

Institute of Computer-aided Product Development Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis

Agile Requirements Engineering in Product Development

Suresh Muthuveerappan

Course of Study:	Information Technology
Examiner:	Univ-Prof. Hon-Prof. Dr. Dieter Roller
Supervisor:	Dipl.-Inf. Akram Chamakh
Commenced:	April 18, 2017
Completed:	October 19, 2017
CR-Classification:	D.2.5, D.2.2, D.2.1

Abstract

The quality of requirements plays a major role in any software development process. In order to ensure such quality, the elicited requirements are analyzed using various criteria. Testability is one such criterion which can be evaluated by creating test cases corresponding to the requirement. Test case design and generation is a tedious manual process performed by experts and it consumes a significant time in the software development lifecycle. In addition to this, the test cases written manually may not offer a complete coverage of the requirements. In processes like Agile, frequent changes in requirements reduce the reusability of the manually written test cases, thereby increasing the cost and time of the testing process.

In order to eliminate the tedious manual processes in testing, this thesis proposes a feasible solution for the automatic generation of test scripts directly from the requirements. In Agile, the requirements are usually in the form of use cases defined in natural language. This thesis proposes a formal method to derive an analyzable model from the requirements in natural language. The derived model is further analyzed to create test cases and later the test scripts. In order to assert the feasibility of the approach, a tool has been developed that generates test scripts automatically from the use case specifications given in a restricted natural language. The tool reduces the effort required to generate the test scripts and also improves the test coverage thereby improving the quality of test suite. The results from the evaluation study are also presented in this thesis.

Contents

1	Introduction	17
1.1	Motivation	17
1.2	Problem Statement	19
1.3	Proposed Solution	20
1.4	Thesis Management	21
2	Background	25
2.1	Evolution of Agile Software Development	25
2.2	Model Driven Software Engineering	27
2.3	Unified Modeling Language	28
2.4	Petri Net	29
3	Related Work and Utilized Software Tools	33
3.1	Related Work	33
3.2	Utilized Software Tools	36
4	The Proposed Solution	39
4.1	Overview of the Solution	39
4.2	Requirements Elicitation in Agile	40
4.3	Requirements in a Restricted Natural Language	41
4.4	Generation of Petri Nets from Use Case Specifications	43
4.5	Generation of Reachability/Coverability Graph	45
4.6	Generation of Test Specifications and Test Scripts	45
5	Design and Implementation of the Tool	47
5.1	Design of the Tool	47
5.2	Implementation of the Tool	49
6	Evaluation of the Tool	55
6.1	Preliminary Preparation for Evaluation	55
6.2	Evaluation Results and Discussion	57
7	Conclusion and Outlook	65
7.1	Summary of the Thesis	65

7.2	Limitations	66
7.3	Outlook for Future Work	66
7.4	Conclusion	67
A	Appendix	69
A.1	M2M Rules for RUCM to Petri Net Conversion	69
A.2	Implementation of M2M Rules for Sub to Integrated Petri Nets Conversion in QVTo	70
A.3	Implementation of M2T Rules for Petri Net to PNML Conversion in Xpand	74
	Bibliography	77

List of Figures

1.1	Overview of the proposed solution	20
1.2	Gannt Chart for the thesis work	22
2.1	Overview of transformations in MDA [CD15]	27
2.2	Examples of UML diagrams.	29
2.3	An example of Petri Net	30
2.4	Concept of transitions in a Petri Net.	32
3.1	Overview of different languages in QVT [EQVT10]	37
4.1	Overview of the proposed solution	40
4.2	Concept of reachability.	44
4.3	An example of Reachability Graph	46
5.1	Overview of different components in the tool	49
5.2	Petri Net Metamodel	50
5.3	RUCM Metamodel	50
5.4	Running example given as input to the tool	52
5.5	Integrated Petri Net for the running example	52
5.6	The final output of the TSTG tool	54
6.1	The procedure used for evaluating the tool	58
6.2	Effort _{TC} and TCP for manual and automatic method	60
6.3	Comparison between different existing approaches	61
6.4	Breakdown of the time taken between different activities	62
6.5	Relation between number of test cases and total time taken	63
A.1	M2M Rules for Initial State and Simple Basic Flow	69
A.2	M2M Rules for Conditional Basic Flow and Loop Basic Flow	69
A.3	M2M Rules for Specific Alternative Flow, Concurrency and Final State	70

List of Tables

4.1	RUCM Use Case Template [YBL13]	42
5.1	Running Example	51
6.1	Evaluation results of manual test script generation method	59
6.2	Evaluation results of automatic test script generation method	59
6.3	Evaluation results with respect to coverage criteria	60

List of Listings

List of Algorithms

List of Abbreviations

CWM	Common Warehouse Metamodel.	27
EFSM	Extended Finite State Machine.	33
EMF	Eclipse Modeling Framework.	36
JDK	Java Development Kit.	48
KDT	Keyword Driven testing.	34
LCS	Lane Centering System.	51
LDWS	Lane Departure Warning System.	51
LKS	Lane Keeping System.	51
LMS	Lane Management System.	50
M2M	Model-to-Model.	36
M2T	Model-to-Text.	36
MBT	Model Based Testing.	18
MDA	Model Driven Architecture.	26
MDD	Model Driven Development.	27
MDSD	Model Driven Software Development.	27
MDSE	Model Driven Software Engineering.	36
MOF	Meta Object Facility.	27
NL	Natural Language.	19
NLP	Natural Language Processing.	34
OCL	Object Constraint Language.	35
OMG	Object Management Group.	26
PIM	Platform Independent Model.	27

PIPE2	Platform Independent Petri Net Editor.	37
PNM	Petri Net Model.	20
PNML	Petri Net Modelling Language.	37
PSM	Platform Specific Model.	27
RNL	Restricted Natural Language.	43
RTCM	Restricted Test Case Modeling.	35
RUCM	Restricted Use Case Modeling.	35
RUCS	Restricted Use Case Specification.	20
SDL	System Description Language.	33
SDLC	Software Development Life Cycle.	18
SRS	Software Requirements Specification.	51
SUT	System Under Test.	35
TCP	Test Case Productivity.	57
TSGT	Test Script Generation Tool.	47
UCS	Use Case Specification.	20
UML	Unified Modelling Language.	18
UMTG	Use Case Modeling for System Tests Generation.	35
XMI	XML Metadata Interchange.	27

1 Introduction

The development of modern day software is becoming an increasingly complex activity which warrants extensive research in the field of Software Engineering. According to IEEE Standards, Software Engineering is described as the application of a systematic, closely controlled, proven approach to the development, operation and maintenance of software. Along these lines, several methodologies were introduced to reduce the cost and time of software development and to improve the quality of software product. Most of the current research in the field of Software Engineering focuses on some of these objectives and this thesis also tries to address one such purpose.

The first half of this chapter illustrates the motivation behind the thesis work, the problem statement and the proposed solution. The second half illustrates how the thesis is planned and managed along with the organization of the thesis report.

1.1 Motivation

The following section describes the motivation behind the thesis from different perspectives.

1.1.1 Agile Requirements Perspective

Agile software development processes [Amb09] were introduced to keep up with the fast changing marketplace along with the view to support frequent changes in requirements, stakeholder involvement and their priorities and finally quality products with shorter deadlines. Agile development processes have shown greater success rate than traditional software development processes in industry and hence had become an inevitable term in the current software industry. In order to ensure quality products within a shorter development lifecycle, development and testing are done in parallel based on the set of elicited requirements. As requirement changes are frequent in an agile process, the need to change test cases also becomes necessary which when performed manually is time

consuming and extensive. Hence an automated process of test case generation directly from the elicited requirements plays a huge role in the Agile development process.

1.1.2 Software Testing Perspective

Software Testing is one of the salient steps in the software development lifecycle which ensures whether the developed product meets its purported requirements. Test cases are generally derived from functional requirements, which are usually written in common natural language and this requires a test engineer to manually elicit and develop system test case specifications and also to convert it to a test scripting language for automatic execution. Particularly in the context of safety critical systems, the traceability between system test cases and the requirements and the test coverage with respect to requirements are imperative. The efficiency of the manual process depends on the domain expertise of the test engineer and it also does not offer a systematic way to establish test coverage and traceability. Hence, it can be stated firmly that the manual process of test case creation is a time consuming, expensive and a non-systematic approach. In order to overcome these challenges, this thesis focuses on the automatic generation of system test cases directly from requirements specifications written in natural language.

1.1.3 Model Based Testing Perspective

Automatic test case generation has been the topic of study in software engineering for quite a long period. But most of the already proposed approaches need customized artifacts as input for test case generation. For example, Model Based Testing (MBT) is one approach in which models of the system under test is used for test case generation. The functional behavior of the system is modeled in any formal specification notations [CFB+13] which are then used for the generation of test scenarios and test oracles.

The most common artifact used in such cases is the Unified Modelling Language (UML) model that captures the behavioral or functional aspect of the system. Many approaches need the requirement specifications to be modeled as UML behavioral models like state charts [RG99b], activity diagrams [LJX+04] and sequence diagrams [NFLJ06]. But these approaches when applied in complex industrial projects require the presence of precise behavioral models representing the system. This again is a complex and time consuming process which defeats our core purpose. MBT also becomes impractical when an industry never uses formal models in its software development life cycle. Hence the best approach would be to generate test cases directly from requirement specifications without the need for manual creation of any input artifacts.

1.2 Problem Statement

Software testing is a challenging, time consuming and expensive process in Software Development Life Cycle (SDLC). The process of software testing includes the creation of test cases, test execution and test evaluation. In the mentioned processes, test execution and evaluation are relatively easy and simple whereas creation of test cases utilizes nearly 40 – 70 % of the total effort spent on testing [KJ14]. This is because, while test execution and evaluation can be easily automated, eliciting test cases from requirements specification and deciding whether the test cases are sufficient to verify the entire system are done manually by testing experts. Such a process of manual elicitation of test cases has the following drawbacks.

1. The efficiency of the process is hugely dependent on the experience of the testing experts and his knowledge on the particular domain.
2. The process of deriving test cases is not systematic and it solely depends on the experience of the expert.
3. The traceability between requirements and test cases are not systematically established which are necessary for establishing a standard process.
4. There is no standardized process to establish the test coverage and this also leads to the increase in the number of redundant test cases.
5. The requirements written in Natural Language (NL) are often ambiguous and are interpreted differently by different domain experts. The same holds true for test cases written in NL and its interpretation by the tester.
6. Finally, the process is manual, error-prone and time consuming.

In order to overcome these drawbacks of the current practices, this thesis focuses on providing an ideal situation where textual, easy to read and traceable test cases are automatically generated from requirements defined in NL. This process should also provide a systematic approach for test case generation, ensure necessary test coverage, and provide means to automatically convert these textual test cases into executable test cases along with test input data and thereby improving the quality of the product. In short, the thesis tries to address the following questions.

1. Can we create a tool that can automatically generate test cases from requirements defined in natural language without any intermediate input from domain expert?
2. Can the tool be used to create some intermediate output that can be used in other software engineering process?

3. Can the tool automatically create executable test cases with minimal input from domain experts?

1.3 Proposed Solution

The proposed solution is to create a tool that takes the requirements specification written in natural language as an input and convert it into any intermediate formal notation such as Petri Net. The requirements are usually written as Restricted Use Case Specifications (RUCSs) which are Use Case Specifications (UCSs) with a defined template and a set of restriction rules. The usage of such rules is to reduce the imprecision and incompleteness in UCS. The intermediate Petri Net Model (PNM) can be analyzed using any search algorithms for test scenario generation. The test scenarios are then converted into executable test cases along with test input. The conversion into formal models such as PNM has other advantages as they can also be used for other purposes such as checking the completeness, consistency, correctness and unambiguity of the given requirement descriptions [SLA15]. The skeleton of the proposed work is shown in Figure 1.1.

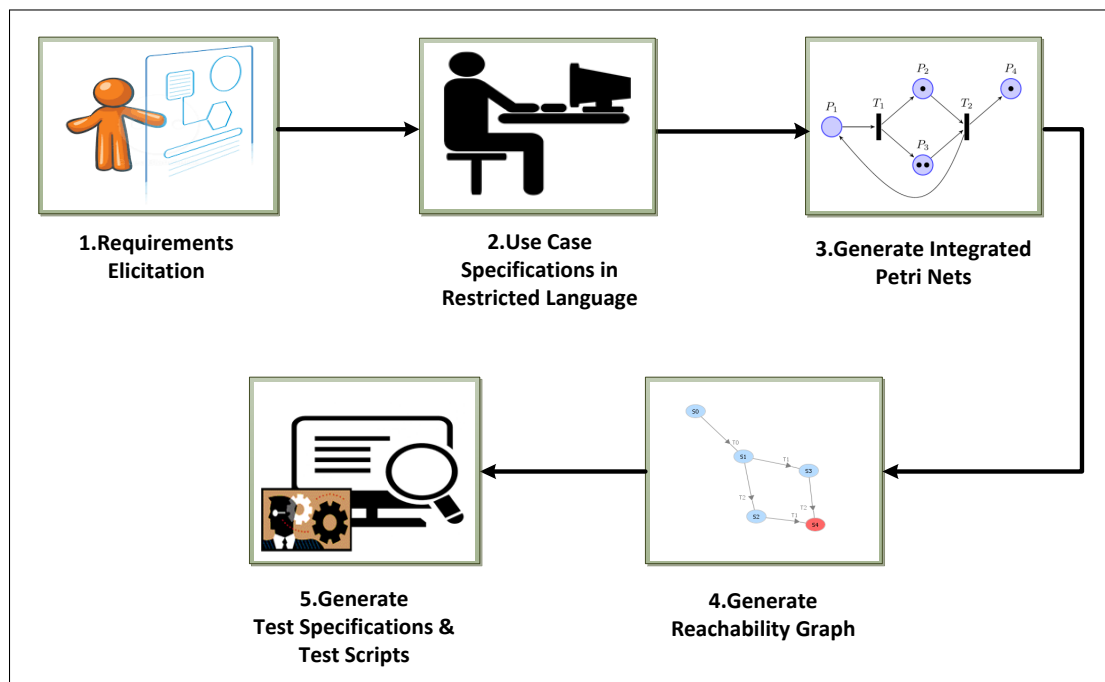


Figure 1.1: Overview of the proposed solution

1.4 Thesis Management

A project plan was created in order to guarantee an organized approach to this thesis work. Various milestones were created to monitor the progress of the project with time.

1.4.1 Project Plan

A Waterfall model with overlapping phases was created for project planning. The Gantt chart in Figure 1.2 shows the various phases and the time duration planned for each phase. The significant phases are enlisted below.

1. Literature Survey
2. Design and Development
3. Testing and further development
4. Documentation

From the Gantt chart, it can be seen some phases are overlapped. For example, the overall design of the approach and tool selection for implementation can be made immediately once a feasible approach has been identified in the literature survey. The identification of a suitable system and the enumeration of requirements for basic functionalities of the proposed tool can be done in parallel to the above mentioned phase.

1.4.2 Literature Survey

This thesis mainly revolves around the work done by Edgar et. al., 2015 [SACS15], Wang et. al., 2015 [WPG+15b] and Toa Yue et. al., 2015 [YAZ15]. The literature survey started with the books from the Library of University Stuttgart and moved towards online resources like the IEEE Xplore catalog, the ACM digital library, the E-Books from various publishers like Springer, the search engines from Internet such as Google¹ and specific search engines for scientific journals like Google Scholar². Apart from the scientific journals and publications, numerous other resources had been used from the internet regarding the availability of different tools, their user manuals and their documentation.

¹<https://www.google.de>

²<https://scholar.google.de>

1 Introduction

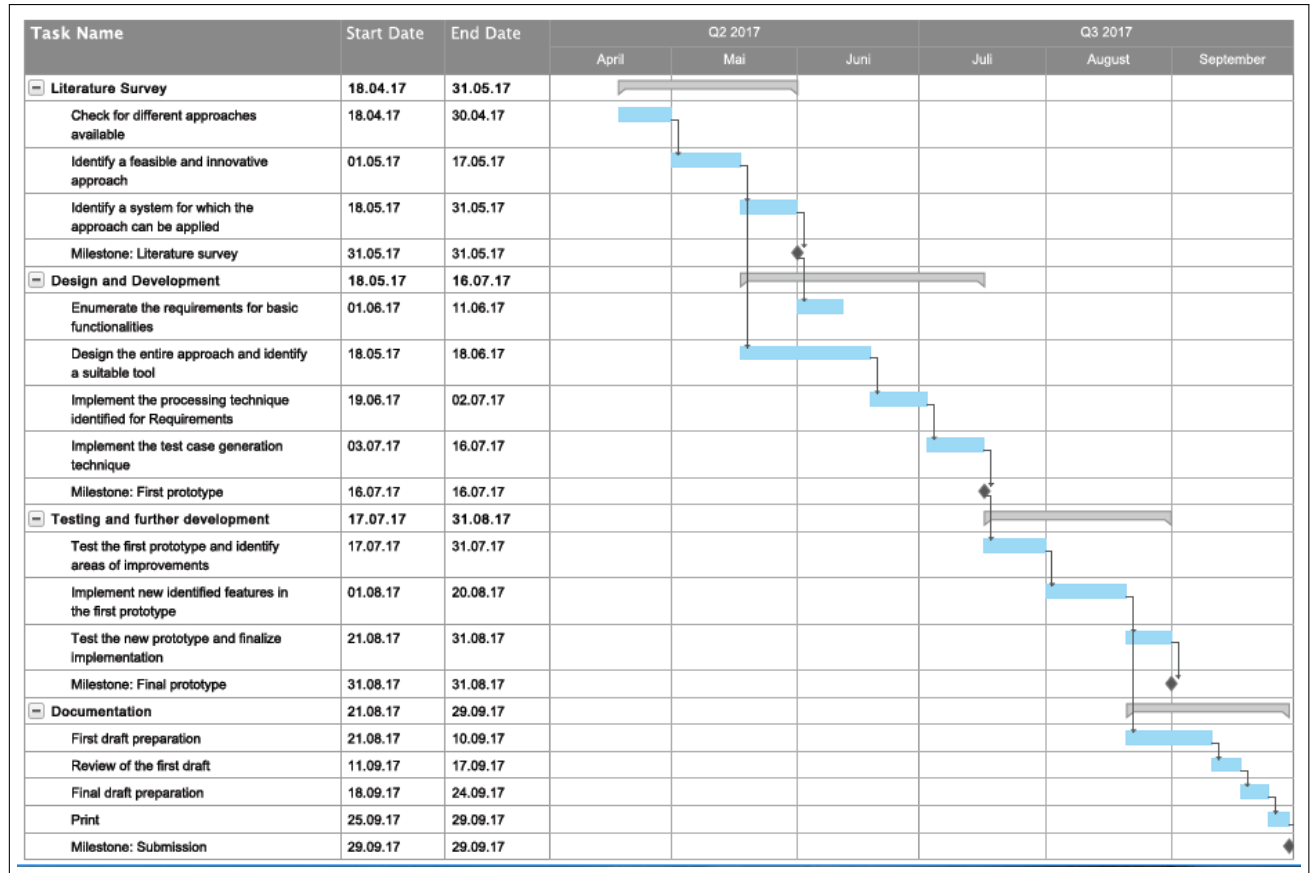


Figure 1.2: Gantt Chart for the thesis work

1.4.3 Thesis Structure

The thesis has been organized as the following chapters.

Chapter 1 This chapter gives a short introduction to the thesis work, the motivation behind the thesis and a rough idea of the proposed solution.

Chapter 2 This chapter gives a short introduction to the different methodologies and terminologies that is used throughout the rest of the thesis.

Chapter 3 This chapter provides an overview of related research work in the domain of automatic test case generation and it also gives a basic introduction to the technologies used in the implementation stage of the thesis.

Chapter 4 This chapter provides a detailed account of the proposed approach and the various methodologies used for test script generation.

Chapter 5 This chapter gives a deeper insight into the design of the tool and the implementation of the different methodologies introduced in the previous chapter.

Chapter 6 This chapter illustrates the evaluation of the proposed tool and its comparison with other similar approaches.

Chapter 7 This chapter provides a short overview of the thesis and details the different limitations and contributions of the current work.

2 Background

This chapter gives a short introduction to topics regarding Agile processes and their definitions, Model Driven Architecture, Unified Modeling Language and Petri Nets. Basic knowledge in these topics helps us to appreciate the various design and implementation decisions taken in Chapter 3 and 4. This chapter also acts a guide to understand the various terminologies used throughout this work.

2.1 Evolution of Agile Software Development

Agile Software Development was introduced in the year 2001 by Agile Alliance to keep up with growing demands of the software industry. Traditional software development methods like Waterfall model couldn't deliver the expected results on circumstances of frequent requirement changes and increased software complexity. The main reason that can be attributed to the failure of traditional development methods is the single flow of sequential development phases without any iterative phases.

For example, in Waterfall Model, the business analyst along with the client creates a set of requirements and a model of the final product. A requirement specification document is created which acts as the base document for the next development phases like Analysis, Design, Development and Testing of the product. The client is never involved during the development process and would view the product only after the final testing is completed. If the requirements of the client were not captured correctly or if the client has a changed requirement, then the entire development process has to be repeated. This kind of rework increases the cost, time and resources needed for the project and subsequently leads to its failure. ([VONE02], [ZEP07])

Agile Software Development, on the other hand, is built on the foundation of iterative approaches, in which product development happens incrementally and stakeholder participates in the entire SDLC from the conception to the delivery of the product. One such methodology is *Sprint* in which a larger functionality is broken into smaller pieces called *User Stories* which can be delivered in a time span of just two weeks. This approach has several advantages such as the time consuming documentation is replaced

with face to face communication with the stakeholders reducing the misinterpretation and hence a providing better understanding of stakeholder requirements.

Agile also provides separate methodologies for testing which makes testing easier with a quick feedback from stakeholders. A user story is marked complete if it passes all the acceptance tests. They are then evaluated whether to retain them for regression testing. A difference in interpretation of the requirement can be found out immediately with feedback and only a small rework will be needed in case of a difference. Test team members create the test plan, write test specifications and test cases and manage the testing activities within a sprint. The testing activities can be classified as follows.

2.1.1 Requirements-Based Testing

In this testing methodology, the objective is to verify whether the deliverables' design and code meets the application's requirements. Hence the test cases, conditions and data are generally derived from the requirements specification document. The testing can be done to verify either the functional attributes or the non-functional attributes like performance, reliability or usability. [TVKB01]

2.1.2 Model-Based Testing

This methodology involves the generation of test cases from models describing the system requirements and behavior. Even though this methodology requires the building of models up-front in the development cycle, it has several advantages like finding inconsistencies in the requirements itself and detection of discrepancies even before the code is generated. [DSVT07]

2.1.3 Code-Based Testing

In this methodology, test cases are used to evaluate the code to verify whether different test paths of the system are covered. The benefits of this methodology are that the parts of the software not tested in other techniques are covered. [PSVS05]

The main objective of this thesis is to simplify the existing test management process. As bulk of the effort is spent on elicitation and generation of test cases, this thesis aims to simplify the task of test case generation by using an automated tool. Most of the existing approaches for automated generation of test cases can be put under the above three categories. This thesis tries to automate test case generation for the first category which is Requirements-Based Testing.

2.2 Model Driven Software Engineering

An important paradigm shift happened in the field of software development after the introduction of Model Driven Architecture (MDA) by Object Management Group (OMG). The underlying idea of MDA is to make use of models, a key tool in engineering, for software development. In general, models can be defined as abstraction of the system and its environment. An advantage with models is that it can be used to provide different levels of abstraction, with each level highlighting specific aspects or features of the system. Hence a model is essentially a representation of the necessary characteristics of the underlying system, leaving out the rest and thus contains less complexity. Less complexity of the models provides an easy way of system behavior prediction, examination of specific properties and unambiguous communication among software developers.

One of the motivations for MDA approach is that the developed software will be deployed to one or more platforms. The volatility of the platforms is higher than the higher-level models of the system and the objective of MDA is to create models that are increasingly independent of the target platform. In MDA, Platform Independent Models (PIMs) are initially designed in a platform independent language (e.g. UML), which are then translated into Platform Specific Models (PSMs) by mapping them to some implementation language (e.g. C++, Java) as illustrated in Figure 2.1.

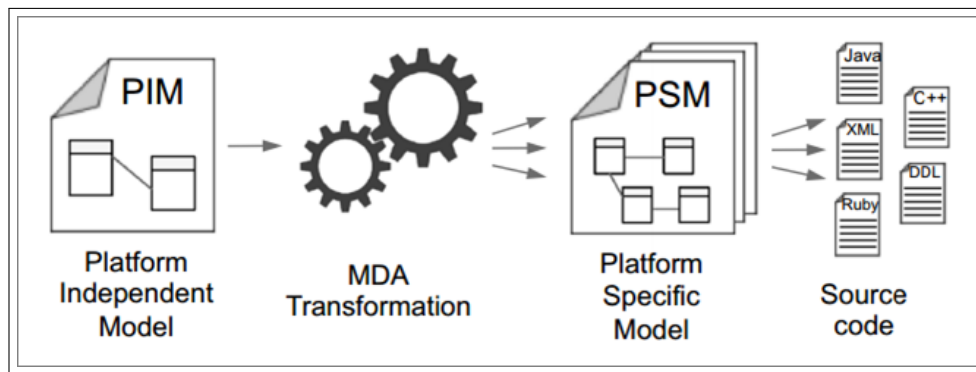


Figure 2.1: Overview of transformations in MDA [CD15]

A number of OMG standards such as The UML, XML Metadata Interchange (XMI), Common Warehouse Metamodel (CWM), Meta Object Facility (MOF) form the core of MDA concepts. Among these standards, UML is used to define the PIM models which will be discussed in detail in the Section 2.3 whereas the other standards are not in the scope of this thesis.

The term Model Driven Software Development (MDSD) or Model Driven Development (MDD) describes the family of engineering approaches that use models and their trans-

formations for creating software products. MDSD takes advantage of the MDA facilities, as a result of which the code can either be automatically or semi-automatically generated from models that are described in standard specification languages. The main priority of MDSD is to enable the validation of software by the end users and customer stakeholders as early as possible in SDLC.

2.3 Unified Modeling Language

UML is a standard from OMG and is a de-facto industry standard for modeling business applications in MDSD [CD15]. UML models help to understand the requirements of the system graphically. They also provide aides to design the system, its components and to model the relationship existing between them right from early stages of software development. UML also helps the developers and end users to maintain consistency in their interpretation of the design specification. UML's rapid gain in popularity in object oriented software development has attracted a great deal of research on UML in software testing. ([NFLJ03],[BBN03],[NFLJ03])

UML can be classified broadly into two categories namely structural and behavioral models. The structural diagrams represent the static structure of the system and the relationship between them. The different structural diagrams represent different abstraction and implementation levels. On the other hand, behavioral diagrams represent the dynamic behavior of the objects in the system i.e. the changes that happen in the system over time. Some of the important UML diagrams which are used in upcoming sections are elaborated below.

Class Diagram is one example of structural diagrams that acts as a blueprint of the system or subsystem under development. It is extensively used to model the objects that make up the system, the relationship between them, their description and the functions they provide. Class Diagrams usage is across different levels in the development cycle. In analysis stage, it is used to understand the requirements of the problem domain whereas, in the implementation stage, it can be used to create actual software classes. An example of Class Diagram is shown in Figure 2.2a.

Activity Diagram is one example of behavioral diagrams which illustrates the sequence of actions in a process. Activity Diagrams usage across different levels in the development cycle are illustrated below. In requirements stage, it can be used to model the flow of events that the use cases describe and in design and implementation stage, it can be used to define the behavior of operations. An example of Activity Diagram is shown in Figure 2.2b.

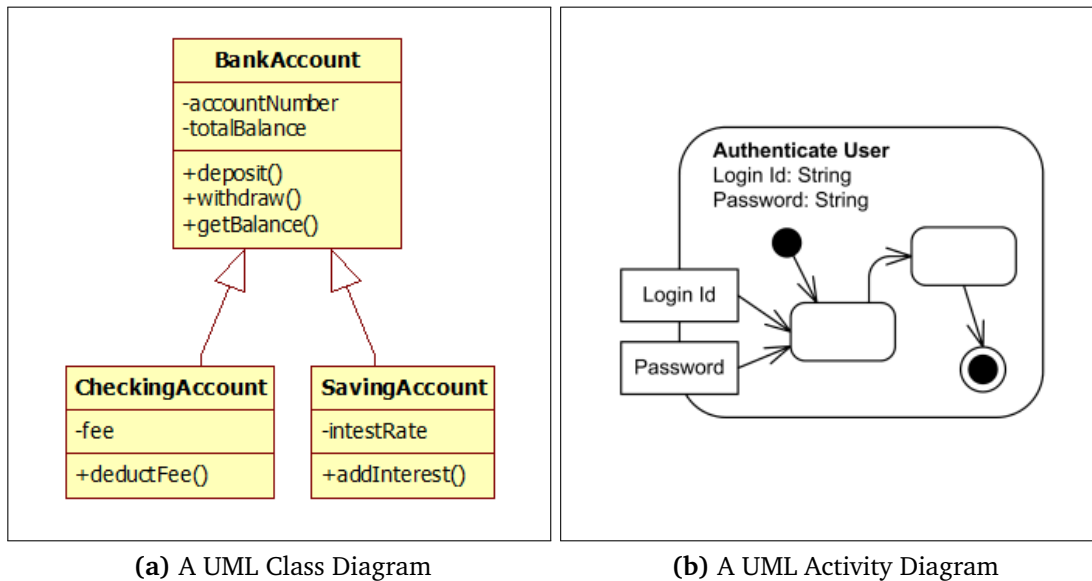


Figure 2.2: Examples of UML diagrams.

2.4 Petri Net

Petri Net is a graphical and mathematical modeling tool with well defined semantics suitable for formal analysis. The concept of Petri Net was first introduced by Carl Adam Petri in the year 1961, after which it had grown tremendously to support different domains like workflow management, manufacturing, real time systems, distributed systems, embedded systems, protocols and networks, performance evaluation and much more.

In the domain of computer science engineering, Petri nets are mainly used in information processing systems that can be categorized as parallel, concurrent, distributed, asynchronous, non-deterministic and stochastic. Petri nets can be used as communication documents like flow charts, block diagrams and networks but can also simulate the dynamic and concurrent activities of the system with the concept of tokens. Also, it can model the mathematical representations of the systems using algebraic and state equations.

2.4.1 Basics of Petri Net

There are different kinds of Petri Nets depending upon the amount of information that the nets can carry. One such example of low level Petri Net is the Place/Transition Net (PT-Net) which is used in this thesis.

2 Background

A Petri Net is an example of directed, bipartite and weighted graph in between two nodes called places and transitions. Arcs run between places and transitions and each place can hold specific items called tokens. An example Petri Net is shown in Figure 2.3 and the various elements are elaborated below.

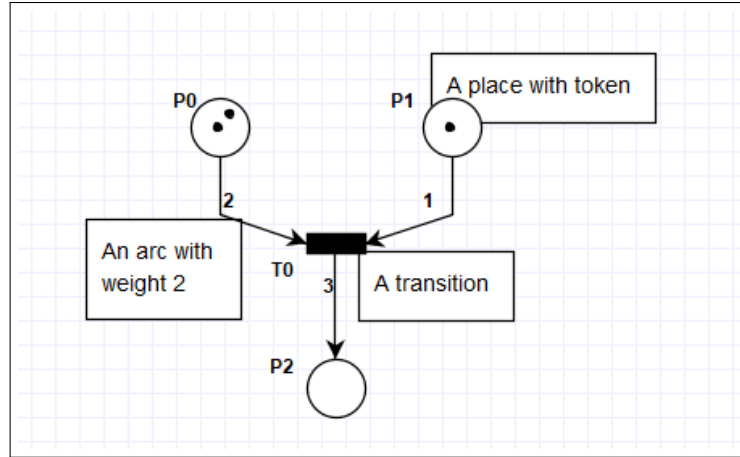


Figure 2.3: An example of Petri Net

Places are the passive component of the net, and they represent the buffer, communication medium or in general a geographical location. The current state of the system is determined by the number of tokens present in a place and this state is represented by the term *Marking* in Petri Nets.

Transitions are the active components of the net, which represent the activities that change the state of the system. Transitions are fired only when certain preconditions are met and these conditions are represented in the net by means of tokens.

Tokens are present inside the places and each token represents a condition to be fulfilled or a synchronization condition. In general, tokens represent a physical or information object.

Arcs run only between places and transitions or vice versa. In the first case, these are called *input arcs* whereas in the second case these are called *output arcs*. Each arc is also associated with a specific weight that determines the number of tokens required for firing the particular transition.

2.4.2 Formal Definition and Basic Terminology

The terms defined in the above section can be formally defined as the following. The definitions are an excerpt from Calisaya et. al., 2015 [Cal16].

Definition 2.4.1

A **place-transition Petri Net** [Rei12] is a five-tuple $PN = (P, T, F, W, M_0)$ where $P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \dots, t_n\}$ is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $W : F \subseteq \rightarrow \{1, 2, \dots\}$ is a weight function, $M_0 : P \rightarrow \{0, 1, 2, \dots\}$ is the initial marking and $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

Definition 2.4.2

For a $PN = (P, T, F, W, M_0)$, a **marking** is a function $M : P \rightarrow \{0, 1, 2, \dots\}$, where $M(p)$ is the number of tokens in p . M_0 represents the initial marking of PN .

Definition 2.4.3

A **transition** t is enabled for firing at a marking M if $M(p) \geq W(p, t)$ for any $p \in p_{in}$ where p_{in} is the set of input places of t . On firing t , M is changed to M' such that $\forall p \in P: M'(p) = M(p) - W(p, t) + W(t, p)$.

Definition 2.4.4

For a PN , a sequence of transitions $\sigma = \langle t_1, t_2, \dots, t_n \rangle$ is called a **firing sequence** if and only if $M_0[t_1], [t_2], \dots, [t_n]M_n$. In notation, $M_0[PN, \sigma]M_n$ or $M_0[\sigma]M_n$.

Definition 2.4.5

For a $PN = (P, T, F, W, M_0)$, a marking M is said to be **reachable** if and only if there exist a firing sequence σ such that $M_0[\sigma]M$.

2.4.3 Analysis of Petri Nets

One of the main features of Petri Net is the ability to perform analysis on model properties i.e. it can detect defects related to structural and dynamic properties [Mur89]. A simple transversal of the flow relation between places and transitions can detect the structural properties whereas initial markings and final markings after transitions can detect the dynamic properties. As defined in [Rei12], the defects due to dynamic properties of boundedness, liveness, deadlock free and reachability can be detected using methods like simulation, reachability/coverability or invariant analysis.

The transition behavior in a Petri Net is shown in Figure 2.4. Figure 2.4 (a & b) show Petri Nets with markings M_0 and M_1 respectively. P_0 and P_1 are the places and T_0 is the transition. The places contain tokens and arcs with arc weight of two and one respectively. This marking, which is the required precondition, enables the firing of the transition T_0 and subsequently results in the marking as shown in Figure 2.4b. The final marking consists of place P_2 with three tokens since the arc from T_0 to P_2 is of weight 3.

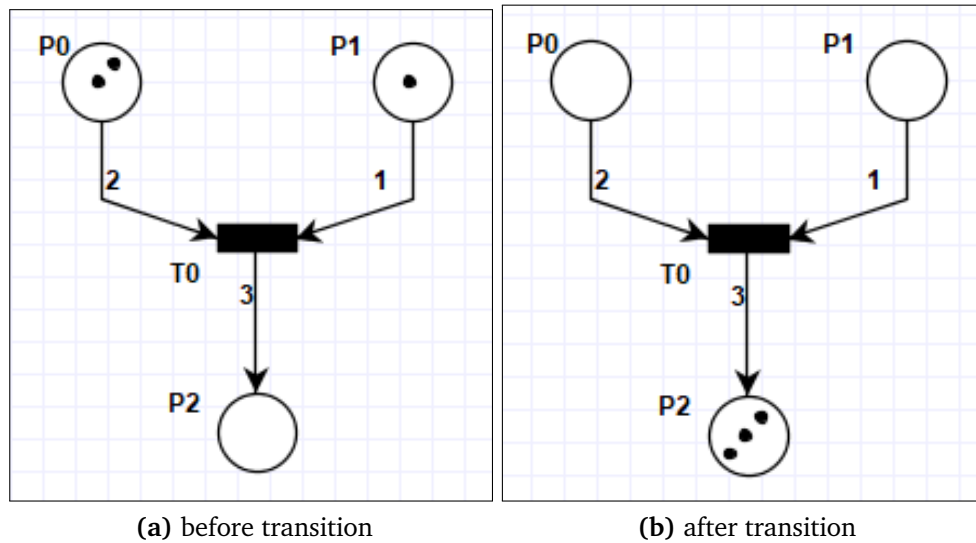


Figure 2.4: Concept of transitions in a Petri Net.

3 Related Work and Utilized Software Tools

The foundation for every research work is provided by other related research work in the same domain. The first section of this chapter provides an overview of such related research works for test case generation. The second half of this chapter focuses on some of the technologies used for implementing the proposed approach.

3.1 Related Work

This section provides an overview of already existing research and approaches in the domain of automatic test case generation. It also describes the way in which the proposed approach differs from the already existing approaches and how the disadvantages of the already existing approaches are eliminated. These related research works are grouped according to the type of inputs needed for test case generation.

3.1.1 Model Based Testing Related

The following section describes the related research for automatic test case generation using a formal model of the requirements as an input artifact. Most of these approaches use UML diagrams as a formal model.

Shirole et. al. [SK13] gives an overview of different existing methodologies for test case generation from UML models. Most of these approaches need the requirements to be converted into UML behavioral models like state-charts ([RG99b], [BG09]), sequence diagrams ([NFLJ03], [LJX+04], [BL02]) and activity diagrams ([NFLJ06], [NS12]). Some of these approaches are elaborated below.

Tahat et. al. [TVKB01] proposed a technique to generate test cases from requirements defined in System Description Language (SDL) ([ALH+12], [BPBM97], [BH95]). Initially, each requirement is first converted into a SDL system model and then a combined SDL model for the entire set of requirements is created manually. This combined model

is converted into Extended Finite State Machine (EFSM). This acts as an input artifact for the test case generation process.

The drawbacks of the above approaches are as follows. The creation of formal models is common only in case of object oriented applications and there are domains like embedded systems where models are not so common. In such scenarios, creation of formal models is an overhead and an approach that could evade this is greatly appreciated. Even in industries that have embraced MDA approach, the creation of precise behavior models is expensive and hence it is not a frequently used practice.

Our work differs from all the above mentioned methodologies in that it does not need a formal model either for test creation or for obtaining test data and is completely automated that it does not require any manual intervention in any stage of test case generation.

3.1.2 Natural Language Processing Related

Different approaches that generate test case using Natural Language Processing (NLP) techniques are present but mostly these methods require additional behavioral model inputs like state diagrams [RG99b], activity diagrams [HGB08], labeled transition diagram [KK06] or they need the provision of additional inputs such as test data or manual test derivation [BG03]. There are also several other approaches existing ([FL00], [YBL10], [YBL13]) for generating UML models from NL requirements. They can also be easily adapted for test case generation.

The approach proposed by Kulkarni and Joglekar [KJ14] makes use of NLP tools to parse the requirements written in natural language and convert them into graph-based structures called Knowledge Representation. This graph is then analyzed to create test cases. The objective of authors Ryser et. al. [RG99a] is to create system test cases from scenarios expressed in NL. But since scenarios described in natural language tend to be more ambiguous, an intermediate formal representation is created. State Machines are used as a formal representation, which are then used for test case generation.

Barros et. al. [BNHT11] makes use of use case specifications to generate test cases. But the language used to define the specifications is controlled i.e. only part of natural language grammar and structure can be used. In this work, this controlled language is called ucsCNL and test cases are automatically generated from use case specifications.

Our model makes use of NL requirements as inputs but does not require the formal intermediate UML models and presence of domain experts to verify these models. Instead, it is a completely automatic process where a NL requirement input results in corresponding executable test scripts.

3.1.3 Keyword Driven Testing Related

Keyword Driven testing (KDT), also called Table Driven Testing or Action Word Testing, is a testing methodology which defines keywords to describe operations and actions. It is greatly suited for test automation and there exists a variety of approaches that can generate executable test scripts. Even though the following literatures have not directly contributed to the test case generation in our work, they were of immense help in defining a test script generation methodology.

Tang et. al. [TCM08] has proposed a methodology in which test cases can be derived for KDT. This technique even produces test scripts suitable for different test applications or System Under Tests (SUTs) under different environments. Keywords identify the atomic operations in test execution and are very much dependent on their application. For example, *Click* and *Connect* are the keywords for GUI and Database Applications respectively. Test cases are created by the users by specifying a sequence of keywords i.e. the actions and these test cases are then converted into test scripts which act as an input to the test driver specific to an environment.

Little and Miller [LM06] proposed a similar approach where keyword commands are converted to executable test scripts. Hametner et. al. [HWZ12] suggested a test case generation technique for KDT in the field of industrial automation systems. Here, test engineers write test cases manually using predefined keywords and a predefined tabular template in a high abstraction level. The automated generation of test cases is not supported in this technique but can be easily adapted. Thummalapenta et. al. [TSSC12] put forth a technique in which test scripts can be created from manually written test cases which consist sequences of tuples like *action-target-data*.

3.1.4 Restricted Natural Language Related

This section describes an approach which is a hybrid between different approaches seen above. Wang et. al., 2015 [WPG+15b] describes a semi-automatic approach for generating test cases. The approach is labeled as Use Case Modeling for System Tests Generation (UMTG). The requirements are specified in a restricted natural language in a defined template called Restricted Use Case Modeling (RUCM) [YAZ15]. They are used as one of the inputs whereas the other input is the domain model of the system which has to be manually developed. This approach also needs the restrictions of the model to be modeled manually as Object Constraint Language (OCL) constraints. The workflow of this approach consists of four manual stages out of its total seven stages. Zhang et. al., 2014 [YAZ15] proposes another method for test case generation using RUCM called

Restricted Test Case Modeling (RTCM) which again uses textual transformations with less priority to test data generation.

3.2 Utilized Software Tools

3.2.1 Eclipse Modeling Project

Eclipse Modeling Project [ECL01] is a collection of tools and frameworks that can be used for model-based development. The principal element of Eclipse Modeling Project is the Eclipse Modeling Framework (EMF) which is used for describing, creating, and evaluating structured information. It provides tools to generate a set of Java classes and also provides features that enable command based editing of a model using adapter classes. In general, for any application, the data model is expected to be independent of the logic or user interface. Similarly, EMF comprises a set of plug-ins which helps to create data models independently and in addition, it helps to generate the code or any other expected output from these data models. The added advantage with EMF is that the Java code can be regenerated from the model at any time. Also, the metamodel can be created in different representations such as XMI, UML, etc.

3.2.2 Eclipse QVT

Model-to-Model (M2M) transformation is a key technique in Model Driven Software Development and QVT facilitates such model transformations. QVT stands for Query/View/-Transformation and it has a standard set of languages that help in model transformations [EQVT10]. QVT languages are compliant with OMG standard and there are totally three different model transformation languages. QVT Operational which is designed for writing unidirectional transformations, QVT Relations that allows both unidirectional and bidirectional model transformations and QVT Core which is a simple language designed to act as a translation for QVT Relations. Finally, an additional mechanism called QVT Black Box is used to invoke transformation facilities stated in other languages. Figure 3.1 gives an overview of the different languages and its relations as defined in the standard. Among the different languages, QVT Operational is used in this thesis work since the transformation rules are written in an imperative language similar to other programming constructs.

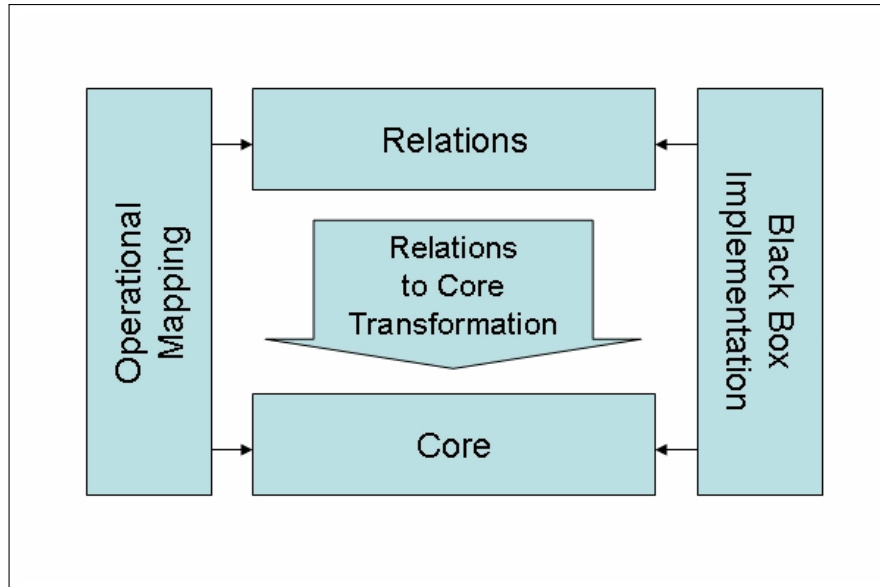


Figure 3.1: Overview of different languages in QVT [EQVT10]

3.2.3 Xpand

Xpand is a language released as an Eclipse plug-in for generation of code from EMF models [XPAND07]. In Model Driven Software Engineering (MDSE), once the models are developed, it is difficult to directly execute them. For further processing, it is required that these models should be converted into executable artifacts. The Xpand language is used for such a transformation. Though there are many template languages available, Xpand is focused on effective template development for code generation with the help of a domain-specific Model-to-Text (M2T) transformation language and good tool support. Here the templates are written by the user which are then used by the generator that runs the development workflow. In addition, the Xpand language has an inbuilt, small but sufficient vocabulary. Xpand's domain-specific features and its familiarity in high level MDSD makes it an automatic choice over other languages.

3.2.4 PIPE2

Platform Independent Petri Net Editor (PIPE2) is an open source, platform independent tool developed for the creation and analysis of Petri Nets [BLPK07]. PIPE2 was developed as a research project at the Imperial College in 2003. It is completely written in Java to make it platform independent. It also provides a simple user interface which creates, saves and loads Petri Nets in conformance with Petri Net Modelling Language (PNML) interchange format. The major advantage of PIPE2 is its user friendly GUI, its platform

independence and its easy extensibility feature. PIPE2 also features a complete set of analysis modules to check behavioral properties, derive performance statistics and other capabilities like Petri Nets comparison and classification. It also offers modules to perform different kinds of qualitative and quantitative analysis. The research group continuously works on additional enhancements to make the tool much more effective for the Petri Nets creation and analysis.

4 The Proposed Solution

This chapter gives a detailed explanation of the proposed approach and the various steps involved in it. It elaborates on the theoretical design behind the conversion of agile requirements described in natural language into formal models and the subsequent test case generation methodologies.

The first section of this chapter gives an overview of the proposed solution and the different steps involved in it. The following sections elaborate the steps defined in the overview and the justification on which the theoretical ideas are based.

4.1 Overview of the Solution

The proposed solution of test script generation from requirements can be broadly classified into five important steps.

1. Requirements Elicitation in Agile.
2. Requirements in a Restricted Natural Language.
3. Generation of Petri-Nets from Use Case Specifications.
4. Generation of Reachability/Coverability Graph.
5. Generation of Test Specifications and Test Scripts.

Figure 4.1 shows the skeleton of the proposed work. The detailed explanation of these steps can be found in the following sections.

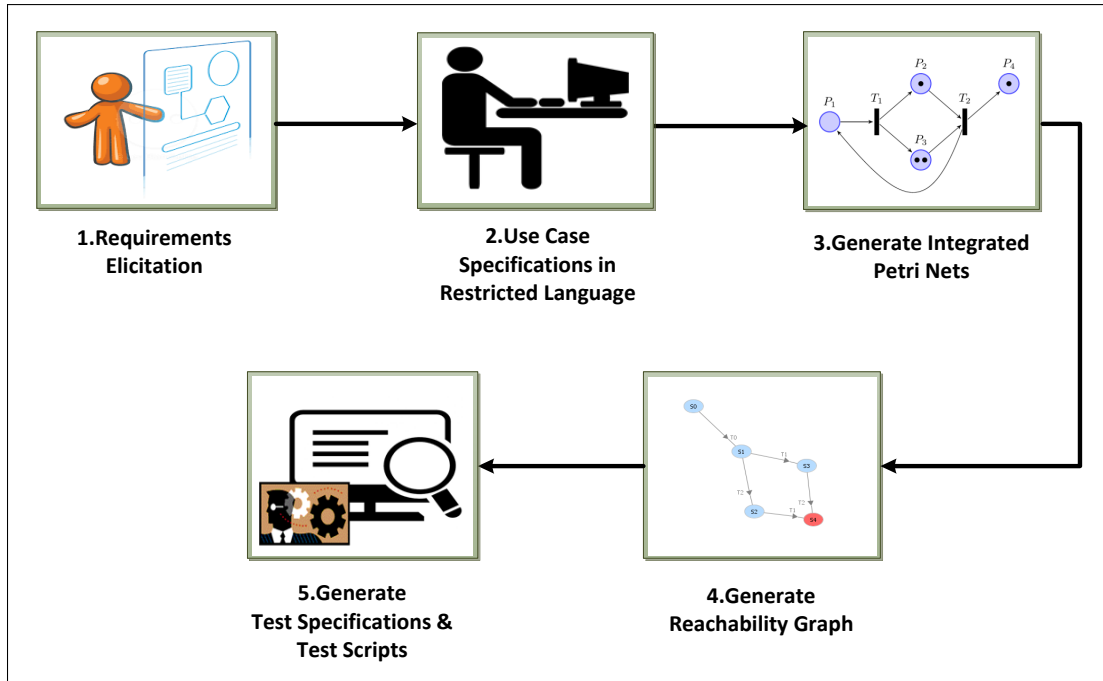


Figure 4.1: Overview of the proposed solution

4.2 Requirements Elicitation in Agile

The Requirements Engineering process often dictates the quality of the project and hence it plays a vital role in any software development process. Agile method, which relies on iterative development, embraces changing requirements during the software development process. Hence a long and detailed documentation process as in traditional Requirements Engineering process is useless. In the Agile process, one of the frameworks common in the industry is the *Scrum*. Here once the requirements are elicited, they are split into smaller features defined in the form of *User Stories* or *Use Cases*. The choice between the two is dependent on the project since both the representations have its respective pros and cons. This thesis makes use of Use Case for representing requirements.

Next step is to analyze the elicited requirements for its various properties like atomicity, uniqueness, completeness, consistency and unambiguousness, testability, etc [Guru13]. There are approaches ([SLA15], [Cal16]) that analyze some of these properties by converting the requirements into formal models and then applying NLP techniques. Our proposed approach can also be used to analyze the property *testability* by generating executable test cases directly from the requirements using formal models. In a SDLC, it is the imperative to ensure the quality of requirements as early as possible to reduce the

cost of software development and to meet the scheduled deadlines and our approach adds to this objective.

4.3 Requirements in a Restricted Natural Language

The main objective of this approach is to generate test cases directly from requirements specified in natural language. In the industry, one of the important documents used to communicate requirements between the different stakeholders is UCSs. In order to reduce the incompleteness and imprecision in UCSs, a template with restriction rules are usually used, thereby making it a semi-formal representation.

4.3.1 RUCM – A Restricted Natural Language

RUCM [YBL13] is one such use case format that provides restriction rules and specific keywords, hence confining the use of natural language for UCSs. It also corresponds to a specific metamodel called UCMeta, which facilitates automatic analysis of use cases. The selection of a restricted NL template depends upon factors like simplicity, expressive power, better understandability and quality of models derived from it. Experiment results [YBL13] shows that the template is beneficial and is applicable in various scenarios with the above mentioned characteristics.

4.3.2 Restriction Rules and Keywords

RUCM has also defined a list of standard keywords. A short overview of the rules is given below. For example, a conditional logic sentence is defined by the keyword IF-THEN-ELSEIF-ELSE-ENDIF, a concurrency statement by the keyword MEANWHILE, condition checking sentences by the keyword VALIDATES THAT and iteration sentences by DO-UNTIL. RESUME STEP keyword is used when the alternative flow should return back to its reference flow. Similarly, ABORT keyword is used to describe an exit action due to some exceptions. The complete set of keywords can be found in the template reference given in [YBL13]. In addition to the keywords, RUCM also specifies restriction rules that are detailed in [YBL13]. A table describing the basic elements of a RUCM template is given in Table 4.1.

RUCM was basically created for analysis of use cases and not for test case generation. Hence few additional restrictions were added to RUCM by the authors of [WPG+15a]. In order to establish composite conditions with multiple specific alternative flows, the

Table 4.1: RUCM Use Case Template [YBL13]

Element	Description	
Use Case Name	The name of the use case. It usually starts with a verb.	
Brief Description	Summarizes the use case in a short paragraph.	
Precondition	What should be true before the use case is executed.	
Primary Actor	The actor which initiates the use case.	
Secondary Actors	Other actors the system relies on to accomplish the services of the use case.	
Dependency	Include and extend relationships to other use cases.	
Generalization	Generalization relationships to other use cases.	
Basic Flow	Specifies the main successful path, also called “happy path.”	
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the basic flow executes.
Specific Alternative Flows	Applies to one specific step of the basic flow.	
	RFS	A reference flow step number where flow branches from.
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.
Global Alternative Flows	Applies to all the steps of the basic flow.	
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.
Bounded Alternative Flows	Applies to more than one step of the basic flow, but not all of them.	
	RFS	A list of reference flow steps where flow branches from.
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.

authors decided that the specific alternative flow should also begin with the keyword IF....THEN. Further additions to the restriction rules can be found in [WPG+15a]. This work has also made some assumptions and changes to the RUCM template which will be detailed in Chapter 5.

4.3.3 Comparison with a related RNL

Another widely used Restricted Natural Language Specification is the Scenario Specifications. It also uses restricted vocabulary and introduces restriction rules similar to RUCM. According to the author in [Cal16], a scenario is a collection of partially ordered event occurrences, each guarded by a set of conditions (pre or post) or restricted by

constraints. Further details on the scenario specifications can be found in the literature ([PHDK00], [Cal16]).

An initial comparison between the two templates revealed the following conclusions. Scenario Specification is deprived of certain properties like the metamodel defined is not exhaustive and hence it cannot be used for complete test case generation. RUCM is more suited for safety critical embedded software and an industrial case study with this template yielded good results [WPG+15a]. Test data generation is easier with the help of RUCM since natural language processing can be introduced early in the process because of the detailed metamodel.

Scenario Specification, on the other hand, has been used to specify requirements in natural language and subsequently to automate the check of correctness, consistency, completeness and unambiguity properties of the requirements in [Cal16]. The work in the mentioned literature also makes use of Petri Nets as a formal model for the analysis. This work uses RUCM as a template for Restricted Natural Language (RNL) requirement specifications and the work using Scenario Specifications as a foundation for transformation to formal models.

4.4 Generation of Petri Nets from Use Case Specifications

One of the main objectives of this work is to establish a formal model for the requirements specified in natural language. The formal model in our work is Petri Nets, to be precise Place-Transition Petri Net, which is a sub category of Petri Nets. To establish the conversion from UCS to PN, we use the concept of Model-to-Model (M2M) transformations. For this purpose, the UCSs are described as a metamodel and we use RUCM as a template for this. Petri Net, on the other hand, can also be described as a metamodel.

The transformation method from UCS to PN consists of three main steps.

1. The first step defines mapping rules that translate the RUCM elements into corresponding Petri Net elements according to transformation rules defined in Appendix A.1. The expected result of this process is a Sub Petri Net which consists of at least one place, transition and arc.
2. The second step is to compose the sub Petri Nets generated from the previous step into a whole Petri Net. This is done by the fusion of output and input dummy places between sequential sub Petri Nets or by integrating Petri Nets of other referenced UCSs.

3. The third step is to generate an integrated Petri Net from the partial Petri Nets generated for each UCS. According to RUCM specification, each UCS can refer to another UCS and hence the need arises to transform referenced UCS into a Petri Net. This step integrates the partial Petri Nets into an Integrated Petri Net and hence it reflects the properties of the synthesized partial Petri Nets.

The different transformations and metamodells are detailed in Chapter 5 along with a running example. As a result of the above transformations, we obtain a Petri Net that is equivalent for the given UCS.

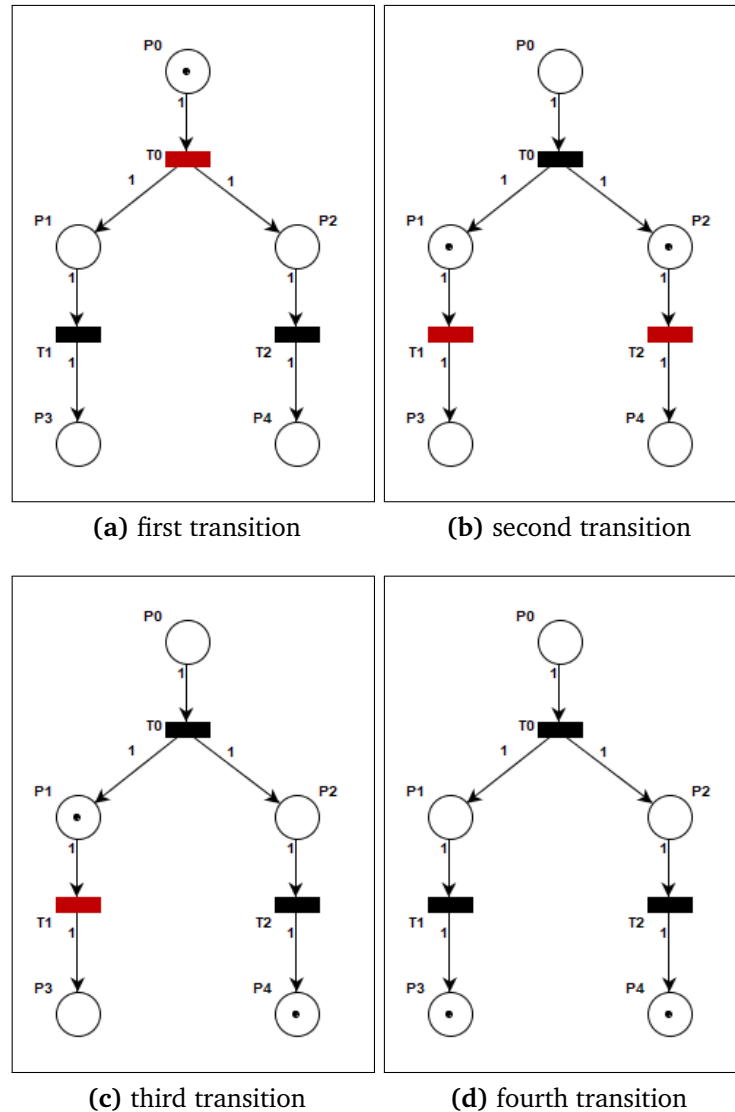


Figure 4.2: Concept of reachability.

4.5 Generation of Reachability/Coverability Graph

As mentioned in Chapter 2, one of the important properties of Petri Net for dynamic analysis is reachability. As already described, when a transition is enabled for firing, tokens are transferred between places. This causes a change in marking of the Petri Net and each marking is defined by the term *state*. Thus a new state is reached from the initial state. A *reachable state* is defined as a state that can be reached from the current state (marking).

The Figure 4.2 explains the above process with the help of Petri Nets. The firing sequence of transitions described in the figure is $\{T0, T2, T1\}$. Initial marking of the Petri Net is given by $\{1, 0, 0, 0, 0\}$ which corresponds to $\{P0, P1, P2, P3, P4\}$. After the first firing, the marking of the Petri Net changes to $\{0, 1, 1, 0, 0\}$. As defined, these markings are also termed as states and in our case, they are termed as $S0$ and $S1$ respectively. Hence it can be seen that $S1$ is a reachable state from $S0$.

A reachability or coverability graph is a graph that contains reachable markings or states as nodes and transitions, which cause the flow of tokens, as arcs. This graph contains all the possible reachable states from the initial state and can even have multiple paths. Reachability graph for the configuration given in Figure 4.2a is shown in Figure 4.3. Here, it can be seen that for the state $S1$ there are two possible reachable states namely $S2$ and $S3$ and hence contains two paths. The path described in Figure 4.2c is $\{S0, S1, S2, S4\}$ whereas there is an additional path $\{S0, S1, S3, S4\}$ in the reachability graph.

4.6 Generation of Test Specifications and Test Scripts

The Reachability Graph acts as an input for the generation of Test Specifications and Scripts. A Reachability Graph is composed of tangible and vanishing states with transitions (from Petri Nets) acting as arcs. A tangible state is a final reachable state from a given initial state whereas a vanishing state is a state that is traversed before reaching a tangible state. Now each path from the initial state to a tangible state forms a test scenario since each arc represents a transition from Petri Net. A test scenario is a sequence of actions performed to reach a particular state. In order to make these test scenarios into complete test specifications, we add other elements like Pre-Conditions, Post-Conditions and conditions from logical statements, etc.

The test specifications consist of a set of test cases and each test case, in turn, is made up of a set of test steps. In practice, the test scripts are realized as a set of test steps that are implemented as packages in platform specific languages. A mapping table is created to map each of the test steps to specific packages in platform specific languages.

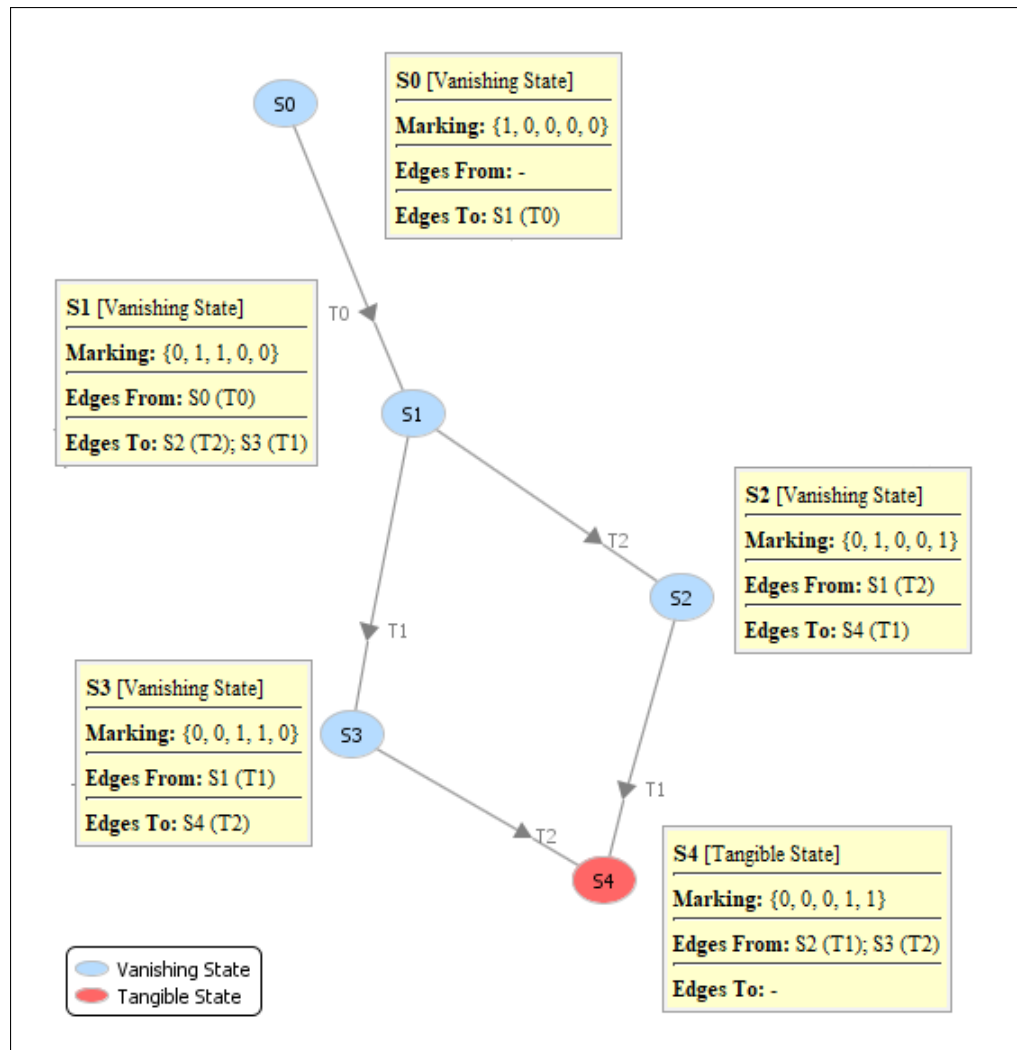


Figure 4.3: An example of Reachability Graph

For test data, the numerical values from test steps are extracted and used for test script generation.

5 Design and Implementation of the Tool

This chapter speaks in detail about the Test Script Generation Tool (TSGT). It elaborates on the design and implementation details of the tool. The entire implementation is explained with the help of a running example.

The chapter can be broadly classified into two sections. The first section speaks about the various objectives that the system has to fulfill, following that the constraints regarding the design are discussed and finally the tool design. The second section explains the various implementation details and intermediate artifacts.

5.1 Design of the Tool

The following sections elaborate on the different aspects of the tool design and the different considerations made for it.

5.1.1 Design Goals

The high level objective of the tool is to automate the test script generation process using any formal method. While analyzing the requirements of the tool in detail, following salient objectives of the tool are identified.

- The main goal of the TSGT is to generate test scripts for system testing from requirements defined in the natural language.
- The tool should also provide means for extension so that test script generation for component and unit level testing can be accommodated.
- The primary output of the tool has to be automatically generated Test Scripts that can be directly executed on a given system. The tool should also provide Test Specifications as an intermediate output.
- The tool should be able to provide a formal model as an intermediate output. The objective is to use the formal model for other processes in SDLC.

- The intermediate output should be represented in a standard format so that the artifact is compatible with other tools already available in research and industry.
- The transformation between different models should have defined rules and should be implemented using established processes. This is to ensure that the same transformation does not yield different outputs.
- The tool must improve the coverage of the requirements than the manual process of test case generation.
- The tool should be automated as much as possible to reduce the time taken for test script generation.
- The tool must try to reduce the number of intermediate input artifacts which should be manually created. This ensures increased automation.

5.1.2 Design Constraints

In this section, some of the constraints that hinder the tool implementation are discussed. This section also gives an overview of the considerations made in the design to circumvent these constraints.

One of the most important constraints is that the metamodel provided for RUCM is too detailed. This increases the complexity of converting it into a formal model i.e. Petri Net. Therefore, a less complex metamodel is created which addresses some of the main required features. This metamodel can be considered as a subset of the original metamodel. Some of the rules in RUCM are also replaced with new rules according to the requirements of our current implementation. A brief overview of the changes is given in the respective sections.

Next important constraint is the need for an intermediate input artifact that describes the environment for which the test script has to be generated. In industry, usually the test scripts are implemented using tools and a mapping is needed to interface our implementation with the specific tool implementation. Hence for this purpose, an implementation similar to a Lookup table is used in our tool design.

5.1.3 System Design of the Tool

Figure 5.1 shows the high level system design of the tool. The design tries to incorporate the design goals described in the previous section taking into account the different limitations. The design can be divided into three blocks based on the platform on which

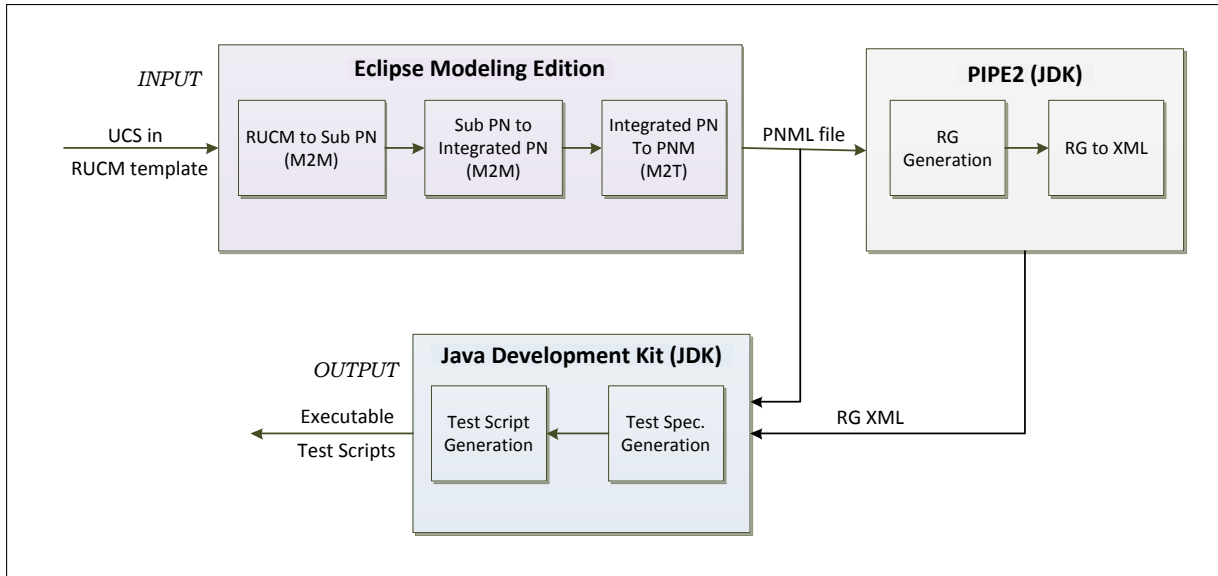


Figure 5.1: Overview of different components in the tool

they are implemented. The three blocks are implemented in Eclipse Modeling Edition, PIPE2 and Java Development Kit (JDK) respectively. The input of the system is UCS in natural language and the outputs are executable test scripts. Figure 5.1 also gives the details of different sub components in each block.

5.2 Implementation of the Tool

This section describes the implementation of various blocks defined in the previous section.

5.2.1 Implementation in Eclipse Modeling Edition

Metamodels Developed

This section explains in detail about the different metamodels that are developed for the implementation of the tool. The metamodels are developed using the Eclipse Modeling Project [ECL01].

Petri Nets in our tool is based on the metamodel shown in Figure 5.2. As explained earlier, it consists of Places, Transitions and Arcs.

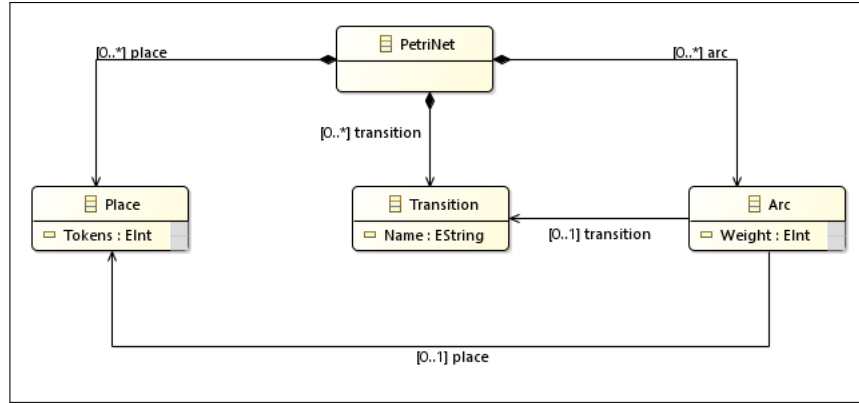


Figure 5.2: Petri Net Metamodel

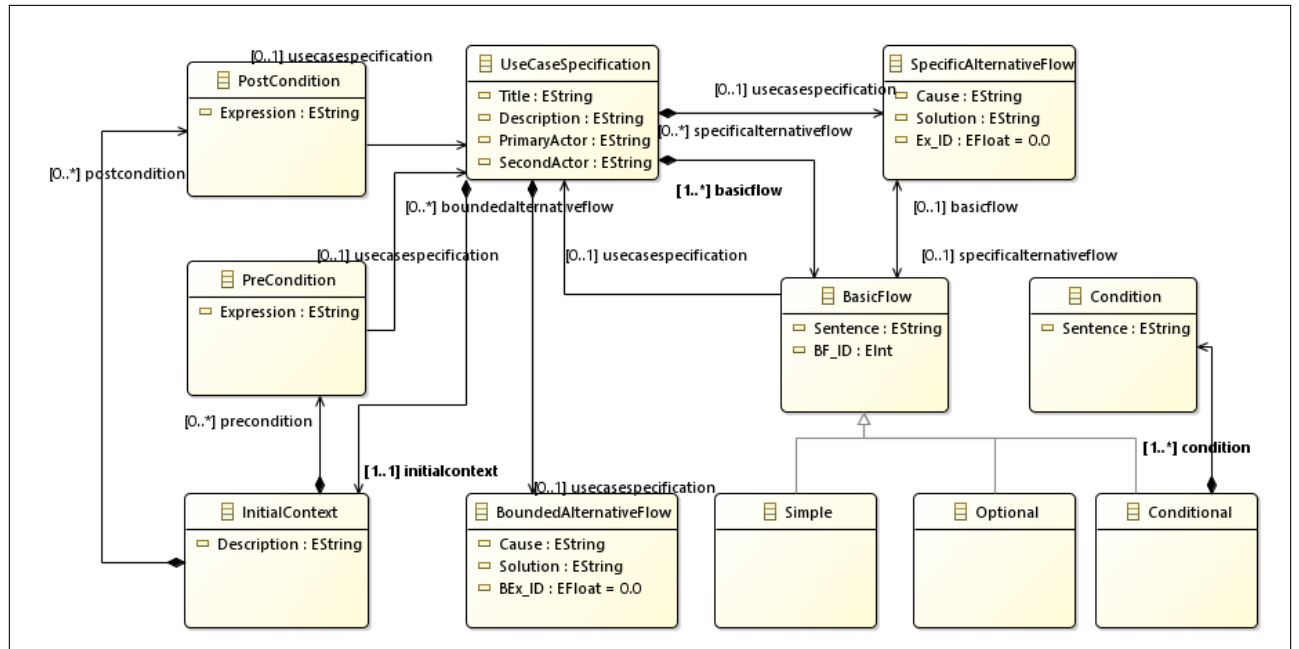


Figure 5.3: RUCM Metamodel

The second metamodel developed is for the RNL template. The RNL used in our case is RUCM and the metamodel for its template is shown in Figure 5.3. Here some customization according to our design goals is made. The developed metamodel is a simplified version of the metamodel in [YBL13] and models only the key features necessary for our tool. It is based more on the metamodel provided in [Cal16]. Some changes are made to the restriction rules such as the rule that mandates each alternative flow should begin with a conditional statement is added and the rule that mandates the presence of post condition in each basic flow is withdrawn.

Running Example

The tool implementation described in the next sections is explained with the help of a running example. The system under consideration is Lane Management System (LMS) from Automotive Software. The LMS is used to enhance the safety of an automobile. The purpose of the system is to help the driver to keep the vehicle in their lane to avoid crashes.

Table 5.1: Running Example

<i>Element</i>	<i>Description</i>
Use Case Name	<i>Disable System</i>
Brief Description	The LMS should be turned off at certain conditions.
Precondition	1. LMS is activated
	2. Vehicle Speed is greater than 15 kmph
Primary Actor	Driver, LMS Control Unit
Secondary Actors	LCS, LKS and LWKS Control Units
Basic Flow	1. The driver deaccelerates the vehicle.
	2. The driver accelerates the vehicle.
	3. The driver drifts away from the current lane.
	4. The driver activates the LKS system.
	5. The LKS steers the vehicle.
	6. The vehicle is moving through a road under repair.
Specific Alternative Flows	1.1 IF Vehicle Speed less than 15 kmph THEN deactivate LMS system with an audio alert.
	3.1 IF the departure is unintentional THEN create an audio alert and deactivate LMS system.
	5.1 IF the driver manually takes over THEN raise an alert and deactivate LKS.
	6.1 IF the road markers are unclear THEN raise an alert and deactivate LMS

The LMS consists of three main components namely a Lane Centering System (LCS), a Lane Departure Warning System (LDWS) and a Lane Keeping System (LKS). The LDWS alerts the driver in case of unintentional lane changes. The LCS and LKS will work together to take control of the vehicle and keep the vehicle in driver defined center of the lane. Table 5.1 is an extract from the Software Requirements Specification (SRS) document of LMS and it illustrates a specific Use Case called Disable System. The template used for specifying this Use Case is RUCM template and it is based on the metamodel described in the previous section.

To give inputs to the tool, we should create an instance of the RUCM metamodel. The running example is given as input for the tool and the structure of the user input can be seen in Figure 5.4.

Model Transformations

This section describes how the UCS in RUCM metamodel is converted into an integrated Petri Net. Model transformations, which are the core of the MDSE technique, are used for this purpose.

The first step of the transformation process is to transform elements from RUCM metamodel to Petri Net metamodel elements. For this purpose, the concept of M2M transformation is used and it needs specific rules for transformation from one metamodel to another. In our tool, these rules are implemented using QVTo [EQVT10].

Once this transformation is executed, we get a collection of Sub Petri Nets. These should be integrated again into a single Petri Net using the rules defined in Section 4.4. These rules are again implemented using QVTo as a M2M transformation. Appendix A.2 shows the implementation of these rules.

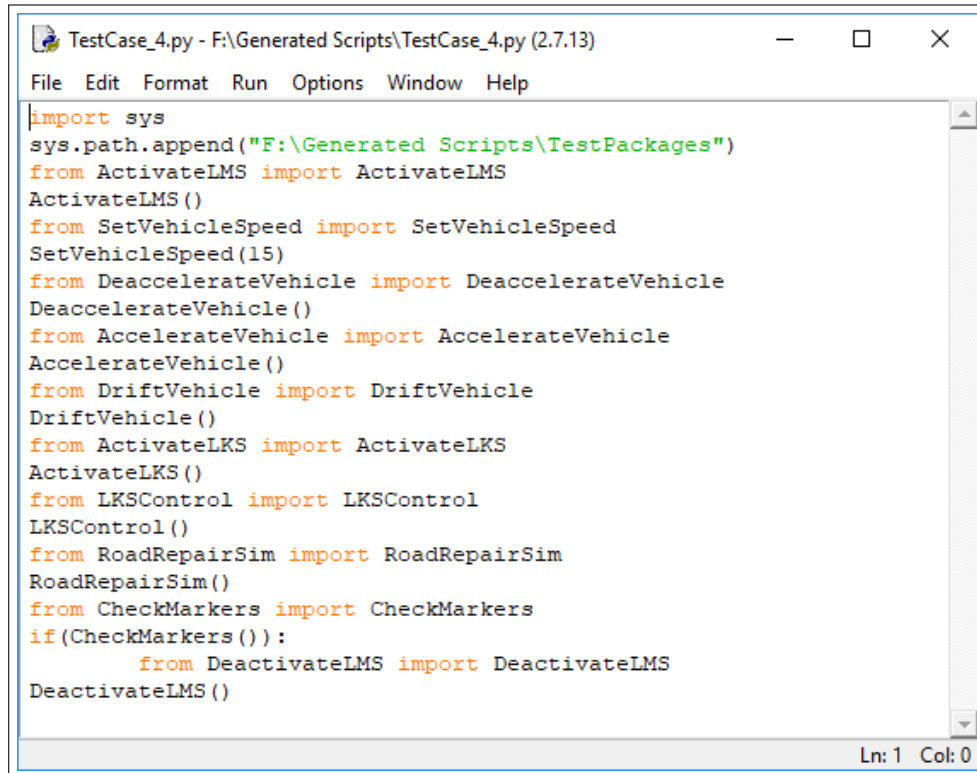
The output of the above transformation yields an integrated Petri Net. In order to make this output compatible with other Petri Net tools, the available Petri Net is converted into a standard xml file called PNML. This transformation from Petri Net metamodel to textual representation is done using M2T transformation implemented in Xpand language [EM2T07]. Appendix A.3 gives the details of this implementation. A graphical representation of the intermediate output for our running example in PIPE2 tool is shown in Figure 5.5.

5.2.2 Implementation in PIPE2 tool

This block generates the reachability graph. The integrated Petri Net in the form of the XML file is given as input to the PIPE2 tool. From this, the reachability graph is generated using the corresponding menu in the tool. In order to use this graph in the subsequent steps, it is exported as an XML file. For this, a package is added to the existing tool and this exports the XML file in a specified template. The implementation is done using JDK.

5.2.3 Implementation in JDK environment

This block is responsible for the generation of test specifications and test scripts. Inputs to this section are the Reachability Graph and the PNML file. The test specification is generated by identifying the reachable paths from the Reachability Graph and then the conditions from the PNML file are added. For identifying all reachable paths, an *All Path Search* is done from initial state to tangible states using *Depth First Search Algorithm*.



```
TestCase_4.py - F:\Generated Scripts\TestCase_4.py (2.7.13)
File Edit Format Run Options Window Help
import sys
sys.path.append("F:\Generated Scripts\TestPackages")
from ActivateLMS import ActivateLMS
ActivateLMS()
from SetVehicleSpeed import SetVehicleSpeed
SetVehicleSpeed(15)
from DeaccelerateVehicle import DeaccelerateVehicle
DeaccelerateVehicle()
from AccelerateVehicle import AccelerateVehicle
AccelerateVehicle()
from DriftVehicle import DriftVehicle
DriftVehicle()
from ActivateLKS import ActivateLKS
ActivateLKS()
from LKSControl import LKSControl
LKSControl()
from RoadRepairSim import RoadRepairSim
RoadRepairSim()
from CheckMarkers import CheckMarkers
if(CheckMarkers()):
    from DeactivateLMS import DeactivateLMS
    DeactivateLMS()
```

Figure 5.6: The final output of the TSTG tool

Once a test specification is ready, each test case in the test specification is elicited out and the test steps for the test cases are generated using an implementation similar to a Lookup table. This Lookup table maps each test step to a package predefined in python and finally, executable test cases are generated in python. Figure 5.6 shows a generated test script for one of the test cases of our running example. The entire implementation is done in JDK using Eclipse Java Standard Edition.

6 Evaluation of the Tool

This chapter gives detailed information on how our tool is evaluated. Evaluation of the tool is done on the basis of parameters used by similar tools to access the quality of generated test scripts. These parameters facilitate correlating our tool with other tools.

The first section of the chapter speaks on the various methods and criteria used for evaluation. The second section of this chapter discusses the different results obtained from the evaluation and also compares our tool with similar tools available in the research domain.

6.1 Preliminary Preparation for Evaluation

This section gives an idea of the system to be used for evaluation, the different criteria and the procedure used for evaluation.

6.1.1 System for Evaluation

For evaluation of TSGT tool, we use a system called LMS. The LMS is safety critical automotive software used in most of the present-day automobiles. The SRSs for LMS provided in [BDFM] serves a foundation for our evaluation. This document provides an overview of the entire system and the various requirements needed for system development. It also provides detailed requirements for specific functionalities and in addition, includes models describing the functionality and interactions of the system. The requirements are organized in the form of use cases and there are totally 13 use cases which act as inputs for our tool.

6.1.2 Evaluation Criteria

The evaluation of the tool is based on two main criteria.

1. The time taken for the generation of test scripts.
2. The coverage of requirements by the generated test cases.

Time Criteria

Definition 6.1.1

The effort required to generate a test case is defined as the average time taken to generate the test cases [EMB15].

$$\text{Effort}_{TC} = \frac{\text{Time taken to generate the test cases}}{\text{Total number of generated test cases}}$$

In case of the automated process, the time taken to generate the test cases is inclusive of the time that is required to specify the requirements in RNL, whereas in case of the manual process, it is inclusive of the time taken to convert the test cases into executable test scripts.

Definition 6.1.2

Test Case Productivity is defined as the ratio of the number of test steps generated to the time taken (in minutes) to generate these test steps [Gul09].

$$\text{Test Case Productivity (TCP)} = \frac{\text{Number of test steps generated}}{\text{Total Time taken to generate the test steps}}$$

Coverage Criteria

The following criterion is necessary to analyze the quality of the test case generation process. This criterion also evaluates our methodology for Petri Net derivation since the test case generation depends upon the derived Petri Net.

Definition 6.1.3

The generated test cases must ensure that all the requirements in SRS are covered at least once [ISO13]. This is also a requirement of ISO-26262 [ISO11], a standard for functional safety of automobiles.

$$\text{Requirement Coverage} = \frac{\text{Number of requirements covered}}{\text{Number of requirements present in the SRS}} \times 100$$

Definition 6.1.4

The generated test cases must ensure that all possible paths from the initial place to the final place in a Petri Net are covered at least once [ISO13].

$$\text{Path Coverage} = \frac{\text{Number of paths covered}}{\text{Number of paths present}} \times 100$$

6.1.3 Evaluation Procedure

The sequence of steps followed for evaluation of the tool is shown graphically in Figure 6.1. The evaluation is done by comparing the manual and automatic generation of test scripts. The SRS document provided in [BDFM] acts as input for the domain expert. In case of manual method, the domain expert analysis the SRS document and creates the test specification. Once it is done, the tester creates the test scripts corresponding to the test specification. In case of automatic method, the domain expert writes the requirements in RNL and provides it as an input to the TSGT tool which directly generates the test scripts.

6.2 Evaluation Results and Discussion

This section lists out the results obtained from the evaluation process and further discusses the various inferences from the results.

6.2.1 Analysis of Quantitative Data**Based on Time Criteria**

The results of the tool evaluation are first discussed with respect to time criteria discussed in Section 6.1.2 . Table 6.1 and 6.2 describe the different parameters for manual and tool generated test scripts respectively. The tables provide data of five different use cases from our evaluation system and also the average of different parameters.

According to Definition 6.1.1, effort per test case is indicative of the average time required to generate each test case. The tables show that the effort required for each test case in the manual method is 10.6 minutes per test case on an average whereas in the automatic generation it is 1.6 minutes per test case. This denotes an 85% improvement over the manual method.

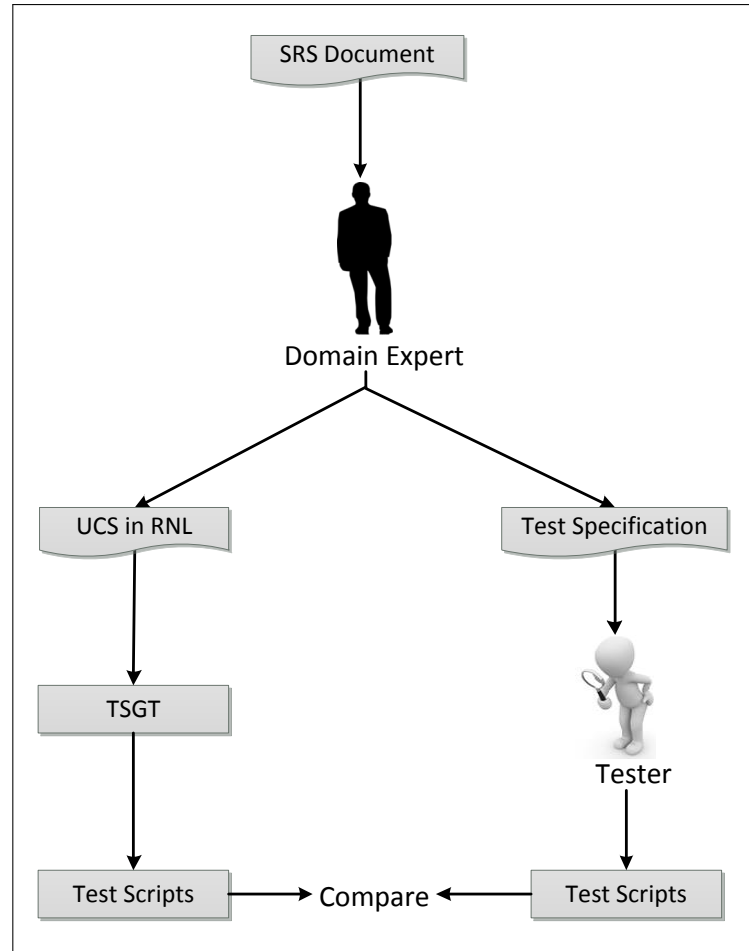


Figure 6.1: The procedure used for evaluating the tool

According to Definition 6.1.2, Test Case Productivity (TCP) shows the number of test steps that can be generated in one minute. From the tables, it can be seen that our tool generated 3.3 more test steps in a minute because TCP stands at 0.6 and 3.9 for manual and automatic methods respectively. Figure 6.2 shows a chart that contains the details of all the individual use cases. From the tables, it can be inferred that the total time taken for generating the test scripts manually takes 54 minutes on an average whereas the total time taken for automatic test script generation is only 13 minutes. This implies that there is a 76% reduction in time when the tool is used for test script generation.

Table 6.1: Evaluation results of manual test script generation method

Manual	Use Case					
	1	2	3	4	5	Average
<i>Total Time Taken (Manual)</i>	40	45	60	50	75	54.0
<i>Number of test cases (Manual)</i>	3	5	6	5	7	5
<i>Effort_{TC}</i>	13.3	9.0	10.0	10.0	10.7	10.6
<i>Number of test steps</i>	24	32	34	34	45	34
<i>Test Case Productivity (TCP)</i>	0.6	0.7	0.6	0.7	0.6	0.6

Table 6.2: Evaluation results of automatic test script generation method

TSGT	Use Case					
	1	2	3	4	5	Average
<i>Total Time Taken (TSGT)</i>	10	10	15	12	18	13.0
<i>Number of test cases (TSGT)</i>	5	7	9	8	12	8
<i>Effort_{TC}</i>	2.0	1.4	1.7	1.5	1.5	1.6
<i>Number of test steps</i>	30	45	52	52	78	51
<i>Test Case Productivity (TCP)</i>	3.0	4.5	3.5	4.3	4.3	3.9

Based on Coverage Criteria

This segment deals with the results of the tool evaluation based on coverage criteria discussed in Section 6.1.2. Table 6.3 describes the different parameters for manual and tool generated test scripts respectively.

From the table, it can be inferred that the number of test cases generated by manual method is always less than the automatic method. The coverage percentage in case of manual method is solely dependent on the individual extracting the test cases. Hence we see a difference in coverage percentage in manual method and it is arbitrary. But in the automatic method, requirement and path coverage are always 100%. This is due to

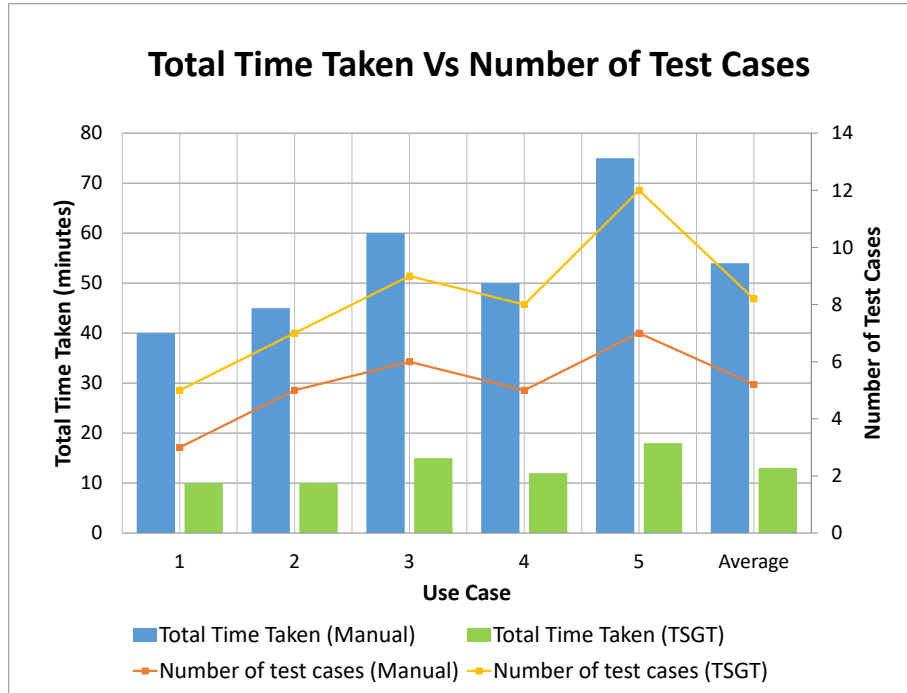
Figure 6.2: Effort_{TC} and TCP for manual and automatic method

Table 6.3: Evaluation results with respect to coverage criteria

Parameters	Use Case					
	1	2	3	4	5	Average
<i>Number of test cases (Manual)</i>	3	5	6	5	7	5.2
<i>Requirement Coverage (Manual)</i>	60%	80%	85%	90%	70%	77%
<i>Path Coverage (Manual)</i>	100%	90%	85%	70%	90%	87%
<i>Number of test cases (TSGT)</i>	5	7	9	8	12	8.2
<i>Requirement Coverage (TSGT)</i>	100%	100%	100%	100%	100%	100%
<i>Path Coverage (TSGT)</i>	100%	100%	100%	100%	100%	100%

the formal method of test case generation and this also ensures that each requirement and path is covered with the generated test cases.

6.2.2 Discussion of the Results

This section compares our tool i.e. TSGT with other closely related test script generation tools available in the research field. Figure 6.3 shows a graphical representation of the comparison between different approaches based on the effort per test case criterion. The data for the manual and TSGT approach is taken out from our evaluation. [YAZ15] provides the data for RTCM and UMTG approach.

The pros and cons of the different approaches are discussed below. UMTG provides an exhaustive test data for test script generation but it requires domain models for test data generation. The need for domain model generation along with test data generation using OCL increases the effort required for test case generation. RTCM provides the best solution but it requires manual intervention to improve some of the intermediate artifacts which might increase the effort in some cases.

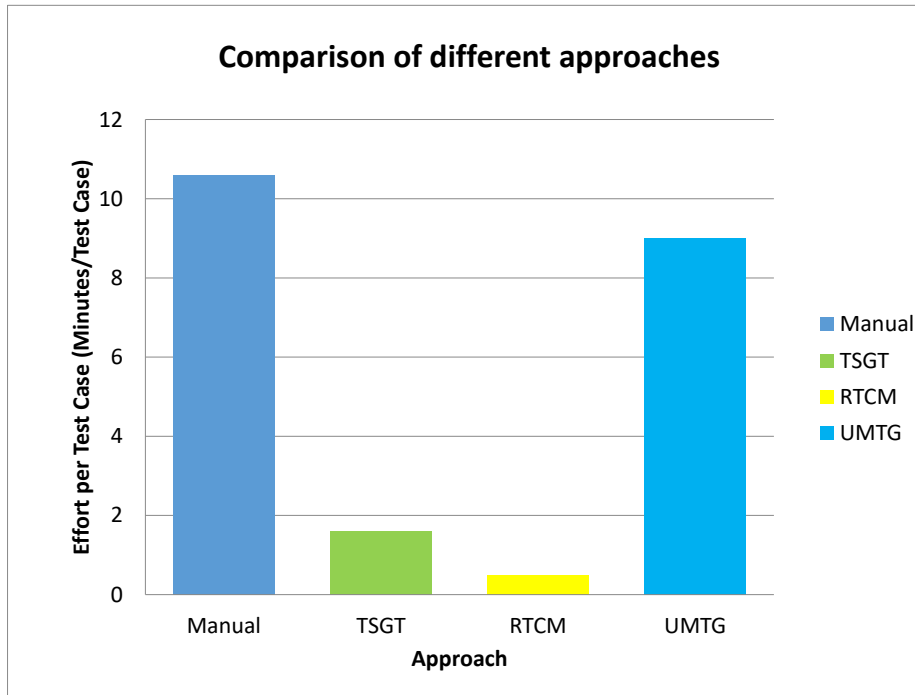


Figure 6.3: Comparison between different existing approaches

The total time taken for test case generation in our tool is inclusive of the time taken to specify the UCSs in RUCM template and to present them as input to our tool. This solely depends on the expertise of the user in RUCM template and also the experience in our tool. The time taken for test script generation after the input section is negligible. Figure 6.4 shows how the total time taken for test script generation from each use case

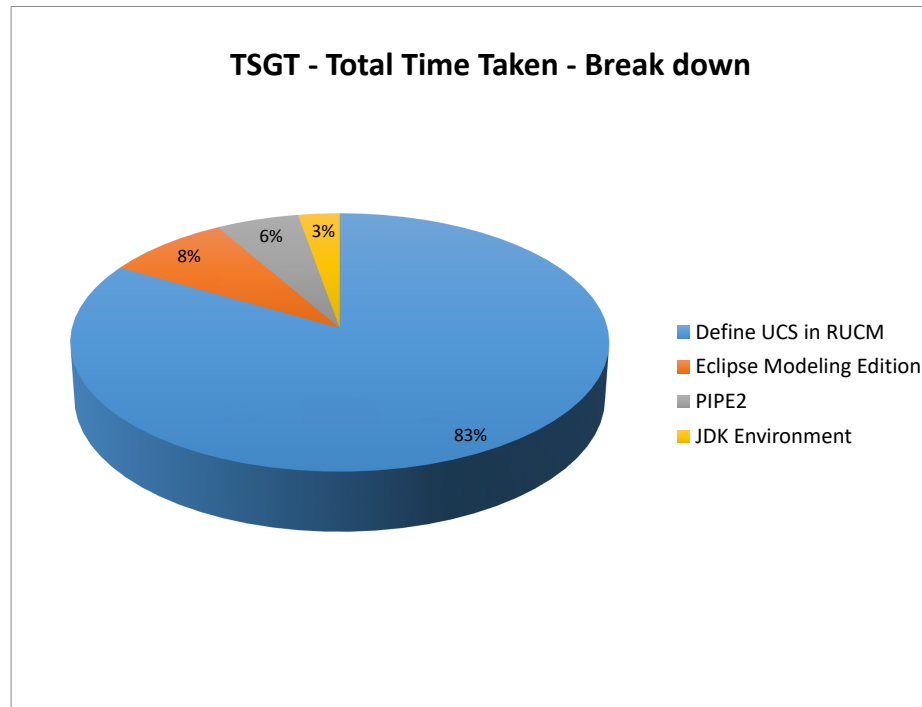


Figure 6.4: Breakdown of the time taken between different activities

is split between different blocks in our tool. From the chart, it is obvious that the UCS to RUCM template utilizes most of the time taken.

The advantage of our tool is that the time taken to specify complex UCSs does not add much to the total time taken, but produces more number of test cases. This scenario can be explained with the help of a graph as shown in Figure 6.5. By comparing Use Case 3 and Use Case 5, there is not much difference in the time required to specify the UCSs but there is a drastic increase in the number of test cases that are generated thereby reducing the effort required to generate them. In the manual method, it can also be seen that more complex the use cases become, lesser the number of test cases identified. The average effort required to generate a test case will improve drastically when more number of complex use cases are evaluated.

6.2.3 Summary

This section summarizes the trends observed in the evaluation results and our tool characteristics. Evaluation criteria considering time and coverability have been compared between TSGT and manual methods. The evaluation shows that TSGT requires 85% less effort than the manual method and it also provides 100% coverability.

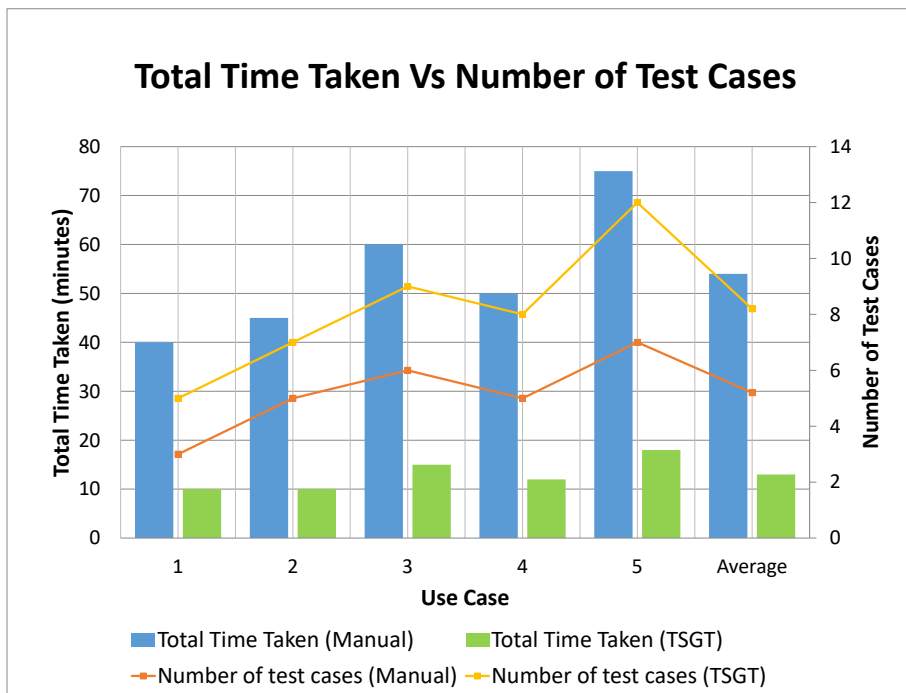


Figure 6.5: Relation between number of test cases and total time taken

Our tool has also been compared with other similar approaches in the research field and has been found to be good. Also, the limitations of our tool in reducing the effort required for each test case has been discussed and ways to improve them are suggested. From all the above discussions, it can be concluded that the automatic test case generation using TSGT is definitely better than manual test case generation and other existing approaches.

7 Conclusion and Outlook

This chapter provides a short summary of the thesis work and concludes the thesis work. This chapter also provides a short note on the limitations of the proposed work and the possibilities of future research. The final section concludes the chapter by enumerating the contributions of this thesis-work to the research in the field of software engineering.

7.1 Summary of the Thesis

The modern day software demands a software development process that produces high quality software with reduced cost and time. Agile methodology, which was developed with these objectives, has a frequent change in requirements as one of its main feature. Consequently, the testing process must also adapt itself to the changing requirements. A need for a formal method for test case generation is paramount because most of the testing process is dependent on the expertise of the stakeholders. The above statements provide the basic motivation for the thesis.

The thesis provides an approach in which the test scripts can be generated directly from the requirements. In Agile, the requirements are expressed in the form of use cases written in natural language. In order to automate the test script generation, we need a model like Petri Net which has semantics useful for formal analysis.

Now the need for a formalized method to convert use cases from natural language into Petri Nets arises. For this, a RNL template called RUCM is used and it has predefined metamodels that facilitate easy conversion into Petri Net. Once the Petri Net is elicited from the requirements, it can be easily simulated to derive test cases. Finally, test scripts are created using a Lookup table that has mappings to test steps that are predefined according to the platform on which the generated test scripts should be executed.

A tool called TSGT has been developed based on the above proposed approach. The tool on comparison with manual method shows 76% reduction in the time taken for test case generation and 85% improvement on the effort required for each test. The coverage

criteria have also greatly improved than that of the manual method. On an average, the tool provides 21% more coverage than the manual method.

The proposed approach also has some limitations and hence some assumptions have to be made for developing the tool. It also provides the possibility for future work since some of its functionalities can be enhanced and extended with more work. The next two sections provide details on the limitations and outlook regarding future work respectively.

7.2 Limitations

The main assumption in this work is that the tool is used only within the scope of Agile software development process. In Agile, the larger requirements are broken down into smaller features called User Stories or Use Cases depending upon the project. These have a relatively simpler workflow when compared to other forms of requirement specifications. Currently, the tool cannot support much complex workflows such as the interaction between multiple Use Cases and concurrent running of Use Cases.

The work has the following limitations that directly affect the evaluation metrics of the tool. First, the quality of the generated test cases depends solely on the quality of the UCSs entered by the user. Therefore, the user should be aware of the restriction rules and grammar of the RUCM template. This not only ensures good quality of generated test cases but also plays a major role in improving the effort taken for test case generation.

Second, the efficiency of the test script generation depends upon the accuracy of the Lookup table according to predefined test steps. Hence, Lookup table implementation has to be continuously improved for good quality test scripts. Finally, the work has been implemented considering just two coverage criteria. There is scope for inclusion of further criteria like branch, decision, boundary value, etc. and for this more research has to be performed on the part of test data generation.

7.3 Outlook for Future Work

There are several directions in which this work can be extended and improved. The next sections highlight the different enhancements that can be carried out in future on the current work.

- *A comprehensive solution for requirements:* The work can be extended to make it applicable for requirements of different software development processes. Currently, only the Agile methodology is supported.
- *Integration with Requirements Engineering:* In practice, the quality of requirements is evaluated not only by analyzing them but also by creating corresponding test cases. Hence, the method can be enhanced such that it supports feedback about the quality of requirements by integrating it into existing approaches like [Som07] and [SLA15] that use Petri Nets for formal analysis of requirements.
- *Integration with other processes in SDLC:* The work can be developed such that it can be integrated with other processes in SDLC. For example, some UML behavioral diagrams like Activity Diagrams are based on the semantics of Petri Nets. A formal approach can convert our intermediate artifacts into standard models. Similarly, some approaches like [Xu11] and [HW05] use Petri Nets for automatic code generation.
- *Integration of NLP tools:* The current work can be improved in areas like identifying constraints and test data, by employing NLP. The advantage of using NLP techniques is that it helps to understand the context of the test scenario. The quality of the test cases would improve if the tool could understand the meaning of the test steps.
- *Integration with MDSE processes:* The current implementation of the tool has various formal models. Since MDSE uses models as the foundation for software development, these formal models can be integrated into the MDSE processes thereby improving the productivity.

7.4 Conclusion

A formal method that supports auto-generation of test scripts from requirements within an Agile SDLC has been designed and implemented in this thesis. The thesis is concluded with the contributions of the proposed approach to the existing research in the field of Agile software engineering. The contributions are

- *A formal technique for test case elicitation from the requirements:* This eliminates the need for domain experts and their proficiency from test case generation process.
- *A formal technique to convert requirements into analyzable formal models:* This facilitates the use of the formal models for other processes in SDLC.

- *A restricted natural language template for use case specifications:* A template has been analyzed and adapted for test case generation process in this work.
- *A tool for test script generation:* A tool that can reduce the effort for test script generation along with improved test coverage has been designed and realized in this thesis.

As final remarks, it can be concluded that a formal approach for test script generation from requirements in natural language is feasible and has been realized in this work. It can also be concluded that the tool built in this work is found to be more advantageous than the manual method of test script generation. In addition, the thesis also provides various prospects for integration into the mainstream processes of software development.

A Appendix

A.1 M2M Rules for RUCM to Petri Net Conversion

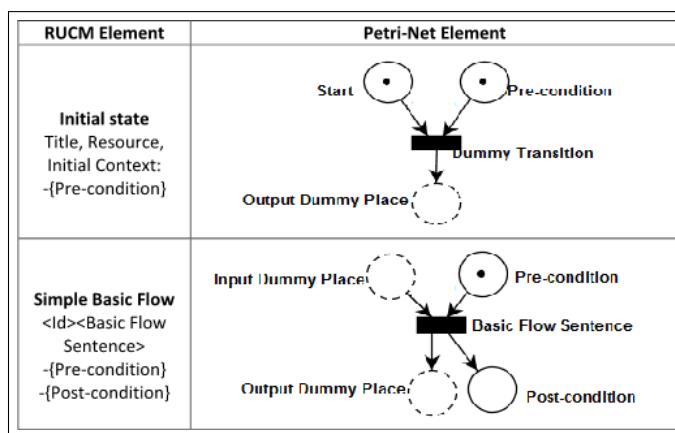


Figure A.1: M2M Rules for Initial State and Simple Basic Flow

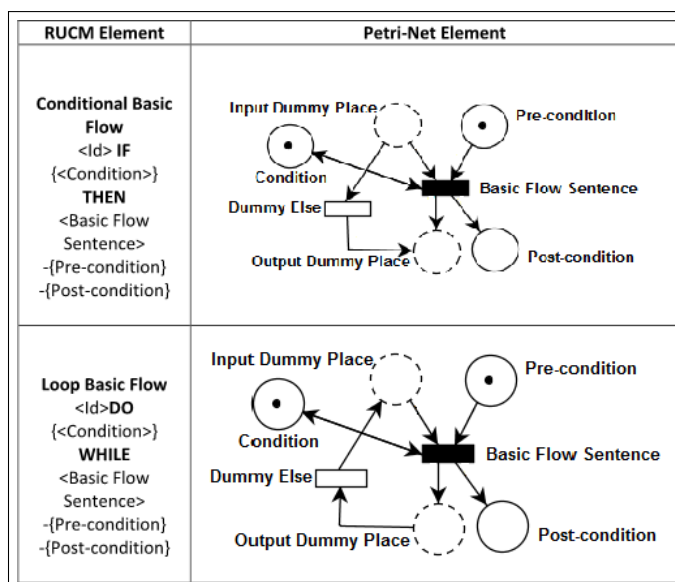


Figure A.2: M2M Rules for Conditional Basic Flow and Loop Basic Flow

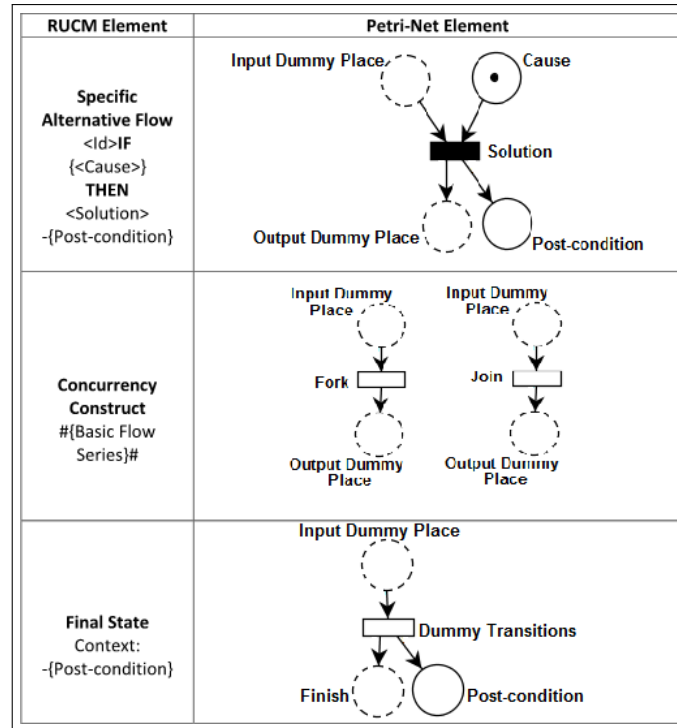


Figure A.3: M2M Rules for Specific Alternative Flow, Concurrency and Final State

A.2 Implementation of M2M Rules for Sub to Integrated Petri Nets Conversion in QVTo

```

modeltype MMA uses "http://www.example.org/PetriNetMetaModel";
modeltype MMB uses "http://www.example.org/PetriNetMetaModel";

transformation PNtoPNTransformation(in Source: MMA, out Target: MMB);

main() {
    Source.rootObjects()[PetriNet] -> map PetriNetToPetriNet();
}

mapping PetriNet :: PetriNetToPetriNet() : PetriNet {
    /*This section makes calls to the different mappings
    required to integrate the sub Petri Nets*/
    result.place :=self.place -> map CopyOtherPlaceToNewPlace();
    result.place +=self.place -> map InputDummyPlaceToNewPlace();
    result.transition := self.transition -> map TransitionToNewTransition();

    result.arc := self.arc -> map InputArcToNewInputArc();
    result.arc += self.arc -> map InputArcToNewPlace();

```

A.2 Implementation of M2M Rules for Sub to Integrated Petri Nets Conversion in QVTo

```
result.arc += self.arc -> map OutputArcToNewOutputArc();

result.arc += self.arc -> map ExceptionInputArcToEpisodePlace();
result.arc += self.arc -> map ConditionElseInputArcToEpisodePlace();
result.arc += self.arc -> map ConditionElseOutputArcToEpisodePlace();
}

/*This following section contains the different mappings rules are
   implemented according to their constraints*/
mapping Arc :: ConditionElseOutputArcToEpisodePlace() : Arc
when {self.Type.equalsIgnoreCase("Output")
    and self.place.Name.startsWith("I_Dummy_Place")
    and self.transition.Name.equalsIgnoreCase("T_Dummy_Else")} {
    result.Weight := self.Weight;
    result.transition := self.transition
        .resolveoneIn(Transition :: TransitionToNewTransition, Transition);
    result.Type := self.Type;

    var var_Append: String;
    var_Append:= self.place.Name.at(15);
    var var_Next:Integer;
    var_Next:= var_Append.toInteger()+1;

    var temp : Place;
    temp := self.place.container().allSubobjectsOfType(Place)
        ->any(Name.toString().startsWith("I_Dummy_Place_"+var_Append));
    result.place := temp.resolveoneIn( Place :: InputDummyPlaceToNewPlace, Place);
}

mapping Arc :: ConditionElseInputArcToEpisodePlace() : Arc
when {self.Type.equalsIgnoreCase("Input")
    and self.place.Name.startsWith("O_Dummy_Place")
    and self.transition.Name.equalsIgnoreCase("T_Dummy_Else")} {
    result.Weight := self.Weight;
    result.transition := self.transition
        .resolveoneIn(Transition :: TransitionToNewTransition, Transition);
    result.Type := self.Type;

    var var_Append: String;
    var_Append:= self.place.Name.at(15);
    var var_Next:Integer;
    var_Next:= var_Append.toInteger()+1;

    var temp : Place;
    temp := self.place.container().allSubobjectsOfType(Place)
        ->any(Name.toString().startsWith("I_Dummy_Place_"+var_Next.toString()));
    result.place := temp.resolveoneIn( Place :: InputDummyPlaceToNewPlace, Place);
}
```

A Appendix

```
mapping Arc :: ExceptionInputArcToEpisodePlace() : Arc
when {self.Type.equalsIgnoreCase("Input")} {
    and self.place.Name.startsWith("I_E_Dummy") } {
    result.Weight := self.Weight;
    result.transition := self.transition
        .resolveoneIn(Transition :: TransitionToNewTransition, Transition);
    result.Type := self.Type;

    var var_Append: Integer;
    var_Append:= self.place.Name.at(11).toInteger()+1;
    var temp : Place;
    var var_Count:Integer;
    var var_Set:Collection(Place);
    var_Set:= self.place.container().allSubobjectsOfType(Place)
        ->select(Name.toString().startsWith("I_Dummy_Place_"));
    var_Count := var_Set->size();

    var var_Next:Integer;
    var_Next:= var_Append;

    if(var_Next = var_Count)then{
        temp := self.place.container().allSubobjectsOfType(Place)
            ->any(Name.toString().startsWith("I_Dummy_Place_E"));
        result.place := temp.resolveoneIn( Place :: InputDummyPlaceToNewPlace, Place);
    }else{
        temp := self.place.container().allSubobjectsOfType(Place)
            ->any(Name.toString().startsWith("I_Dummy_Place_"+var_Append.toString()));
        result.place := temp.resolveoneIn( Place :: InputDummyPlaceToNewPlace, Place);
    }endif;
}

mapping Arc :: OutputArcToNewOutputArc() : Arc
when {self.Type.equalsIgnoreCase("Output")} {
    and self.place.Name.startsWith("O_Dummy_Place") } {
    result.Weight := self.Weight;
    result.transition := self.transition
        .resolveoneIn(Transition :: TransitionToNewTransition, Transition);
    result.Type := self.Type;

    var var_Count:Integer;
    var var_Set:Collection(Place);
    var_Set:= self.place.container().allSubobjectsOfType(Place)
        ->select(Name.toString().startsWith("I_Dummy_Place_"));
    var_Count := var_Set->size();

    var var_Append: String;
    var_Append:= self.place.Name.at(15);
    if(var_Append.equalsIgnoreCase("S"))then{
        var_Append := 1.toString();
    }
```


A.2 Implementation of M2M Rules for Sub to Integrated Petri Nets Conversion in QVTo

```
var temp : Place;
temp := self.place.container().allSubobjectsOfType(Place)
    ->any(Name.toString()
        .startsWith("I_Dummy_Place_" + var_Append.toString()));
result.place := temp.resolveoneIn(Place :: InputDummyPlaceToNewPlace, Place);
}else{
var var_Next:Integer;
var_Next:= var_Append.toInteger()+1;
var temp : Place;

if(var_Next = var_Count)then{
temp := self.place.container().allSubobjectsOfType(Place)
    ->any(Name.toString().startsWith("I_Dummy_Place_E"));
result.place := temp.resolveoneIn(Place :: InputDummyPlaceToNewPlace, Place);
}else{
temp := self.place.container().allSubobjectsOfType(Place)
    ->any(Name.toString().startsWith("I_Dummy_Place_" + var_Next.toString()));
result.place := temp.resolveoneIn(Place :: InputDummyPlaceToNewPlace, Place);
}endif;
}endif;
}

mapping Arc :: InputArcToNewPlace() : Arc
when {(self.Type.equalsIgnoreCase("Input")
    and self.place.Name.startsWith("I_Dummy_Place"))} {
result.Weight := self.Weight;
result.transition := self.transition
    .resolveoneIn(Transition :: TransitionToNewTransition, Transition);
result.place := self.place.resolveoneIn(Place :: InputDummyPlaceToNewPlace, Place);
result.Type := self.Type;
}

mapping Arc :: InputArcToNewInputArc() : Arc
when {(self.Type.equalsIgnoreCase("Input")
    and not self.place.Name.startsWith("I_Dummy_Place")
    and not self.place.Name.startsWith("I_E_Dummy")
    and not self.place.Name.startsWith("O_Dummy_Place"))
    or (self.Type.equalsIgnoreCase("Output")
    and not (self.place.Name.startsWith("O_Dummy_Place")
    or self.place.Name.startsWith("I_Dummy_Place")))} {
result.Weight := self.Weight;
result.transition := self.transition
    .resolveoneIn(Transition :: TransitionToNewTransition, Transition);
result.place := self.place.resolveoneIn(Place :: CopyOtherPlaceToNewPlace, Place);
result.Type := self.Type;
}

mapping Transition :: TransitionToNewTransition() : Transition {
result.Name:=self.Name;
```

```

}

mapping Place :: InputDummyPlaceToNewPlace() : Place
when {self.Name.startsWith("I_Dummy_Place")} {
    result.Tokens:=self.Tokens;
    result.Name:= "P" + self.Name.toString().at(15);
}

mapping Place :: CopyOtherPlaceToNewPlace() : Place
when {not (self.Name.startsWith("O_Dummy_Place")
    or self.Name.startsWith("I_Dummy_Place")
    or self.Name.startsWith("I_E_Dummy") )}{
    result.Tokens:=self.Tokens;
    result.Name:= self.Name;
}

```

A.3 Implementation of M2T Rules for Petri Net to PNML Conversion in Xpand

```

«IMPORT PetriNetMetaModel»

/*To create net details in the corresponding tag in PNML*/
«DEFINE main FOR PetriNet»
«FILE "IntegratedPetriNet.xml"»<?xml version="1.0" encoding="ISO-8859-1"?><pnml>
<net id="Net-One" type="P/T net">
<token id="Default" enabled="true" red="0" green="0" blue="0"/>
«EXPAND placedetails FOREACH place»
«EXPAND transdetails FOREACH transition»
«EXPAND arcdetails FOREACH arc»</net>
</pnml>
«ENDFILE»
«ENDDEFINE»

/*To create the place details in the corresponding tag in PNML*/
«DEFINE placedetails FOR Place»<place id="«this.Name»">
<name>
<value>«this.Name»</value>
</name>
<initialMarking>
<value>Default,«this.Tokens»</value>
</initialMarking>
<capacity>
<value>«this.Tokens»</value>
</capacity>
</place>

```

A.3 Implementation of M2T Rules for Petri Net to PNML Conversion in Xpand

```
«ENDDEFINE»
```

```
/*To create transition details in the corresponding tag in PNML*/
«DEFINE transdetails FOR Transition»<transition id="«this.Name»">
<name>
<value>«this.Name»</value>
</name>
<rate>
<value>1.0</value>
</rate>
<timed>
<value>«if this.Name.contains("T_Dummy_End") then "true" else "false"»</value>
</timed>
<infiniteServer>
<value>>false</value>
</infiniteServer>
<priority>
<value>1</value>
</priority>
</transition>
«ENDDEFINE»
```

```
/*To create arc details in the corresponding tag in PNML*/
«DEFINE arcdetails FOR Arc»<arc id="
«if this.Type.contains("Input") then this.place.Name else this.transition.Name» to
«if this.Type == "Input" then this.transition.Name else this.place.Name»" source="
«if this.Type == "Input" then this.place.Name else this.transition.Name»" target="
«if this.Type == "Input" then this.transition.Name else this.place.Name»">
<inscription>
<value>Default,1</value>
</inscription>
<tagged>
<value>>false</value>
</tagged>
<type value="normal"/>
</arc>
«ENDDEFINE»
```


Bibliography

- [ALH+12] B. Algayres, Y. Lejeune, F. Hugonnet, R. Braek, A. Sarma. “GOAL: Observing SDL behaviors with Geode.” In: *The Proceedings of SDL’95* (2012), pp. 223–230 (cit. on p. 33).
- [Amb09] S. W. Ambler. “The Agile Scaling Model (ASM): Adapting agile methods for complex environments.” In: *Environments* (2009) (cit. on p. 17).
- [BBN03] M. Badri, L. Badri, M. Naha. “A use case driven testing process: Towards a formal approach based on UML collaboration diagrams.” In: *International Workshop on Formal Approaches to Software Testing*. Springer. 2003, pp. 223–235 (cit. on p. 28).
- [BDFM] B. Blazy, A. DeLine, B. Frey, M. Miller. “Software Requirements Specification (SRS) Lane Management System.” In: () (cit. on pp. 55, 57).
- [BG03] A. Bertolino, S. Gnesi. “Use case-based testing of product lines.” In: *ACM SIGSOFT Software Engineering Notes* 28.5 (2003), pp. 355–358 (cit. on p. 34).
- [BG09] A. Bandyopadhyay, S. Ghosh. “Test input generation using UML sequence and state machines models.” In: *Software Testing Verification and Validation, 2009. ICST’09. International Conference on*. IEEE. 2009, pp. 121–130 (cit. on p. 33).
- [BH95] L. Brömstrup, D. Hogrefe. “TESDL: Experience with generating test cases from SDL specifications.” In: *Conformance testing methodologies and architectures for OSI protocols*. IEEE Computer Society Press. 1995, pp. 455–467 (cit. on p. 33).
- [BL02] L. Briand, Y. Labiche. “A UML-based approach to system testing.” In: *Software and Systems Modeling* 1.1 (2002), pp. 10–42 (cit. on p. 33).
- [BLPK07] P. Bonet, C. M. Lladó, R. Puijaner, W. J. Knottenbelt. “PIPE v2. 5: A Petri net tool for performance modelling.” In: *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*. 2007 (cit. on p. 37).
- [BNHT11] F. A. Barros, L. Neves, É. Hori, D. Torres. “The ucsCNL: A Controlled Natural Language for Use Case Specifications.” In: *SEKE*. 2011, pp. 250–253 (cit. on p. 34).

- [BPBM97] G. v. Bochmann, A. Petrenko, O. Bellal, S. Maguiraga. “Automating the process of test derivation from SDL specifications.” In: *Sdl* 97.22 (1997), pp. 261–276 (cit. on p. 33).
- [Cal16] E. S. Calisaya. “Analysis of Natural Language Scenarios.” PhD thesis. PUC-Rio, 2016 (cit. on pp. 30, 40, 42, 43, 50).
- [CD15] T. Cerny, M. J. Donahoo. “On separation of platform-independent particles in user interfaces.” In: *Cluster Computing* 18.3 (2015), pp. 1215–1228 (cit. on pp. 27, 28).
- [CFB+13] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, M. Blackburn. “Test case generation from natural language requirements based on SCR specifications.” In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM. 2013, pp. 1217–1222 (cit. on p. 18).
- [DSVT07] A. C. Dias Neto, R. Subramanyan, M. Vieira, G. H. Travassos. “A survey on model-based testing approaches: a systematic review.” In: *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. ACM. 2007, pp. 31–36 (cit. on p. 26).
- [ECL01] IBM. *Building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle*. 2001. URL: <https://eclipse.org/modeling/> (cit. on pp. 36, 49).
- [EM2T07] Eclipse. *Generation of textual artifacts from models*. 2007. URL: <https://eclipse.org/modeling/m2t/?project=xpand> (cit. on p. 53).
- [EMB15] R. Elghondakly, S. Moussa, N. Badr. “Waterfall and agile requirements-based model for automated test cases generation.” In: *Intelligent Computing and Information Systems (ICICIS), 2015 IEEE Seventh International Conference on*. IEEE. 2015, pp. 607–612 (cit. on p. 56).
- [EQVT10] Eclipse. *Implementation of the Operational Mappings Language defined by Meta Object Facility 2.0 Query/View/Transformation*. 2010. URL: <https://projects.eclipse.org/projects/modeling.mmt.qvt-oml> (cit. on pp. 36, 37, 53).
- [FL00] P. Fröhlich, J. Link. “Automated test case generation from dynamic models.” In: *ECOOP*. Springer. 2000, pp. 472–492 (cit. on p. 34).
- [Gul09] L. Gulechha. “Software Testing Metrics.” In: *Software Testing Metrics* 1.1 (2009), pp. 7–8 (cit. on p. 56).

- [Guru13] Guru. *Software Requirements Analysis with Example*. 2013. URL: <https://www.guru99.com/learn-software-requirements-analysis-with-case-study.html> (cit. on p. 40).
- [HGB08] B. Hasling, H. Goetz, K. Beetz. “Model based testing of system requirements using UML use case models.” In: *Software Testing, Verification, and Validation, 2008 1st international conference on*. IEEE. 2008, pp. 367–376 (cit. on p. 34).
- [HW05] N. Hagge, B. Wagner. “A new function block modeling language based on petri nets for automatic code generation.” In: *IEEE Transactions on Industrial Informatics* 1.4 (2005), pp. 226–237 (cit. on p. 67).
- [HWZ12] R. Hametner, D. Winkler, A. Zoitl. “Agile testing concepts based on keyword-driven testing for industrial automation systems.” In: *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*. IEEE. 2012, pp. 3727–3732 (cit. on p. 35).
- [ISO11] I. ISO. “26262: Road vehicles-Functional safety.” In: *International Standard ISO/FDIS 26262* (2011) (cit. on p. 56).
- [ISO13] ISO. *ISO/IEC/IEEE 29119-1: 2013 Software and systems engineering-Software testing-Part 1: Concepts and definitions*. 2013 (cit. on pp. 56, 57).
- [KJ14] P. Kulkarni, Y. Joglekar. “Generating and analyzing test cases from software requirements using nlp and hadoop.” In: *International Journal of Current Engineering and Technology (INPRESSCO)* (2014) (cit. on pp. 19, 34).
- [KK06] M. Katara, A. Kervinen. “Making model-based testing more agile: a use case driven approach.” In: *Haifa Verification Conference*. Springer. 2006, pp. 219–234 (cit. on p. 34).
- [LJX+04] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, Z. Guoliang. “Generating test cases from UML activity diagram based on gray-box method.” In: *Software Engineering Conference, 2004. 11th Asia-Pacific*. IEEE. 2004, pp. 284–291 (cit. on pp. 18, 33).
- [LM06] G. Little, R. C. Miller. “Translating keyword commands into executable code.” In: *Proceedings of the 19th annual ACM symposium on User interface software and technology*. ACM. 2006, pp. 135–144 (cit. on p. 35).
- [Mur89] T. Murata. “Petri nets: Properties, analysis and applications.” In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580 (cit. on p. 31).

- [NFLJ03] C. Nebut, F. Fleurey, Y. Le Traon, J.-M. Jézéquel. “Requirements by contracts allow automated system testing.” In: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE. 2003, pp. 85–96 (cit. on pp. 28, 33).
- [NFLJ06] C. Nebut, F. Fleurey, Y. Le Traon, J.-M. Jezequel. “Automatic test generation: A use case driven approach.” In: *IEEE Transactions on Software Engineering* 32.3 (2006), pp. 140–155 (cit. on pp. 18, 33).
- [NS12] A. Nayak, D. Samanta. “Synthesis of Test Scenarios Using UML Sequence Diagrams.” In: *ISRN Software Engineering 2012* (2012) (cit. on p. 33).
- [PHDK00] J. C. S. do Prado Leite, G. D. Hadad, J. H. Doorn, G. N. Kaplan. “A scenario construction process.” In: *Requirements Engineering* 5.1 (2000), pp. 38–61 (cit. on p. 43).
- [PSVS05] M. Prasanna, S. Sivanandam, R. Venkatesan, R. Sundarrajan. “A survey on automatic test case generation.” In: *Academic Open Internet Journal* 15.6 (2005) (cit. on p. 26).
- [Rei12] W. Reisig. *Petri nets: an introduction*. Vol. 4. Springer Science & Business Media, 2012 (cit. on p. 31).
- [RG99a] J. Ryser, M. Glinz. “A practical approach to validating and testing software systems using scenarios.” In: *QWE’99, 3 rd International Software Quality Week Europe*. Citeseer. 1999 (cit. on p. 34).
- [RG99b] J. Ryser, M. Glinz. “A scenario-based approach to validating and testing software systems using statecharts.” In: *Proc. 12th International Conference on Software and Systems Engineering and their Applications*. 1999 (cit. on pp. 18, 33, 34).
- [SACS15] E. Sarmiento, E. Almentero, J. CSP Leite, G. Sotomayor. “Mapping textual scenarios to analyzable Petri-Net models.” In: *Proceedings of the 17th International Conference on Enterprise Information Systems-Volume 2*. SCITEPRESS-Science and Technology Publications, Lda. 2015, pp. 494–501 (cit. on p. 21).
- [SK13] M. Shirole, R. Kumar. “UML behavioral model based test case generation: a survey.” In: *ACM SIGSOFT Software Engineering Notes* 38.4 (2013), pp. 1–13 (cit. on p. 33).
- [SLA15] E. Sarmiento, J. C. S. D. P. Leite, E. Almentero. “Analysis of scenarios with Petri-Net models.” In: *Software Engineering (SBES), 2015 29th Brazilian Symposium on*. IEEE. 2015, pp. 90–99 (cit. on pp. 20, 40, 67).
- [Som07] S. S. Somé. “Petri nets based formalization of textual use cases.” In: *SITE-University of Ottawa, Technical Report TR-2007-11* (2007) (cit. on p. 67).

- [TCM08] J. Tang, X. Cao, A. Ma. “Towards adaptive framework of keyword driven automation testing.” In: *Automation and Logistics, 2008. ICAL 2008. IEEE International Conference on*. IEEE. 2008, pp. 1631–1636 (cit. on p. 35).
- [TSSC12] S. Thummalapenta, S. Sinha, N. Singhanian, S. Chandra. “Automating test automation.” In: *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE. 2012, pp. 881–891 (cit. on p. 35).
- [TVKB01] L. H. Tahat, B. Vaysburg, B. Korel, A. J. Bader. “Requirement-based automated black-box test generation.” In: *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*. IEEE. 2001, pp. 489–495 (cit. on pp. 26, 33).
- [VONE02] V. One. *Manage agile testing: Easy test management for QA teams*. 2002. URL: <https://www.versionone.com/product/lifecycle/test-management/> (cit. on p. 25).
- [WPG+15a] C. Wang, F. Pastore, A. Goknil, L. C. Briand, Z. Iqbal. “UMTG: a toolset to automatically generate system test cases from use case specifications.” In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 942–945 (cit. on pp. 41–43).
- [WPG+15b] C. Wang, F. Pastore, A. Goknil, L. Briand, Z. Iqbal. “Automatic generation of system test cases from use case specifications.” In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM. 2015, pp. 385–396 (cit. on pp. 21, 35).
- [XPAND07] B. Klatt. *Xpand: A Closer Look at the model2text Transformation Language*. 2007. URL: <http://www.bar54.de/benjamin.klatt-Xpand.pdf> (cit. on p. 37).
- [Xu11] D. Xu. “A tool for automated test code generation from high-level Petri nets.” In: *Applications and Theory of Petri Nets (2011)*, pp. 308–317 (cit. on p. 67).
- [YAZ15] T. Yue, S. Ali, M. Zhang. “RTCM: a natural language based, automated, and practical test case generation framework.” In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM. 2015, pp. 397–408 (cit. on pp. 21, 35, 61).
- [YBL10] T. Yue, L. C. Briand, Y. Labiche. “An Automated Approach to Transform Use Cases into Activity Diagrams.” In: *ECMFA*. Springer. 2010, pp. 337–353 (cit. on p. 34).
- [YBL13] T. Yue, L. C. Briand, Y. Labiche. “Facilitating the transition from use case models to analysis models: Approach and experiments.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.1 (2013), p. 5 (cit. on pp. 34, 41, 42, 50).

[ZEP07] Zephyr. *Test management add-ons for atlassian - zephyr*. 2007. URL: <https://www.getzephyr.com/products/zephyr-for-jira> (cit. on p. 25).

All links were last followed on September 20, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature