. The generated formal artefacts including X-machine, state transition graph, Petri nets, and labelled transition system are nondeterministic in nature. A nondeterministic formal notation is used to specify the incomplete behaviour of a software and it can also generate the unreachable states in a software behaviour. A generated Petri net can be used as an input model to the ROMEO and TAPAAL model checkers. Whereas, the generated labelled transition system can be used as an input model to the NuSMV model checker. BLAST model checker can take an ontology instance as an input model. The generated SBVR model is required to be transformed into BPMN model prior to be used as an input model to the BLAST model checker. The generated Z specifications can be used as an input model to SAL model checker after processing. However, a model checker also requires formal specifications along with the input model of a software for model checking. It can be observed that the above generated artefacts can be used as input model but the user of these approaches still requires the formal specifications in the form of LTL or CTL formulas for model checking.

It is also important to note that these approaches produce formal specifications or formal model only. A model checker requires a formal model as well as formal specifications of a software. The user of the existing approaches has to develop the formal specifications or a formal model depending on the generated output of the approach. This is expensive in terms of cost and time.

It can also be observed that only a few of the existing transformation approaches have a published tool. This limitation makes such transformation approaches difficult to be practised in the software industry.

It is also observed that these approaches perform transformation at instance level instead of metamodel-based transformation. A metamodel-based transformation requires the definition of metamodel for input and output artefacts along with the definition of transformation rules at metamodel-level.

There is room for a metamodel-based approach that transform use case to corresponding formal model and formal specifications. The metamodel-based transformation approach can handle all the definable instances of input and can transform these into instances of target metamodel. Whereas, the ability to generate formal specification along with a formal model facilitates a quality engineer to verify a software behaviour.

3.3 Summary

The transformation approaches, discussed in this chapter, take informal software requirements as an input and generate semiformal or formal notations as an output. It is observed that:

- Some of these approaches require additional software artefacts other than informal software artefacts for the transformation process
  - The additional artefacts also include such artefacts that are developed in the design phase of software development. Such approaches can not be practised prior to the design phase.
- It is also observed that some of these approaches require the use of a variant of controlled natural language.
- Some of these approaches are domain specific.
  - These approaches require the services of a domain expert for the definition of domain concepts.

- The generated artefacts of these approaches can not be used for model checking directly.
  - The user of these approaches also has to specify formal software specifications as LTL or CTL formulas.

- In addition to these, the existing approaches do not perform metamodel-based transformation.

- Moreover, only a few of the discussed approaches have a published tool.

This identified limitations of the existing approaches are discussed in this chapter and this also answers RQ:1.

Chapter 4

Use Case to a Kripke Structure and LTL Formulas Generation

This chapter discusses the proposed approach. This approach transforms informal software specifications into formal specifications. The approaches discussed in Chapter 3 also transform informal software requirements. But, it is observed that most of these approaches generate semiformal notations and some of these are domain specific. A number of these approaches require additional artefacts for the transformation. Whereas, only a few of these are capable to perform metamodel-based transformation. These limitations offer an opportunity to propose an approach that is capable to produce formal notations by performing the metamodel-based transformation. In addition to it, the proposed approach is not limited to a domain and also not require any additional artefact. It is also supported with a tool. These features allow the proposed approach suitable for the software industry.

A use case template proposed to specify an input use case along with the *Extended BackusNaur Form* EBNF grammar to validate an input are discussed prior to the transformation process. In addition to these metamodels for the proposed use case template, Kripke structure and LTL formula as well as the metamodel-based transformation process are presented in this chapter.

4.1 Use Case Template

The input use case is required to be specified in a proposed template. As discussed in Chapter 2, there are multiple use case templates available for a use case specification. But none of these templates enlist *inputs* and *outputs* of the software distinctly, these are already identified at specification stage. Moreover, *normal* and *alternate* scenarios are listed in different sections. All these add processing cost as the process has to manipulate different section iteratively. It is also observed in the existing approaches that only the work of Simko et al. uses Cockburn's template for writing a use case description. All other approaches propose a customize template to specify a use case.

This approach proposes a use case description template to facilitate the transformation process. This template distinctly specify *inputs* and *outputs*. In addition to this, *normal* and *alternate* scenario are listed in continuation to facilitate the transformation process. The proposed template requires to use a set of keywords to specify a use case. These keywords inclue: *UseCase*, *ActorSet*, *InputSet*, *OutputSet*, *Scenario*, *Alternate_Scenario*, *End_of_AlternateScenario*, *Continue*, *Include*, *Extend* and *End_of_Usecase*. The proposed use case template is shown in Figure 4.1.

UseCase construct is used to specify a use case name. *ActorSet* lists the actor(s) of a use case. *InputSet* specifies the *input(s)* of a software, specified in a use case. *OutputSet*'s *outputsymbol* lists an *output*. The possible value(s) this *output* is(are) listed alongside. There could be multiple *OutputSet* constructs in a use case to represent multiple *outputs* of a software. *Scenario* specifies the functionality of the use case consisting of actor and software 's actions. *Alternate_Scenario* specifies an alternate scenario of a functionality. The end of an alternate scenario is marked by *End_of_AlternateScenario* or *Continue* constructs. *End_of_AlternateScenario* marks the end of an alternate scenario, where software terminates its execution. Whereas, *Continue* construct marks a software allows a user to provide a valid input to continue its functionality. *Include* allows to add another use case. Whereas, *Extend* construct extends another use case

UseCase: Use Case Name[unique]
ActorSet: List of Actor(s)
InputSet: $Input_1$, $Input_2$, , $Input_k$
OutputSet [$outputsymbol_1$**]**: $Value_1$, $Value_2$, , $Value_l$
OutputSet [$outputsymbol_2$**]**: $Value_1$, $Value_2$, , $Value_m$
.
.
.
OutputSet [$outputsymbol_z$**]**: $Value_1$, $Value_2$, , $Value_n$
Scenario:
Actor action line
System action line
Alternate_Scenario
Actor action line
System action line
End_of_AlternateScenario
Actor action line
System action line
Alternate_Scenario
Actor action line
System action line
Continue
Actor action line
System action line
Include Use Case Name
Extend Use Case Name **Condition** Actor action line
Actor action line
System action line
End_of_Usecase

Figure 4.1: Proposed Use Case Template.

along with a *Condition* keyword that specifies a required user's interaction for this extension. *End_of_Usecase* is used to mark the end of a use case. Only one *Usecase*, *ActorSet*, *InputSet*, *Scenario* and *End_of_Usecase* constructs are allowed in a use case description. However, there is no restriction on the occurrence of other constructs.

The input use case is required to be in the proposed template. An EBNF grammar is also developed to ensure the validity of an input use case. The developed grammar consists of a set of terminal, nonterminal and literal values. The set of terminals includes: *aplha*. The set of nonterminals includes: *Id*, *UcTemplate*, *UcName*, *OtherthanName*, *Actor*, *OtherthanActor*, *InputSymbol*, *OtherthanInput*, *OutputSet*, *OtherthanOutputset*, *Label*, *OutputValue*, *Scenario*, *OtherScenarios*, *ScenarioLine*, *AlternateScenario*, *UserLine*, *SystemLine*, and *EndAlternate*. The literal values include: UseCase, ActorSet, InputSet, OutputSet, Scenario, Alternate_Scenario, End_of_AlternateScenario, Continue, Include, Extend, Condition and End_of_Usecase. The production rules of the developed grammar are listed in Figure 4.2.

```
alpha = 'a'..'z' + 'A'..'Z' + '_'.
Id= alpha {alpha}.
UcTemplate = "UseCase:" UcName OtherthanName.
UcName = Id {Id}.
OtherthanName = "ActorSet:" Actor {"," Actor} OtherthanActor.
Actor = Id.
OtherthanActor = "InputSet:" InputSymbol {"," Inputsymbol} OtherthanInput.
InputSymbol = Id.
OtherthanInput = OutputSet {OutputSet} OtherthanOutputset.
OutputSet = "OutputSet" "["Label "]:" OutputValue {"," OutputValue}. Label = Id.
OutputValue = Id.
OtherthanOutputset = "Scenario:" Scenario {OtherScenarios} "End_of_Usecase".
Scenario = ScenarioLine {ScenarioLine} {AlternateScenario}.
ScenarioLine = UserLine | SystemLine.
AlternateScenario = "Alternate_Scenario" ScenarioLine ScenarioLine EndAlternate.
EndAlternate = "End_of_AlternateScenario" | "Continue". UserLine = Id {Id}.
SystemLine = Id {Id}.
OtherScenarios = Scenario | "Include" UcName | "Extend" UcName "Condition" UserLine.
```

FIGURE 4.2: Extended Backus-Naur Form (EBNF) Grammar for the Use Case Template.

The non-terminal *alpha* allows English alphabets as well as _. An Id consists of one or more *alpha*. For the definition of non-terminal production, it starts with the definition of $UcTemplate$. It produces a literal UseCase and two non-terminals $UcName$ and $OtherthanName$. $OtherthanName$ is produced in a literal ActorSet with one or more *Actor* terminal(s) and a non-terminal $OtherthanActor$. *OtherthanActor* is produced to a literal InputSet with one or more non-terminal $InputSymbol$ and a non-terminal $OtherthanInput$. $OtherthanInput$ non-terminal is generated to one or more *OutputSet* and *OtherthanOutputset* non-terminal. *OutputSet* is allowed to translate into an OutputSet literal with a non-terminal *Label* terminal along with one or more $OutputValue$ non-terminals. $OtherthanO-utputset$ is allowed to produce a Scenario literal with *Scenario* and $OtherScenarios$ non-terminals along with an End_of_Usecase literal. *Scenario* is generated to one or more *ScenarioLine* non-terminals and into an *AlternateScenario* non-terminal. *ScenarioLine* is allowed to generate a $UserLine$ or a *SystemLine* non-terminals. *AlternateScenario* is allowed to produce into Alternate_Scenario literal with one or more *ScenarioLine* non-terminals along with *EndAlternate* non-terminal. *EndAlternate* is allowed to generate into an End_of_AlternateScenario or a Continue literals. $OtherScenarios$ is allowed to produce into a *Scenario* non-terminal or Include construct along with an $UcName$ non-terminal. The other possible production of $OtherScenarios$ allows it to produce a Extend literal with $UcName$ as well as a Condition literal with an $UserLine$ non-terminal.

The non-terminals $UcName$, $Actor$, *label*, $OutputValue$, $UserLine$, $SystemLine$ are allowed to produce Id. This developed grammar is also implemented in the developed tool and if the input use case does not comply with the rules of this grammar, the tool generates an error message.

The approach proposed by Simko et al. requires the definition of annotation tags. This require the services of a domain expert. Furthermore, it is also tedious to specify a use case using these annotation tags. Whereas, the approach of Sing et al. requires the understanding of domain model along with a sequence diagram. A practitioner with sound knowledge of UML can understand these UML artefacts. Though, the proposed approach also requires use case specification in a proposed template. But this specification only requires the identification of a software inputs along with outputs and these are clear at requirements analysis phase. Moreover, the used keywords are common in software development community. In addition to these, the proposed approach does not require the services of a domain expert.

4.2 Use Case Meta Model

Meta model level transformation requires a meta model for the input, i.e, a use case in this approach. The input use case can include or extend other use cases. The $UsecaseFlattener$ process discussed in the following Subsection 4.5.1, handles the included and/or extended use cases and generates a flattened use case. A meta model for the flattened use case is defined and is shown in Figure 4.3.

$UseCase$ contains a *name*, an *ActorSet*, an $InputSet$, one or more *OutputSet* and one or more *Scnarioline*(s). An *ActorSet* can have one or more *Actor* constructs. Each of which have a *name* label. An $InputSet$ can have one or more $InputSymbol$ and each $InputSymbol$ has a *name* label. An *OutputSet* has a *label* and one or more *OutputSymbol*(s). Each of *OutputSymbol* has a *value*. A *ScenaioLine* can be an *ActorActionLine*, a *SystemActionLine*, an *AlternateScenarioLine*, a *ContinueLine*, an *EndAlternateScenarioLine*, an *ExtensionPointLine*, an

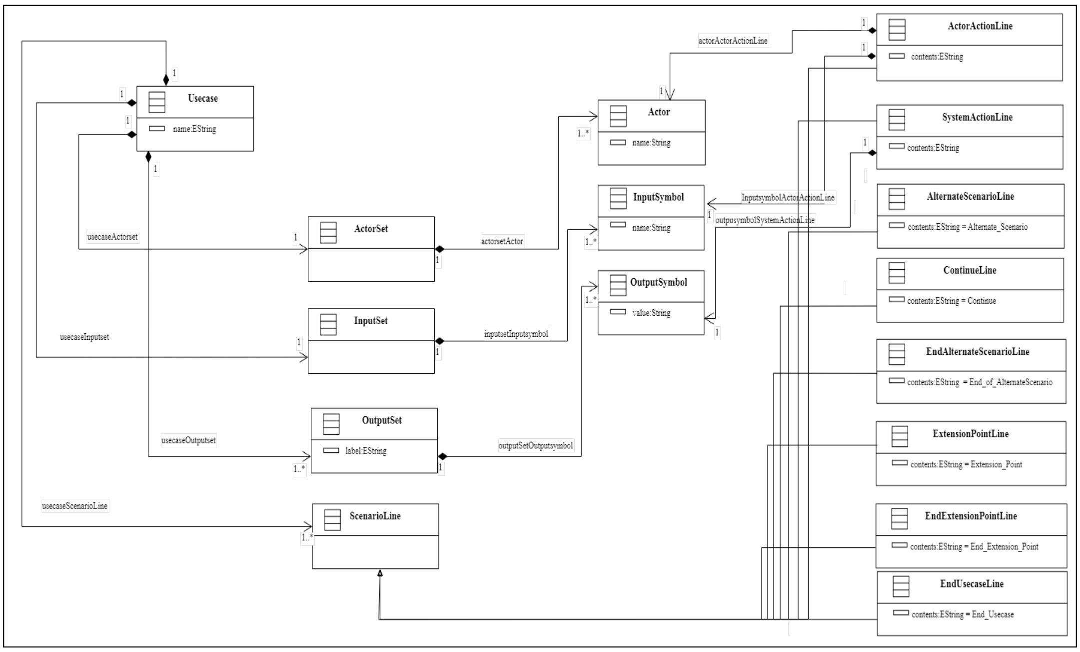


FIGURE 4.3: Flatten Use Case Meta Model.

EndExtensionPointLine or an *EndUseCaseLine*. An *ActorActionLine* contains *Actor* and*InputSymbol* in its *contents*. A *ystemActionLine*'s *contents* has *OutputSymbol*. Whereas, the *AlternateScenarioLine*, *ContinueLine*, *EndAlter−nateScenarioLine*, *ExtensionPointLine*, *EndExtensionPointLine* and *EndUse−caseLine contents* specify *Alternate_Scenario*, *Continue*, *End_of_AlternateSce−nario*, *Extension_Point*, *End_Extension_Point* and *End_of_Usecase* respectively.

This defined use case meta model can be presented as:

$UseCase_{metamodel} = \langle\ name_i,\ ActorSet_i,\ InputSet_i,\ (k \times OutputSet)_i,\ (\ell \times ScenarioLine)_i\ \rangle$. where $i = 1, \ldots, n$, represents the i^{th} instance of use case meta model. $name_i$ is a use case model name. $ActorSet_i = \{\ Actor_1, Actor_2, \ldots, Actor_j\ \}$ and $j \in \mathbb{N}$.

$InputSet_i = \{\ InputSymbol_1, InputSymbol_2, \ldots, InputSymbol_m\ \}$ where $m \in \mathbb{N}$ and *InputSymbolś* is stored in *name* label.

A use case model have k *OutputSet* where $k \in \mathbb{N}$. Whereas, $OutputSet = \{\ label, OutputSymbol_1, OutputSymbol_2, \ldots, OutputSymbol_o\ \}$ where $o \in \mathbb{N}$ and each $OutputSymbol = \{\ value\ \}$.

A use case model have ℓ *ScenarioLines* and a $ScenarioLine = ActorActionLine$, *SystemActionLine*, *AlternateScenarioLine*, *EndAlternateScenarioLine*, *Conti−nueLine*, *ExtensionPointLine*, *EndExtensionPointLine*, *EndUseCaseLine* }.

The proposed meta model is implemented by using Eclipse Modelling Framework (EMF) [109]. Epsilon is a family of Java-based scripting languages for model-to-model transformation and model validation using EMF. It also includes Eclipse-based editors for modelling and model visualization. There are number of other options available including Xtext and Sirius. The use of editor is subjective, however, the metamodels are developed using Ecore by utilizing Epsilon editors.

The input use case is transformed into a Kripke structure and LTL formulas. The meta models for the Kripke structure, LTL formulas along with the transformation rules are provided in the following sections.

4.3 Kripke Structure Meta Model

Meinke et al. [20] propose a variation to the standard definition of a Kripke structure, provided in Chapter 2. They modified it by labelling the state of a Kripke structure with a Boolean bitvector. This allows to represent a system state in the corresponding binary format. The formal definition for this variant of Kripke structure is:

- Q: set of states,
- Σ: set of input symbols,
- δ: $Q \times \Sigma \to Q$, a transition function,
- $q_0 \in Q$: an initial state,
- λ: $Q \to \mathbb{B}^k$, a labelling function.

$\mathbb{B}^k$ is a Boolean bitvector and ($b_1, \ldots, b_k$)and it is an indexing of a set $AP$of k atomic propositions.

The meta model transformation process requires meta model definition of the models being used in the transformation. To the best of knowledge, there is no meta model definition exist for a kripke structure. However, Arcaini et al. [56] propose a meta model for a Finite State Machine (FSM). The developed meta model represents a Mealy machine. Whereas, a Kripke structure is an extension of a Moore machine with binary labels on it states to represent a system state in the binary format. This work also define a meta model for the extended definition of a Kripke structure proposed by Meinke et al. [20].

This work introduces a meta model for a Kripke structure to make this approach suitable for meta model level transformation. The defined meta model for a Kripke structure is shown in Figure 4.4.

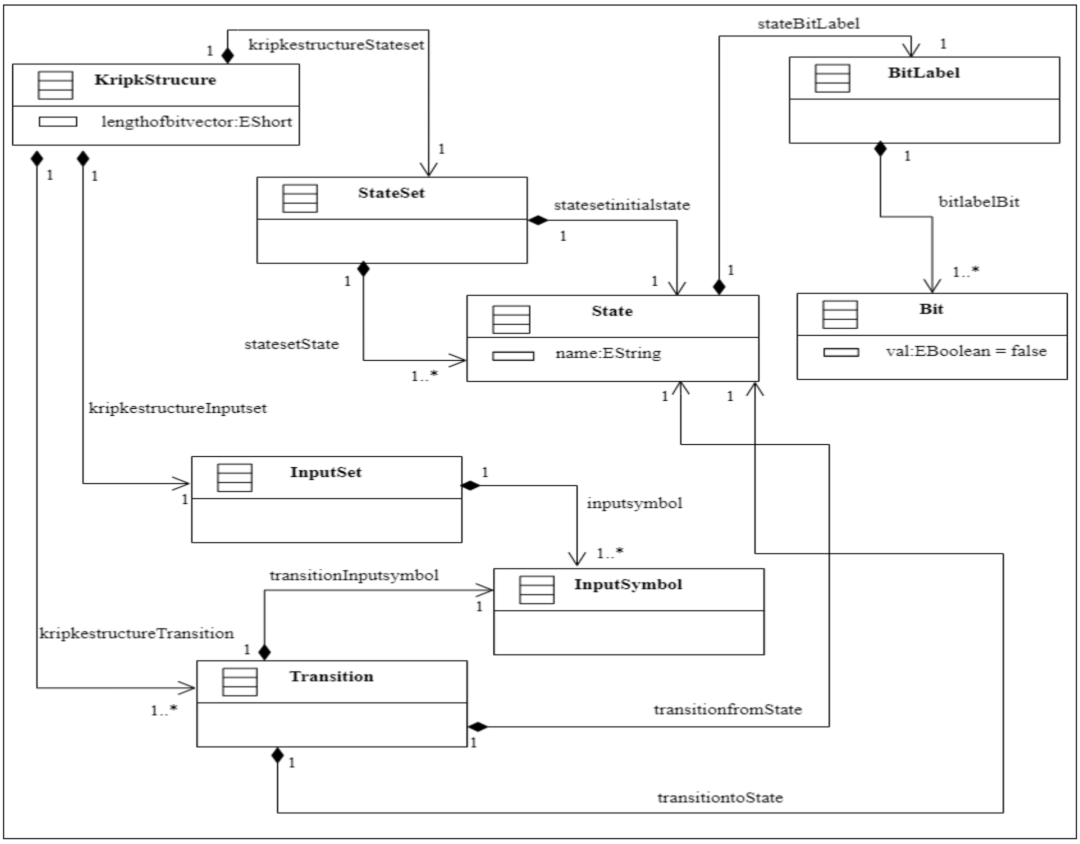


FIGURE 4.4: Kripke Structure Meta Model.

A $KripkeStructure$ contains a $Lengthofbitvector \in \mathbb{N}$, a $StateSet$, an $InputSet$ and one or more $Transition$(s). A $StateSet$ have one or more $State$s including an $initialstate$. A $State$ have a $name$ for state identification, a $BitLable$ consisting of one or more Bits. A $Bit = \{ true, false \}$. An $InputSet$ have one or more $InputSymbol$. An $InputSymbol$ have a $name$ to identify the symbol. A $Transition$ have a $fromstate$, $tostate$ and an $InputSymbol$.

This defined meta model is represented as:

$KripkeStructure_{metamodel} = \langle\ StateSet_i,\ InputSet_i,\ (\ k \times Transition\ )\ _i\ \rangle$ where $i \in \mathbb{N}$ denotes the i^{th} instance of the Kripke structure meta model.

A $StateSet_i = \{\ q_{initial},\ q_1,\ q_2,\ \ldots,\ q_l\ \}$ where $l \in \mathbb{N}$. The $q_{initial} = \{\ name = Initial\_State$ and $BitLabel\ \}$ where $BitLabel = \{\ Bit_1,\ \ldots,\ Bit_{lengthofbitvector}$ and $\forall\ Bit = false\ \}$. Each $q = \{\ name,\ BitLabel\ \}$ where $BitLabel = \{\ Bit_1,\ \ldots,\ Bit_{lengthofbitvector}$ and $\forall\ Bit = \{\ true,\ false\ \}\ \}$

$InputSet_i = \{\ InputSymbol_1,\ InputSymbol_2,\ \ldots,\ InputSymbol_m\ \}$ where $m \in \mathbb{N}$ and each $InputSymbol = name$ to identify it.

$k \in \mathbb{N}$ and $Transition = \{\ q_{fromstate},\ q_{tostate},\ InputSymbol\ \}$. The designed meta model is implemented by using EMF [109].

4.4 Linear Temporal Logic (LTL) Formula Meta Model

The meta model transformation also requires the meta model definition of output artefacts. A meta model for LTL formula is proposed to make this meta model level transformation. The developed meta model for LTL formula is shown in Figure 4.5.

$LTLForm$ have $UntImpExpression$, $UnaryExpression$, $BinaryExpression$ and $Literal$.A $UntImpExpression$ have $UnaryExpression$, $BinaryExpression$ and $Literal$. A $UnaryExpression$ is applied to a $BinaryExpression$, $UnaryExpressi-on$ and a $Literal$. A $BinaryExpression$ consists of $UnaryExpression$, $Literal$ or a $BinaryExpression$.

This defined meta model is represented as:

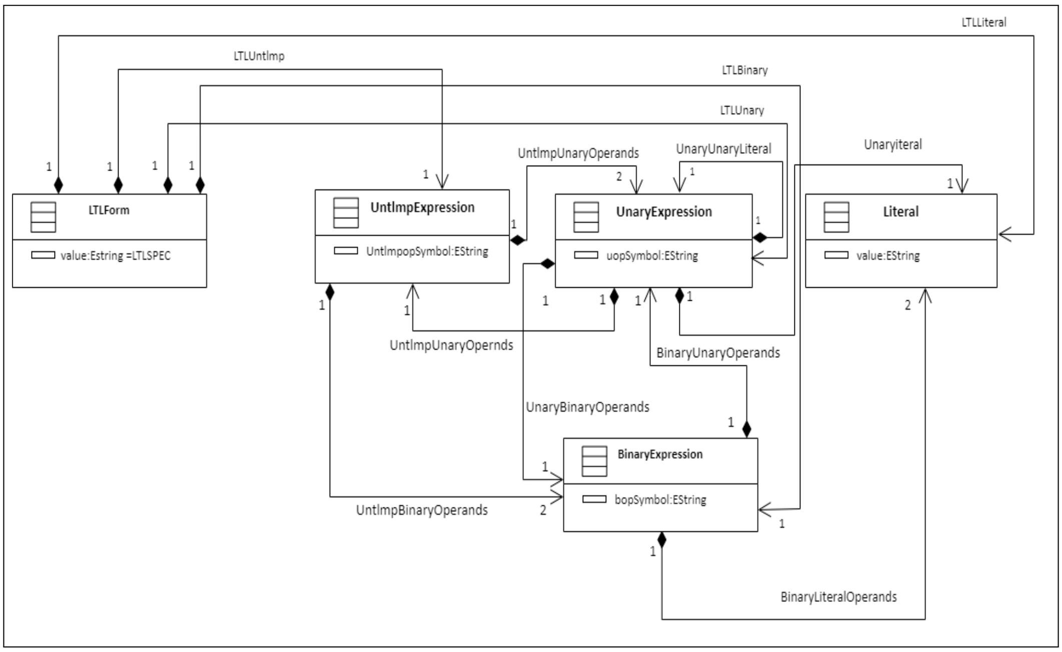


FIGURE 4.5: LTL Formula Meta Model.

$LTLForm_{metamodel} = \langle\, UntImpExpression_i, UnaryExpression_i, BinaryExpres-sion_i,\ Literal_i\ \rangle$ where $i \in \mathbb{N}$ denotes the i^{th} instance of the LTL formula meta model.

A $UntImpExpression = \{UntImpopSymbol, UnaryExpression, BinaryExpre-ssion,\ Literal\ \}$. and an $UntImplopSymbo = \{\ \mathrm{U}, \rightarrow\ \}$

A $UnaryExpression\ =\ \{\ uopSymbol,\ UnaryExpression,\ BinaryExpression,\ Literal\ \}$ and an $uopSymbol = \{\ !, \mathrm{X}, \mathrm{F}, \mathrm{G}\ \}$.

A $BinaryExpression = \{\ \{\ bopSymbol, BinaryExpression, UnaryExpression,\ Literal\ \}\ \}$ and a $bopSymbol = \{\ |, \&\ \}$

A $Literal = \{\ value\ \}$ and $value$ is an atomic predicate. The proposed meta model is implemented by using EMF [109].

4.5 Proposed Approach

This work proposes a domain independent approach that is capable to perform transformation at meta model level. This approach transforms a use case into a

Kripke structure and LTL formulas. The generated Kripke structure and the LTL formulas are formal in nature and can be used as input for model checking. This approach does not require any additional artefact for the transformation. This approach performs transformation at meta model level and it is also supported with a platform independent tool.

The schematic diagram of the proposed approach is shown in Figure 4.6.

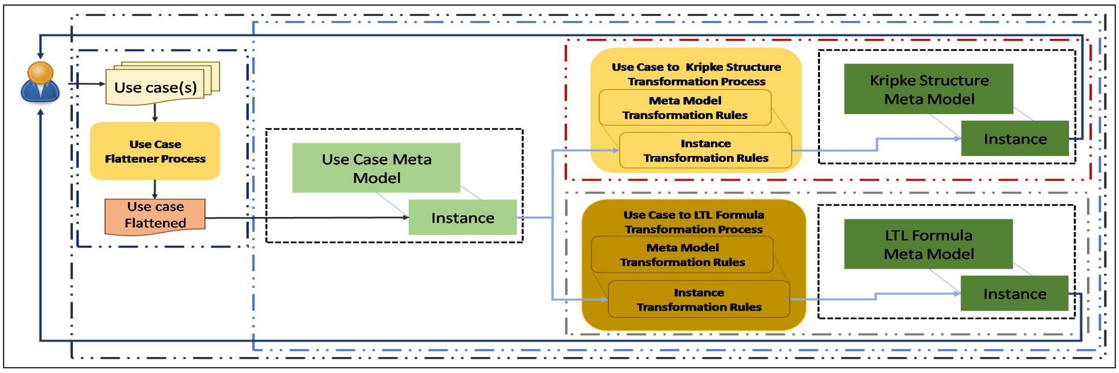


FIGURE 4.6: Proposed Approach Schematic Diagram.

The user of this approach provides a use case model as an input. This approach also handles use case relationships, i.e., *include* and *extend* relationships. *UseCaseFlattener* process takes the input use case. This process checks either the provided use case *include* or *extend* other use case(s). This process flattens the input use case by handling the use case relationships. This process output the same use case, if no other use case being included or extended. This flattened use case is provided as input to *Use Case to Kripke Structure Ttransformation* process and *Use Case to LTL Formula Transformation* process. These processes generate a Kripke structure and LTL formulas as output respectively. *Use Case Flattener* process, *Use Case to Kripke Structure Transformation* process and *Use Case* to *LTL Formula Transformation* process are discussed in detail in the following subsections.

4.5.1 Use Case Flattener Process

Use Case Flattener process takes a use case as input and checks either this use case *include* or *extend* other use cases. It handle the use case relationships and generate a flattened use case as output. The schematic diagram of this process is shown in Figure 4.7.

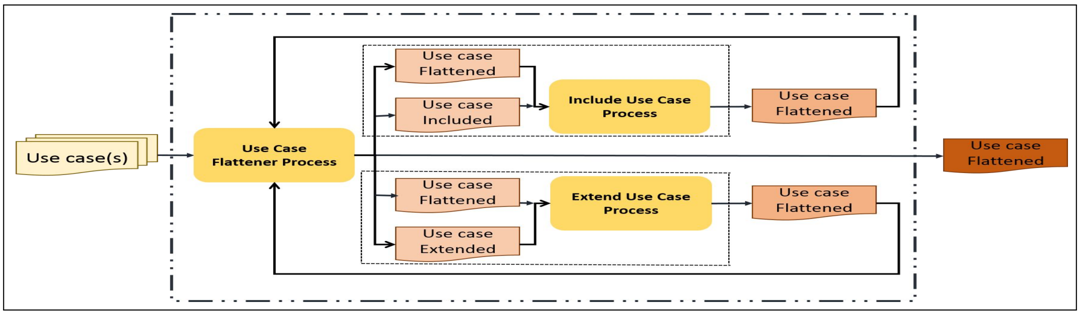


FIGURE 4.7: Use Case Flattener Process Schematic Diagram.

The input use case is required to be specified in the proposed template discussed in Section 4.1. The input use case description is denoted by UC variable and it is passed to Algorithm 1 and this algorithm returns a flattened use case as output represented by $UC_{flattened}$. The algorithm defines a UC_{temp}, $UC_{flattened}$ and initializes $UC_{flattened}$'s constructs namely $ActorSet$, $InputSet$, $OutputSet$ with the corresponding constructs of UC. It reads $UC.Scenario$ line by line for the occurrence of Include or Extend literals. If a line contains Include or Extend literal, this algorithm stores the included or extended use case name to UC_{temp} and calls $IncludeUseCase$ or $ExtendUseCase$ algorithms respectively. $IncludeUseCase$ and $ExtendUseCase$ processes are presented as Algorithm 2 and 3 respectively.

Algorithm 1 UseCaseFlattener

Require: UC as a use case description in the proposed template

Ensure: $UC_{flattened}$ as a use case description in the proposed template

Define UC_{temp}, $UC_{flattened}.ActorSet \leftarrow UC.ActorSet$, $UC_{flattened}.InputSet \leftarrow UC.InputSet$, $UC_{flattened}.OutputSet \leftarrow UC.OutputSet$

for ℓ in $UC.Scenario$ **do**

    if ℓ contains $Include$ **then**

        $UC_{temp} \leftarrow Ucname$

        $UC_{flattened} \leftarrow$ IncludeUseCase($UC_{flattened}$,UC_{temp})

    else if ℓ contains $Extend$ **then**

        $UC_{temp} \leftarrow Ucname$

        $UC_{flattened} \leftarrow$ ExtendUseCase($UC_{flattened}$,UC_{temp},Userline)

else

$UC_{flattened}.Scenario \leftarrow UC_{flattened}.Scenario + \ell$

end if

end for

Algorithm 2 IncludeUseCase

Require: $UC_{flattened}$, $UC_{included}$ as use case descriptions in the proposed template

Ensure: $UC_{flattened}$ as a use case description in the proposed template

$UC_{flattened}.ActorSet \leftarrow UC_{flattened}.ActorSet \cup UC_{included}.ActorSet$

$UC_{flattened}.InputSet \leftarrow UC_{flattened}.InputSet \cup UC_{included}.InputSet$

$UC_{flattened}.OutputSet \leftarrow UCflattened.OutputSet \cup UC_{included}.OutputSet$

$UC_{flattened}.Scenario \leftarrow UC_{flattened}.Scenario + UC_{included}.Scenario$

Algorithm 3 ExtendUseCase

Require: $UC_{flattened}$, $UC_{extended}$ as use case descriptions in the proposed template, Userline as scenario line

Ensure: $UC_{flattened}$ as a use case description in the proposed template

$UC_{flattened}.ActorSet \leftarrow UC_{flattened}.ActorSet \cup UC_{extended}.ActorSet$

$UC_{flattened}.InputSet \leftarrow UC_{flattened}.InputSet \cup UC_{extended}.InputSet$

$UC_{flattened}.OutputSet \leftarrow UCflattened.OutputSet \cup UC_{extended}.OutputSet$

$UC_{flattened}.Scenario \leftarrow UC_{flattened}.Scenario + Extension\_Point + Userline$

$UC_{flattened}.Scenario \leftarrow UC_{flattened}.Scenario + UC_{extended}.Scenario + End\_Extension\_Point$

Algorithm 2 provides the working of $IncludeUseCase$ process. This algorithm requires $UC_{flattend}$ and $UC_{included}$. UC_{temp} is referred as $UC_{included}$. This algorithm merges the $ActorSet$, $InputSet$ and $OutputSet$ constructs of $UC_{included}$ to the corresponding constructs of $UC_{flattened}$. This algorithm also concatenates $Scenario$ of UC_temp to the $Scenario$ of $UC_{flattened}$ and returns $UC_{flattened}$.

Algorithm 3 provides the functionality of $ExtendUseCase$ process. This algorithm accepts the $UC_{flattened}$, $UC_{extended}$ use case descriptions and $UserLine$ a scenario line passed from the $UseCaseFlattener$ process. UC_{temp} is referred as $UC_{extended}$. This algorithm merges *ActorSet*, *InputSet* and *OutputSet* constructs of $UC_{extended}$ to the corresponding constructs of $UC_{flattend}$. It concatenates $ExtensionPoint\ UserLine$ at end of *Scenario* of $UC_{flattened}$ and then after concatenates *Scenario* construct of $UC_{extended}$ to the updated *Scenario* construct of $UC_{flattened}$ with *End_Extension_Point* to the end of it. This process returns the updated $UC_{flattened}$. This flattened use case is then passed to $Use\ Case\ to\ Kripke\ Structure\ Transformation$ process and $Use\ Case\ to\ LTL\ Formula\ Transformation$ process to generate a Kripke structure and LTL formulas. These processes are discussed in the following subsections.

4.5.2 Use Case to Kripke Structure Transformation Process

$Use\ Case\ to\ Kripke\ Structure\ Transformation$ process takes an instance of use case meta model as input and produces a Kripke structure as output. The input use case instance is denoted by UC and the output Kripke structure instance is denoted by KS. Meta models for the flattened use case and Kripke structure are discussed in Section 4.2 and Section 4.3 respectively.

The proposed approach produces an instance of kripke structure meta model from the provided instance of use case meta model. This transformation process is discussed in the following paragraphs:

Rule 1 calculates the length of *bitvector*. This is calculated from the number of *OutputSet* and *OutputSymbol*(s) in each *OutputSet*. This rule also calculates distinct binary equivalent values for each *OutputSymbol*. In addition to it, this rules copies the use case *InputSet* to Kriprke structure instance identified as KS's *InputSet*.

Rule 1 Calculate Binary Values, Bitvector Length and Copy Input Symbols

$ucOPSet_{Binary}$: new $UseCase!OutputSet$, $bitvectorlength \leftarrow 0$, $InputSet_{temp}$: new $KripkeStructure!InputSet$
for $UC.OutputSet$ **do**
  $count_{output} \leftarrow OutputSet.OutputSymbol.count$
  $bit_{req} \leftarrow$ RequiredBits$count_{output}$; $bitvectorlength$ += bit_{req}
  for $OutputSymbol$ **do**
    $ucOPSet_{Binary}.value \leftarrow OutputSymbol.value$
    $ucOPSet_{Binary}.binaryvalue \leftarrow$ random binary value
  end for
end for
for $UC.InputSet.InputSymbol$ **do**
  $InputSet_{temp}.InputSymbol.name \leftarrow UC.InputSet.InputSymbol.name$
end for
$KS.Inputset = InputSet_{temp}$
$KS.bitvectorlength \leftarrow bitvectorlength$

Rule 2 Define Initial and Dead States

$BitLabel_{temp}$: new $KS.BitLable$, $state_{dead}$, $q_{current}$: new $KS.State$
for $BitLabel_{temp}$ **do**
  $BitLabel_{temp}.Bit.val \leftarrow false$
end for
$KS.State.InitialState.BitLabel \leftarrow BitLabel_{temp}$
$state_{dead} \leftarrow BitLabel_{temp}$
$q_{current} \leftarrow KS.State.InitialState$

Rule 2 defines an $InitialState$ and a dead state $state_{dead}$ of KS. This rule also assigns all indices of $BitLabel$ of these states to $false$ and also marks $InitialState$ as $q_{current}$.

Rule 3 parses all the $ScenarioLine$ of input use case UC. This rule marks $isInput$ and $isActor$ turn on the flags marking that an input and actor symbols are read. The read input symbol is stored as σ_{temp}. This rule also stores the read *OutputSymbol* in a scenario line in $output_{temp}$ variable.

Rule 3 Scan Actor, Input and Output Symbols

```
for UC.ScenarioLine do
    for UC.InputSet do
        if ℓ contains σ then
            isInput ← true, σ_temp ← InputSymbol.name
        end if
    end for
    for UC.ActorSet do
        if ℓ contains Actor then
            isActor ← true
        end if
    end for
    for UC.OutputSet do
        for OutputSymbol do
            if ℓ contains OutputSymbol then
                output_temp ← OutputSymbol.value
            end if
        end for
    end for
end for
```

Rule 4 Define the New State and Transitions

```
if isInput AND isActor then
    isInput ← false, isActor ← false
    Define q_new, BitLabel_temp ← q_current.BitLabel
    BitLabel_temp ← BitLabelUpdater ( output_temp, ucOPSet_Binary, BitLabel_temp )
    q_new.BitLabel ← BitLabel_temp
    KS.State.add(q_new)
    if isExtensionPoint then
        KS.Transition.add(q_beforeExtension, q_new, σ_temp)
    else
        KS.Transition.add(q_current, q_new, σ_temp)
    end if
    for UC.InputSet - σ_temp do
        KS.Transitio.add(q_current, state_dead, σ)
    end for
    q_current ← q_new
end if
```

Rule 4 checks for the marked flags *isInput* and *isActor*. This rule resets the values of these flags, if they are marked *true*, it defines a new state q_{new}. This rule copies the $BitLabel$ of $q_{current}$ to $BitLabel_{temp}$ and it also update the corresponding indices of $BitLabel_{temp}$ with the corresponding binary value of read $output_{temp}$. This rule defines a transition from $q_{beforeExtension}$ to this newly created state q_{new} if $isExtensionPoint$ flag is marked true true. This flag is marked true by Rule 8 on reading $ExtensionPointLine$, otherwise if $isExtenstionPoint$ is false then it defines a transition from $q_{current}$ to q_{new} and adds q_new in $KS.state$. This new transition is labelled with read input symbol σ_{temp}. This rule then defines a transition from $q_{current}$ to q_{dead} for each input symbol of $InputSet$ else of σ_{temp} and marks the transitions with corresponding input symbols to make KS deterministic. All of these created transitions are added to $KS.Transtion$. The value of $q_{current}$ is updated to q_{new}.

Rule 5 checks for $AltrenateScenaioLine$ and copies $q_{current}$ in q_{hold} and defines a new state q_{new}. This rule copies the $BitLabel$ of $q_{current}$ to $BitLabel_{temp}$. It updates the corresponding indices of $BitLabel_{temp}$ for $output_{temp}$ and assigns this updated $BitLabel_{temp}$ as q_{new}'s $BitLabel$. A transition from $q_{current}$ to q_{new} is defined and is labelled with the value of σ_{temp}. This rule also defines the transitions for each input symbols else of σ_{temp} in $InputSet$ from $q_current$ to $stated_{dead}$ and marks each of this transition with corresponding input symbol. All of the new transitions are added in $KS.Transition$ and $q_{current}$ is updated to q_{new}.

Rule 5 Process a Alternate Scenario Line

if ℓ.typeOf($UC.AlternateScenarioLine$) **then**
  $q_{hold} \leftarrow q_{current}$
  Define q_{new}, $BitLabel_{temp} \leftarrow q_{current}$.BitLabel
  $BitLabel_{temp} \leftarrow$ BitLabelUpdater ($output_{temp}$, $ucOPSet_{Binary}$, $BitLabel_{temp}$)
  q_{new}.BitLabel $\leftarrow BitLabel_{temp}$
  $KS.State$.add(q_{new})
  $KS.Transition$.add($q_{current}$, q_{new}, σ_{temp})
  for $UC.InputSet$ - σ_{temp} **do**
    $KS.Transition$.add($q_{current}$, $state_{dead}$, σ)
  end for
  $q_{current} \leftarrow q_{new}$
end if

Rule 6 checks the occurrence of $ContinueLine$ and adds a transition in $KS.Transi-tion$ from $q_{current}$ to $q_{current}$ and labels this transition with the value of σ_{temp}. This rule also updates $q_{current}$ to q_{hold}.

Rule 6 Process a Continue Line

1: **if** ℓ.typeOf($UC.ContinueLine$) **then**
2: $KS.Transition.add(q_{current}, q_{current}, \sigma_{temp})$
3: $q_{current} \leftarrow q_{hold}$
4: **end if**

Rule 7 reads the $EndAlternateScenarionLine$ and it adds a transition from $q_{current}$ to $InitialState$ in $KS.Transition$. This transition is labelled with σ_{temp} and the value of $q_{currents}$ is updated to q_{hold}.

Rule 7 Process End of Alternate Scenario Line

1: **if** ℓ.typeOf($UC.EndAlternateScenarioLine$) **then**
2: $KS.Transition$.add($q_{current}$, $KS.States.InititalState$,σ_{temp})
3: $q_{current} \leftarrow q_{hold}$
4: **end if**

Rule 8 Process Extension Point Line

1: **if** ℓ.typeOf($UC.ExtensionPointLine$) **then**
2: $isExtensionPoint \leftarrow true$
3: $q_{beforeExtenstion} \leftarrow q_{current}$
4: **end if**

Rule 8 marks the $isExtenstionPoint$ to true and copies $q_{current}$ to $q_{beforeExtension}$ on reading $ExtensionPointLine$ line.

Rule 9 adds a transition in $KS.Transition$ from $q_{current}$ to $q_{beforExtension}$ and labels it with σ_{temp} values. This rule also updates $q_{current}$ with the value of $q_{beforeExtension}$. It also marks the $isExtensionPoint$ flag's value to $false$. It is to be noted that $EndUsecaseLine$ does not have any impact in the transformation process and it is only used to mark the end of a use case.

Rule 9 Process End Extension Point Line

if ℓ.typeOf($UC.EndExtensionPointLine$) **then**

  $KS.Transition$.add($q_{current}$, $q_{beforeExtension}$, σ_{temp})

  $q_{current} \leftarrow q_{beforeExtenstion}$

  $isExtensionPoint \leftarrow false$

end if

$Use\ Case\ to\ Kripke\ Structure\ Transformation$ process consists of a number of rules and these rules are implemented in Epsilon Transformation Language (ETL). ETL is a model-based transformation language. The transformation rules effecting the input and output metamodel concepts are mapped in Table 4.1.

TABLE 4.1: **Transformation Rules Effecting Source Target Meta Model Summary**

Sr. No.	Source Metamodel	Target Metamodel	Applicable Rule
1	UseCase.InputSet, UseCase.OutpuSet	KripkeStructure.InputSet, KripkeStructure.Lengthofbitvector	Rule1
2	–	KripkeStructure.State	Rule2
3	UseCase.InputSet, UseCase.OutpuSet	–	Rule3
4	–	KripkeStructure.State, KripkeStructure.Transition	Rule4
5	UseCase.AlternateScenarioLine, UseCase.InputSet	KripkeStructure.State, KripkeStructure.Transition	Rule5
6	UseCase.ContinueLine	KripkeStructure.Transition	Rule6
7	UseCase.EndAlternateScenarioLine	KripkeStructure.Transition	Rule7
8	UseCase.ExtensionPointLine	–	Rule8
9	UseCase.EndExtensionPointLine	KripkeStructure.Transition	Rule9

4.5.3 Use Case to LTL Formula Transformation Process

$Use\ Case\ to\ LTL\ Formula\ Transformation$ process takes a flattened use case meta model instance discussed in Section 4.2 as an input and it produces LTL formulas as output. This approach also proposed a meta model for LTL formula and is provided in Section 4.4. This process produces LTL formulas from the provided of use case. $Use\ Case\ to\ LTL\ Formula\ Transformation$ process consists of two rules. One of these rule produces LTL formulas by handling X operator and the other one generates LTL formulas by handling F operator.

Rule 1 takes a use case and it produces LTL formulas by handling X operator. This rule processes the *OutputSet* of *UC* and generates an *OutputLabel* by concatenating the *label* of each *OutputSet* to *null* value. This rule processes scenario lines one by one and scans for the occurrences of *InputSymbol* and actor. It enables *inInput* and *isActor* flags to *true* on reading these. The read input symbol is stored in $InputSymbol_{read}$ variable. The read *OutputSymbol* is stored in $OutputSymbol_{read}$ variable and *isOutput* flag is marked to *true*. The read *OutputSymbol* value is used to update the corresponding value of *OutputSet* in *OutputLabel*. If an *ExtensionPoint* is read then *OutputLabel* is stored in $OutputLabel_{beforExtension}$ that is reverted when *EndExtensionPoint* line is read. When an *Alternatescenarioline* is read, the value of *OutputLabel* is stored in $OutputLabel_{beforeAlternate}$ and this value is restored on reading *Continue* or *End Alternate Scenario* line.

The values of the *isActor*, *isInput* and *isOutput* flags are marked to true and a $Formula_{current}$ is built by arranging the value of *OutputLabel* and other operators as on written on the lines 46-49 of Rule 1.

Rule 1 LTL Next Specifications Generator

$isInput \leftarrow false$, $isActor \leftarrow false$, $isOutput \leftarrow false$

String $OutputLabel$, $OutputLabel_{beforeExtension}$, $OutputLabel_{beforeAlternate}$, $InputSymbol_{read}$, $OutputSymbol_{read}$, $Formula_{current}$

1: **for** *set* in *UC.OutputSet* **do**
2:     $set.OutputSymbols \leftarrow set.OutputSymbols$ + null
3:     $OutputLabel \leftarrow OutputLabel + set.Label +$ "= null"
4: **end for**
5: **for** ℓ in *UC.ScenarioLine* **do**
6:     **for** *inputsymbol* in *UC.InputSet* **do**
7:         **if** ℓ contains *inputsymbol* **then**
8:             $isInput \leftarrow true$
9:             $InputSymbol_{read} \leftarrow inputsymbol$

end if
end for
for *set* in $UC.OutputSet$ **do**
for *outputsymbol* in *set* **do**
if ℓ contains *outputSymbol* **then**
$isOutput \leftarrow true$
$OutputSymbol_{read} \leftarrow outputsymbol$
for $set_{available}$ in $UC.OutputSet$ **do**
if $set.Label = set_{available}$ **then**
Update $OutputLabel.set.Label \leftarrow OutputSymbol_{read}$
end if
end for
end if
end for
end for
for *actor* in $UC.ActorSet$ **do**
if ℓ contains *actor* **then**
$isActor \leftarrow true$
end if
end for
if ℓ.typeof($UC.ExtensionPointLine$) **then**
$OutputLabel_{beforeExtension} \leftarrow OutputLabel$
end if
if ℓ.typeof($UC.EndExtensionPointLine$) **then**
$OutputLabel \leftarrow OutputLabel_{beforeExtension}$
end if
if ℓ.typeof($UC.AlternateScenarioLine$) **then**
$isAlternate \leftarrow true$
$OutputLabel_{beforeAlternate} \leftarrow OutputLabel$
end if
if ℓ.typeof ($UC.ContinueLine$) OR ℓ.typeof (" $UC.EndAlternateScenarioLine$ ") **then**

$isAlternate \leftarrow false$

$OutputLabel \leftarrow OutputLabel_{beforeAlternate}$

end if

if $isActor$ AND $isInput$ AND $isOutput$ **then**

if all $OutputSet.Label.value = null$ **then**

$Formula_{current} \leftarrow$ "LTLSPEC G (state = Initial_State & input = " + $InputSymbol_{read}$ + "− > X (" + $OutputSymbol_{read}$ + ")"

else

$Formula_{current}$ = "LTLSPEC G (" $OutputLabel$ + "& input = " + $InputSymbol_{read}$ + " − > X (" + $OutputSymbol_{read}$ + ")"

end if

end if

LTL formulas = LTL formulas + $Formula_{current}$

$isActor \leftarrow false$, $isOutput \leftarrow false$

end for

Rule 2 generates LTL formulas from a use case for LTL F operator.

Rule 2 LTL Future Specifications Generator

boolean $isInput \leftarrow false$, $isActor \leftarrow false$, $isOutput \leftarrow false$

String $Input_{future}$, $Input_{beforefuture}$, $Input_{beforeExtension}$, $Input_{beforeAlternate}$, $OutputSymbol_{read}$, $Formula_{current}$

$Counter_{input} \leftarrow 0$, $isFirstWritten \leftarrow false$, $isAlternate \leftarrow false$

for ℓ in $UC.ScenarioLine$ **do**

for $inputsymbol$ in $UC.InputSet$ **do**

if ℓ contains $inputsymbol$ **then**

$isInput \leftarrow true$

if $Counter_{input} = 0$ **then**

$Counter_{input}$++

$Input_{future} \leftarrow$ " (input =" + $inputsymbol$ +")"

else

$Input_{beforefuture} \leftarrow Input_{future}$

$Input_{future} \leftarrow Input_{future}$ + " & X (input =" + $inputsymbol$ + ")"
end if
end if
end for
for set in $UC.OutputSet$ **do**
for $outputsymbol$ in set **do**
if ℓ contains $outputSymbol$ **then**
$isOutput \leftarrow true$
$OutputSymbol_{read} \leftarrow outputsymbol$
end if
end for
end for
for $actor$ in $UC.ActorSet$ **do**
if ℓ contains $actor$ **then**
$isActor \leftarrow true$
end if
end for
if ℓ.typeof($UC.ExtensionPointLine$) **then**
$Input_{beforeExtension} \leftarrow Input_{future}$
end if
if ℓ.typeof($UC.EndExtensionPointLine$) **then**
$Input_{future} \leftarrow Input_{beforeExtension}$
end if
if ℓ.typeof($UC.AlternateScenarioLine$) **then**
$isAlternate \leftarrow true$
$Input_{beforeAlternate} \leftarrow Input_{future}$
$Input_{future} \leftarrow Input_{beforefuture}$
end if
if ℓ.typeof($UC.ContinueLine$)
OR ℓ.typeof($UC.EndAlternateScenarioLine$) **then**

40: $isAlternate \leftarrow false$

41: $Input_{future} \leftarrow Input_{beforeAlternate}$

42: **end if**

43: **if** $isActor$ AND $isInput$ AND $isOutput$ **then**

44: $Formula_{current} \leftarrow$ "LTLSPEC G (state = Initial_State &" +

45: $Input_{future}$ +" − > F (" + $OutputSymbol_{read}$ +")"

46: **end if**

47: LTL formulas $\leftarrow$ LTL formulas + $Formula_{current}$

48: $isActor \leftarrow false$, $isOutput \leftarrow false$

49: **end for**

This rule parses each scenario line of the input use case and scans it for the *Inputsymbols*. The read *Inputsymbol* is stored in $Input_{future}$ by concatenating with $input_{future}$ = literal. The *Inputsymbols* read after are concatenated with it by adding a logical operator & and LTL X operator. The read *Output* symbol is stored in $OutputSymbol_{read}$. The value of $Input_{future}$ is stored in $Input_{beforeExtension}$ on reading an *ExtensionPoint* line and this value is reverted when an *EndExtensionPoint* line is read. The value of $Input_{future}$ is stored in $Input_{beforeAlternate}$ and is reverted on reading a *Continue* or *End_of_AlternateSc−enario* line. A $Formula_{current}$ is generated after arranging the generated constructs as on line 43 of Rule 2.

TABLE 4.2: **Transformation Rules Effecting Source Target Meta Model Summary**

Sr. No.	Source Metamodel	Target Metamodel	Applicable Rule
1	UseCase.ActorSet, UseCase.InputSet, UseCase.OutpuSet, UseCase.ScenarioLine	LTLForm.Literal, LTLForm.BinaryExpression, LTLUnaryExpression, LTLImpExpression	Rule1
2	UseCase.ActorSet, UseCase.InputSet, UseCase.OutpuSet, UseCase.ScenarioLine	LTLForm.Literal, LTLForm.BinaryExpression, LTLUnaryExpression, LTLImpExpression	Rule2

The transformation rules effecting the input and output metamodel concepts are mapped in Table 4.2. Ecore metamodel is exportable as Java file. The exported Java files for Use case, Kripke structure and LTL metamodels along with the transformation rules are incorporated in the tool, and are presented in Chapter 5. The usage of these exported Java files enables the definition of valid metamodel instances only.

4.5.4 Proof of Soundness and Completeness

The soundness of $Use\ Case\ to\ Kripke\ Structure\ Transformation\ Process$ can be proved by the induction on the number of scenario lines of a use case description and is listed below:

Base Case:

$ScenarioLineCount = 1$

$KripkeStructure.InputSet := UseCase.InputSet$ by line 14 of Rule 1.

$KripkeStructure.State := \{Inital.State\ ,\ state\_dead\}$

$KripkeStructure.Transition := \{\}$ by line 5-6 of Rule 2.

$KripkeStructure.State := KripkeStructure.State \cup newState$ by line 6-14 of Rule 4.

A single line use case cannot be $Alternate\_Scenario$, $Continue$, $End\_of\_Alternat-eScenario$, $Extension\_Point$ or $End\_Extension\_Point$ line.

Inductive Case:

It is assumed that for n $ScenarioLines$ a well formed Kripke structure is generated. For $ScenarioLineCount = n + 1$

$KripkeStructure.State := KripkeStructure.State \cup newState$

$KripkeStructure.Transition := KripkeStructure.Transition \cup newTransition$ if $ScenarioLine = UseCase.ActorActionLine$ and it is dealt by line 6-14 of Rule 4.

$KripkeStructure.State := KripkeStructure.State \cup newState$

$KripkeStructure.Transition := KripkeStructure.Transition \cup newTransition$ if $ScenarioLine = UseCase.AlternateScenarionLine$ and it is dealt by line 6-10 of Rule 5.

$KripkeStructure.Transition := KripkeStructure.Transition \cup newTransition$ if $ScenarioLine = UseCase.ContinueLine$ and it is dealt by Rule 6.

$KripkeStructure.Transition := KripkeStructure.Transition \cup newTransition$ if $ScenarioLine = UseCase.EndAlternateScenarioLine$ and it is dealt by Rule 7.

$KripkeStructure.Transition := KripkeStructure.Transition \cup newTransition$ if $ScenarioLine = UseCase.End\_of\_Extension\_PointLine$ and it is dealt by Rule 9.

The soundness of $Use\ Case\ to\ LTL\ Formula\ Transformation\ Process$ is also proved by the induction on the number of scenario lines of a use case description and is listed below:

Base Case:
$ScenarioLineCont = 1$
$LTLForm.Literal := UseCase.OutputSet = null$ by line 14 of Rule 1. and line14-21 of Rule 2.
$LTLForm.Literal := LTLForm.Literal \wedge, input = readInputSymbol\}$ by line 5-11 of Rule 1 and by line 1-13 of Rule 2.
$LTLForm.Literal := readOutputSymbol$ by line 12-24 of Rule 1 and line 14-21 of Rule 2.
A single line use case cannot be $Alternate\_Scenario$, $Continue$, $End\_of\_Alternat-eScenario$, $Extension\_Point$ or $End\_Extension\_Point$ line.
Inductive Case:
It is assumed that for $ScenarioLineCoutn = n$
a well formed LTL formula is generated. For $ScenarioLineCoutn = n + 1$
$LTLForm.Literal := LTLForm.Literal \wedge input = readInputSymbol$ by line 5-11 of Rule 1 and line 1-13 of Rule 2.
$LTLForm.Literal := readOutputSymbol$ by line 12-24 of Rule 1 and line 14-21 of Rule 2.
$LTLForm := LTLForm.UnaryExpression \wedge LTLForm$ by line 46, line 48 of Rule 1 and line 10, line 21 of Rule 2
$UseCase.AlternateScenarionLine$ is dealt by line 36-39 of Rule 1 and line 37-37 of Rule 2.
$UseCase.ContinueLine$ and $UseCase.End\_of\_AlternateScenarioLine$ are dealt

by line 40-43 of Rule 1 and line 38-41 of Rule 2.

$UseCase.End\_of\_Extension\_PointLine$ is dealt by line 30-32 of Rule 1 and line 27-29 of Rule 2. $ExtensionPointLine$ has no impact of $LTLForm$

It can be observed that the transformation process handles all the constructs of the input use case. It is possible that a user inputs a use case with zero line. This case is dealt at the input level. An EBNF grammar is proposed to check the input use case prior to the transformation process. The proposed EBNF grammar does not allow this type of use case input and an error message will be generated.

4.5.5 Time Complexity

Time complexity is a function that maps the growth of input to the number of operations it performs to process the input. This approach consists of two main processes including $UseCase\ to\ Kripke\ Structure\ Transformation$ process and $Use\ Case\ to\ LTL\ Formula\ Transformation$ process. $Use\ Case\ to\ Kripke\ Structure\ Transformation$ process takes a use case as an input and generates a Kripke structure. Whereas, $Use\ Case\ to\ LTL\ Formula\ Transformation$ process generates LTL formulas form the input use case. The $UseCaseFlattener$ process flattens the input use case by handling the included and/or extended use cases and the flattened use case is provided as an input to these processes. A flattened use case description consists of actors, input symbols, output symbols with their possible values and scenario lines. The use case flattened process merges input set, output set and scenario lines of input use cases. Its time complexity is always smaller than the time complexity of other two process and thus this is not discussed here. These are represented by following variables

- ac denotes the cardinality of an $ActorSet$
- ip denotes the cardinality of an $InputSet$
- $opCount$ denotes the number of $OutputSets$
- os_i denotes the cardinality of an $OutputSet$, where $i = \{1,2,\ldots,opCount\}$

- ℓ denotes the number of scenario lines in *Scenario*

Time complexity of other two processes are discussed in the following subsections.

Use Case to Kripke Structure Transformation Process

This process consists of a number of rules and the time complexity of each rule is as follows:

Rule 1 contains two main loops starting on line 2 and line 10. The loop on line 2 starts execution with an index i of value equals 1 to a maximum value equals to $opCount$. This loop has a nested loop on line 5. This nested loop executes for os_i times, where i referred to the $OutputSet$ being referred by the outer loop. The worst case for loop on line 2 is $opCount$ * max (os_i). The loop on line 10 executes for ip times. The time complexity of this rule depends on the value of ip or $opCount$ * max(os_i), whichever is greater. The time complexity of this rule is $\mathcal{O}$ (max (ip, ($opCount$ * max (os_i))).

Rule 2 has a loop on line 2. This loop iterates for the size of $BitLabel_{temp}$ and it is denoted by $BitLabelSize$. The time complexity of Rule 2 is $\mathcal{O}$ ($BitLabelSize$).

Rule 3 has a loop on line 3 and this loop iterates for ℓ times. This loop has 3 nested loops on line 2, 7 and 12. The nested loop on line 2 iterates for ip times and the loop on line 7 iterates for ac times. The loop on line 12 iterates for a maximum to the value of $opCount$ and this loop has an inner loop which iterates for os_i times, where i referred to the $OutputSet$ being referred by the outer loop. The worst case for loop on line 12 is $opCount$ * max (os_i) times. The value of ac is equal to one and it can be ignored in the calculation of time complexity. The time complexity of this rule depends on the value of ip or $opCount$ * max(os_i), whichever is greater. The time complexity of this rule is $\mathcal{O}$ (max (ip, ($opCount$ * max (os_i))).

Rule 4 has a loop on line 12 and it is executed for the size of $ip - 1$. The time complexity of it is $\mathcal{O}$ (ip).

Rule 5 has a loop on line 8 and it is executed for $ip - 1$ times. The time complexity of it is $\mathcal{O}$ (ip).

Rule 6, Rule 7, Rule 8 and Rule 9 have a constant execution time, i.e., k.

In the transformation process, Rule 1, Rule 2 and Rule 3 execute one time and Rule 4 execute equal to the number of lines having an actor and an input symbol. It is denoted by ℓ_{acip}. However, Rule 5 executes for the number of *Alternate_Scenario* and it is denoted by $\ell_{alternatescenario}$. Rule 6 executes for the number of *Continue* lines and it is denoted by $\ell_{continue}$. Rule 7 executes for the number of $End\_of\_Alter-nateScenario$ lines and is denoted by $\ell_{endofalternatescenario}$. Rule 8 and Rule 9 execute for the number of $ExtnesionPoint$ and $EndExtensionPoint$ lines, respectively. These are denoted by $\ell_{extensionpoint}$ and $\ell_{endextensionpoint}$, respectively. So, the time complexity of $Use\ Case\ to\ Kripke\ Structure\ Transformation$ process is:

$\mathcal{O}$ ((max $(ip\ ,\ (opCount\ *\ \max\ (os_i))\ ))\ +\ (BitLabelSize)\ +\ (\ell\ *\ \max(\ ip,\ (opCount\ *\ \max\ (os_i))\ ))\ +\ (\ell_{acip}\ *\ ip\ )\ +\ (\ell_{alternatescenario}\ *\ ip))\ +\ ((\ell_{continue}\ +\ \ell_{endofalternatescenario}\ +\ \ell_{extensionpoint}\ +\ \ell_{endextensionpoint})\ *\ k\ )$.

The smaller terms are ignored and the time complexity of this process becomes

$\mathcal{O}$ $((\ell\ *\ \max(\ ip,\ (opCount\ *\ \max\ (os_i))\ ))\ +\ (\ell_{acip}\ *\ ip\ )\ +\ (\ell_{alternatescenario}\ *\ ip))\ +\ ((\ell_{continue}\ +\ \ell_{endofalternatescenario}\ +\ \ell_{extensionpoint}\ +\ \ell_{endextensionpoint})\ *\ k\ )$.

The $\ell_{continue} + \ell_{endofalternatescenario} = \ell_{alternatescenario}$. The ℓ_{acip}, $\ell_{alternatescenario}$, $\ell_{extensionpoint}$, $\ell_{endextensionpoint} \subset \ell$. So the $\mathcal{O}$ is

$\mathcal{O}$ $(\ell\ *\ \max(\ ip,\ (opCount\ *\ \max\ (os_i))\ ))$.

Use Case to LTL Formula Transformation Process

This transformation process consists of Rule 1 and Rule 2. These rules take a use case as input and generate the LTL formulas as output. Rule 1 has two loops starting on line 1 and on line 5. The loop on line 1 iterates for $opCount$ times.

The loop on line 5 executes for ℓ times and it has the inner loops on lines 6, line 12 and line 25. The inner loop on line 6 executes for ip times. The loop on line 12 executes for $opCount$ times. This loop has two nested loops on line 13 and on line 17. The nested loop on line 13 executes for os times of op being referred by the outer loop. The nested loop on line 17 executes for $opCount$ times. Whereas, the loop on line 25 executes for ac times. The value of time complexity of this rule is dependent on the iterations of loop on line 5 and its nested loop on line 12 because the term $opCount$ * $\max(os_i)$ * $opCount$ larger than ip. The loop on line 5 executes for ℓ times and loop on line 12 executes for $opCount$ * $\max(os_i)$ * $opCount$ times. The time complexity of this rule is $\mathcal{O}$ (ℓ * $opCount^2$ * $\max(os_i$)).

Rule 2 has a loop and this loop starts on line 1. This loop executes for ℓ times. This loop has the nested loops on line 2, line 14 and line 22. The loop on line 2 executes for ip times. The loop on line 14 executes for $opCount$ times and this loop has an inner loop on line 15 that executes for os times of op being referred by the outer loop. The nested loop on line 22 executes for ac times. The time complexity of this rules is dependent on the loop on line 1 and the nested loop on line 14. The loop on line 1 executes for ℓ times and the loop on line 14 executes for $opCount$ * $\max(os_i)$ times. The time complexity of this rule is denoted by $\mathcal{O}$ (ℓ * ($opCount$ * $\max(os_i)$)).

Use Case to LTL Formula Transformation process executes the Rule 1 and Rule 2 for one time. The time complexity of this process is $\mathcal{O}$ ((ℓ * $opCount^2$ * $\max(os_i)$) + (ℓ * $opCount$ * $\max(os_i)$)). The smaller terms are ignored and the time complexity of this process is $\mathcal{O}$ (ℓ * $opCount^2$ * $\max(os_i$).

Both the *Use Case to Kripke Structure Transformation* process and *Use Case to LTL Formula Transformation* process execute for one time. The time complexity of this approach is $\mathcal{O}$((ℓ * max (ip, ($opCount$ * max(os_i)))) + (ℓ * $opCount^2$ * $\max(os_i)$)). The smaller terms are ignored and the time complexity of this approach is $\mathcal{O}$ (ℓ * $opCount^2$ * max(os_i)) and it is quartic.

This approach generates both LTL formulas and behaviour model of a software. The time complexity of this process is quartic and is relatively large. But the generation of both behaviour model and LTL formulas make software verification possible at the early stage of requirements engineering.

4.6 Summary

The proposed approach transforms a use case into a Kripke structure and LTL formulas. This approach does not require any additional artefacts for the transformation. The transformation process is performed at meta model level. The generated formal artefacts are useful for software model verification and validation activities. This chapter presents meta models for input and output artefacts. In addition to it, *Use Case to Kripke Structure Transformation* and *Use Case to LTL Formula Fransformation* processes are presented. Time complexity along with soundness and completeness of these processes are also discussed in this chapter.

The discussed approach in this chapter generates both behavioural model and formal software specifications of a software. This objective is set in RQ2 and is achieved.

Chapter 5

Tool Support

This chapter discusses a tool, developed to implement the proposed approach. The developed tool is platform independent. This tool takes a use case in the proposed template and generates a Kripke structure in *.dot*, *.gml*, *.smv* and *.png* formats. In addition to these, LTL formulas are generated and stored in *.txt* format for further usage. The tool is available on Harvard Dataverse [110]. The developed metamodels and the transformation rules are implemented using Eclipe tools for Epsilon modelling framework [109]. Epsilon modelling framework does not provide the constructs to develop a GUI. A GUI tool is developed in Java language. This makes this tool a platform independent due to inherent features of Java language. Tool's architecture, class diagram along with other features are discussed in this chapter..

5.1 Architecture Diagram

The tool consists of two layers including $User\ Interface$ and *Application Logic* Layer. The architecture diagram of the tool is shown in Figure 5.1.

The $User\ Interface$ layer consists of a number of panels including *Path*, $Usecase$, *Progress Messages*, *Kripke structure*, *LTL Next Formulas* and *LTL Future*

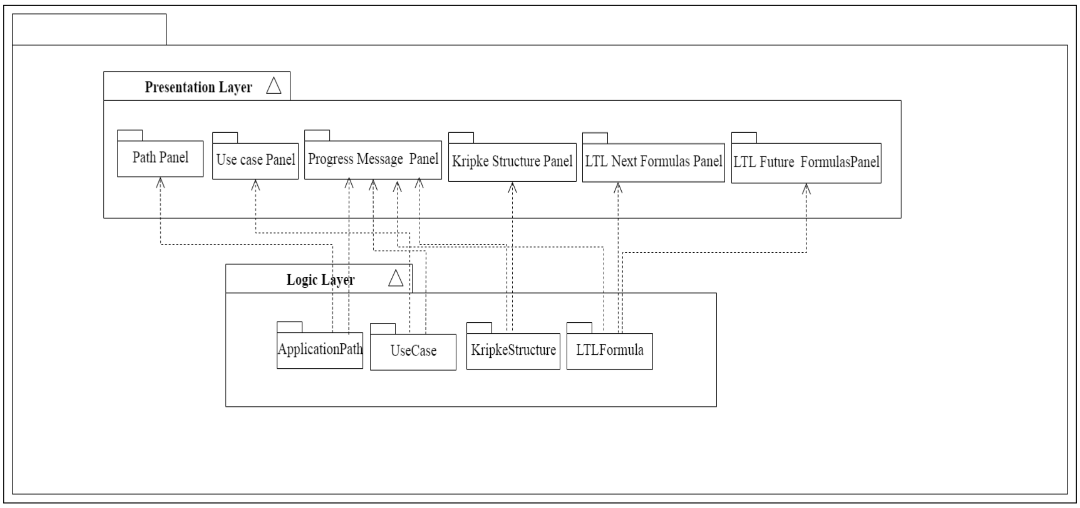


FIGURE 5.1: Tool Architecture Diagram.

Formulas panels. The user of the tool interacts with these constructs to process the provided use case.

Path panel is responsible to collect the paths of *GraphViz* API [111] and *use case*. The *GrpahViz* API is used to generate the output Kripke structure in *.gml* and *.png* formats from the generated *.dot* format. The user of the tool selects these paths by exploring the directory structure of the system. When a user selects a use case, the directory path of the selected use case is automatically fetched. It is required that the included or extended use cases with the main use case should by in th same directory. The generated outputs of the software are also stored by creating a subdirectory named *Output* in this directory.

The *Use case* panel displays the contents of the use case, provided as an input to the software. The *Progress* panel displays the messages about the status of the transformation along with the error messages, if generated. The progress messages include, the flattening use case, generating Kripke structure, generating LTL formulas and writing output files on the system. This panel also displays error messages for the user. The error messages include invalid *GraphViz* path, provided use case description does not comply with the use case template, unable to find included or extended use case(s). In addition to these, if some unexpected error occurs during the execution of the software, an error message is also generated for the user.

The generated Kripke structure and LTL formulas are displayed in *Kripke structure*, *LTL Next Formulas* and *LTL Future*4Formulas*panelsrespectively*.

The *Application layer* consists of *UserInterface*, *ApplicationPath* and *Usecase*, *Kripkestructure* and *LTLFormla*. The *UserInterface* is responsible to populate the user interface and also manages provided input and the generated output. *ApplicatoinPath* manages the provided paths by the user through *Path* panel. The selected use case is handled by *Usecase* to and the contents of the selected use case in *Use case* Panel and it is also used by *Kripke Structure Generator* and *LTL Formula Generator* for the production of a Kripke structure and LTL formulas.

The internal structure of the tool is discussed with the help of a class diagram. This class diagram is discussed in the following section.

5.2 Class Diagram

Tool's internal structure is shown, as a class diagram, in Figure 5.2. The *Interface Handler* is the main construct and it contains an *Application Path Handler*, a *Use case*, a *Use case flattened*, a *Kripke structure* and multiple *LTL Formula*s. Whereas, *Kripke structure* consists of multiple *state* and *Transition* constructs.

The *UserInterface*'s responsibilities include *PathValuesCollector*, *usecaseLoader*, *usecasVerifier*, *usecaseFlattener*, *kripkestructureGenerator*, *ltlNextSpecificat−ionGenerator* and *ltlFutureSpecificationGenerator*.

The *PathValuesCollector* function is responsible to record the selected paths of *GraphViz* API and use case directory. These paths are used by *Application Path Handler*'s *graphvizPath* and *usecaseDirectory*'s constructs respectively. This function is also responsible to verify either the provided *GrpahViz* API path is valid or invalid. In case of the provided path is invalid, it generates an error message. The *usecaseLoader* function is responsible to store the use case contents to *Use case*'s *actor*, *inputSymbols*, *outputSymbols* and *scenario* constructs.

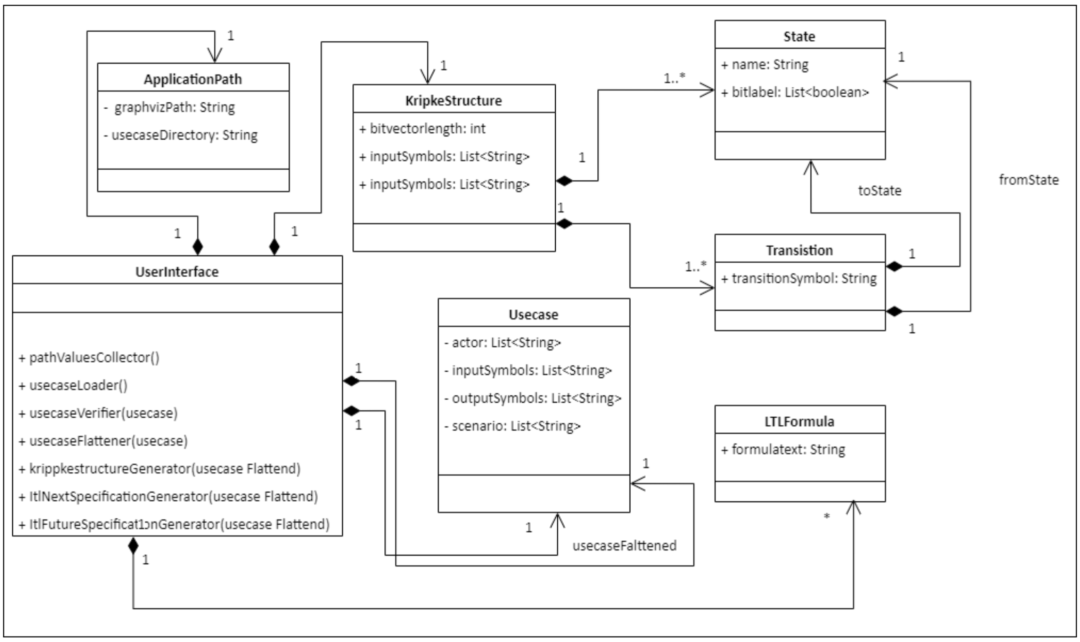


FIGURE 5.2: Tool Class Diagram.

An EBNF grammar is developed to verify the syntax of the input use case(s) and is discussed in Section 4.1. The compiler generator Coco/R [112] is used to generate Java files with the parsing abilities to parse the provided use case. These generated files are then customized to generate customize messages and are embedded in the developed tool as *usecaseVerifier* function.

The *usecaseFlattener* function flattens the provided use case by handling included and extended use cases. The included and extended use cases are required to be available in the provided use case directory. If any of the included or extended use case are not found in the provided use case directory, an error message is generated. The contents of resultant use case by *usecaseFlattener* is assigned to *use case flattened*'s *actor*, *inputSymbols*, *outputSymbols* and *scenario* constructs.

The *kripkestrucureGenerator* function is responsible to generate a Kripke structure from the provided *use case flattened*. This function is also responsible to write the generated Kripke structure in *.dot* and *.smv* formats. Both of these are stored on the system in the *OutputFiles* subdirectory. It is automatically created in the use case directory. In addition to it, this function also generates the Kripke structure in *.gml* and *.png* formats using *GraphViz* API. This function also generates error message, if any of the above operation do not execute successfully.

The $ltlNextSpecificationGenerator$ and $ltlFutureSpecificationGernerator$ functions are responsible to generate LTL formulas from the input use case. This function writes the generated LTL formulas in *.txt* files. In addition to it, these functions are also responsible to generate the error messages in case of any failure.

5.3 User Interface

A GUI based tool is developed to implement the proposed approach. The user interface consists of a number of graphical panels including *Path Info* panel, *Use case* panel, *Progress Message* panel, *Kripke structure* panel, *LTL Next Formulas* panel and *LTL Future Formulas* panel. The interface is shown in Figure 5.3. The interfaces panels are labelled with alpha characters.

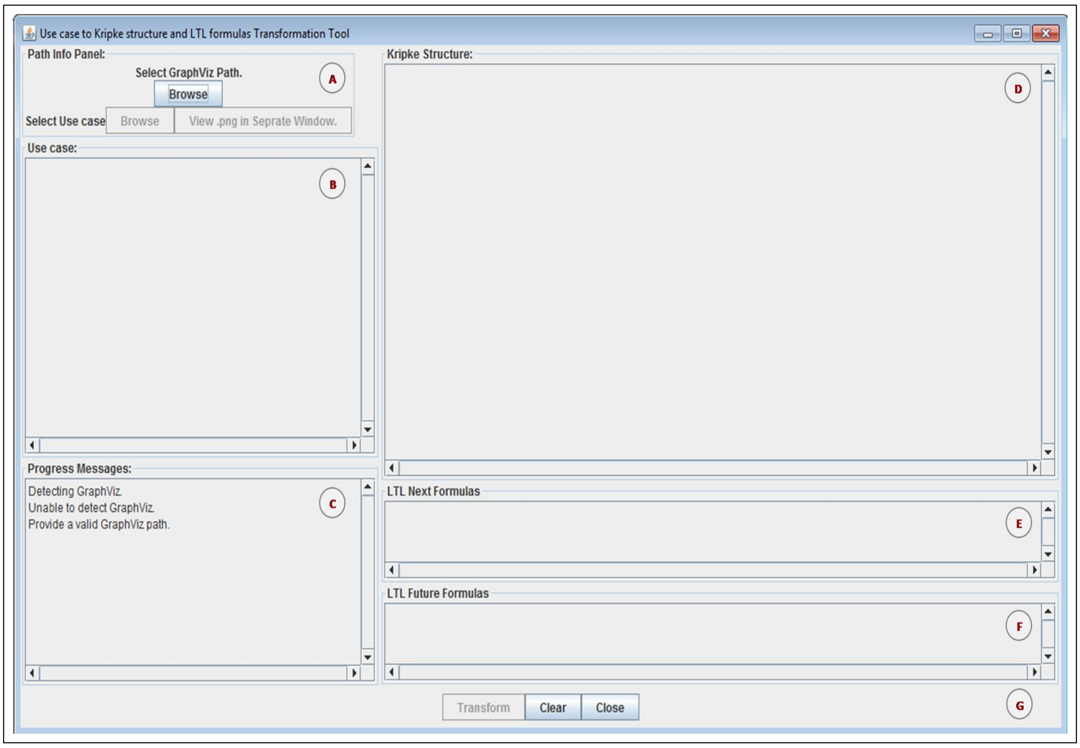


FIGURE 5.3: User Interface.

The user of the tool provides a *GraphViz* API path using *Browse* button as shown in Figure 5.4. The software verifies for the existence of *GraphViz* API on the provided path. It enables the *Browse* button on occurrence of *GraphViz*

API and the software displays a message in the *Progress Message* pane *GrphViz detected*.

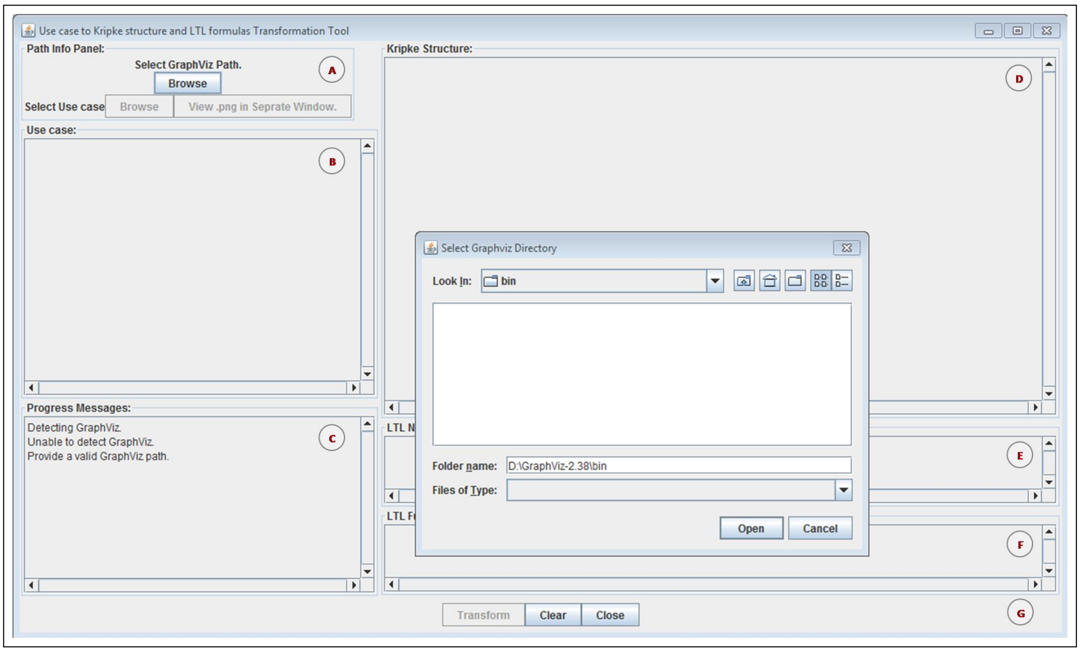


FIGURE 5.4: GraphViz Directory Selection Interface.

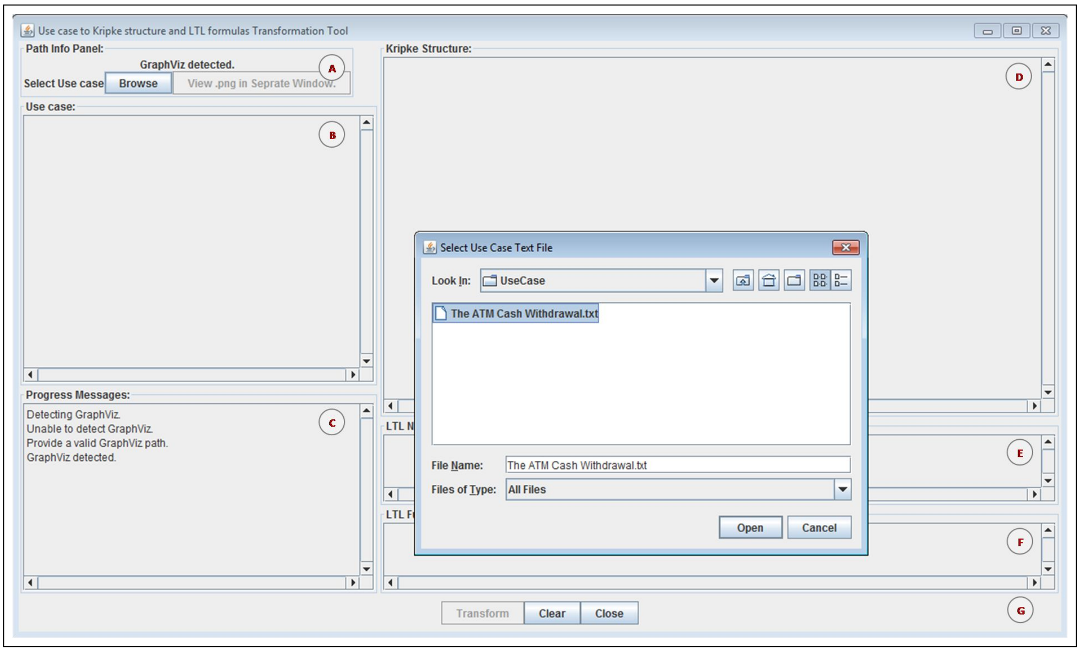


FIGURE 5.5: Use Case Selection Interface.

The user of the software selects a use case to be transformed into a Kripke structure and LTL formulas by clicking the *Browse* button as shown in Figure 5.5. When the user of the software selects a use case, the software checks for the existence of

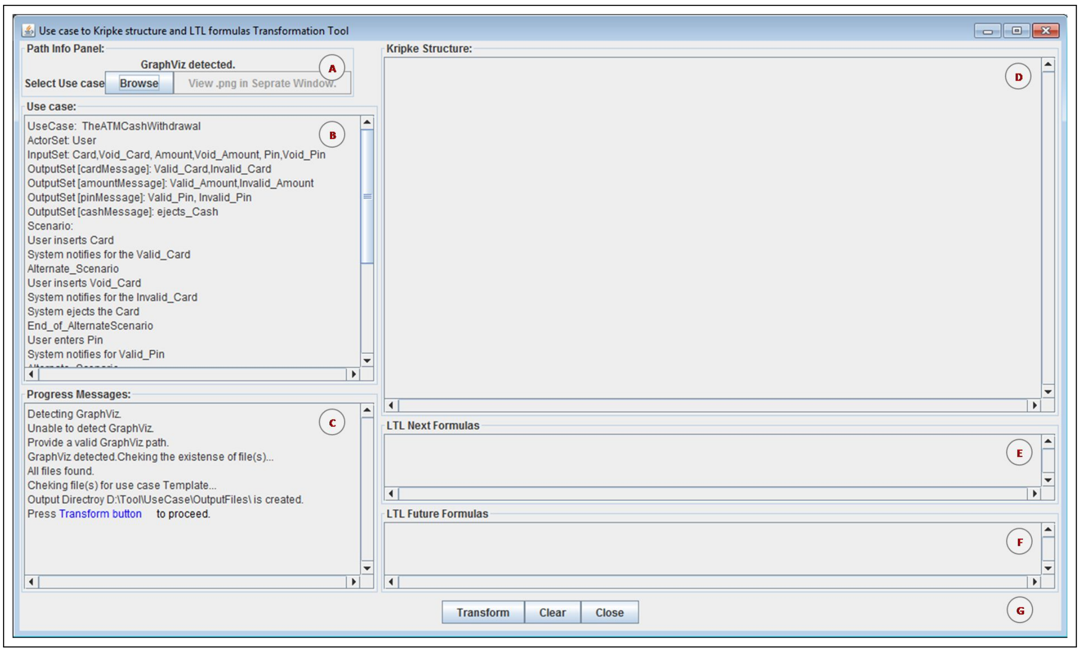


FIGURE 5.6: Syntax Correct Message on the Interface.

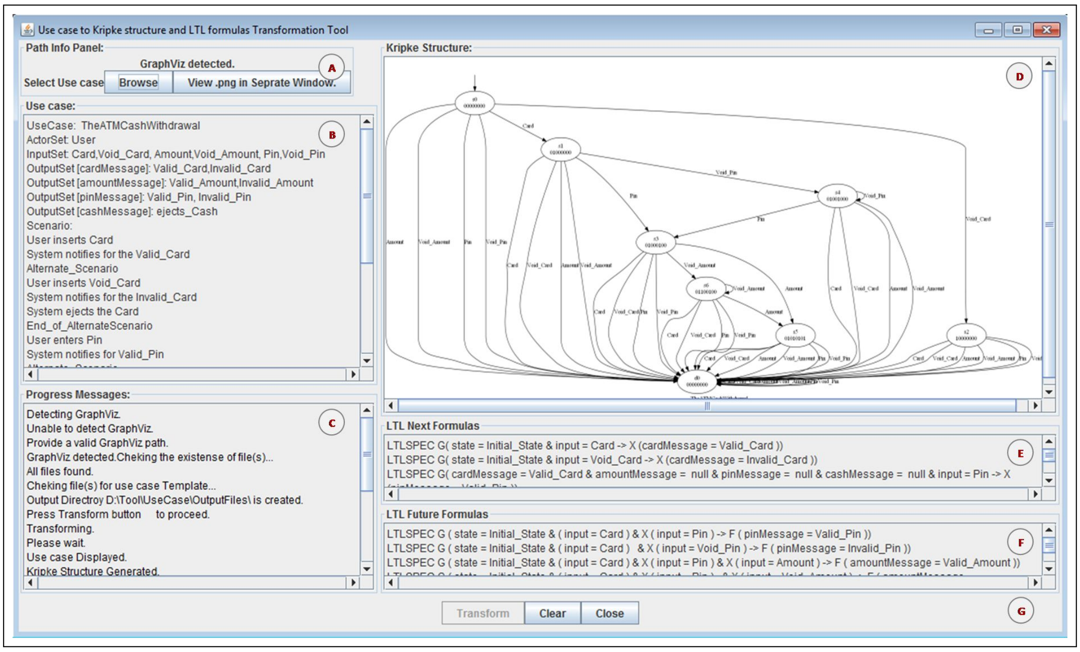


FIGURE 5.7: Generated Kripke structure and LTL formulas on the Interface.

included and extended use cases. The software displays a message *All files found* in the *Progress Message* pane. The software then checks the provided use cases against the proposed use case template. It creates a subdirectory *OutputFiles* in the directory from where the use case description is selected. The tool displays the contents of the selected use case in the *Use case* pane as shown in Figure 5.6. The tool, then, enables the *Transform* button. When, the user clicks this button, the software generates the resultant Kripke structure and LTL formulas and also displays the generated artefacts in *Kripke structure* pane, *LTL Next Formula* pane and *LTL Future Formula* pane as shown in Figure 5.7. The tool also saves the generated Kripke structure in *.dot*, *.gml*, *.smv* and *.png* formats in the *OutputFiles* subdirectory. In addition to these, the software also writes the generated LTL formulas in *.txt* in the *OutputFiles* subdirectory.

5.4 Summary

A GUI based tool is developed in Java language to implement the proposed approach. The inherit features make the developed tool platform independent. This tool takes a use case as input and generates the corresponding Kripke structure and LTL formulas. The tool, also, checks the provided use case description against the proposed template before the transformation. The architecture diagram, class diagram and features of the developed software are discussed in this chapter.

Chapter 6

Case Studies and Results Validation

The developed tool discussed in Chapter 5 is used to formalize the use cases of two pedagogical examples and two industrial case studies. The pedagogical examples include *ATM Cash Withdrawal* example and *SIM Vending Machine* example. Whereas, the industrial case studies include *Work Flow Manager*, developed by Elixir technologies, the other case study is *Touch'D*, an android application developed by *BitSol Inc*. The use cases of these pedagogical examples and case studies are presented in the following subsections. Some of the generated artefacts are provided in appendices due to their large size.

6.1 ATM Cash Withdrawal Example

The user presents a card to the software. The software intimates about the validity of the card. If the card is not valid, the software ejects the card. If the card is valid, the software allows to enter a Personal Identification Number (PIN). The software verifies the validity of the provided PIN and inform the user. If the provided PIN is invalid, the software allows to re-enter a PIN. On provision of a valid PIN the software allows the user to enter the amount to be withdrawn. If the entered amount is valid, the software dispenses the amount along with the card. If the provided amount is invalid, the software allows to re-enter a valid amount. The use case diagram is shown in Figure 6.1 and the use case description in the proposed template is shown in Figure 6.2.

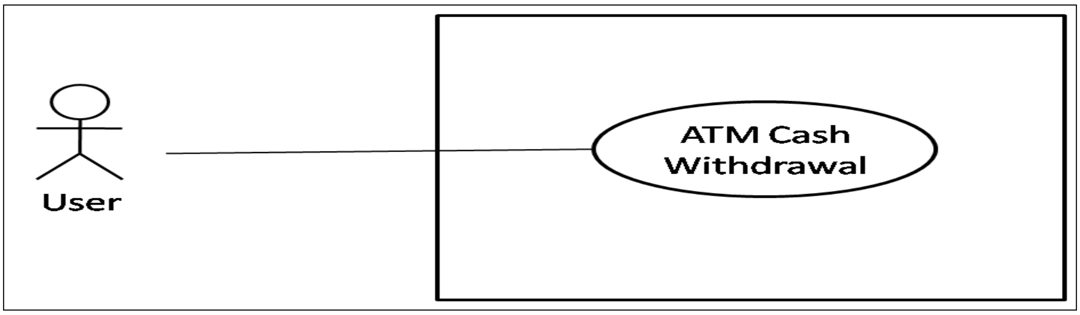


FIGURE 6.1: ATM Cash Withdrawal Use Case Diagram.

UseCase: The ATM Cash Withdrawal
ActorSet: User
InputSet: Card,Void_Card, Amount,Void_Amount, Pin,Void_Pin
OutputSet [cardMessage]: Valid_Card,Invalid_Card
OutputSet [amountMessage]: Valid_Amount,Invalid_Amount
OutputSet [pinMessage]: Valid_Pin, Invalid_Pin
OutputSet [cashMessage]: ejects_Cash
Scenario:
User inserts Card
System notifies for the Valid_Card
Alternate_Scenario
User inserts Void_Card
System notifies for the Invalid_Card
System ejects the Card
End_of_AlternateScenario
User enters Pin
System notifies for Valid_Pin
Alternate_Scenario
User enters Void_Pin
System notifies for Invalid_Pin
Continue
User enters Amount
System notifies for Valid_Amount
Alternate_Scenario
User enters Void_Amount
System notifies for Invalid_Amount
Continue
System ejects_Cash
End_of_Usecase

FIGURE 6.2: ATM Cash Withdrawal Use Case Description.

6.2 SIM Dispensing Machine Example

The user of the software provides a Computerized National Identification Number (CNIC) and the software verifies the provided CNIC. The software intimates the user about the validity of the provided CNIC. If the provided CNIC is invalid, the software ends the transaction. The user can check the number of registered Subscriber Identification Module (SIM) against the valid CNIC. The user of the software can also purchase a new SIM, modify the SIM plan and can check the balance history. The use case diagram of this is shown in Figure 6.3.

The use case descriptions of *Start A Transaction* is shown in Figure 6.4. The use case *Check SIM Registered* includes *Start A Transaction* use case and its use case description is shown in Figure 6.5.

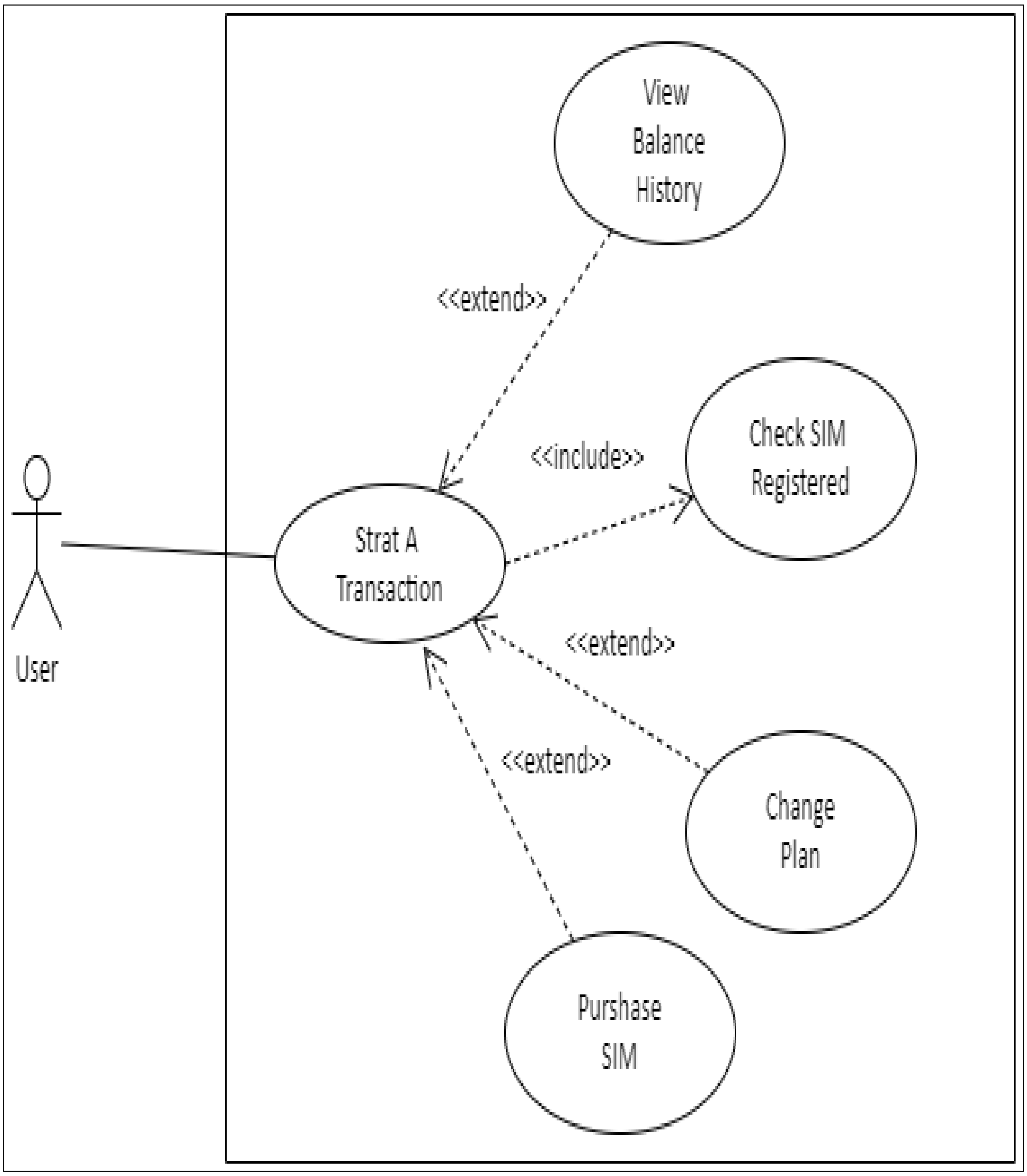


FIGURE 6.3: SIM Vending Machine Use Case Diagram.

UseCase: Start A Transaction
ActorSet: User
InputSet: valid_CNIC, invalid_CNIC, purchase_SIM_option, balance_check_option, change_plan_option
OutputSet [cardMessage]: valid_card_message, invalid_card_message
Scenario:
User enters valid_CNIC
System displays valid_card_message
Alternate_Scenario
User enters invalid_CNIC
System displays invalid_card_message
End_of_AlternateScenario
Include Check SIM Registered
Extend Purchase SIM **Condition** User selects purchase_SIM_option
Extend Balance History **Condition** User selects balance_check_option
Extend Change Plan **Condition** User selects change_plan_option
End_of_Usecase

Figure 6.4: Start a Transaction Use Case Description.

UseCase: Check SIM Registered
ActorSet: User
InputSet: number_of_registered_SIMs
OutputSet [checkSIMMessage]: list_of_registered_SIMs
Scenario:
User requests for the number_of_registered_SIMs
System displays the list_of_registered_SIMs
End_of_Usecase

Figure 6.5: Check SIM Registered Use Case Description.

The use case *Purchase SIM* is extended by *Start A Transaction* use case and its use case description is shown in Figure 6.6.

UseCase: Purchase SIM
ActorSet: User
InputSet: SIM_type_option, valid_amount, invalid_amount, valid_thumb_impression, invalid_thumb_impression, dispense_SIM
OutputSet [purchaseSIMOptionMessage]: available_option
OutputSet [amountMessage]: deposit_amount_message, invalid_amount_message
OutputSet [thumbMessage]: thumb_place_message, valid_thumb_message, invalid_thumb_message
OutputSet [issueSIMMessage]: issues_SIM
Scenario:
System displays available_option
User selects SIM_type_option
System displays deposit_amount_message
User enters valid_amount
System displays thumb_place_message
Alternate_Scenario
User enters invalid_amount
System displays invalid_amount_message
End_of_AlternateScenario
User scans valid_thumb_impression
System displays valid_thumb_message
Alternate_Scenario
User scans invalid_thumb_impression
System displays invalid_thumb_message
Continue
User asks to dispense_SIM
System issues_SIM
End_of_Usecase

Figure 6.6: Purchase SIM Use Case Description.

UseCase: View Balance History
ActorSet: User
InputSet: valid_mobile_number,invalid_mobile_number
OutputSet [balanceHistoryMessage]: check_balance, balance_history, invalid_mobile_message
Scenario:
System asks for mobile number to check_balance
User enters valid_mobile_number
System displays balance_history
Alternate_Scenario
User enters invalid_mobile_number
System displays invalid_mobile_message
End_of_AlternateScenario
End_of_Usecase

FIGURE 6.7: View Balance History Use Case Description.

UseCase: Change Plan
ActorSet: User
InputSet: SIM_plan_type
OutputSet [changePlanMessage]: SIM_current_plan, SIM_plan_option, SIM_plan_type_change_message
Scenario:
System displays the SIM_current_plan message
System displays available SIM_plan_option
User selects a SIM_plan_type
System displays SIM_plan_type_change_message
End_of_Usecase

FIGURE 6.8: Change Plan Use Case Description.

The use case *View Balance History* is extended by *Start A Transaction* use case and its use case description is shown in Figure 6.7. The use case *Change Plan* is extended by *Start A Transaction* use case and its use case description is shown in Figure 6.8.

6.3 Work Flow Manager Case Study

Work Flow Manager is a customized application application developed by Elixir Technologies. This application allows an organization to keep track of interaction between its employees and clients. The application follows a sequence of operations. It starts with a request from an employee, called requester in this application. Requester's request is viewed by Business Analyst (BA). BA can ask for an opinion from Subject Matter Expert (SME). BA can approve or reject this interaction request. Once BA approves a request it is forwarded to the client. The application functionalities are described in the form of user stories. The use case diagrams and along with use case descriptions are listed in the following subsection.

6.3.1 Request New Letter

A requester initiates a new request. The requester is required to be logged-in before the request initiation. The use case diagram is shown in Figure 6.9.

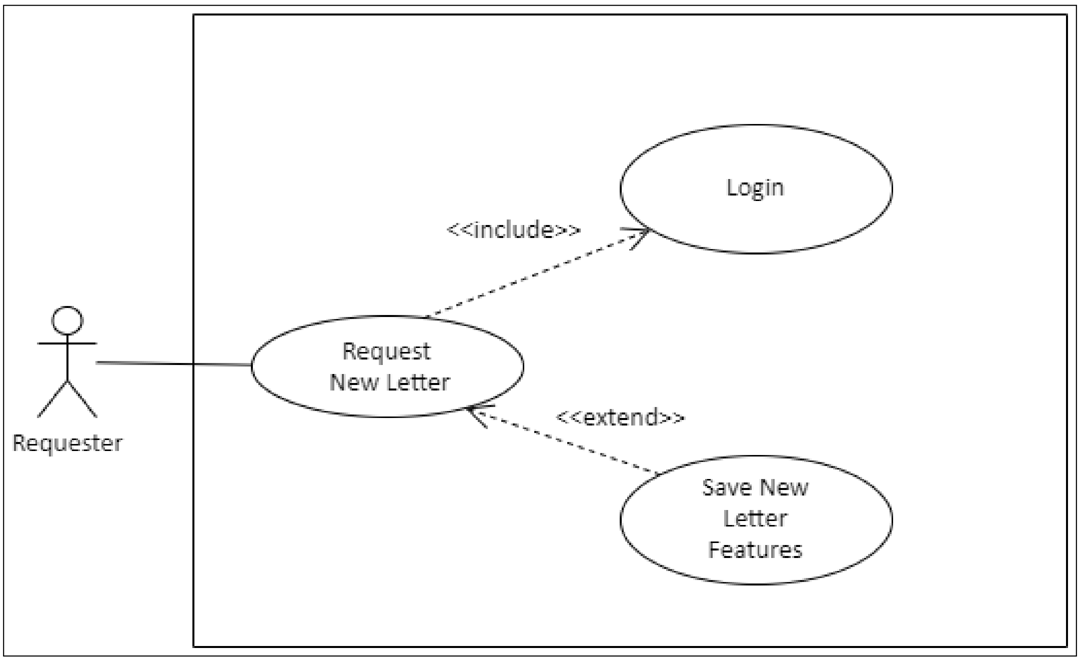


FIGURE 6.9: Request New Letter Use Case Diagram.

The use case description of *Requester Request New Letter* is shown in Figure 6.10.

UseCase: Requester Request New Letter
ActorSet: Requester
InputSet: new_letter, complete_features, save_letter_features, incomplete_features
OutputSet [feature_notification]: new_features_page
OutputSet [letter_status]: registers_letter
OutputSet [analyst_status]: buisness_analyst
OutputSet [error_message]: error_feature_message
Scenario:
Requester initiates new_letter request
System displays new_features_page
Include Login
Extend Save New Letter Features **Condition** Requester selects save_letter_features
Requester provides complete_features of letter
System registers_letter
System assigns buisness_analyst
End_of_Usecase

FIGURE 6.10: Request New Letter Use Case Description.

Login use case included by *Develop New Letter* use case. Its use case description is shown in Figure 6.11.

UseCase: Login
ActorSet: Requester
InputSet: valid_credentials, Invalid_credentials
OutputSet [screen_notification]: main_screen, error_login_message
Scenario: Requester provides valid_credentials to login
System displays main_screen
Alternate_Scenario
Requester provides Invalid_credentials to login
System displays error_login_message
End_of_AlternateScenario
End_of_Usecase

FIGURE 6.11: Login Use Case Description.

Save Letter Info extended by *Request New Review Letter Request* use case. Its use case description is shown in Figure 6.12.

UseCase: Save New Letter Features
ActorSet: Requester
InputSet: no
OutputSet [save_notification_message]: letter_features_save_message
Scenario: System displays letter_features_save_message
End_of_Usecase

FIGURE 6.12: Save New Letter Features Use Case Description.

6.3.2 Business Analyst Review Letter Request

A business analyst reviews the request initiated by a requester. A business analyst is required to be logged-in and cam call a meeting . A business analyst can view requester information, add information or save letter information. The use case diagram of this is shown in Figure 6.13.

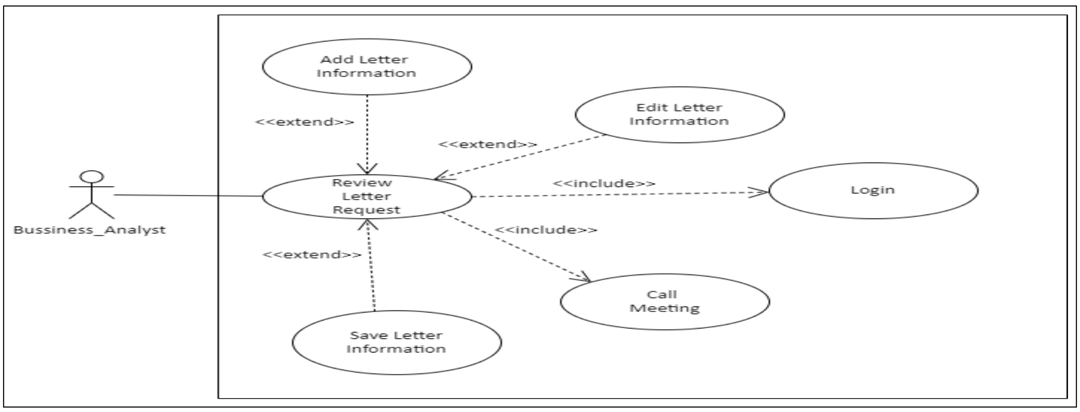


FIGURE 6.13: Business Analyst Review Letter Request Use Case Diagram.

The use case description of *Business Analyst Review Letter Request*, in the proposed template, is shown in Figure 6.14. The *Login* use case included by *Business Analyst Review Letter Request* use case. Its use case description is shown in Figure 6.15.

UseCase: Business Analyst Review Letter Request
ActorSet: Business_Analyst
InputSet: selects_letter, edit_option, add_info_option, save_info_option
OutputSet [letterInfoMessage]: letter_information
Scenario:
Business_Analyst selects_letter
System display letter_information
Include Analyst Login
Extend Edit Letter Information **Condition** Business_Analyst selects edit_option
Extend Add Letter Information **Condition** Business_Analyst selects add_info_option
Extend Save Letter Information **Condition** Business_Analyst selects save_info_option
Include Call Meeting
End_of_Usecase

FIGURE 6.14: Business Analyst Review Letter Request Use Case Description.

UseCase: Analyst Login
ActorSet: Business_Analyst
InputSet: valid_credentials, Invalid_credentials
OutputSet [loginMessage]: main_screen, error_login_message
Scenario:
Business_Analyst provides valid_credentials to login
System displays main_screen
Alternate_Scenario
Business_Analyst provides Invalid_credentials to login
System displays error_login_message
End_of_AlternateScenario
End_of_Usecase

FIGURE 6.15: Login Use Case Description.

The *Call Meeting* use case included by *Business Analyst Review Letter Request* use case. Its use case description is shown in Figure 6.16.

UseCase: Call Meeting
ActorSet: Business_Analyst
InputSet: meeting_details
OutputSet [meetingMessage]: meeting_confirmation
Scenario:
Business_Analyst provides meeting_details
System displays meeting_confirmation message
End_of_Usecase

FIGURE 6.16: Call Meeting Use Case Description.

The *Add Letter Information* use case is extended by *Business Analyst Review Letter Request* use case. Its use case description is shown in Figure 6.17.

UseCase: Add Letter Information
ActorSet: Business_Analyst
InputSet: additional_letter_information
OutputSet [addLetterInfoMessage]: additional_info
OutputSet [infoMessage]: save_info
Scenario:
System displays additional_info box
Business_Analyst adds additional_letter_information
System displays save_info message
End_of_Usecase

FIGURE 6.17: Add Letter Information Use Case Description.

The *Edit Letter Information* use case is extended by *Business Analyst Review Letter Request* use case. Its use case description is shown in Figure 6.18.

UseCase: Edit Letter Information
ActorSet: Business_Analyst
InputSet: edited_information
OutputSet [editLetterMessage]: edited_letter_information
OutputSet [updateLetterMessage]: updated_letter
Scenario:
System displays edited_letter_information
Business_Analyst provides edited_information
System displays updated_letter message
End_of_Usecase

FIGURE 6.18: Edit Letter Information Use Case Description.

The *Save Letter Information* use case is extended by *Business Analyst Review Letter request* use case. Its use case description is shown in Figure 6.19.

UseCase: Save Letter Information
ActorSet: Business_Analyst
InputSet: no
OutputSet [saveLetterMessage]: letter_information_save_message
Scenario:
System displays letter_information_save_message
End_of_Usecase

FIGURE 6.19: Save Letter Information Use Case Description.

6.3.3 Letter Writer Develops New Letter

A letter writer on the request of the requester develops a new letter, The letter writer requires to be logged-in for the new letter development. The letter writer can view requester and business analyst information. Moreover, letter writer can add information. The use case diagram of this is shown in Figure 6.20.

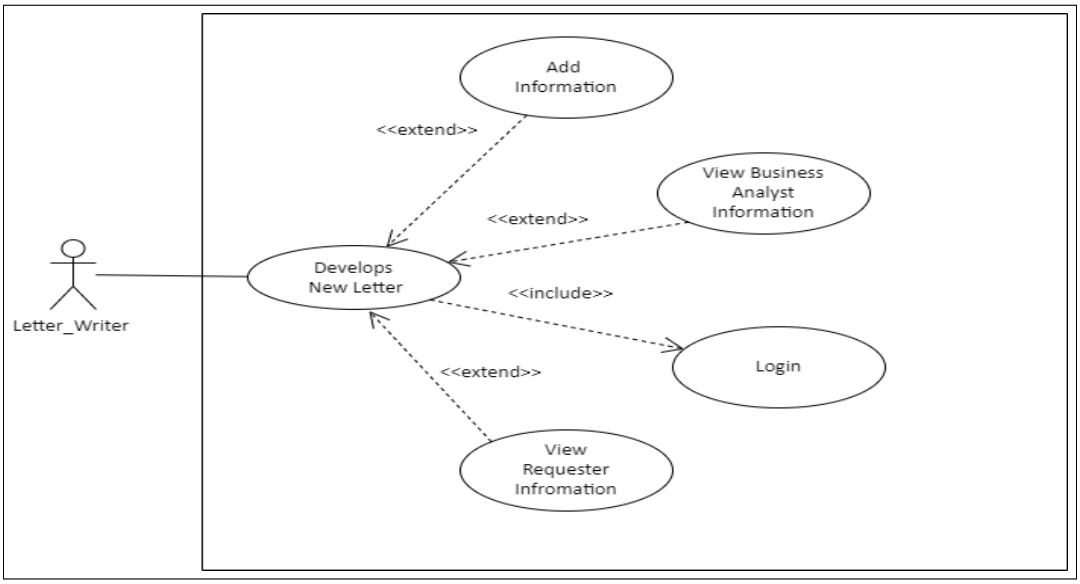


FIGURE 6.20: Letter Writer Develops a New Letter Use Case Diagram.

UseCase: Letter Writer Login **ActorSet**: Letter_Writer **InputSet**: valid_credentials, Invalid_credentials **OutputSet**[screen_notification]: main_screen, error_login_message **Scenario**: Letter_Writer provides valid_credentials to login System displays main_screen **Alternate_Scenario** Letter_Writer provides Invalid_credentials to login System displays error_login_message **End_of_AlternateScenario** **End_of_Usecase**

FIGURE 6.21: Login Use Case Description.

The *Login* use case is included by *Develop New Letter* use case. Its use case description is shown in Figure 6.21. The use case description of *Letter Writer Develops New Letter*, in the proposed template, is shown in Figure 6.22.

UseCase: Letter Writer Develops New Letter **ActorSet**: Letter_Writer **InputSet**: view_requester_info, view_bus_analyst_info, add_info, generates_letter, letter_to_stakeholders **OutputSet** [letter_completion]: final_letter **OutputSet** [stakeholder_status]: submitted_stakeholders **Scenario**: Letter_Writer generates_letter System generate final_letter **Include** Letter Writer Login **Extend** View Requester Information **Condition** Letter_Writer select view_requester_info **Extend** View Buisness Analyst Information **Condition** Letter_Writer select view_bus_analyst_info **Extend** Add Information **Condition** Letter_Writer select add_info Letter_Writer submits letter_to_stakeholders System displays submitted_stakeholders list **End_of_Usecase**

FIGURE 6.22: Letter Writer Develops a New Letter Use Case Description.

The *Add Information* use case is extended by *Develops New Letter* use case. Its use case description is shown in Figure 6.23.

UseCase: Add Information
ActorSet: Letter_Writer
InputSet: letter_writer_provided_information
OutputSet[letter_status]: letter_information
OutputSet[information_status]: information_added
Scenario:
System asks to add letter_information
Letter_Writer provides letter_writer_provided_information
System displays information_added message
End_of_Usecase

FIGURE 6.23: Add Information Use Case Description.

The *View Requester Information* use case is extended by *Develops New Letter* use case. Its use case description is shown in Figure 6.24.

UseCase: View Requester Information
ActorSet: Letter_Writer
InputSet: noinput
OutputSet [information_provide_status]: requester_provided_information
Scenario:
System displays requester_provided_information
End_of_Usecase

FIGURE 6.24: View Requester Information Use Case Description.

The *View Business Analyst Information* use case is extended by *Develop New Letter* use case. Its use case description is shown in Figure 6.25.

UseCase: View Buisness Analyst Information
ActorSet: Letter_Writer
InputSet: no
OutputSet[business_analyst_status]: bus_analyst_provided_information
Scenario:
System displays bus_analyst_provided_information
End_of_Usecase

FIGURE 6.25: View Business Analyst Information Use Case Description.

6.3.4 Business Analyst Review Developed Letter

Once the letter writer develops the letter, the business analyst can review the developed letter. The business analyst required to be logged-in to review the developed letter. The business analyst can approve or reject the developed letter. The use case diagram of this interaction is shown in Figure 6.26.

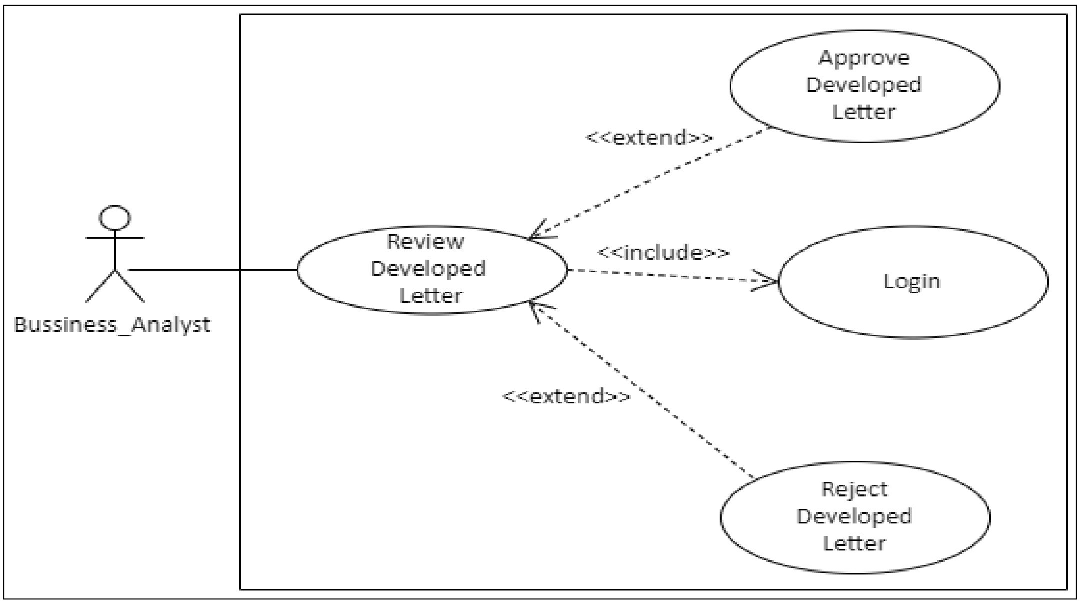


FIGURE 6.26: Business Analyst Review Developed Letter Use Case Diagram.

The use case description of *Letter Writer Develops New Letter*, in the proposed template, is shown in Figure 6.27.

UseCase: Buisness Analyst Review Developed Letter **ActorSet**: Business_Analyst **InputSet**: developed_letter_review, reject_option, approve_option **OutputSet** [letter_notification]: letter_information **Scenario**: Business_Analyst select a developed_letter_review System displays letter_information **Include** Business Analyst Login **Extend** Reject Developed Letter **Condition** Business_Analyst selects reject_option **Extend** Approve Developed Letter **Condition** Business_Analyst selects approve_option **End_of_Usecase**

FIGURE 6.27: Business Analyst Review Developed Letter Use Case Description.

The *Business Analyst Login* use case is included by *Business Analyst Review Developed Letter* use case. Its use case description is shown in Figure 6.28.

UseCase: Business Analyst Login **ActorSet**: Business_Analyst **InputSet**: valid_credentials, Invalid_credentials **OutputSet** [loginMessage]: main_screen, error_login_message **Scenario**: Business_Analyst provides valid_credentials to login System displays main_screen **Alternate_Scenario** Business_Analyst provides Invalid_credentials to login System displays error_login_message **End_of_AlternateScenario** **End_of_Usecase**

FIGURE 6.28: Login Use Case Description.

The *Approve Developed Letter* use case is extended by *Business Analyst Review Developed Letter* use case. Its use case description is shown in Figure 6.29.

UseCase: Approve Developed Letter
ActorSet: Business_Analyst
InputSet: noinput
OutputSet[Approval_notification]: approval_message
Scenario:
System displays approval_message
End_of_Usecase

FIGURE 6.29: Approve Developed Letter Use Case Description.

The *Reject Developed Letter* use case is extended by *Business Analyst Review Developed Letter* use case. Its use case description is shown in Figure 6.30.

UseCase: Reject Developed Letter
ActorSet: Business_Analyst
InputSet: no
OutputSet[rejection_status]: rejection_message
Scenario:
System displays rejection_message
End_of_Usecase

FIGURE 6.30: Login Use Case Description.

6.3.5 Requester Approves Developed Letter

The requester can approve the developed letter. The requester is required to be logged-in. The requester can approve or reject the letter. The use case diagram of requester approves developed letter is shown in Figure 6.31.

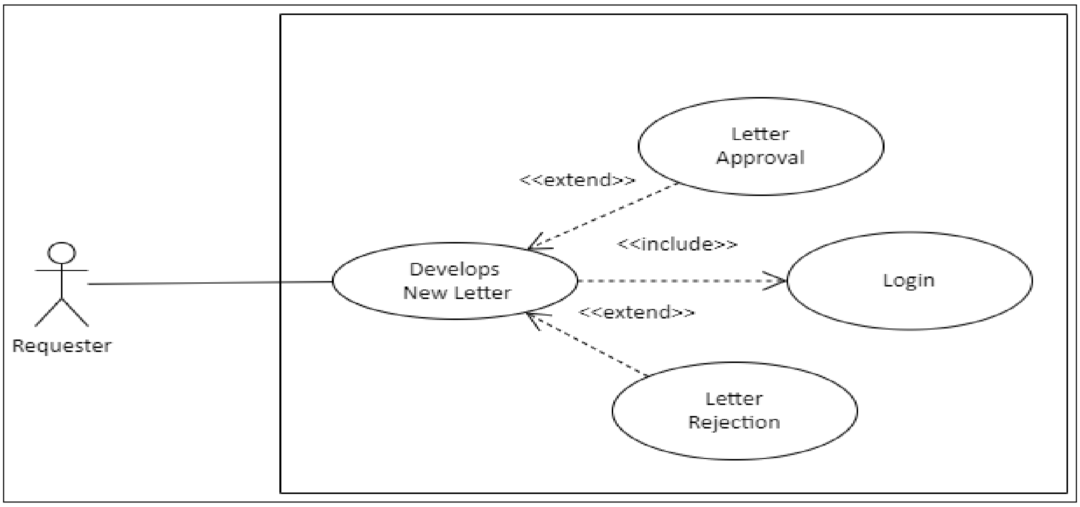


FIGURE 6.31: Requester Approves Developed Letter Use Case Diagram.

The use case description of *Requester Approves Developed Letter*, in the proposed template, is shown in Figure 6.32.

UseCase: Requester Approves Developed Letter
ActorSet: Requester
InputSet: request,rejects_request,accepts_request
OutputSet [save_notification]: save_message
Scenario:
Requester initiates request
Include Requester Login
Extend Letter Rejection **Condition** Requester rejects_request
Extend Letter Approval **Condition** Requester accepts_request
End_of_Usecase

FIGURE 6.32: Requester Approves Developed Letter Use Case Description.

The *Login* use case is included by *Requester Approves Developed Letter* use case. Its use case description is shown in Figure 6.33.

UseCase:Login
ActorSet: Requester
InputSet: valid_credentials, Invalid_credentials
OutputSet [screen_notification]: main_screen, error_login_message
Scenario:
Requester provides valid_credentials to login
System displays main_screen
Alternate_Scenario Requester provides Invalid_credentials to login
System displays error_login_message
End_of_AlternateScenario
End_of_Usecase

FIGURE 6.33: Login Use Case Description.

The *Letter Approval* use case is extended by *Requester Approves Developed Letter* use case. Its use case description is shown in Figure 6.34.

UseCase: Letter Approval
ActorSet: Requester
InputSet: no
OutputSet [approaval_notification]: approval_message
Scenario:
System displays approval_message
End_of_Usecase

FIGURE 6.34: Letter Approval Use Case Description.

UseCase: Letter Rejection
ActorSet: Requester
InputSet: suggestion
OutputSet [rejection_status]: rejection_comments
Scenario:
System asks to enter rejection_comments
Requester adds suggestion
System displays save_message
End_of_Usecase

FIGURE 6.35: Letter Rejection Use Case Description.

The *Letter Rejection* use case is extended by *Requester Approves Developed Letter* use case. Its use case description is shown in Figure 6.35.

6.4 Touch'D Case Study

Touch'D is an android application, it allows a user to create a profile, view a contact and call a contact. This application keeps a user updated regarding the contacts in the contact list. It provides different summaries including the number of interactions in a week, contact health. In addition to these, this application also allows a user to record short notes about a contact. There are three kinds of interactions broadly provided to the user. The interactions include *User Makes a Profile*, *User Views a Contact* and *User Calls a Contact*. The use case diagrams along with the use case descriptions of these interactions are provided in the following subsections.

6.4.1 User Makes a Profile

The *User Makes a Profile*, allows a user to create a profile by setting the personal information and the privacy level. In addition to it, this interaction allows the user to view recently contacted, view today and last week interaction summary. The use case diagram of *User Makes a Profile* is shown in Figure 6.36.

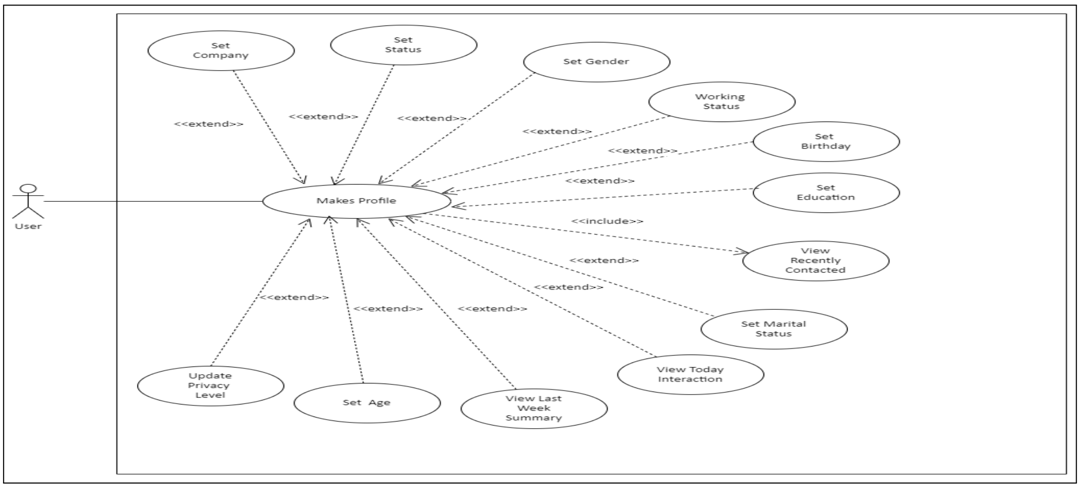


FIGURE 6.36: User Makes A Profile Use Case diagram.

UseCase: User Makes a Profile
ActorSet: User
InputSet: makeProfileOption, setAge, setMarital, setCompanyName, setWorkingStatus, setBirthday, setEducation, setGender, updatePrivacy, setStatus,viewweekSummary, viewTodayInteraction
OutputSet[profile_status]: profileOptions
Scenario:
User selects makeProfileOption
System displays profileOptions
Extend Set Age **Condition** User select setAge option
Extend Set Marital Status **Condition** User selects setMarital option
Extend Set Company **Condition** User selects setCompanyName option
Extend Working Status **Condition** User selects setWorkingStatus option
Extend Set Birthday **Condition** User selects setBirthday option
Extend Set Education **Condition** User selects setEducation option
Extend Set Gender **Condition** User selects setGender option
Extend Update Privacy Level **Condition** User selects updatePrivacy option
Extend Set Status **Condition** User selects setStatus option
Extend View Last Week Summary **Condition** User selects viewweekSummary Option
Extend View Today Interaction Summary **Condition** User selects viewTodayInteraction Option
Include View Recentaly Contacted
End_of_Usecase

FIGURE 6.37: User Makes Profile Use Case Description.

The use case description of *User Makes a Profile*, in the proposed template, is shown in Figure 6.37.

The *View Recently Contacted* use cases is included by *User Makes a Profile*. The use case description of *View Recently Contacted*, in the proposed template, is shown in Figure 6.38.

UseCase: View Recentaly Contacted
ActorSet: User
InputSet: recentlyContactedData
OutputSet [recent_contact_status]: recentlyContactedValue
Scenario:
User selects display recentlyContactedData option
System displays recentlyContactedValue
End_of_Usecase

FIGURE 6.38: View Recently Contacted Use Case Description.

The *Set Age* use cases is extended by *User Makes a Profile*. The use case description of *Set Age*, in the proposed template, is shown in Figure 6.39.

UseCase: Set Age
ActorSet: User
InputSet: ageEnterOption, validAge
OutputSet [age_status]: updatedAge
Scenario:
System displays ageEnterOption
User provides validAge
System displays updatedAge
End_of_Usecase

FIGURE 6.39: Set Age Use Case Description.

The *Set Marital Status* use cases is extended by *User Makes a Profile*. The use case description of *Set Marital Status*, in the proposed template, is shown in Figure 6.40.

UseCase: Set Marital Status
ActorSet: User
InputSet: maritalType
OutputSet [marital_status]: maritalSelectOption
OutputSet [update_marital]: updatedMaritalValue
Scenario:
System displays maritalSelectOption
User selects maritalType
System displays updatedMaritalValue
End_of_Usecase

FIGURE 6.40: Set Marital Status Use Case Description.

The *Set Company* use cases is extended by *User Makes a Profile*. The use case description of *Set Company*, in the proposed template, is shown in Figure 6.41.

UseCase: Set Comapny
ActorSet: User
InputSet: companyTextBox, companyName
OutputSet [company_status]: updatedCompanyName
Scenario:
System displays companyTextBox
User provide companyName
System displays updatedCompanyName
End_of_Usecase

FIGURE 6.41: Set Company Use Case Description.

The *Set Working Status* use cases is extended by *User Makes a Profile*. The use case description of *Set Working Status*, in the proposed template, is shown in Figure 6.42.

UseCase: Set Working Status
ActorSet: User
InputSet: workingStatusData
OutputSet [working_Status]: workingStatusOption
Scenario:
System displays workingStatusOption
User selects workingStatusData
System displays updatedWorkingStatus
End_of_Usecase

FIGURE 6.42: Set Working Status Use Case Description.

The *Set Birthday* use cases is extended by *User Makes a Profile*. The use case description of *Set Birthday*, in the proposed template, is shown in Figure 6.43.

UseCase: Set Birthday
ActorSet: User
InputSet: birthdayData
OutputSet [birthday_stauts]: birthdayProvideOption
OutputSet [update_birth_status]: updatedBirthdayValue
Scenario:
System displays birthdayProvideOption
User provides birthdayData
System displays updatedBirthdayValue
End_of_Usecase

FIGURE 6.43: Set Birthday Use Case Description.

UseCase: Set Education
ActorSet: User
InputSet: educationTypeData
OutputSet [education_staus]: educationProvideOption
OutputSet [update_edu_status]: updatedEducationValue
Scenario:
System displays educationProvideOption
User selects educationTypeData
System displays updatedEducationValue
End_of_Usecase

FIGURE 6.44: Set Education Use Case Description.

The *Set Education* use cases is extended by *User Makes a Profile*. The use case description of *Set Education*, in the proposed template, is shown in Figure 6.44.

The *Set Gender* use cases is extended by *User Makes a Profile*. The use case description of *Set Gender*, in the proposed template, is shown in Figure 6.45.

UseCase: Set Education
ActorSet: User
InputSet: educationTypeData
OutputSet [education_staus]: educationProvideOption
OutputSet [update_edu_status]: updatedEducationValue
Scenario:
System displays educationProvideOption
User selects educationTypeData
System displays updatedEducationValue
End_of_Usecase

FIGURE 6.45: Set Gender Use Case Description.

UseCase: Update Privacy Level
ActorSet: User
InputSet: privacyTypeData
OutputSet [privacy_status]: privacyProvideOption
OutputSet [update_privacy_value]: updatedPrivacyValue
Scenario:
System displays privacyProvideOption
User selects privacyTypeData
System displays updatedPrivacyValue
End_of_Usecase

FIGURE 6.46: Update Privacy Level Use Case Description.

The *Update Privacy Level* use cases is extended by *User Makes a Profile*. The use case description of *Update Privacy Level*, in the proposed template, is shown in Figure 6.46.

The *Set Status* use cases is extended by *User Makes a Profile*. The use case description of *set Status*, in the proposed template, is shown in Figure 6.47.

UseCase: Set Status
ActorSet: User
InputSet: statusData
OutputSet [update_status]: statusProvideOption
OutputSet [update_status_value]: updatedStatusValue
Scenario:
System displays statusProvideOption
User selects statusData
System displays updatedStatusValue
End_of_Usecase

FIGURE 6.47: Set Status Use Case Description.

UseCase: View Last week Summary
ActorSet: User
InputSet: nosymbol
OutputSet [week_status]: lastweekData
Scenario:
System displays lastweekData
End_of_Usecase

FIGURE 6.48: View Last Week Summary Use Case Description.

UseCase: View Today Interaction Summary
ActorSet: User
InputSet: noinput
OutputSet [today_interact_status]: todayInteractionSummary
Scenario:
System displays todayInteractionSummary
End_of_Usecase

FIGURE 6.49: View Today Interaction Summary Use Case Description.

The *View Last Week Summary* use cases is extended by *User Makes a Profile*. The use case description of *View Last Week Summary*, in the proposed template, is shown in Figure 6.48.

The *View Today Interaction Summary* use cases is extended by *User Makes a Profile*. The use case description of *View Today Interaction Summary*, in the proposed template, is shown in Figure 6.49.

6.4.2 User Calls a Contact

The *User Calls a Contact* dials a contact number is the main use case and this use case include the *View a Note* use case. The use case diagram of *User Calls a Contact* is shown in Figure 6.50.

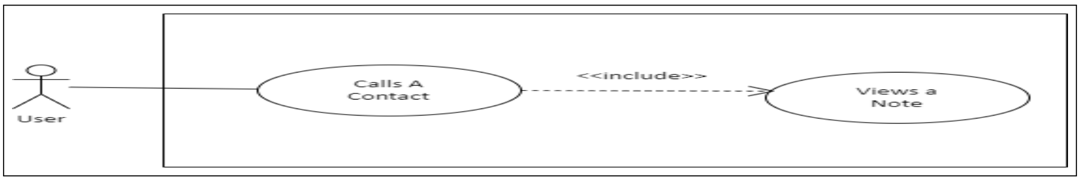


FIGURE 6.50: User Calls Contact Use Case Diagram.

The use case description of *User Calls a Contact*, in the proposed template, is shown in Figure 6.51.

UseCase: User Calls a Contact
ActorSet: User
InputSet: contactNumber, incorrectContactNumber
OutputSet [contact_notification]: dialingContactMessage, incorrectContactMessage
Scenario:
User provides a contactNumber to call
System displays dialingContactMessage
Alternate_Scenario
User provides incorrectContactNumber
System displays incorrectContactMessage
End_of_AlternateScenario
Include View a Note
End_of_Usecase

FIGURE 6.51: User Calls a Contact Use Case Description.

UseCase: View a Note
ActorSet: User
InputSet: Number
OutputSet [save_notification_status]: lastSavedNote
Scenario:
User dialing a Number
System displays lastSavedNote
End_of_Usecase

FIGURE 6.52: View a Note Use Case Description.

The *View a Note* use cases is included by *User Calls a Contact*. The use case description of *View a Note*, in the proposed template, is shown in Figure 6.52.

6.4.3 User Views a Contact

The *User Views a Contact* is the main use case and this use case include *View Last Interaction Type* and extend *View Communication Summary*, *View Communication Starter*, *View Relationship Health*, *Send Message* and *Add a Note* use cases. The use case diagram of *User Views a Contact* is shown in Figure 6.53.

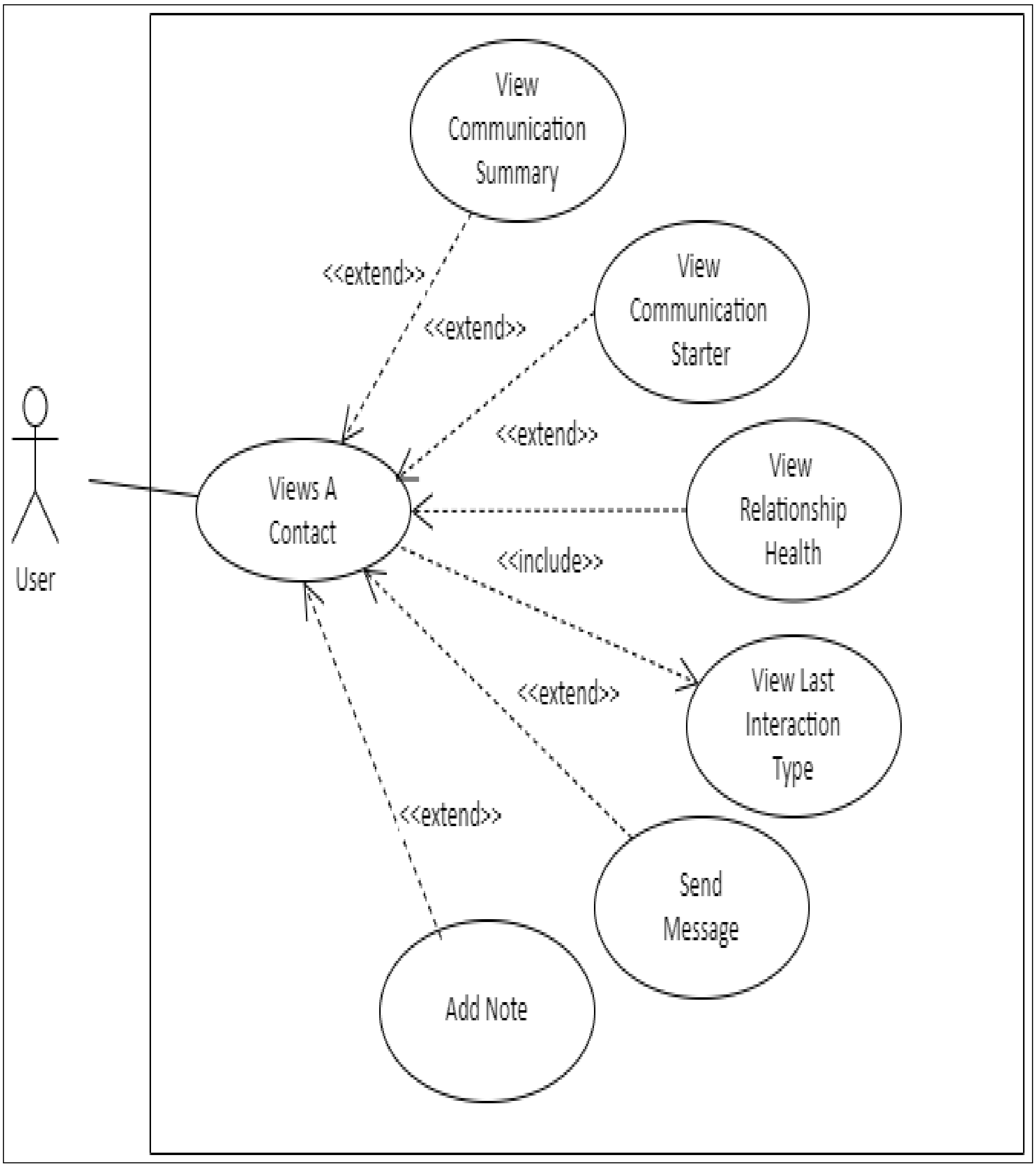


FIGURE 6.53: User Views a Contact Use Case Diagram.

The use case description of *User Views a Contact*, in the proposed template, is shown in Figure 6.54.

UseCase: User Views a Contact
ActorSet: User
InputSet: contact, addNewNote, viewCommunicationSummary, viewCommunicationStarterSummary, sendNewMessage, viewRelationshipHealth
OutputSet [contactMessage]: selectedContactDetails
Scenario:
User selects a contact
System displays selectedContactDetails
Extend Add Note **Condition** User selects addNewNote option
Extend View Communication Summary **Condition** User selects viewCommunicationSummary option
Extend View Communication Starter Summary **Condition** User selects viewCommunicationStarterSummary option
Extend Send Message **Condition** User selects sendNewMessage option
Extend View Relationship Health **Condition** User selects viewRelationshipHealth option
Include View Last Interaction Type
End_of_Usecase

FIGURE 6.54: User Views a Contact Use Case Description.

UseCase: View Last Interaction Type
ActorSet: User
InputSet: contacttoDisplay
OutputSet [lastInteractionTypeMessage]: lastInteractionTypeValue
Scenario:
User select a contacttoDisplay
System displays lastInteractionTypeValue
End_of_Usecase

FIGURE 6.55: View Last Interaction Type Use Case Description.

UseCase: Add a Note
ActorSet: User
InputSet: noteContents, saveNote, cancelNote
OutputSet [noteTextBoxMessage]: noteTextBox
OutputSet [saveNoteOptionMessage]: saveNoteOption
OutputSet [noteFinalMessage]: noteSaveMessage, noteCancelMessage
Scenario:
System displays noteTextBox
User types noteContents
System enables saveNoteOption
User selects saveNote
System displays noteSaveMessage
Alternate_Scenario
User selects cancelNote
System displays noteCancelMessage
End_of_AlternateScenario
End_of_Usecase

FIGURE 6.56: Add Note Use Case Description.

The *View Last Interaction Type* use case is included by *User Views a Contact*. The use case description of *View Last Interaction Type*, in the proposed template, is shown in Figure 6.55.

The *Add a Note* use case is extended by *User Views a Contact*. The use case description of *Add a Note*, in the proposed template, is shown in Figure 6.56.

The *View Communication Summary* use case is extended by *User Views a Contact*. The use case description of *View Communication Summary*, in the proposed template, is shown in Figure 6.57. The *View Communication Starter Summary* use case is extended by *User Views a Contact*. The use case description of *View Communication Starter Summary*, in the proposed template, is shown in Figure 6.58.

UseCase: View Communication Summary
ActorSet: User
InputSet: nosymbol
OutputSet [communicationDataMessage]: communicationSummaryData
Scenario:
System displays communicationSummaryData
End_of_Usecase

FIGURE 6.57: View Communication Summary Use Case Description.

UseCase: View Communication Starter Summary
ActorSet: User
InputSet: no
OutputSet [commStartSummaryDataMessage]: communicationstarterSummaryData
Scenario:
System displays communicationstarterSummaryData
End_of_Usecase

FIGURE 6.58: View Communication Starter Summary Use Case Description.

UseCase: Send Message
ActorSet: User
InputSet: messageContents, sendMessage, cancelMessage
OutputSet [messageTextBoxMessage]: messageTextBox
OutputSet [sendMessageOptionMessage]: sendMessageOption
OutputSet [messageFinalMessage]: messageSentMessage, messageCancelMessage
Scenario:
System displays messageTextBox
User types messageContents
System enables sendMessageOption
User selects sendMessage
System displays messageSentMessage
Alternate_Scenario
User selects cancelMessage
System displays messageCancelMessage
End_of_AlternateScenario
End_of_Usecase

FIGURE 6.59: Send Message Use Case Description.

UseCase: View Relationship Health
ActorSet: User
InputSet: noinput
OutputSet [relationshipHealthMessage]: relationshipHealthData
Scenario:
System displays relationshipHealthData
End_of_Usecase

FIGURE 6.60: View Relationship Health Use Case Description.

The *Send Message* use case is extended by *User Views a Contact*. The use case description of *Send Message*, in the proposed template, is shown in Figure 6.59. The *View Relationship Health* use case is extended by *User Views a Contact*. The use case description of *View Relationship Health*, in the proposed template, is shown in Figure 6.60. The features of the input use cases are enlisted in Table 6.1.

TABLE 6.1: **Input Examples and Case Studies Features**

Sr. No	Name	NO. of Use Case	NO. of Input Symbol-s/UC	NO. of Out-put Set/UC	NO. of Output Symbol-s/UC	NO. of Sce-nario Lines/UC	NO. of Alter-nate Scenar-ios/UC	NO. of In-cluded Use Cas-es/UC	NO. of Ex-tended Use Cas-es/UC
1	ATM Cash With-drawal Example	1	6	4	7	22	3	0	0
2	SIM Dispensing Ma-chine Example	5	1-6	1-4	1-7	4-19	0-2	1	3
3	Work Flow Manage Case Study	22	1-6	1-4	1-4	1-10	0-1	0-2	0-3
4	Touch'D Case Study	22	1-11	1-3	1-4	1-16	0-1	0-1	0-11

6.5 Generated Artefacts and Their Validation

The generated artefacts including a Kripke structure in *.png*, *.gml* and *.dot* formats along with the *.smv* model file and LTL formulas for **SIM Dispensing Machine**, **Work Flow Manager** and **TouchD́** case studies are placed at Harvard data-verse [110] due to their large size. However, the generated artefacts for **ATM Cash Withdrawal** example are provided in Appendix A.

The proposed approach consists of two main processes and these are discussed in 4.5.2 and 4.5.3. These processes formalize the provided use case into the corresponding Kripke structure and LTL formulas. The generated Kripke structure is produced in different formats including *.dot*, *.png*, *.gml* and *.smv* formats. The *.dot* file is used to generate *.png* file, using *GraphViz* API. The generated *.png* file is useful for the visual representation of the generated Kripke structure. The generated *.gml* format is useful for further graphical-based processing by using an editor with capabilities to process graph markup language.

A model checker requires a model of a software and the formal specifications for the validation of a software. Model checking is an automated process. It requires minimal human intervention during the validation process. It also generates counterexamples for the invalid scenarios. The generated *.smv* file is used to provide the generated Kripke structure as a model to NuSMV model checker. The generated LTL formulas are generated by following the syntax requirements of NuSMV model checker and are used to specify formal specification to NuSMV model checker. The generated *.smv* file for *ATM Cash Withdrawn Example* is provided as a model to the NuSMV model checker. The generated LTL formulas for this example are used as formal specification are used to verify the provided input model. NuSMV does not generate any counter example and it is shown in Figure 6.61.

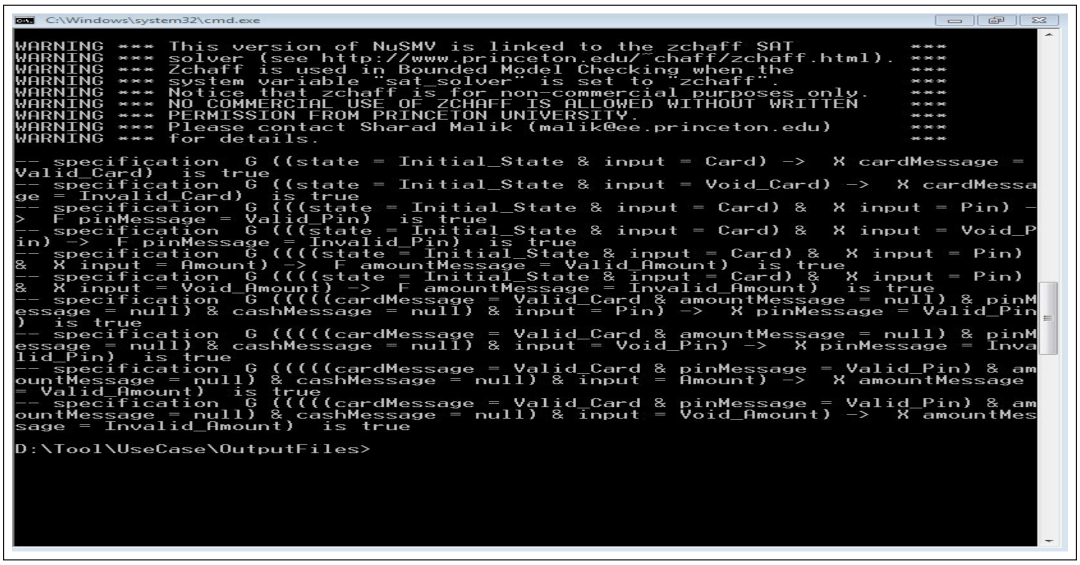


Figure 6.61: NuSMV Model Checker Output for The ATM Cash Withdrawal Example.

It is observed that *GraphViz* API successfully generates the resultant *.png* file. This generated *.png* is viewable in any photo editor. The generated *.gml* file is successfully opened and editable in *yEd* graph editor, an editor to process graph markup language. The generated *.smv* file is provided to NuSMV model checker and the model checker successfully loaded the provided model without any error.

The generated LTL formulas are provided to the NuSMV model checker for the verification of provided model. It is observed that NuSMV does not generate any syntax errors for the provided LTL formulas. In addition to it, NuSMV model checker does not generate any counter example for any provided LTL formulas. It is important to note that both Kripke structure and LTL formulas are generated by two independent processes from the provided use case model. The generated formal artefacts by two different processes validate one an other.

It is also important to verify if an invalid LTL formula is provided as an input along with the generated Kripke structure whether NuSMV model checker generates a counterexample or not. An LTL formula for this purpose is selected randomly and the selected LTL formula is:

LTLSPEC G ($state = Initial\_State$ & ($input = Card$) & **X** ($input = Pin$) $\rightarrow$ F ($pinMessage = Valid\_Pin$))

This formulas is modified and value of $pinMessage$ is set to $Invalid\_Pin$. The modified formula becomes:

LTLSPEC G ($state = Initial\_State$ & ($input = Card$) & **X** ($input = Pin$) $\rightarrow$ F ($pinMessage = Invalid\_Pin$))

This altered formula along with the generated Kripke structure are provided as input to NuSMV model checker. The model checker generates a counterexample and is shown in Figure 6.62.

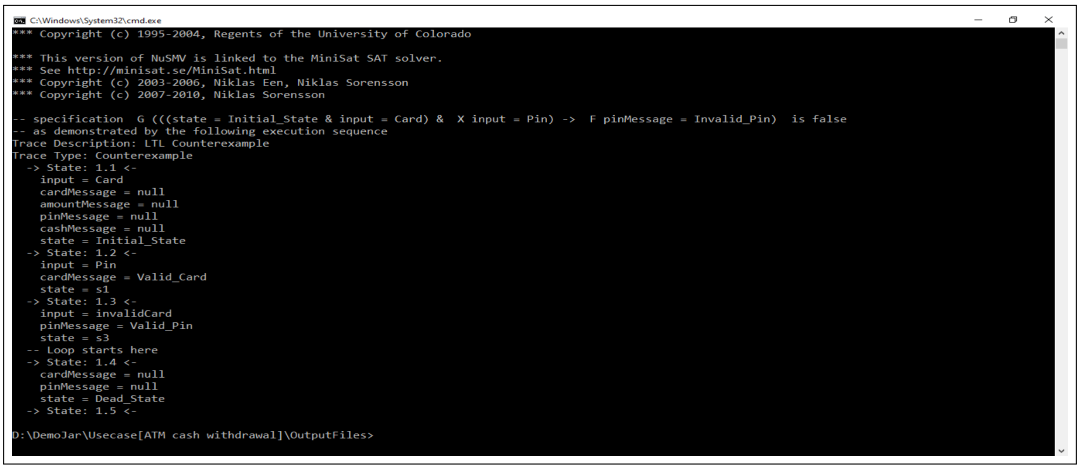


FIGURE 6.62: NuSMV Model Checker Counterexample Screenshot.

6.5.1 Conformance of the Generated Artefacts

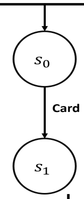
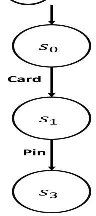
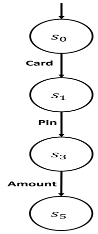
Conformance of the generated artefacts requires that the generated artefacts are valid and represents the information provided in the input use case model. A Kripke structure is valid and deterministic in nature, if it has only one initial state, all the generated states are unique in terms of their labelling function and there should only be one transition from each state for each input symbol. Moreover, the bit label of a state correspond to the binary equivalent values of the output symbols. The transitions of the generated kripke structure are labelled with an input symbol of a use case.

In addition to it, the generated LTL formulas specify a software behaviour. An LTL formula represents a particular output after receiving an or combination of input symbols on a particular state. The conformance of the generated artefacts for *ATM Cash Withdrawal Example*, *SIM Dispensing Machine Example* , *Work Flow Manger* case study and *Touch'D* case study are discussed in the following.

6.5.1.1 ATM Cash Withdrawal Example

The use case description for *ATM Cash Withdrawal Example* is shown in Figure 6.2. This use case allows a user to withdraw cash after presenting a valid card, a valid PIN and a valid amount. The use case scenario lines, generated states along with the transitions of resultant Kripke structure and the generated LTL formulas are listed in Table 6.2.

Table 6.2: **ATM Cash Withdrawal Use Case Conformance to the Generated Artefacts**

Scenario Lines	States	Transitions	LTL formulas	State Diagram
User inserts Card System notifies for the Valid_Card	s_1	$s_0 \xrightarrow{Card} s_1$	$LTLSPEC\ G(state = Initial\_State\ \&\ input = Card \rightarrow X(cardMessage = Valid\_Card))$	
User enters Pin System notifies for Valid_Pin	s_3	$s_1 \xrightarrow{Pin} s_3$	$LTLSPEC\ G(state = Initial\_State\ \&\ (input = Card)\ \&\ X(input = Pin) \rightarrow F(pinMessage = Valid\_Pin))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = Card)\ \&\ X(input = Pin) \rightarrow F(pinMessage = Valid\_Pin))$	
User enters Amount System notifies for Valid_Amount System ejects_Cash	s_5	$s_3 \xrightarrow{Amount} s_5$	$LTLSPEC\ G(cardMessage = Valid\_Card\ \&\ pinMessage = Valid\_Pin\ \&\ amountMessage = null\ \&\ cashMessage = null\ \&\ input = Amount \rightarrow X(amountMessage = Valid\_Amount))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = Card)\ \&\ X(input = Pin)\ \&\ X(input = Amount) \rightarrow F(amountMessage = Valid\_Amount))$	

continued . . .

...continued

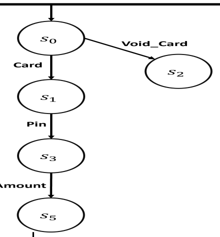
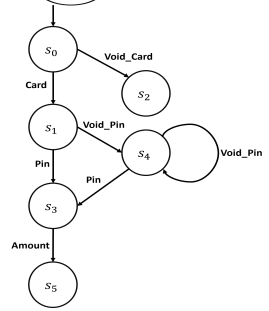
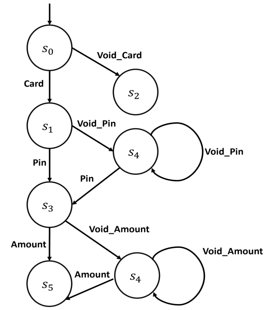
Scenario Lines	States	Transitions	LTL formulas	State Diagram
Alternate_Scenario User inserts Void_Card System notifies for the Invalid_Card End_of_AlternateScenario	s_2	$s_0 \xrightarrow{Void\_Card} s_2$	$LTLSPEC\ G(state = Initial\_State\ \&\ input = Void\_Card \rightarrow X(cardMessage = Invalid\_Card))$	
Alternate_Scenario User enters Void_Pin System notifies for Invalid_Pin Continue	s_4	$s_1 \xrightarrow{Void\_Pin} s_4$ $s_4 \xrightarrow{Void\_Pin} s_4$ $s_4 \xrightarrow{Pin} s_3$	$LTLSPEC\ G(state = Initial\_State\ \&\ (input = Card)\ \&\ X(input = Void\_Pin) \rightarrow F(pinMessage = Invalid\_Pin))$ $LTLSPEC\ G(cardMessage = Valid\_Card\ \&\ amountMessage = null\ \&\ pinMessage = null\ \&\ cashMessage = null\ \&\ input = Void\_Pin \rightarrow X(pinMessage = Invalid\_Pin))$	
Alternate_Scenario User enters Void_Amount System notifies for Invalid_Amount Continue	s_6	$s_3 \xrightarrow{Void\_Amount} s_6$ $s_6 \xrightarrow{Void\_Amount} s_6$ $s_6 \xrightarrow{Amount} s_5$	$LTLSPEC\ G(cardMessage = Valid\_Card\ \&\ pinMessage = Valid\_Pin\ \&\ amountMessage = null\ \&\ cashMessage = null\ \&\ input = Void\_Amount \rightarrow X(amountMessage = Invalid\_Amount))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = Card)\ \&\ X(input = Pin)\ \&\ X(input = Void\_Amount) \rightarrow F(amountMessage = Invalid\_Amount))$	

Table 6.2 first row enlists the scenario line where a user inserts a *Card* and the system notifies for the *Valid_Card*. The system moves from the initial state s_0 to the state s_1 with an input *Card*. This state represents a system's state with a *Valid_Card*. A transition for this execution is represented by $s_0 \xrightarrow{Card} s_1$. The LTL formula denotes the transition from *Initial_State* to the state where the output symbol *cardMessage* equals to *Valid_Card* with input value equals to *Card*. The other lines of the table represents the corresponding states, transitions and LTL formulas to the enlisted scenario lines.

6.5.1.2 Work Flow Manager

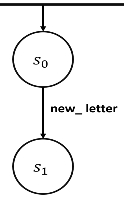
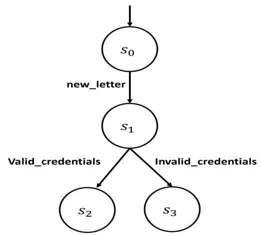
The working of this case study is discussed in Section 6.3. This case study consists of five user stories including *Request New Letter*, *Business Analyst Review Letter Request*, *Letter Writer Develops New letter*, *Business Analyst Review Developed Letter* and *Requester Approves Developed Letter*. The conformance of the generated artefacts of these user stories are discussed in the following subsections.

Request New Letter

The use case description of *Request New Letter* is listed in Figure 6.10. This use case allows to initiate a new letter. The use case scenario lines, generated states along with the transitions of the resultant Kripke structure and the generated LTL formulas are listed in Table 6.3.

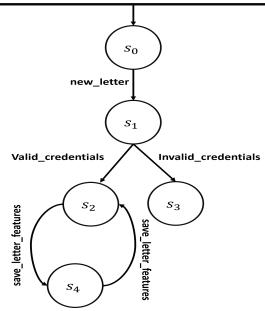
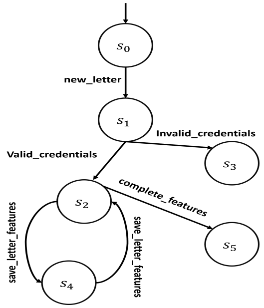
Table 6.3 enlists scenario lines and the starting lines are, a requester initiates *new_letter* request and the system displays *new_features_page*. This system state is represented by s_1 that denotes the display of *new_features_page*. A transition from s_0 to the s_1 is defined and it is labelled with the input symbol *new_letter*. This activity is represented by the LTL formula with input value equals to *new_letter* and with *feature_notification* value equals to *new_features_page*. The remaining scenario lines with the corresponding states, transitions and LTL formulas are listed in the remaining table rows.

Table 6.3: **Request New Letter Conformance to the Generated Kripke Structure and LTL Formulas**

Scenario Lines	States	Transitions	LTL formulas	State Diagram
Requester initiates new_letter request System displays new_features_page	S_1	$s_0 \xrightarrow{new\_letter} s_1$	LTLSPEC G(state = Initial_State & input = new_letter → X(feature_notification = new_features_page))	
Include Login	S_2 S_3	$s_1 \xrightarrow{Valid\_credentials} s_2$ $s_1 \xrightarrow{Invalid\_credentials} s_3$	LTLSPEC G(feature_notification = new_features_page & letter_status = null & analyst_status = null & error_message = null & screen_notification = null & save_notification_message = null & input = valid_credentials → X(screen_notification = main_screen)) LTLSPEC G(feature_notification = new_features_page & letter_status = null & analyst_status = null & error_message = null & screen_notification = null & save_notification_message = null & input = Invalid_credentials → X(screen_notification = error_login_message)) LTLSPEC G(state = Initial_State & (input = new_letter) & X(input = valid_credentials) → F(screen_notification = main_screen)) LTLSPEC G(state = Initial_State & (input = new_letter) & X(input = Invalid_credentials) → F(screen_notification = error_login_message))	

continued ...

. . . continued

Scenario Lines	States	Transitions	LTL formulas	State Diagram
Extend Save New Letter Features Condition Requester selects save_letter_features	s_4	$s_2 \xrightarrow{save\_letter\_features} s_4$ $s_4 \xrightarrow{save\_letter\_features} s_2$	$LTLSPEC\ G(feature\_notification = new\_features\_page\ \&\ screen\_notification = main\_screen\ \&\ letter\_status = null\ \&\ analyst\_status = null\ \&\ error\_message = null\ \&\ save\_notification\_message = null\ \&\ input = save\_letter\_features \rightarrow X(save\_notification\_message = letter\_features\_save\_message))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = new\_letter)\ \&\ X(input = valid\_credentials)\ \&\ X(input = save\_letter\_features) \rightarrow F(save\_notification\_message = letter\_features\_save\_message))$	
Requester provides complete_features of letter System registers_letter	s_5	$s_2 \xrightarrow{complete\_features} s_5$	$LTLSPEC\ G(feature\_notification = new\_features\_page\ \&\ screen\_notification = main\_screen\ \&\ letter\_status = null\ \&\ analyst\_status = null\ \&\ error\_message = null\ \&\ save\_notification\_message = null\ \&\ input = complete\_features \rightarrow X(letter\_status = registers\_letter))$ $LTLSPECG(state = Initial\_State\ \&\ (input = new\_letter)\ \&\ X(input = valid\_credentials)\ \&\ X(input = complete\_features) \rightarrow F(letter\_status = registers\_letter))$	

Business Analyst Review Letter Request

The use case description of *Business Analyst Review Letter Request* is listed in Figure 6.27. This use case allows to update the information of a selected letter request, if required, and also allows to call a meeting. The use case scenario lines, generated Kripke structure states along with the transitions and the generated LTL formulas are listed in Table 6.4.

Table 6.4 enlists scenario lines and the starting lines are, a *Business_Analyst select_letter* and the system displays *letter_information*. This system state is represented by s_1 that denotes the display of *letter_information*. A transition from s_0 to the s_1 is defined and is labelled with the input symbol *selects_letter*. This activity is represented by the LTL formula with input value equals to *selects_letter* and with *letterInforMessage* value equals to *letter_information*. The remaining scenario lines with the corresponding states, transitions and LTL formulas are listed in the remaining table rows.

Letter Writer Develops New Letter

This use case description specified in Figure 6.22. This use case allows a letter writer to develop a new letter by providing the letter information. The letter writer can also view the requester and business analyst information. The use case scenario lines and the generated Kripke structure states along with its transitions and the generated LTL formulas are listed in Table 6.5.

Table 6.5 enlists scenario lines and the starting lines are, a *Letter_Writer generates_letter* and the system generates *final_letter*. This system state is represented by s_1 that denotes the display of *final_letter*. A transition from s_0 to the s_1 is defined and is labelled with the input symbol *generates_letter*. This activity is represented by the LTL formula with input value equals to *generates_letter* and with *letter_completion* value equals to *final_letter*. The remaining scenario lines with the corresponding states, transitions and LTL formulas are listed in the remaining table rows.

TABLE 6.4: **Business Analyst Review Letter Request Use Case Conformance to the Generated Artefacts.**

Scenario Lines	States	Transitions	LTL formulas
Business_Analyst selects_letter System display letter_information	s_1	$s_0 \xrightarrow{selects\_letter} s_1$	$LTLSPEC\ G(state = Initial\_State\ \&\ input = selects\_letter \rightarrow X(letterInfoMessage = letter\_information))$
Include Analyst Login	s_2 s_3	$s_1 \xrightarrow{valid\_credentials} s_2$ $s_1 \xrightarrow{Invalid\_credentials} s_3$	$LTLSPEC\ G(letterInfoMessage = letter\_information\ \&\ loginMessage = null\ \&$ $editLetterMessage = null\&updateLetterMessage = null\ \&\ addLetterInfoMessage = null\ \&$ $infoMessage = null\ \&\ saveLetterMessage = null\ \&\ meetingMessage = null\ \&\ input = valid\_credentials$ $\rightarrow X(loginMessage = main\_screen))$ $LTLSPEC\ G(letterInfoMessage = letter\_information\ \&\ loginMessage = null\ \&$ $editLetterMessage = null\ \&\ updateLetterMessage = null\ \&\ addLetterInfoMessage = null\ \&$ $infoMessage = null\ \&\ saveLetterMessage = null\ \&\ meetingMessage = null\ \&\ input = Invalid\_credentials$ $\rightarrow X(loginMessage = error\_login\_message))$ $LTLSPEC\ G(state = Initial\_State\&(input = selects\_letter)\ \&\ X(input = valid\_credentials)$ $\rightarrow F(loginMessage = main\_screen))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = selects\_letter)\ \&\ X(input = Invalid\_credentials)$ $\rightarrow F(loginMessage = error\_login\_message))$
Extend Edit Letter Information Condition Business_Analyst selects edit_option	s_4 s_5	$s_2 \xrightarrow{edit\_option} s_4$ $s_4 \xrightarrow{edited\_information} s_5$ $s_5 \xrightarrow{edited\_information} s_2$	$LTLSPEC\ G(letterInfoMessage = letter\_information\ \&\ loginMessage = main\_screen\ \&$ $editLetterMessage = null\ \&\ updateLetterMessage = null\ \&\ addLetterInfoMessage = null\ \&$ $infoMessage = null\ \&\ saveLetterMessage = null\ \&\ meetingMessage = null\ \&\ input = edit\_option$ $\rightarrow X(editLetterMessage = edited\_letter\_information))$ $LTLSPEC\ G(letterInfoMessage = letter\_information\ \&\ loginMessage = main\_screen\ \&$ $editLetterMessage = edited\_letter\_information\ \&\ updateLetterMessage = null\ \&$ $addLetterInfoMessage = null\ \&\ infoMessage = null\ \&\ saveLetterMessage = null\ \&$ $meetingMessage = null\ \&\ input = edited\_information \rightarrow X(updateLetterMessage = updated\_letter))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = selects\_letter)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = edit\_option) \rightarrow F(editLetterMessage = edited\_letter\_information))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = selects\_letter)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = edit\_option)\ \&\ X(input = edited\_information) \rightarrow F(updateLetterMessage = updated\_letter))$

continued ...

. . . continued

Scenario Lines	States	Transitions	LTL formulas
Extend Add Letter Information Condition Business_Analyst selects add_info_option	s_6 s_7	$s_2 \xrightarrow{add\_info\_option} s_6$ $s_6 \xrightarrow{*} s_7$ $s_7 \xrightarrow{*} s_2$ $(*) additional\_letter\_information$	LTLSPEC G(letterInfoMessage = letter_information & loginMessage = main_screen & editLetterMessage = null & updateLetterMessage = null & addLetterInfoMessage = null & infoMessage = null & saveLetterMessage = null & meetingMessage = null & input = add_info_option → X(addLetterInfoMessage = additional_info)) LTLSPEC G(letterInfoMessage = letter_information & loginMessage = main_screen & addLetterInfoMessage = additional_info & editLetterMessage = null & updateLetterMessage = null & infoMessage = null & saveLetterMessage = null & meetingMessage = null & input = additional_letter_information → X(infoMessage = save_info)) LTLSPEC G(state = Initial_State & (input = selects_letter) & X(input = valid_credentials) & X(input = add_info_option) → F(addLetterInfoMessage = additional_info)) LTLSPEC G(state = Initial_State & (input = selects_letter) & X(input = valid_credentials) & X(input = add_info_option) & X(input = additional_letter_information) → F(infoMessage = save_info))
Extend Save Letter Information Condition Business_Analyst selects save_info_option	s_8	$s_2 \xrightarrow{save\_info\_option} s_8$ $s_8 \xrightarrow{save\_info\_option} s_2$	LTLSPEC G(letterInfoMessage = letter_information & loginMessage = main_screen & editLetterMessage = null & updateLetterMessage = null & addLetterInfoMessage = null & infoMessage = null & saveLetterMessage = null & meetingMessage = null & input = save_info_option → X(saveLetterMessage = letter_information_save_message)) LTLSPEC G(state = Initial_State & (input = selects_letter) & X(input = valid_credentials) & X(input = save_info_option) → F(saveLetterMessage = letter_information_save_message))
Include Call Meeting	s_9	$s_2 \xrightarrow{meeting\_details} s_9$	LTLSPEC G(letterInfoMessage = letter_information&loginMessage = main_screen&editLetterMessage = null&updateLetterMessage = null&addLetterInfoMessage = null&infoMessage = null&saveLetterMessage = null&meetingMessage = null&input = meeting_details → X(meetingMessage = meeting_confirmation)) LTLSPEC G(state = Initial_State&(input = selects_letter)&X(input = valid_credentials)&X(input = meeting_details) → F(meetingMessage = meeting_confirmation))

Table 6.5: **Letter Writer Develops New Letter Use Case Conformance to the Generated Artefacts.**

Scenario Lines	States	Transitions	LTL formulas
Letter_Writer generates_letter System generate final_letter	s_1	$s_0 \xrightarrow{generates\_letter} s_1$	$LTLSPEC\ G(state = Initial\_State\ \&\ input = generates\_letter \rightarrow X(letter\_completion = final\_letter))$
Include Letter Writer Login	s_2 s_3	$s_1 \xrightarrow{valid\_credentials} s_2$ $s_1 \xrightarrow{Invalid\_credentials} s_3$	$LTLSPEC\ G(letter\_completion = final\_letter\ \&\ stakeholder\_status = null\ \&$ $screen\_notification = null\ \&\ information\_provide\_status = null\ \&\ business\_analyst\_status = null\ \&$ $letter\_status = null\ \&\ information\_status = null\ \&\ input = valid\_credentials$ $\rightarrow X(screen\_notification = main\_screen))$ $LTLSPEC\ G(letter\_completion = final\_letter\ \&\ stakeholder\_status = null\ \&$ $screen\_notification = null\ \&\ information\_provide\_status = null\ \&\ business\_analyst\_status = null\ \&$ $letter\_status = null\ \&\ information\_status = null\ \&\ input = Invalid\_credentials$ $\rightarrow X(screen\_notification = error\_login\_message))$
Extend View Requester Information Condition Letter_Writer select view_requester_info	s_4	$s_2 \xrightarrow{view\_requester\_info} s_4$ $s_4 \xrightarrow{view\_requester\_info} s_2$	$LTLSPEC\ G(letter\_completion = final\_letter\ \&\ screen\_notification = main\_screen\ \&$ $stakeholder\_status = null\ \&\ information\_provide\_status = null\ \&\ business\_analyst\_status = null\ \&$ $letter\_status = null\ \&\ information\_status = null\ \&\ input = view\_requester\_info$ $\rightarrow X(information\_provide\_status = requester\_provided\_information))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = generates\_letter)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = view\_requester\_info) \rightarrow F(information\_provide\_status = requester\_provided\_information))$
Extend View Buisness Analyst Information Condition Letter_Writer select view_bus_analyst_info	s_5	$s_2 \xrightarrow{*} s_5$ $s_5 \xrightarrow{*} s_2$ (*)view_bus_analyst_info	$LTLSPEC\ G(letter\_completion = final\_letter\ \&\ screen\_notification = main\_screen\ \&$ $stakeholder\_status = null\ \&\ information\_provide\_status = null\ \&\ business\_analyst\_status = null\ \&$ $letter\_status = null\ \&\ information\_status = null\ \&\ input = view\_bus\_analyst\_info$ $\rightarrow X(business\_analyst\_status = bus\_analyst\_provided\_information))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = generates\_letter)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = view\_bus\_analyst\_info) \rightarrow F(business\_analyst\_status = bus\_analyst\_provided\_information))$

continued ...

. . . continued

Scenario Lines	States	Transitions	LTL formulas
Extend Add Information Condition Letter_Writer select add_info	s_6 s_7	$s_2 \xrightarrow{add\_info} s_6$ $s_6 \xrightarrow{*} s_7$ $s_7 \xrightarrow{*} s_2$ (*)letter_writer_provided_information	$LTLSPEC\ G(letter\_completion = final\_letter\ \&\ screen\_notification = main\_screen\ \&$ $stakeholder\_status = null\ \&\ information\_provide\_status = null\ \&\ business\_analyst\_status = null\ \&$ $letter\_status = null\ \&\ information\_status = null\ \&\ input = add\_info \rightarrow X(letter\_status = letter\_information))$ $LTLSPEC\ G(letter\_completion = final\_letter\ \&\ screen\_notification = main\_screen\ \&$ $letter\_status = letter\_information\ \&\ stakeholder\_status = null\ \&$ $information\_provide\_status = null\ \&\ business\_analyst\_status = null\ \&\ information\_status = null\ \&$ $input = letter\_writer\_provided\_information \rightarrow X(information\_status = information\_added))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = generates\_letter)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = add\_info) \rightarrow F(letter\_status = letter\_information))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = generates\_letter)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = add\_info)\ \&\ X(input = letter\_writer\_provided\_information)$ $\rightarrow F(information\_status = information\_added))$
Letter_Writer submits letter_to_stakeholders System displays submitted_stakeholders list	s_8	$s_2 \xrightarrow{letter\_to\_stakeholders} s_8$	$LTLSPEC\ G(letter\_completion = final\_letter\ \&\ screen\_notification = main\_screen\ \&$ $stakeholder\_status = null\ \&\ information\_provide\_status = null\ \&\ business\_analyst\_status = null\ \&$ $letter\_status = null\ \&\ information\_status = null\ \&\ input = letter\_to\_stakeholders$ $\rightarrow X(stakeholder\_status = submitted\_stakeholders))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = generates\_letter)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = letter\_to\_stakeholders) \rightarrow F(stakeholder\_status = submitted\_stakeholders))$

Business Analyst Review Developed Letter

The use case description of use case *Business Analyst Review Developed Letter* is provided in Figure 6.27. This use case allows a user to accept or reject the developed letter. The use case lines and the generated Kripke structure states and its transitions along with the generated LTL formulas are presented in Table 6.6.

Table 6.6 enlists scenario lines and the starting lines are, a *Business_Analyst* select a *developed_letter_review* and the system displays *letter_information*. This system state is represented by s_1 that denotes the display of *letter_information*. A transition from s_0 to the s_1 is defined and it is labelled with the input symbol *developed_letter_review*. This activity is represented by the LTL formula with input value equals to *developed_letter_review* and with *letter_notification* value equals to *letter_information*. The remaining scenario lines with the corresponding states, transitions and LTL formulas are listed in the remaining table rows.

Requester Approves Developed Letter

The use case description of *Requester Approves Developed Letter* is provided in Figure 6.32. This use case allows a user to accept or reject the developed letter. The use case scenario lines and the generated Kripke structure states along with its transitions and the generated LTL formulas are listed in Table 6.7. Table 6.7 enlists a scenario line *Include Requester Login*. It includes *Requester Login* use case. This use case displays *main_screen* when a user login to the system with the valid credentials. The system will notify the error login message if the user provides invalid credentials. This system state is represented by s_1 that denotes the display of *main_screen*. A transition from s_0 to the s_1 is defined and is labelled with the input symbol *valid_credentials*. This activity is represented by the LTL formula with input value equals to *valid_credentials* and with *screen_notification* value equals to *main_screen*. This system state having *invalid_credentials* is represented by s_2 and denotes *error_login_message*.

Table 6.6: **Business Analyst Review Developed Letter Use Case Conformance to the Generated Artefacts.**

Scenario Lines	States	Transitions	LTL formulas
Business_Analyst select a developed_letter_review System displays letter_information	s_1	$s_0 \xrightarrow{*} s_1$ $(*) developed\_letter\_revi-$ ew	$LTLSPEC\ G(state = Initial\_State \,\&\, input = developed\_letter\_review$ $\rightarrow X(letter\_notification = letter\_information))$
Include Business Analyst Login	s_2 s_3	$s_1 \xrightarrow{valid\_credentials} s_2$ $s_1 \xrightarrow{Invalid\_credentials} s_3$	$LTLSPEC\ G(letter\_notification = letter\_information \,\&\, loginMessage = null \,\&$ $rejection\_status = null \,\&\, Approval\_notification = null \,\&\, input = valid\_credentials$ $\rightarrow X(loginMessage = main\_screen))$ $LTLSPEC\ G(letter\_notification = letter\_information \,\&\, loginMessage = null \,\&$ $rejection\_status = null \,\&\, Approval\_notification = null \,\&\, input = Invalid\_credentials$ $\rightarrow X(loginMessage = error\_login\_message))$ $LTLSPEC\ G(state = Initial\_State \,\&\, (input = developed\_letter\_review) \,\&\, X(input = valid\_credentials)$ $\rightarrow F(loginMessage = main\_screen))$ $LTLSPEC\ G(state = Initial\_State \,\&\, (input = developed\_letter\_review) \,\&\, X(input = Invalid\_credentials)$ $\rightarrow F(loginMessage = error\_login\_message))$
Extend Reject Developed Letter Condition Business_Analyst selects reject_option	s_4	$s_2 \xrightarrow{reject\_option} s_4$ $s_4 \xrightarrow{reject\_option} s_2$	$LTLSPEC\ G(letter\_notification = letter\_information \,\&\, loginMessage = main\_screen \,\&$ $rejection\_status = null \,\&\, Approval\_notification = null \,\&\, input = reject\_option$ $\rightarrow X(rejection\_status = rejection\_message))$ $LTLSPEC\ G(state = Initial\_State \,\&$ $(input = developed\_letter\_review) \,\&\, X(input = valid\_credentials) \,\&\, X(input = reject\_option)$ $\rightarrow F(rejection\_status = rejection\_message))$
Extend Approve Developed Letter Condition Business_Analyst selects approve_option	s_5	$s_2 \xrightarrow{approve\_option} s_5$ $s_5 \xrightarrow{approve\_option} s_2$	$LTLSPEC\ G(letter\_notification = letter\_information \,\&\, loginMessage = main\_screen \,\&$ $rejection\_status = null \,\&\, Approval\_notification = null \,\&\, input = approve\_option$ $\rightarrow X(Approval\_notification = approval\_message))$ $LTLSPEC\ G(state = Initial\_State \,\&\, (input = developed\_letter\_review) \,\&$ $X(input = valid\_credentials) \,\&\, X(input = approve\_option) \rightarrow F(Approval\_notification = approval\_message))$

Table 6.7: **Requester Approves Developed Letter Use Case Conformance to the Generated Artefacts.**

Scenario Lines	States	Transitions	LTL formulas
Include Requester Login	s_1 s_2	$s_0 \xrightarrow{valid\_credentials} s_1$ $s_0 \xrightarrow{Invalid\_credentials} s_2$	$LTLSPEC\ G(state = Initial\_State\ \&\ input = valid\_credentials \rightarrow X(screen\_notification = main\_screen))$ $LTLSPEC\ G(state = Initial\_State\ \&\ input = Invalid\_credentials$ $\rightarrow X(screen\_notification = error\_login\_message))$
Extend Letter Rejection Condition Requester rejects_request	s_3 s_4	$s_1 \xrightarrow{rejects\_request} s_3$ $s_3 \xrightarrow{suggestion} s_4$ $s_4 \xrightarrow{suggestion} s_1$	$LTLSPEC\ G(screen\_notification = main\_screen\ \&\ save\_notification = null\ \&$ $rejection\_status = null\ \&\ approaval\_notification = null\ \&\ input = rejects\_request$ $\rightarrow X(rejection\_status = rejection\_comments))$ $LTLSPEC\ G(screen\_notification = main\_screen\ \&$ $rejection\_status = rejection\_comments\ \&\ save\_notification = null\ \&\ approaval\_notification = null\ \&$ $input = suggestion \rightarrow X(save\_notification = save\_message))$ $LTLSPEC\ G(state = Initial_State\ \&\ (input = request)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = rejects\_request) \rightarrow F(rejection\_status = rejection\_comments))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = request)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = rejects\_request)\ \&\ X(input = suggestion) \rightarrow F(save\_notification = save\_message))$
Extend Letter Approval Condition Requester accepts$_request$	s_5	$s_1 \xrightarrow{accept\_request} s_5$ $s_5 \xrightarrow{accept\_request} s_1$	$LTLSPEC\ G(screen\_notification = main\_screen\ \&$ $save\_notification = null\ \&\ rejection\_status = null\ \&\ approaval\_notification = null\ \&\ input = accepts\_request$ $\rightarrow X(approaval\_notification = approval\_message))$ $LTLSPEC\ G(state = Initial\_State\ \&\ (input = request)\ \&\ X(input = valid\_credentials)\ \&$ $X(input = accepts\_request) - > F(approaval\_notification = approval\_message))$

A transition from s_0 to the s_2 is defined and it is labelled with the input symbol *invalid_credentials*. This activity is represented by the LTL formula with input value equals to *invalid_credentials* and with *screen_notification* value equals to *error_login_message*. The remaining scenario lines with the corresponding states, transitions and LTL formulas are listed in the remaining table rows.

6.6 Comparison With the Existing Approaches

There are a number of approaches exist that formalize a use case into the corresponding formal notations and these are discussed in Chapter 3. The discussed approaches transform a use case into corresponding formal and semiformal notations. A number of these approaches are selected for the purpose to compare with the proposed approach. The selected approaches take a use case and generate corresponding formal notation. Because, the proposed approach also takes a use case as an input and generates formal notation including a Kripke structure and LTL formulas as output. The selected approaches are the work of *Somé* [61], *Simko et al.* [63], *Cuote et al.*[64] and the work of *Yang et al.* [66]. These approaches are compared on the basis of *Use case Relationships Handled*, *Usage of Controlled Input Language*, *Domain Specific*, *Manual Required Effort*, *Generated Artefacts*, *Nature of Generated Artefacts*, *Tool Support* and *Meta model Level Transformation Support*. The selected approaches and the proposed approach with the values of above mentioned attributes are listed in Table 6.8.

It is observed that only the proposed approach and the approach proposed by *Simko et al.* [63] handle use case relationships. The use case relationships allow to specify a use case model in more realistic way and enable to reuse of specified use cases. The lack of ability to handle these relationships in the transformation process restricts the approach to handle a single use case. The generated artefacts do not reflect the complete software functionality performed due to lack of the ability to handle use case relationships. However, the proposed approach handles the use case relationships during the formalization process and as a result the generated formal artefacts also represent the complete software functionality. It is also important to note that the proposed approach does not require from its user to use a controlled natural language for the specification of use case description. All of the other listed approaches use controlled language for the specification of a use case. It can be seen that the approach proposed by *Somé* [61] uses *Restricted Natural Language* and the approach proposed by *Simko et al.* [63] uses *RUCM*.

Table 6.8: **Analysis of the Existing Approaches to the Proposed Approach**

Author	Use case Relationships Handled	Usage of Controlled Input Language	Domain Specific	Manual Required Effort	Generated Artefacts	Nature of Generated Artefact	Tool Support	Metamodel-based Transformation Support
Somé	Yes	Controlled Natural Language (CNL)	No	Describe pre- and post- conditions of a use case as predicate and specify a use case description using constructs of CNL	Petri nets	non deterministic formal notation	Yes	No
Simko et al.	Yes	Annotation tags	No	Definition and usage of annotation tags for specification of use case description	Labelled Transition System	Non-deterministic formal notation	No	No

continued . . .

. . . continued

Author	Use case Relationships Handled	Usage of Controlled Input Language	Domain Specific	Manual Required Effort	Generated Artefacts	Nature of Generated Artefact	Tool Support	Metamodel-based Transformation Support
Cuoto et al.	No	Restricted Use case Statement (RUS)	Yes	Define individuals, relations, terms and data properties of the domain and specify a use case description using RUS	Ontology instance	Formal notation	Yes	No
Yang et al.	No	Boilerplates	Yes	Definition of Boilerplates and specification of a use case description using boilerplates	Ontology instance	Formal notation	Yes	No
Proposed Approach	*Yes*	*No*	*No*	*Definition of input, output with its possible values and specification of use case in the proposed template*	*Kripke structure and LTL formulas*	*Formal notation*	*Yes*	*Yes*

The work of Cuoto et al. [64] requires the use of RUS and the approach of Yang et al. [66] requires the usage of boiler plate for the specification of a use case. Along with the controlled statement structure, all the approaches including the proposed approach, use keywords. However, the keywords required by the proposed approach for the specification of a use case are common to the requirements engineers. The usage of controlled language and keywords, other than known to requirements engineer, add additional efforts required from a user to practice these approaches. However, the user of the proposed approach does not require usage of any controlled natural language for the specification of the input use case. It is important to note that the proposed approach also requires the specification of a use case in the proposed template. But, this requires identification of input and output symbols that are clear at the requirements engineering stage. A user can, then, simply specify a use case scenario in natural language.

It is also noted that the work of Cuoto et al. [64] and Yang et al. [66] are domain specific. However, the approaches proposed by Someé [61], Simko et al. [63] and the proposed approach are domain independent. The domain specific approaches are operable in the domain for which they are developed. Moreover the concepts of a domain are get evolved with the evolution of a domain. This requires to regenerate the input and output artefacts. It is important to note that the proposed approach is a domain independent.

The generated artefacts produced by the approaches of Somé [61] and Simko et al. [63] are nondeterministic in nature. However, these can be used as input to a model checker. However, the formal specification are also required along with the generated model for model checking. In addition to it, the nondeterministic nature of the generated artefacts can generate unreachable states. Whereas, the proposed approach produces a deterministic Kripke structure and LTL formulas as formal specification and these two can be provided to a model checker for model checking. The generated artefacts of the proposed approach can be provided as input to NuSMV model checker. The task to use the generated artefacts directly by a model checker, required in RQ3 is also achieved.

It is also notable that only the approaches proposed by Somé [61], Cuoto et al. [64] and Yang et al. [66] along with the proposed approach are supported with a developed tool form their authors. However, the tool developed along with the proposed approach is published and it is freely available on Harvard data repository for public use [110]. The developed tool is also platform independent.

It is also important to note that only the proposed approach is supported with the developed meta models for input and output artefacts. Other approaches including the work of Cuote et al. [64] and yang et al. [66] use ontology definition of other authors. Moreover the transformation rules presented in all other works perform instance level transformation. However, the proposed approach provides meta models for input and output artefacts along with the transformation rules at meta model level. This facilitate to transform all definable instance of input meta models to the resultant meta models instances. This make this approach a generalized approach.

Chapter 7

Conclusions and Future work

This work proposes an approach that takes a use case model as an input and generates corresponding formal artefacts including a Kripke structure and the LTL formulas. The input use case model is required to be specified in a template.

7.1 Answers to Research Questions

This work is initiated with a number of research questions to be answered. The answers to these questions are enlisted in the following paragraphs:

- RQ1 is related to find the shortcomings of the existing approaches. It is that the most of the existing approaches do not handle use case relationships and are domain specific. A number of the exist approaches require additional artefacts for the transformation process. Most of these approaches perform transformation at model level. The findings for this research question are discussed in Chapter 3.

- RQ2 requires to propose a metamodel-based approach that generates both behavioural model and formal software specifications. This task is accomplished in Chapter 4. It also requires the definition of meta models for input use case along with the output Kripke structure model and LTL formulas. Additionally, it also requires the definition of meta model level transformation rules. The meta models for the input and output artefacts are defined and the transformation rules are also defined at meta model level. A platform independent tool is also developed to practice this approach.

- RQ3 requires the definition of generated artefacts in such a format that can be used for software verification. The generated artefacts are produced in NuSMV model checker input format. The generated artefacts usage with NuSMV model checker is discussed in Chapter 5.

7.2 Conclusions

Conclusions of this work is listed in the following paragraphs:

- The proposed approach handles use case relationships, i.e., include and extend relationships.
  - This allows to generate artefacts that reflect use case model instead of a single use case.

- This approach, also, does not require any additional artefacts like system sequence diagram, class diagram, sequence diagram or ontology of the domain for the formalization.
  - This allows to practise this approach at early stage of requirement analysis of software development process.

- The proposed approach is domain-independent.
  - It does not require the definition of domain concepts for use case specification.

- The proposed approach performs metamodel-based transformation.
  - This facilitates to transform all definable instances of source metamodel to instances of target metamodels.

- The generated artefacts, i,e. Kripke structure and LTL formulas are usable for model checking.
  - This allows to verify a software behaviour at the early stage of requirements analysis.

- A GUI tool is developed in Java to make this approach useful for the software development industry.

7.3 Limitations and Future Work

The existing approach can only transform use case specified in the proposed template. In addition to it, the generated artefacts are usable with NuSMV model checker.

In future, the work can be extended to make this approach usable with the other use case templates. UML does not provide a standard template for the use case specification. However, the templates proposed by Cockburn, Duran and Ivar Jacobson are commonly used in the industry. The approach can be enabled to accept the input use model in these templates. Moreover, the proposed approach can be extended to generate the CTL formulas. CTL formulas are extension of LTL formulas. CTL allows to validate a software behaviour on all its execution paths. In addition to these, the existing approach can be extended to generate formal notation compatible to the model checkers like SPIN and SAL.

Bibliography

[1] C. Ghezzi, M. Jazayeri, and D. Mandrioli, Fundamentals of Software Engineering. Prentice Hall, 2002.

[2] C. Jones, Software engineering best practices. McGraw-Hill, Inc., 2009.

[3] H. v. Vliet, Software Engineering: Principles and Practice. Wiley Publishing, 2008.

[4] R. Wleringa and E. Dubois, "Integrating semi-formal and formal software specification techniques," Information Systems, vol. 23, no. 3-4, pp. 159–178, 1998.

[5] R. H. Thayer, S. C. Bailin, and M. Dorfman, Software requirements engineerings. IEEE Computer Society Press, 1997.

[6] K. E. Wiegers, "Writing quality requirements," Software Development, vol. 7, no. 5, pp. 44–48, 1999.

[7] V. A. de Santiago Júnior, "Solimva: A methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications," Ph.D. dissertation, Thesis (PhD in applied computing), Instituto Nacional de Pesquisas Espaciais , 2011.

[8] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso, "Model checking early requirements specifications in tropos," in Proceedings Fifth IEEE International Symposium on Requirements Engineering. IEEE, 2001, pp. 174–181.

[9] M. Kamalrudin, “Automated software tool support for checking the inconsistency of requirements,” in 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2009, pp. 693–697.

[10] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, “Elaborating requirements using model checking and inductive learning,” IEEE Transactions on Software Engineering, vol. 39, no. 3, pp. 361–383, 2012.

[11] M. D. Fraser and V. K. Vaishnavi, “A formal specifications maturity model,” Communications of the ACM, vol. 40, no. 12, pp. 95–103, 1997.

[12] F. Ipate, M. Gheorghe, and R. Lefticaru, “Test generation from p systems using model checking,” The Journal of Logic and Algebraic Programming, vol. 79, no. 6, pp. 350–362, 2010.

[13] B. Zeng and L. Tan, “Test criteria for model-checking-assisted test case generation: a computational study,” in 2012 IEEE 13th International Conference on Information Reuse & Integration (IRI). IEEE, 2012, pp. 600–607.

[14] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, “Formal methods: Practice and experience,” ACM computing surveys (CSUR), vol. 41, no. 4, pp. 1–36, 2009.

[15] J. P. Bowen and M. G. Hinchey, “Ten commandments of formal methods,” Computer, vol. 28, no. 4, pp. 56–63, 1995.

[16] C. Ghezzi, D. Mandrioli, and A. Morzenti, “Trio: A logic language for executable specifications of real-time systems,” Journal of Systems and software, vol. 12, no. 2, pp. 107–123, 1990.

[17] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, Model checking. MIT press, 2018.

[18] B. Schürmann, D. Heß, J. Eilbrecht, O. Stursberg, F. Köster, and M. Althoff, “Ensuring drivability of planned motions using formal methods,” in 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC). IEEE, 2017, pp. 1–8.

[19] S. Liu and S. Nakajima, "Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing," IEEE Transactions on Software Engineering, 2020.

[20] K. Meinke and M. A. Sindhu, "Lbtest: a learning-based testing tool for reactive systems," in 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. IEEE, 2013, pp. 447–454.

[21] H. Khosrowjerdi and K. Meinke, "Learning-based testing for autonomous systems using spatial and temporal requirements," in Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, 2018, pp. 6–15.

[22] B. K. Aichernig, C. Burghard, and R. Korošec, "Learning-based testing of an industrial measurement device," in NASA Formal Methods Symposium. Springer, 2019, pp. 1–18.

[23] M. Nanda, J. Jayanthi, and Y. Jeppu, "Formal methodsa need for practical applications," in Formal Methods for Safety and Security. Springer, 2018, pp. 1–12.

[24] J. A. Bubenko, "Challenges in requirements engineering," in Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95). IEEE, 1995, pp. 160–162.

[25] T. Gilb and S. Finzi, Principles of software engineering management. Addison-wesley Reading, MA, 1988, vol. 11.

[26] M. Cohn, User stories applied: For agile software development. Addison-Wesley Professional, 2004.

[27] K. Pohl, Requirements engineering: fundamentals, principles, and techniques. Springer Publishing Company, Incorporated, 2010.

[28] K. R. Linberg, "Software developer perceptions about software project failure: a case study," Journal of Systems and Software, vol. 49, no. 2-3, pp. 177–192, 1999.

[29] D. R. Wallace and R. U. Fujii, “Software verification and validation: an overview,” Ieee Software, vol. 6, no. 3, pp. 10–17, 1989.

[30] J. F. Peters and W. Pedrycz, Software engineering: an engineering approach. John Wiley & Sons, Inc., 1998.

[31] D. Rosenberg and K. Scott, Use case driven object modeling with UML. Springer, 1999.

[32] H. Kirk, “Use of decision tables in computer programming,” Communications of the ACM, vol. 8, no. 1, pp. 41–43, 1965.

[33] D. E. Knuth, “Computer-drawn flowcharts,” Communications of the ACM, vol. 6, no. 9, pp. 555–563, 1963.

[34] S. L. Pfleeger and J. M. Atlee, Software engineering: theory and practice. Pearson Education India, 1998.

[35] C. Wohlin et al., Engineering and managing software requirements. Springer Science & Business Media, 2005.

[36] W. S. Humphrey, A discipline for software engineering. Pearson Education India, 1995.

[37] K. Johannisson, Formal and informal software specifications. Department of Computer Science and Engineering, Chalmers University of Technology and Gteborg Univ., 2005.

[38] E. M. Clarke and J. M. Wing, “Formal methods: State of the art and future directions,” ACM Computing Surveys (CSUR), vol. 28, no. 4, pp. 626–643, 1996.

[39] C. M. Macal, “Model verification and validation, workshop on:” threat anticipation: Social science methods and models,” The University of Chicago and Argonne National Laboratory, 2005.

[40] J. M. Spivey, Understanding Z: a specification language and its formal semantics. Cambridge University Press, 1988, vol. 3.

[41] A. Pnueli, "The temporal logic of programs," in 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). IEEE, 1977, pp. 46–57.

[42] E. A. Emerson and E. M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," Science of Computer programming, vol. 2, no. 3, pp. 241–266, 1982.

[43] N. Plat, J. van Katwijk, and H. Toetenel, "Application and benefits of formal methods in software development," Software Engineering Journal, vol. 7, no. 5, pp. 335–346, 1992.

[44] A. Hall, "Realising the benefits of formal methods," in International Conference on Formal Engineering Methods. Springer, 2005, pp. 1–4.

[45] P. Rodrigues, M. Ecar, S. V. Menezes, J. P. S. da Silva, G. T. Guedes, and E. M. Rodrigues, "Empirical evaluation of formal method for requirements specification in agile approaches," in Proceedings of the XIV Brazilian Symposium on Information Systems, 2018, pp. 1–8.

[46] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Strategies for incorporating formal specifications in software development," Communications of the ACM, vol. 37, no. 10, pp. 74–87, 1994.

[47] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini, "Assisting requirement formalization by means of natural language translation," Formal Methods in System Design, vol. 4, no. 3, pp. 243–263, 1994.

[48] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," IEEE Transactions on Software Engineering, no. 1, pp. 2–13, 1980.

[49] J. S. Collofello, "Introduction to software verification and validation," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 1988.

[50] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," Synthesis lectures on software engineering, vol. 3, no. 1, pp. 1–207, 2017.

[51] S. Beydeda, M. Book, V. Gruhn et al., Model-driven software development. Springer, 2005, vol. 15.

[52] C. Gonzalez-Perez and B. Henderson-Sellers, Metamodelling for software engineering. Wiley Publishing, 2008.

[53] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in European Conference on Object-Oriented Programming. Springer, 2007, pp. 600–624.

[54] T. Yue, L. C. Briand, and Y. Labiche, "An automated approach to transform use cases into activity diagrams," in European Conference on Modelling Foundations and Applications. Springer, 2010, pp. 337–353.

[55] H. Kaindl, M. Smiałek, P. Wagner, D. Svetinovic, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, H. Schwarz, D. Bildhauer et al., "Requirements specification language definition," ReDSeeDS Project, Project Deliverable D, vol. 2, pp. 4–2, 2009.

[56] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra, "A model-driven process for eineering a ttolset for a formal method," Software: Practice and Experience, vol. 41, no. 2, pp. 155–166, 2011.

[57] C. Arora, M. Sabetzadeh, S. Nejati, and L. Briand, "An active learning approach for improving the accuracy of automated domain model extraction," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 28, no. 1, pp. 1–34, 2019.

[58] D. Harel and R. Marelly, Come, lets play: scenario-based programming using LSCs and the play-engine. Springer Science & Business Media, 2003, vol. 1.

[59] D. Dranidis, K. Tigka, and P. Kefalas, "Formal modelling of use cases with x-machines," in Proc. 1st South-East European Workshop on Formal Methods, 2003, pp. 72–83.

[60] S. S. Some, "An approach for the synthesis of state transition graphs from use cases." in Software Engineering Research and Practice, 2003, pp. 456–464.

[61] S. Some S, "Formalization of textual use cases based on petri nets," International Journal of Software Engineering and Knowledge Engineering, vol. 20, no. 05, pp. 695–737, 2010.

[62] M. Smialek, A. Kalnins, E. Kalnina, A. Ambroziewicz, T. Straszak, and K. Wolter, "Comprehensive system for systematic case-driven software reuse," in International Conference on Current Trends in Theory and Practice of Computer Science. Springer, 2010, pp. 697–708.

[63] V. Simko, P. Hnetynka, T. Bures, and F. Plasil, "Foam: A lightweight method for verification of use-cases," in 2012 38th Euromicro Conference on Software Engineering and Advanced Applications. IEEE, 2012, pp. 228–232.

[64] R. Couto, A. N. Ribeiro, and J. C. Campos, "Application of ontologies in identifying requirements patterns in use cases," in 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, France, 2014.

[65] M. Selway, G. Grossmann, W. Mayer, and M. Stumptner, "Formalising natural language specifications using a cognitive linguistic/configuration based approach," Information Systems, vol. 54, pp. 191–208, 2015.

[66] C.-W. Yang, V. Dubinin, and V. Vyatkin, "Automatic generation of control flow from requirements for distributed smart grid automation control," IEEE Transactions on Industrial Informatics, 2019.

[67] M. Singh, A. Sharma, and R. Saxena, "Formal transformation of uml diagram: Use case, class, sequence diagram with z notation for representing the static and dynamic perspectives of system," in Proceedings of International Conference on ICT for Sustainable Development. Springer, 2016, pp. 25–38.

[68] R. Wieringa, "Design science methodology: principles and practice," in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, 2010, pp. 493–494.

[69] I. Sommerville and P. Sawyer, Requirements engineering: a good practice guide. John Wiley & Sons, Inc., 1997.

[70] J. Shore et al., The Art of Agile Development: Pragmatic guide to agile software development. " O'Reilly Media, Inc.", 2007.

[71] R. H. Thayer and P. Bjorke, "Concept of operations," Encyclopedia of Software Engineering, 2002.

[72] C. Hood, S. Wiedemann, S. Fichtinger, and U. Pautz, Requirements management: The interface between requirements development and all other systems engineering processes. Springer Science & Business Media, 2007.

[73] D. Martin and G. Estrin, "Models of computations and systemsevaluation of vertex probabilities in graph models of computations," Journal of the ACM (JACM), vol. 14, no. 2, pp. 281–299, 1967.

[74] I. Jacobson, I. Spence, and B. Kerr, "Use-case 2.0," Communications of the ACM, vol. 59, no. 5, pp. 61–69, 2016.

[75] O. CORBA and I. Specification, "Object management group," Joint revised submission OMG document orbos/99-02, 1999.

[76] B. Dobing and J. Parsons, "How uml is used," Communications of the ACM, vol. 49, no. 5, pp. 109–113, 2006.

[77] A. B. Durán, B. Ruiz, and A. Toro, "A requirements elicitation approach based in templates and patterns." in In proceedings 2nd Workshop on Requirements Engineering (WER 99). Argentina, 1999.

[78] A. Cockburn, "Use case template," CU-Boulder: Computer Science, 1998.

[79] P. Kruchten, The rational unified process: an introduction. Addison-Wesley Professional, 2004.

[80] A. v. Lamsweerde, "Formal specification: A roadmap," in Proceedings of the Conference on The Future of Software Engineering, ser. ICSE 00. New York, NY, USA: Association for Computing Machinery, 2000, p. 147159. [Online]. Available: https://doi.org/10.1145/336512.336546

[81] E. F. Moore et al., "Gedanken-experiments on sequential machines," Automata studies, vol. 34, pp. 129–153, 1956.

[82] G. H. Mealy, "A method for synthesizing sequential circuits," The Bell System Technical Journal, vol. 34, no. 5, pp. 1045–1079, 1955.

[83] S. A. Kripke, "Semantical analysis of modal logic i normal modal propositional calculi," Mathematical Logic Quarterly, vol. 9, no. 5-6, pp. 67–96, 1963.

[84] R. Gorrieri, "Labeled transition systems," in Process Algebras for Petri Nets. Springer, 2017, pp. 15–34.

[85] M. Holcombe, "X-machines as a basis for dynamic system specification," Software Engineering Journal, vol. 3, no. 2, pp. 69–76, 1988.

[86] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, "Introducing obj," in Software Engineering with OBJ. Springer, 2000, pp. 3–167.

[87] S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing, "Larch: languages and tools for formal specification," Springer-Verlag Texts and Monographs in Computer Science, 1993.

[88] M. J. Gordon and T. F. Melham, Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, 1993.

[89] S. Owre, J. Rushby, N. Shankar, and F. Von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs," IEEE Transactions on Software Engineering, vol. 21, no. 2, pp. 107–125, 1995.

[90] R. M. Balzer, N. M. Goldman, and D. S. Wile, "Operational specification as the basis for rapid prototyping," ACM SIGSOFT Software Engineering Notes, vol. 7, no. 5, pp. 3–16, 1982.

[91] J. L. Peterson, "Petri nets," ACM Computing Surveys (CSUR), vol. 9, no. 3, pp. 223–252, 1977.

[92] B. Selic, "The pragmatics of model-driven development," IEEE software, vol. 20, no. 5, pp. 19–25, 2003.

[93] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," Science of computer programming, vol. 72, no. 1-2, pp. 31–39, 2008.

[94] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," in International Conference on Theory and Practice of Model Transformations. Springer, 2008, pp. 46–60.

[95] D. Kolovos, L. Rose, and R. Page, "The epsilon object language (eol)," in European Conference on Model Driven Architecture-Foundations and Applications. Springer, 2006, pp. 128–142.

[96] L. M. Rose, R. F. Paige, D. S. Kolovos, and Polack, "Epsilon validation language (evl)," The Epsilon Book2011, pp. 67–88, 2011.

[97] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, "The epsilon generation language," in European Conference on Model Driven Architecture-Foundations and Applications. Springer, 2008, pp. 1–16.

[98] J. B. Warmer and A. G. Kleppe, The object constraint language: getting your models ready for MDA. Addison-Wesley Professional, 2003.

[99] C. Snook and M. Butler, “Uml-b: Formal modeling and design aided by uml,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 15, no. 1, pp. 92–122, 2006.

[100] T. Yue, L. C. Briand, and Y. Labiche, “atoucan: an automated framework to derive uml analysis models from use case models,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 24, no. 3, pp. 1–52, 2015.

[101] T. S. Chow, “Testing software design modeled by finite state machines,” IEEE Transactions on Software Engineering, no. 3, pp. 178–187, 1978.

[102] G. Eleftherakis and P. Kefalas, “Model checking safety critical systems specified as x-machines,” Analele Universitatii Bucharest, Matematica-Informatica series, vol. 49, pp. 59–70, 2000.

[103] G. Gardey, D. Lime, M. Magnin, and O. H. . Roux, “Romeo: A tool for analyzing time petri nets,” in Computer Aided Verification, K. Etessami and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 418–423.

[104] A. David, L. Jacobsen, M. Jacobsen, K. Y. Jørgensen, M. H. Møller, and J. Srba, “Tapaal 2.0: Integrated development environment for timed-arc petri nets,” in Tools and Algorithms for the Construction and Analysis of Systems, C. Flanagan and B. König, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 492–497.

[105] M. Saaltink, “The z/eves system,” in International Conference of Z Users. Springer, 1997, pp. 72–85.

[106] J. Derrick, S. North, and A. J. Simons, “Z2sal-building a model checker for z,” in International Conference on Abstract State Machines, B and Z. Springer, 2008, pp. 280–293.

[107] M.-H. Chu, D.-H. Dang, N.-B. Nguyen, M.-D. Le, and T.-H. Nguyen, “Usl: Towards precise specification of use cases for model-driven development,”

in Proceedings of the Eighth International Symposium on Information and Communication Technology, 2017, pp. 401–408.

[108] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker b last," International Journal on Software Tools for Technology Transfer, vol. 9, no. 5-6, pp. 505–525, 2007.

[109] F. Budinsky, R. Ellersick, D. Steinberg, T. J. Grose, and E. Merks, Eclipse modeling framework: a developer's guide. Addison-Wesley Professional, 2004.

[110] Q. Zaman, A. Nadeem, and M. A. Sindhu, "Use Case to Kripke Structure and LTL Formulas Generator Tool (UCKSLTL)," 2020. [Online]. Available: https://doi.org/10.7910/DVN/S9HQYD

[111] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz-open source graph drawing tools," in International Symposium on Graph Drawing. Springer, 2001, pp. 483–484.

[112] H. Moessenboeck, "Coco/r: A generator for fast compiler front-ends," ETH, Eidgenössische Technische Hochschule Zürich, Departement Informatik, Institut für Computer Systeme, vol. 127, 1990.

Appendix A

ATM cash withdrawal example generated artefacts

The generated artefacts for ATM cash withdrawal example are listed in this appendix. These artefacts include a Kripke structure in png, dot, gml and smv formats. In addition to these the generated LTL formal specifications are also included.

The generated Kripke structure in png format is shown in Figure A.1

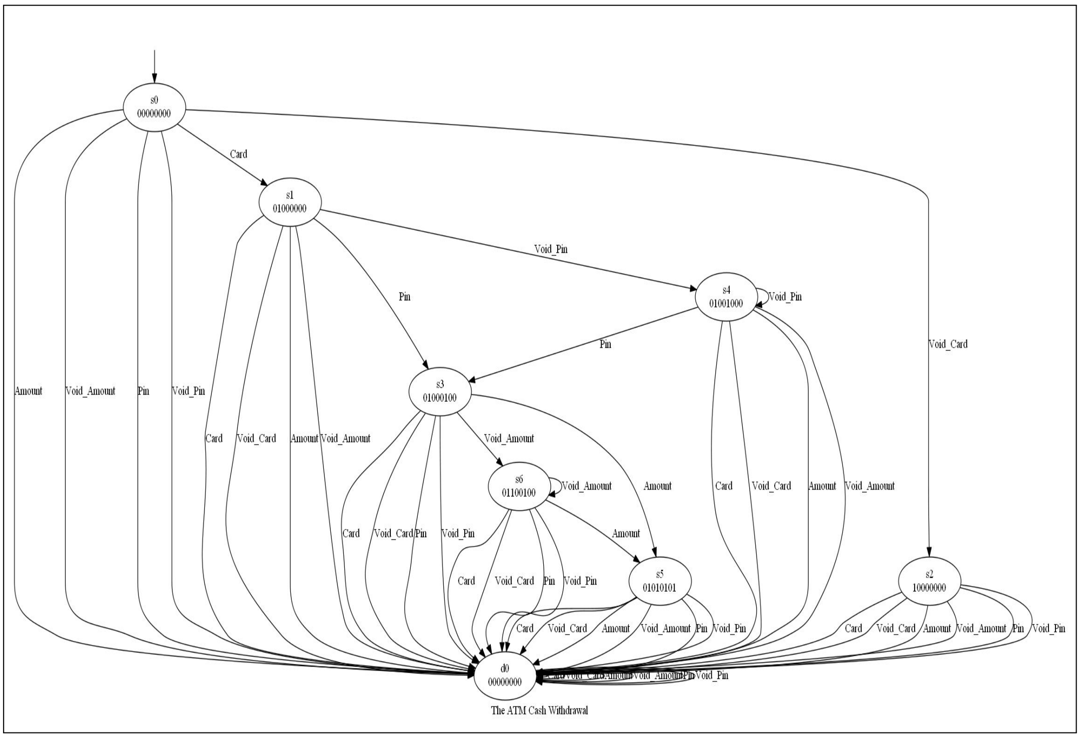


FIGURE A.1: Kripke Structure for ATM Cash Withdrawal Example.

A.1 Generated Kripke Structure in *.dot* Format

The generated Kripke structure in dot format is listed below:

digraph g {
label="The ATM Cash Withdrawal"
s0[label="s0\n00000000"]
d0[label="d0\n00000000"]
s1[label="s1\n01000000"]
s2[label="s2\n10000000"]
s3[label="s3\n01000100"]
s4[label="s4\n01001000"]
s6[label="s6\n01100100"]
s5[label="s5\n01010101"]
s[style=invis]
s − > s0
s0 − > s1[label = Card]
s0 − > s2[label = Void_Card]
s1 − > s3[label = Pin]
s1 − > s4[label = Void_Pin]
s4 − > s4[label = Void_Pin]
s4 − > s3[label = Pin]
s3 − > s5[label = Amount]
s3 − > s6[label = Void_Amount]
s6 − > s6[label = Void_Amount]
s6 − > s5[label = Amount]
s0 − > d0[label = Amount]
s0 − > d0[label = Void_Amount]
s0 − > d0[label = Pin]
s0 − > d0[label = Void_Pin]
d0 − > d0[label = Card]
d0 − > d0[label = Void_Card]
d0 − > d0[label = Amount]
d0 − > d0[label = Void_Amount]
d0 − > d0[label = Pin]
d0 − > d0[label = Void_Pin]
s1 − > d0[label = Card]
s1 − > d0[label = Void_Card]
s1 − > d0[label = Amount]
s1 − > d0[label = Void_Amount]
s2 − > d0[label = Card]
s2 − > d0[label = Void_Card]
s2 − > d0[label = Amount]
s2 − > d0[label = Void_Amount]
s2 − > d0[label = Pin]
s2 − > d0[label = Void_Pin]
s3 − > d0[label = Card]
s3 − > d0[label = Void_Card]
s3 − > d0[label = Pin]
s3 − > d0[label = Void_Pin]
s4 − > d0[label = Card]
s4 − > d0[label = Void_Card]
s4 − > d0[label = Amount]
s4 − > d0[label = Void_Amount]
s6 − > d0[label = Card]
s6 − > d0[label = Void_Card]
s6 − > d0[label = Pin]
s6 − > d0[label = Void_Pin]
s5 − > d0[label = Card]
s5 − > d0[label = Void_Card]
s5 − > d0[label = Amount]
s5 − > d0[label = Void_Amount]
s5 − > d0[label = Pin]
s5 − > d0[label = Void_Pin]}

The generated Kripke structure in smv format is listed below.

MODULE main
VAR
input : { Card , Void_Card , Amount , Void_Amount , Pin , Void_Pin };
cardMessage: {null , Valid_Card , Invalid_Card};
amountMessage: { null , Valid_Amount , Invalid_Amount};
pinMessage: {null, Valid_Pin, Invalid_Pin };
cashMessage: { null , ejects_Cash};
state : { Initial_State, Dead_State , s1 , s2 , s3 , s4 , s6 , s5};
ASSIGN
init(cardMessage) := null;
init(amountMessage) := null;
init(pinMessage) := null;
init(cashMessage) := null;
init(state) := Initial_State; next(state) := case
state = Initial_State & input = Card : s1;
state = Initial_State & input = Void_Card : s2;
state = s1 & input = Pin : s3;
state = s1 & input = Void_Pin : s4;
state = s4 & input = Void_Pin : s4;
state = s4 & input = Pin : s3;
state = s3 & input = Amount : s5;
state = s3 & input = Void_Amount : s6;
state = s6 & input = Void_Amount : s6;
state = s6 & input = Amount : s5;
state = Initial_State & input = Amount : Dead_State;
state = Initial_State & input = Void_Amount : Dead_State;
state = Initial_State & input = Pin : Dead_State;
state = Initial_State & input = Void_Pin : Dead_State;
state = Dead_State & input = Card : Dead_State;

state = Dead_State & input = Void_Card : Dead_State;
state = Dead_State & input = Amount : Dead_State;
state = Dead_State & input = Void_Amount : Dead_State;
state = Dead_State & input = Pin : Dead_State;
state = Dead_State & input = Void_Pin : Dead_State;
state = s1 & input = Card : Dead_State;
state = s1 & input = Void_Card : Dead_State;
state = s1 & input = Amount : Dead_State;
state = s1 & input = Void_Amount : Dead_State;
state = s2 & input = Card : Dead_State;
state = s2 & input = Void_Card : Dead_State;
state = s2 & input = Amount : Dead_State;
state = s2 & input = Void_Amount : Dead_State;
state = s2 & input = Pin : Dead_State;
state = s2 & input = Void_Pin : Dead_State;
state = s3 & input = Card : Dead_State;
state = s3 & input = Void_Card : Dead_State;
state = s3 & input = Pin : Dead_State;
state = s3 & input = Void_Pin : Dead_State;
state = s4 & input = Card : Dead_State;
state = s4 & input = Void_Card : Dead_State;
state = s4 & input = Amount : Dead_State;
state = s4 & input = Void_Amount : Dead_State;
state = s6 & input=Card:Dead_State;
state=s6&input=Void_Card:Dead_State;
state = s6 & input = Pin : Dead_State;
state=s6&input=Void_Pin:Dead_State;
state = s5 & input = Card : Dead_State;
state=s5&input=Void_Card:Dead_State;
state=s5&input=Amount:Dead_State;
state = s5 & input = Void_Amount : Dead_State;
state = s5 & input = Pin : Dead_State;
state=s5&input=Void_Pin:Dead_State;
esac;

next(cardMessage) := case
state = Initial_State & input = Card:Valid_Card;
state=Initial_State & input=Void_Card : Invalid_Card;
state = s1 & input = Pin : Valid_Card;
state=s1&input=Void_Pin:Valid_Card;
state = s4 & input = Void_Pin : Valid_Card;
state = s4 & input = Pin : Valid_Card;
state = s3 & input = Amount : Valid_Card;
state = s3 & input = Void_Amount : Valid_Card;
state = s6 & input = Void_Amount : Valid_Card;
state = s6 & input = Amount : Valid_Card;
state = Initial_State & input = Amount : null;
state = Initial_State & input = Void_Amount : null;
state = Initial_State & input = Pin : null;
state = Initial_State & input = Void_Pin : null;
state = Dead_State & input = Card : null;
state = Dead_State & input = Void_Card : null;
state = Dead_State & input = Amount : null;
state = Dead_State & input = Void_Amount : null;
state = Dead_State & input = Pin : null;
state = Dead_State & input = Void_Pin : null;
state = s1 & input = Card : null;
state = s1 & input = Void_Card : null;
state = s1 & input = Amount : null;
state = s1 & input = Void_Amount : null;
state = s2 & input = Card : null;
state = s2 & input = Void_Card : null;
state = s2 & input = Amount : null;
state = s2 & input = Void_Amount : null;
state = s2 & input = Pin : null;
state = s2 & input = Void_Pin : null;
state = s3 & input = Card : null;
state = s3 & input = Void_Card : null;
state = s3 & input = Pin : null;
state = s3 & input = Void_Pin : null;
state = s4 & input = Card : null;
state = s4 & input = Void_Card : null;
state = s4 & input = Amount : null;
state = s4 & input = Void_Amount : null;
state = s6 & input = Card : null;
state = s6 & input = Void_Card : null;
state = s6 & input = Pin : null;
state = s6 & input = Void_Pin : null;
state = s5 & input = Card : null;
state = s5 & input = Void_Card : null;
state = s5 & input = Amount : null;
state = s5 & input = Void_Amount : null;
state = s5 & input = Pin : null;
state = s5 & input = Void_Pin : null;
esac;

next(amountMessage) := case
state = Initial_State & input=Card:null;
state = Initial_State & input=
Void_Card:null;
state = s1 & input = Pin : null;
state = s1 & input = Void_Pin : null;
state = s4 & input = Void_Pin : null;
state = s4 & input = Pin : null;
state = s3 & input = Amount :
Valid_Amount;
state = s3 & input =Void_Amount:
Inv_Amount;

state = s6 & input =Void_Amount:
Inv_Amount;
state = s6 & input = Amount :
Valid_Amount;
state = Initial_State & input = Amount :
null;
state = Initial_State &
input = Void_Amount : null;
state = Initial_State & input = Pin : null;
state = Initial_State & input = Void_Pin :
null;
state = Dead_State & input = Card : null;
state = Dead_State & input = Void_Card :
null;
state = Dead_State & input = Amount : null;
state = Dead_State & input = Void_Amount :
null;
state = Dead_State & input = Pin : null;
state = Dead_State & input = Void_Pin :
null;
state = s1 & input = Card : null;
state = s1 & input = Void_Card : null;
state = s1 & input = Amount : null;
state = s1 & input = Void_Amount : null;
state = s2 & input = Card : null;
state = s2 & input = Void_Card : null;
state = s2 & input = Amount : null;
state = s2 & input = Void_Amount : null;
state = s2 & input = Pin : null;
state = s2 & input = Void_Pin : null;
state = s3 & input = Card : null;
state = s3 & input = Void_Card : null;
state = s3 & input = Pin : null;
state = s3 & input = Void_Pin : null;
state = s4 & input = Card : null;
state = s4 & input = Void_Card : null;
state = s4 & input = Amount : null;
state = s4 & input = Void_Amount : null;
state = s6 & input = Card : null;
state = s6 & input = Void_Card : null;
state = s6 & input = Pin : null;
state = s6 & input = Void_Pin : null;
state = s5 & input = Card : null;
state = s5 & input = Void_Card : null;
state = s5 & input = Amount : null;
state = s5 & input = Void_Amount : null;
state = s5 & input = Pin : null;
state = s5 & input = Void_Pin : null;
esac;

next(pinMessage) := case
state = Initial_State & input = Card : null;
state = Initial_State & input = Void_Card :
null;
state = s1 & input = Pin : Valid_Pin;
state = s1 & input = Void_Pin : Invalid_Pin;
state = s4 & input = Void_Pin : Invalid_Pin;
state = s4 & input = Pin : Valid_Pin;
state = s3 & input = Amount : Valid_Pin;
state = s3 & input = Void_Amount : Valid_Pin;
state = s6 & input = Void_Amount : Valid_Pin;
state = s6 & input = Amount : Valid_Pin;
state = Initial_State & input = Amount : null;
state=Initial_State&input=Void_Amount:null;
state = Initial_State & input = Pin : null;
state = Initial_State & input = Void_Pin : null;
state = Dead_State & input = Card : null;
state = Dead_State & input = Void_Card :
null;
state = Dead_State & input = Amount : null;
state=Dead_State&input=Void_Amount:null;
state = Dead_State & input = Pin : null;
state = Dead_State & input = Void_Pin : null;
state = s1 & input = Card : null;
state = s1 & input = Void_Card : null;
state = s1 & input = Amount : null;
state = s1 & input = Void_Amount : null;
state = s2 & input = Card : null;
state = s2 & input = Void_Card : null;
state = s2 & input = Amount : null;
state = s2 & input = Void_Amount : null;
state = s2 & input = Pin : null;
state = s2 & input = Void_Pin : null;
state = s3 & input = Card : null;
state = s3 & input = Void_Card : null;
state = s3 & input = Pin : null;
state = s3 & input = Void_Pin : null;
state = s4 & input = Card : null;
state = s4 & input = Void_Card : null;
state = s4 & input = Amount : null;
state = s4 & input = Void_Amount : null;
state = s6 & input = Card : null;
state = s6 & input = Void_Card : null;
state = s6 & input = Pin : null;
state = s6 & input = Void_Pin : null;
state = s5 & input = Card : null;
state = s5 & input = Void_Card : null;
state = s5 & input = Amount : null;
state = s5 & input = Void_Amount : null;
state = s5 & input = Pin : null;
state = s5 & input = Void_Pin : null;
esac;

next(cashMessage) := case
state = Initial_State & input = Card :
null;
state = Initial_State & input = Void_Card :
null;
state = s1 & input = Pin : null;
state = s1 & input = Void_Pin : null;
state = s4 & input = Void_Pin : null;
state = s4 & input = Pin : null;
state = s3 & input = Amount : null;
state = s3 & input = Void_Amount : null;

```
state = s6 & input = Void_Amount : null;
state = s6 & input = Amount : null;
state = Initial_State & input = Amount : null;
state = Initial_State & input = Void_Amount
: null;
state = Initial_State & input = Pin : null;
state = Initial_State & input = Void_Pin : null;
state = Dead_State & input = Card : null;
state = Dead_State & input = Void_Card :
null;
state = Dead_State & input = Amount : null;
state = Dead_State & input = Void_Amount :
null;
state = Dead_State & input = Pin : null;
state = Dead_State & input = Void_Pin : null;
state = s1 & input = Card : null;
state = s1 & input = Void_Card : null;
state = s1 & input = Amount : null;
state = s1 & input = Void_Amount : null;
state = s2 & input = Card : null;
state = s2 & input = Void_Card : null;
state = s2 & input = Amount : null;
state = s2 & input = Void_Amount : null;
state = s2 & input = Pin : null;
state = s2 & input = Void_Pin : null;
state = s3 & input = Card : null;
state = s3 & input = Void_Card : null;
state = s3 & input = Pin : null;
state = s3 & input = Void_Pin : null;
state = s4 & input = Card : null;
state = s4 & input = Void_Card : null;
state = s4 & input = Amount : null;
state = s4 & input = Void_Amount : null;
state = s6 & input = Card : null;
state = s6 & input = Void_Card : null;
state = s6 & input = Pin : null;
state = s6 & input = Void_Pin : null;
state = s5 & input = Card : null;
state = s5 & input = Void_Card : null;
state = s5 & input = Amount : null;
state = s5 & input = Void_Amount : null;
state = s5 & input = Pin : null;
state = s5 & input = Void_Pin : null;
esac;
```

A.2 Generated Kripke Structure in *.gml* Format

The generated Kripke structure in gml format is listed below:

```
graph [ version 2 directed 1
bb ”0,0,1696,778.18”
label ”The ATM Cash Withdrawal”
lheight 0.21 lp ”848,11.5” lwidth 2.18
node [ id 0 name ”s0” label ”s0\n00000000”
graphics [x 226 y 678.31 w 100.411 H 53.7401]
LabelGraphics [text ”s0\n00000000”]]

node [id 1 name ”d0” label ”d0\n00000000”
graphics [x 792 y 49.87 w 100.411 H 53.7401 ]
LabelGraphics [text ”d0\n00000000”]]

node [id 2 name ”s1” label ”s1\n01000000”
graphics [x 445 y 573.57 w 100.411 H 53.7401]
LabelGraphics [text ”s1\n01000000”]]

node [ id 3 name ”s2” label ”s2\n10000000”
graphics [x 1477 y 154.61 w 100.411 H 53.7401]
LabelGraphics [text ”s2\n10000000”]]

node [ id 4 name ”s3” label ”s3\n01000100”
graphics [x 687 y 364.09 w 100.411 H 53.7401 ]
LabelGraphics [text ”s3\n01000100”]]

node [id 5 name ”s4” label ”s4\n01001000”
graphics [x 1149 y 468.83 w 100.411 H 53.7401]
LabelGraphics [text ”s4\n01001000”]]

node [id 6 name ”s6” label ”s6\n01100100”
graphics [x 815 y 259.35 w 100.411 H 53.7401]
LabelGraphics [text ”s6\n01100100”]]

node [id 7 name ”s5” label ”s5\n01010101”
graphics [x 1042 y 154.61 w 100.411 H 53.7401]
LabelGraphics [text ”s5\n01010101”]]

node [id 8 name ”s” label ”s”
graphics [x 226 y 760.18 w 54 H 36 visible 0]
LabelGraphics [text ”s” ]]

edge [ id 1 source 0 target 1 label ”Amount” lp
”23,364.09”
graphics [ Line [
point [ x 175.84 y 675.88 ]
point [ x 109.26 y 670.99 ]
point [ x 0 y 651.29 ]
point [ x 0 y 574.57 ]
point [ x 0 y 574.57 ]
point [ x 0 y 574.57 ]
point [ x 0 y 153.61 ]
point [ x 0 y 113.64 ]
point [ x 30.982 y 109.83]
point [ x 68 y 94.74 ]
point [ x 128.62 y 70.032 ]
point [ x 568.42 y 56.519 ]
point [ x 731.43 y 52.316 ]
point [ x 741.76 y 52.052 ]]]

LabelGraphics [ text ”Amount” ]]

edge [ id 2 source 0 target 1
label ”Void_Amount”
lp ”122.5,364.09” graphics [
Line [point [ x 181.23 y 665.79 ]
point [ x 138.96 y 651.9 ]
point [ x 82 y 624.01 ]
point [ x 82 y 574.57 ]
point [ x 82 y 574.57 ]
point [ x 82 y 574.57 ]
point [ x 82 y 153.61 ]
point [ x 82 y 100.88 ]
point [ x 134.56 y 110.11 ]
point [ x 185 y 94.74 ]
point [ x 285.87 y 64.014 ]
point [ x 598.43 y 54.532 ]
point [ x 731.27 y 51.846 ]
point [ x 741.66 y 51.642 ]]]
LabelGraphics [text ”Void_Amount”]]

edge [id 3 source 0 target 1 label ”Pin”
lp ”208.5,364.09” graphics [
Line [point [ x 215.33 y 651.62 ]
point [ x 207.78 y 631.13 ]
point [ x 199 y 601.5 ]
point [ x 199 y 574.57 ]
point [ x 199 y 574.57 ]
point [ x 199 y 574.57 ]
point [ x 199 y 153.61 ]
point [ x 199 y 121.73 ]
point [ x 212.34 y 110.61 ]
point [ x 240 y 94.74 ]
point [ x 281.85 y 70.727 ]
point [ x 597.57 y 57.396 ]
point [ x 731.85 y 52.762 ]
point [ x 741.97 y 52.417 ]]]
LabelGraphics [text ”Pin”]]

edge [ id 4 source 0 target 1
label ”Void_Pin”
lp ”281,364.09” graphics [
Line [point [ x 236.91 y 652.07 ]
point [ x 244.77 y 631.59 ]
```

point [x 254 y 601.75]
point [x 254 y 574.57]
point [x 254 y 574.57]
point [x 254 y 574.57]
point [x 254 y 153.61]
point [x 254 y 112.28]
point [x 287.76 y 110.44]
point [x 326 y 94.74]
point [x 397.91 y 65.212]
point [x 622.63 y 55.316]
point [x 731.57 y 52.201]
point [x 741.75 y 51.921]]]
LabelGraphics [text "Void_Pin"]]

edge [id 5 source 0 target 2 label "Card"
lp "362,625.94" graphics [Line [
point [x 263.15 y 659.88]
point [x 300.62 y 642.3]
point [x 358.56 y 615.12]
point [x 398.94 y 596.18]
point [x 408.03 y 591.91]]]
LabelGraphics [text "Card"]]

edge [id 6 source 0 target 3 label "Void_Card"
lp "1507,416.46" graphics [Line [
point [x 276.12 y 675.86]
point [x 509.1 y 668.79]
point [x 1475 y 635.8]
point [x 1475 y 574.57]
point [x 1475 y 574.57]
point [x 1475 y 574.57]
point [x 1475 y 258.35]
point [x 1475 y 236.27]
point [x 1475.5 y 211.53]
point [x 1476 y 191.86]
point [x 1476.2 y 181.79]]]
LabelGraphics [text "Void_Card"]]

edge [id 7 source 1 target 1 label "Card"
lp "874.2,49.87" graphics [Line [
point [x 842.26 y 51.438]
point [x 852.7 y 51.273]
point [x 860.2 y 50.75]
point [x 860.2 y 49.87]
point [x 860.2 y 49.32]
point [x 857.27 y 48.909]
point [x 852.53 y 48.639]
point [x 842.26 y 48.302]]]
LabelGraphics [text "Card"]]

edge [id 8 source 1 target 1
label "Void_Card"
lp "920.2,49.87" graphics [Line [
point [x 842.07 y 53.056]
point [x 866.26 y 53.403]
point [x 888.2 y 52.341]
point [x 888.2 y 49.87]
point [x 888.2 y 47.746]
point [x 872 y 46.663]
point [x 852.09 y 46.621]
point [x 842.07 y 46.684]]]
LabelGraphics [text "Void_Card"]]

edge [id 9 source 1 target 1 label "Amount"
lp "975.2,49.87" graphics [Line [
point [x 841.68 y 53.956]
point [x 890.52 y 55.79]
point [x 952.2 y 54.428]
point [x 952.2 y 49.87]
point [x 952.2 y 45.624]
point [x 898.66 y 44.152]
point [x 851.85 y 45.453]
point [x 841.68 y 45.784]]]
LabelGraphics [text "Amount"]]

edge [id 10 source 1 target 1
label "Void_Amount"
lp "1038.7,49.87" graphics [Line [
point [x 841.4 y 54.745]
point [x 905.13 y 57.995]
point [x 998.2 y 56.37]
point [x 998.2 y 49.87]
point [x 998.2 y 43.713]
point [x 914.69 y 41.93]
point [x 851.71 y 44.522]
point [x 841.4 y 44.995]]]
LabelGraphics [text "Void_Amount"]]

edge [id 11 source 1 target 1 label "Pin"
lp "1088.7,49.87" graphics [Line [
point [x 841.45 y 55.168]
point [x 927.25 y 60.248]
point [x 1079.2 y 58.482]
point [x 1079.2 y 49.87]
point [x 1079.2 y 41.586]
point [x 938.61 y 39.637]
point [x 851.53 y 44.021]
point [x 841.45 y 44.572]]]
LabelGraphics [text "Pin"]]

edge [id 12 source 1 target 1
label "Void_Pin"
lp "1125.2,49.87" graphics [Line [
point [x 841.16 y 55.906]
point [x 932.48 y 62.394]
point [x 1098.2 y 60.382]
point [x 1098.2 y 49.87]
point [x 1098.2 y 39.738]
point [x 944.24 y 37.503]
point [x 851.34 y 43.164]
point [x 841.16 y 43.834]]]
LabelGraphics [text "Void_Pin"]]

edge [id 13 source 2 target 1 label "Card"

lp ”322,311.72” graphics [Line [
point [x 402.45 y 559.07]
point [x 387.05 y 552.22]
point [x 370.81 y 542.38]
point [x 360 y 528.7]
point [x 356.28 y 524]
point [x 308.73 y 325.17]
point [x 308 y 319.22]
point [x 307.19 y 312.6]
point [x 307.43 y 310.86]
point [x 308 y 304.22]
point [x 316.03 y 210.1]
point [x 274.82 y 159.07]
point [x 344 y 94.74]
point [x 371.85 y 68.846]
point [x 616.29 y 56.968]
point [x 731.9 y 52.761]
point [x 742.01 y 52.4]]]
LabelGraphics [text ”Card”]]

edge [id 14 source 2 target 1 label ”Void_Card”
lp ”391,311.72” graphics [Line [
point [x 433.35 y 547.39]
point [x 413.36 y 502.65]
point [x 373.3 y 405.84]
point [x 359 y 319.22]
point [x 343.78 y 227.05]
point [x 304.45 y 197.15]
point [x 431 y 94.74]
point [x 475.89 y 58.411]
point [x 640.99 y 51.601]
point [x 731.56 y 50.673]
point [x 741.69 y 50.591]]]
LabelGraphics [text ”Void_Card”]]

edge [id 15 source 2 target 1 label ”Amount”
lp ”468,311.72” graphics [Line [
point [x 445 y 546.39]rpoint [x 445 y 525.63]
point [x 445 y 495.87]
point [x 445 y 469.83]
point [x 445 y 469.83]
point [x 445 y 469.83]
point [x 445 y 153.61]
point [x 445 y 122.94]
point [x 455.36 y 111.57]
point [x 481 y 94.74]
point [x 521.13 y 68.388]
point [x 653.21 y 57.532]
point [x 731.74 y 53.322]
point [x 741.99 y 52.794]]]
LabelGraphics [text ”Amount”]]

edge [id 16 source 2 target 1
label ”Void_Amount”
lp ”534.5,311.72” graphics [Line [
point [x 453.9 y 546.76]
point [x 455.75 y 540.87]
point [x 457.56 y 534.6]
point [x 459 y 528.7]
point [x 469.67 y 484.79]
point [x 511.71 y 170.26]
point [x 527 y 127.74]
point [x 532.73 y 111.81]
point [x 532.15 y 104.47]
point [x 546 y 94.74]
point [x 575.34 y 74.129]
point [x 669.35 y 61.844]
point [x 732.29 y 55.727]
point [x 742.49 y 54.762]]]
LabelGraphics [text ”Void_Amount”]]

edge [id 17 source 2 target 4 label ”Pin”
lp ”630.5,468.83” graphics [Line [
point [x 482.85 y 555.71]
point [x 497.57 y 548.36]
point [x 514.17 y 539.06]
point [x 528 y 528.7]
point [x 580.53 y 489.35]
point [x 631.68 y 432.39]
point [x 661.22 y 397.1]
point [x 667.9 y 389.05]v]]
LabelGraphics [text ”Pin”]]

edge [id 18 source 2 target 5
label ”Void_Pin”
lp ”866,521.2” graphics [Line [
point [x 493.11 y 565.55]
point [x 618.37 y 547.27]
point [x 954.29 y 498.25]
point [x 1090.5 y 478.37]
point [x 1100.7 y 476.88]]]
LabelGraphics [text ”Void_Pin”]]

edge [id 19 source 3 target 1
label ”Card”
lp ”1354,102.24” graphics [Line [
point [x 1432 y 142.5]
point [x 1404.7 y 134.95]
point [x 1369.5 y 123.79]
point [x 1340 y 109.74]
point [x 1329 y 104.5]
point [x 1328.5 y 98.702]
point [x 1317 y 94.74]
point [x 1232.5 y 65.744]
point [x 971.37 y 55.457]
point [x 852.29 y 52.209]
point [x 842.25 y 51.943]]]
LabelGraphics [text ”Card”]]

edge [id 20 source 3 target 1 label ”Void_Card”
lp ”1423,102.24” graphics [Line [
point [x 1440.6 y 136.09]
point [x 1425.1 y 128.38]
point [x 1407 y 118.98]

point [x 1391 y 109.74]
point [x 1380.4 y 103.63]
point [x 1379.6 y 98.66]
point [x 1368 y 94.74]
point [x 1273.9 y 62.831]
point [x 980.43 y 54.036]
point [x 852.56 y 51.69]
point [x 842.55 y 51.513]]]
LabelGraphics [text ”Void_Card”]]

edge [id 21 source 3 target 1 label ”Amount”
lp ”1489,102.24” graphics [Line [
point [x 1473.8 y 127.73]
point [x 1470.9 y 115.76]
point [x 1465.4 y 102.49]
point [x 1455 y 94.74]
point [x 1407.3 y 59.277]
point [x 1007 y 52.481]
point [x 852.5 y 51.178]
point [x 842.25 y 51.097]]]
LabelGraphics [text ”Amount”]]

id 22 source3 target1 label”Void_Amount”
lp ”1560.5,102.24” graphics [Line [
point [x 1505.1 y 132.19]
point [x 1517.3 y 120.43]
point [x 1526.6 y 105.99]
point [x 1516 y 94.74]
point [x 1493.4 y 70.703]
point [x 1021.9 y 56.607]
point [x 852.22 y 52.296]
point [x 842.2 y 52.044]]]
LabelGraphics[text ”Void_Amount”]]

edge [id 23 source 3 target 1 label ”Pin”
lp ”1620.5,102.24” graphics [Line [
point [x 1525.8 y 147.48]
point [x 1572.7 y 139.73]
point [x 1632.5 y 123.77]
point [x 1605 y 94.74]
point [x 1579.2 y 67.487]
point [x 1036.1 y 55.252]
point [x 852.25 y 51.88]
point [x 842.21 y 51.698]
]]
LabelGraphics [text ”Pin”]]

edge [id 24 source 3 target 1 label ”Void_Pin”
lp ”1669,102.24” graphics [Line [
point [x 1526.7 y 150.58]
point [x 1584.8 y 145.31]
point [x 1668.7 y 131.27]
point [x 1634 y 94.74]
point [x 1607.1 y 66.445]
point [x 1040.9 y 54.85]
point [x 852.43 y 51.765]
point [x 842.15 y 51.6]
]]
LabelGraphics [text ”Void_Pin”]]

edge [id 25 source 4 target 1 label ”Card”
lp ”543,206.98” graphics [Line [
point [x 655.19 y 343]
point [x 644.87 y 335.95]
point [x 633.61 y 327.67]
point [x 624 y 319.22]
point [x 572.98 y 274.37]
point [x 517.81 y 266.48]
point [x 529 y 199.48]
point [x 537.11 y 150.93]
point [x 527.84 y 127.02]
point [x 565 y 94.74]
point [x 589.47 y 73.476]
point [x 673.46 y 61.579]
point [x 732.1 y 55.695]
point [x 742.37 y 54.699]]]
LabelGraphics [text ”Card”]]

edge [id 26 source 4 target 1
label ”Void_Card”
lp ”612,206.98” graphics [Line [
point [x 662.9 y 340.08]
point [x 636.48 y 312.98]
point [x 595.75 y 265.18]
point [x 580 y 214.48]
point [x 567.77 y 175.1]
point [x 547.66 y 150.74]
point [x 603 y 94.74]
point [x 621.01 y 76.509]
point [x 684.4 y 64.451]
point [x 732.84 y 57.669]
point [x 742.93 y 56.301]]]
LabelGraphics [text ”Void_Card”]]

edge [id 27 source 4 target 1 label ”Pin”
lp ”656.5,206.98” graphics [Line [
point [x 679.6 y 337.2]
point [x 667.57 y 294.65]
point [x 644.76 y 211.66]
point [x 641 y 181.48]
point [x 636.23 y 143.23]
point [x 617.85 y 125.57]
point [x 641 y 94.74]
point [x 652.47 y 79.46]
point [x 696.55 y 67.773]
point [x 734.15 y 60.318]
point [x 744.04 y 58.422]]]
LabelGraphics [text ”Pin”]]

edge [id 28 source 4 target 1
label ”Void_Pin”
lp ”716,206.98” graphics [Line [
point [x 686.87 y 337.14]
point [x 686.79 y 273.47]

point [x 688.08 y 113.36]
point [x 703 y 94.74]
point [x 712.36 y 83.051]
point [x 725.63 y 74.242]
point [x 739.07 y 67.704]
point [x 748.36 y 63.534]]]
LabelGraphics [text "Void_Pin"]]

edge [id 29 source 4 target 6
label "Void_Amount"
lp "798.5,311.72" graphics [Line [
point [x 714.15 y 341.3]
point [x 733.32 y 325.91]
point [x 759.19 y 305.14]
point [x 779.93 y 288.5]
point [x 787.89 y 282.11]]]
LabelGraphics [text "Void_Amount"]]

edge [id 30 source 4 target 7 label "Amount"
lp "1038,259.35" graphics [Line [
point [x 737.09 y 361.1]
point [x 798.55 y 356.27]
point [x 903.18 y 340.08]
point [x 973 y 286.22]
point [x 1003.8 y 262.44]
point [x 1022.4 y 220.92]
point [x 1032.3 y 191.07]
point [x 1035.4 y 181.44]]]
LabelGraphics [text "Amount"]]

edge [id 31 source 5 target 1 label "Card"
lp "1145,259.35" graphics [Line [
point [x 1142.3 y 442.1]
point [x 1132.1 y 398.48]
point [x 1115.2 y 307.47]
point [x 1131 y 232.48]
point [x 1144.5 y 168.13]
point [x 1228.8 y 143.75]
point [x 1185 y 94.74]
point [x 1163.2 y 70.414]
point [x 956.94 y 58.021]
point [x 852.28 y 53.255]
point [x 842.18 y 52.805]]]
LabelGraphics [text "Card"]]

edge [id 32 source 5 target 1 label "Void_Card"
lp "1222,259.35" graphics [Line [
point [x 1153.8 y 442.05]
point [x 1169.1 y 359.72]
point [x 1214.7 y 107.76]
point [x 1203 y 94.74]
point [x 1180 y 69.149]
point [x 960.73 y 57.278]
point [x 852.22 y 52.934]
point [x 842.07 y 52.537]]]
LabelGraphics [text "Void_Card"]]

edge [id 33 source 5 target 1 label "Amount"
lp "1304,259.35" graphics [Line [
point [x 1191.9 y 454.5]
point [x 1230.4 y 439.54]
point [x 1281 y 411.12]
point [x 1281 y 365.09]
point [x 1281 y 365.09]
point [x 1281 y 365.09]
point [x 1281 y 153.61]
point [x 1281 y 114.96]
point [x 1252.2 y 110.61]
point [x 1217 y 94.74]
point [x 1153.3 y 66.058]
point [x 953.42 y 55.835]
point [x 852.11 y 52.434]
point [x 842.04 y 52.107]]]
LabelGraphics [text "Amount"]]

edge [id 34 source 5 target 1
label "Void_Amount"
lp "1380.5,259.35" graphics [Line [
point [x 1195.3 y 458.01]
point [x 1219.1 y 451.3]
point [x 1247.5 y 440.54]
point [x 1269 y 423.96]
point [x 1283.2 y 413.05]
point [x 1283.7 y 406.8]
point [x 1292 y 390.96]
point [x 1346.3 y 287.33]
point [x 1365.3 y 221.24]
point [x 1295 y 127.74]
point [x 1279.7 y 107.34]
point [x 1271.9 y 103.59]
point [x 1248 y 94.74]
point [x 1177 y 68.49]
point [x 959.34 y 57.019]
point [x 852.48 y 52.864]
point [x 842.18 y 52.473]]]
LabelGraphics [text "Void_Amount"]]

edge [id 35 source 5 target 4 label "Pin"
lp "954.5,416.46" graphics [Line [
point [x 1103 y 457.6]
point [x 1017.7 y 438.62]
point [x 835.04 y 398.01]
point [x 742.79 y 377.5]
point [x 732.8 y 375.28]]]
LabelGraphics [text "Pin"]]

edge [id 36 source 5 target 5
label "Void_Pin"
lp "1244.2,468.83" graphics [Line [
point [x 1195.9 y 478.48]
point [x 1208 y 477.89]
point [x 1217.2 y 474.67]
point [x 1217.2 y 468.83]
point [x 1217.2 y 464.82]

point [x 1212.9 y 462.04]
point [x 1206.2 y 460.51]
point [x 1195.9 y 459.18]]]
LabelGraphics [text "Void_Pin"]]

edge [id 37 source 6 target 1 label "Card"
lp "729,154.61" graphics [Line [
point [x 797.98 y 234.02]
point [x 788.75 y 222.28]
point [x 776.48 y 208.76]
point [x 763 y 199.48]
point [x 744.23 y 186.56]
point [x 727.98 y 200.21]
point [x 715 y 181.48]
point [x 693.04 y 149.79]
point [x 695.63 y 128.07]
point [x 715 y 94.74]
point [x 721.1 y 84.244]
point [x 730.76 y 76.05]
point [x 741.19 y 69.737]
point [x 750.09 y 64.867]]]
LabelGraphics [text "Card"]]

edge [id 38 source 6 target 1
label "Void_Card"
lp "807,154.61" graphics [Line [
point [x 802.19 y 233.15]
point [x 780.87 y 191.19]
point [x 740.39 y 111.4]
point [x 740 y 109.74]
point [x 738.46 y 103.25]
point [x 737.19 y 100.79]
point [x 740 y 94.74]
point [x 742.94 y 88.407]
point [x 747.3 y 82.659]
point [x 752.26 y 77.564]
point [x 759.87 y 70.574]]]
LabelGraphics [text "Void_Card"]]

edge [id 39 source 6 target 1 label "Pin"
lp "863.5,154.61" graphics [Line [
point [x 831.1 y 233.72]
point [x 847.5 y 205.23]
point [x 867.33 y 158.32]
point [x 843 y 127.74]
point [x 820.85 y 99.898]
point [x 787.15 y 137.58]
point [x 765 y 109.74]
point [x 758.63 y 101.74]
point [x 760.2 y 91.993]
point [x 764.86 y 82.752]
point [x 770.12 y 74.121]]]
LabelGraphics [text "Pin"]]

edge [id 40 source 6 target 1
label "Void_Pin"
lp "913,154.61" graphics [Line [
point [x 839.86 y 235.78]
point [x 867.2 y 208.23]
point [x 903.9 y 161.19]
point [x 877 y 127.74]
point [x 853.62 y 98.665]
point [x 819.12 y 138.21]
point [x 795 y 109.74]
point [x 789.59 y 103.36]
point [x 787.33 y 95.045]
point [x 786.76 y 86.689]
point [x 786.8 y 76.601]]]
LabelGraphics [text "Void_Pin"]]

edge [id 41 source 6 target 6
label "Void_Amount"
lp "923.7,259.35" graphics [Line [
point [x 861.86 y 269]
point [x 874.03 y 268.41]
point [x 883.2 y 265.19]
point [x 883.2 y 259.35]
point [x 883.2 y 255.34]
point [x 878.87 y 252.56]
point [x 872.2 y 251.03]
point [x 861.86 y 249.7]]]
LabelGraphics [text "Void_Amount"]]

edge [id 42 source 6 target 7 label "Amount"
lp "987,206.98" graphics [Line [
point [x 857.18 y 244.49]
point [x 880.21 y 236.5]
point [x 909.04 y 225.81]
point [x 934 y 214.48]
point [x 956.43 y 204.3]
point [x 980.57 y 191.34]
point [x 1000.3 y 180.18]
point [x 1009.1 y 175.19]]]
LabelGraphics [text "Amount"]]

edge [id 43 source 7 target 1 label "Card"
lp "824,102.24" graphics [Line [
point [x 1004.7 y 136.51]
point [x 996.07 y 133.13]
point [x 986.84 y 129.95]
point [x 978 y 127.74]
point [x 941.58 y 118.62]
point [x 838.95 y 133.65]
point [x 810 y 109.74]
point [x 802.94 y 103.9]
point [x 798.58 y 95.375]
point [x 795.92 y 86.61]
point [x 793.58 y 76.746]]]
LabelGraphics [text "Card"]]

edge [id 44 source 7 target 1 label "Void_Card"
lp "893,102.24" graphics [Line [
point [x 1004.3 y 136.68]
point [x 995.75 y 133.33]

point [x 986.67 y 130.12]
point [x 978 y 127.74]
point [x 927.27 y 113.79]
point [x 908.83 y 131.65]
point [x 861 y 109.74]
point [x 852.6 y 105.89]
point [x 837.42 y 93.217]
point [x 823.43 y 80.607]
point [x 815.83 y 73.664]]]
LabelGraphics [text "Void_Card"]]

edge [id 45 source 7 target 1 label "Amount"
lp "971,102.24" graphics [Line [
point [x 1003.4 y 137.09]
point [x 986.19 y 129.33]
point [x 965.79 y 119.63]
point [x 948 y 109.74]
point [x 937.33 y 103.81]
point [x 936 y 100.03]
point [x 925 y 94.74]
point [x 900.06 y 82.746]
point [x 871.01 y 72.746]
point [x 846.43 y 65.337]
point [x 836.55 y 62.423]]]
LabelGraphics [text "Amount"]]

edge [id 46 source 7 target 1
label "Void_Amount"
lp "1051.5,102.24" graphics [Line [
point [x 1027.4 y 128.72]
point [x 1019.1 y 116.69]
point [x 1007.6 y 103.08]
point [x 994 y 94.74]
point [x 951.19 y 68.474]
point [x 894.81 y 57.815]
point [x 852.5 y 53.549]
point [x 842.33 y 52.619]]]
LabelGraphics [text "Void_Amount"]]

edge [id 47 source 7 target 1 label "Pin"
lp "1109.5,102.24" graphics [Line [
point [x 1076.6 y 135.03]
point [x 1093.5 y 123.57]
point [x 1107.9 y 108.47]
point [x 1096 y 94.74]
point [x 1065.2 y 59.153]
point [x 932.25 y 51.787]
point [x 852.73 y 50.641]
point [x 842.36 y 50.523]]]
LabelGraphics [text "Pin"]]

edge [id 48 source 7 target 1
label "Void_Pin"
lp "1154,102.24" graphics [Line [
point [x 1084.7 y 140.37]
point [x 1112 y 129.73]
point [x 1139.4 y 113.5]
point [x 1123 y 94.74]
point [x 1105.4 y 74.694]
point [x 942.28 y 60.809]
point [x 851.78 y 54.578]
point [x 841.65 y 53.892]]]
LabelGraphics [text "Void_Pin"]]

edge [id 49 source 8 target 0
lp "" graphics [Line [
point [x 226 y 741.85]
point [x 226 y 734.12]
point [x 226 y 724.7]
point [x 226 y 715.54]
point [x 226 y 705.41]]]]]

A.3 Generated Linear Temporal Logic Formulas

The generated LTL formal specifications are listed below:

LTLSPEC G (state = Initial_State & (input = Card) & X (input = Pin)
→ F (pinMessage = Valid_Pin))

LTLSPEC G (state = Initial_State & (input = Card) & X (input = Void_Pin)
→ F (pinMessage = Invalid_Pin))

LTLSPEC G (state = Initial_State & (input = Card) & X (input = Pin) &
X (input = Amount) → F (amountMessage = Valid_Amount))

LTLSPEC G (state = Initial_State & (input = Card) & X (input = Pin) &
X (input = Void_Amount) → F (amountMessage = Invalid_Amount))

LTLSPEC G(state = Initial_State & input = Card → X (cardMessage = Valid_Card))

LTLSPEC G(state = Initial_State & input = Void_Card
→ X (cardMessage = Invalid_Card))

LTLSPEC G(cardMessage = Valid_Card & amountMessage = null & pinMessage = null
& cashMessage = null & input = Pin → X (pinMessage = Valid_Pin))

LTLSPEC G(cardMessage = Valid_Card & amountMessage = null & pinMessage = null
& cashMessage = null & input = Void_Pin → X (pinMessage = Invalid_Pin))

LTLSPEC G(cardMessage = Valid_Card & pinMessage = Valid_Pin &
amountMessage = null & cashMessage = null & input = Amount
→ X (amountMessage = Valid_Amount))

LTLSPEC G(cardMessage = Valid_Card & pinMessage = Valid_Pin &
amountMessage = null & cashMessage = null & input = Void_Amount
→ X (amountMessage = Invalid_Amount))