MODEL-DRIVEN ARCHITECTURE BASED TESTING USING SOFTWARE ARCHITECTURE VIEWPOINTS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Burak Uzun

June, 2015

MODEL-DRIVEN ARCHITECTURE BASED TESTING USING SOFTWARE ARCHITECTURE VIEWPOINTS
By Burak Uzun
June, 2015

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

_____

Assist. Prof. Dr. Bedir Tekinerdoğan (Advisor)

_____

Prof. Dr. Ali Hikmet Doğru

_____

Assoc. Prof. Dr. Buğra Gedik

Approved for the Graduate School of Engineering and Science:

_____

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

MODEL-DRIVEN ARCHITECTURE BASED TESTING USING SOFTWARE ARCHITECTURE VIEWPOINTS

Burak Uzun
M.S. in Computer Engineering
Advisor: Assist. Prof. Dr. Bedir Tekinerdoğan
June, 2015

Software testing is the process of checking whether a system meets the specifications and fulfills its intended purpose. Testing a system requires executing the test cases that can detect the potential defects in the program. In general, exhaustive testing is not possible or practical for most real programs due to the large number of possible inputs and sequences of operations. Because of the large set of possible tests only a selected set of tests can be executed within feasible time limits. As such, the key challenge of testing is how to select the tests that are most likely to expose failures in the system. Model-based testing (MBT) relies on models of system requirements and behavior to automate the generation of the test cases and their execution. Model based testing can use different representations of the system to generate testing procedures for different aspects of the software systems. Example models include finite state machines (FSMs), Petri Nets, I/O automata, and Markov Chains.

A recent particular trend in MBT is to adopt architecture models to identify the defects related to systemic properties. These systemic properties are typically defined in architecture views which represent the gross level structure of the system from particular concern perspective. Assessing software system correctness with respect to architectural specifications is called *architecture based testing (ABT)*. Many studies have focused on architecture based testing in which different models have been applied. However none of these have so far explicitly focused on adopting architecture views for deriving the test cases.

In this thesis, we first provide a systematic review on existing model-driven architecture based testing. We define all the existing processes in the literature and discuss the current limitations. Based on the result of the systematic review and our own analysis we provide a novel model-driven architecture based testing approach using architecture views. With the approach we focus on detecting the deviations in the code from the architectural views. For this we use models of architecture views together with executable transformation model to generate the test cases which are then executed on the real code. Our approach has been evaluated within a real industrial context of The Scientific and Technological Research Council of Turkey Software Technologies

Institute (STRCT-STI). The results of the industrial case study showed that model-driven architecture based testing can be effective for reducing the time to generate and execute the test cases, and enhancing the reliability of the system.

Keywords: Systematic Literature Review, Software Architecture Viewpoints, Architecture Based Testing, Model Based Testing.

ÖZET

YAZILIM MİMARİSİ BAKIŞ AÇILARI KULLANILARAK MODEL GÜDÜMLÜ MİMARİ TABANLI TEST ETME

Burak Uzun
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Yrd. Doç. Dr. Bedir Tekinerdoğan
Haziran, 2015

Yazılım test etme bir sistemin amacını yerine getirip getirmediğini ve istenilen özellikleri karşılayıp karşılamadığını denetleme sürecidir. Bir sistemi test etmek için olası program hatalarını keşfedebilen test durumlarını çalıştırmak gerekir. Genelde birçok gerçek program için olası girdi ve operasyon dizilerinin çok sayıda olması sebebiyle ayrıntılı test etmek mümkün ve pratik değildir. Büyük test setlerinden sadece seçilen olası test durumları kısıtlı bir zamanda koşturulabilir. Görüldüğü gibi, yazılım test etmede ki esas sorun sistemdeki hataları ortaya çıkarabilen test durumlarını seçebilmektir. Modele dayalı test etme (MDT) sistemin gereksinimlerinin ve davranımlarının modellerine dayanarak test durumlarının oluşturulmasını ve koşturulmasını otomatikleştirir. Modele dayalı test etme sistemin farklı temsillerini kullanarak yazılım sistemin farklı yönleri için test prosedürleri oluşturur. Örnek modeller içerisinde sonlu makineler, Petri Netler, otomatalar ve Markov zincirleri bulunur.

Modele dayalı test etmede yazılım mimarisi kullanılarak sistemik özelliklerde bulunan hataları ortaya çıkarmak yeni bir eğilim olarak görülmektedir. Bu sistemik özellikler tipik olarak mimari bakışlarında tanımlanmıştır. Bir yazılım sisteminin belirtilen mimari özelliklere göre doğruluğunu ölçmek *mimari tabanlı yazılım testi* (MBYT) olarak adlandırılır. Birçok çalışma farklı modeller kullanarak mimari tabanlı test yöntemleri üzerinde yoğunlaştı. Ancak bu çalışmaların hiç birisinde yazılım mimarisi bakış açıları test durumlarını üretirken kullanılması benimsenmedi.

Bu tezde biz, öncellikle var olan model güdümlü mimari tabanlı test etme yöntemlerinin sistematik incelemesini sunuyoruz. Literatürde var olan yöntemleri tanımlıyor ve yöntemlerin sınırlarını tartışıyoruz. Sistematik incelememiz ve analizlerimiz sonucunda yeni bir mimari bakış açılarını kullanan model güdümlü mimari tabanlı test yöntemi geliştiriyoruz. Yöntemimizde mimari bakış açılarında belirtilen tanımlardan sapan kodlar üzerinde yoğunlaşıyoruz. Bunun için mimari bakış açısı modellerini kullanan dönüşüm modellerini koşturarak sistem üzerinde koşturulacak test durumlarını üretiyoruz. Yöntemimiz Türkiye Bilimsel ve Teknolojik Araştırma Kurumu - Yazılım

Teknolojileri Enstitüsü'ndeki (TÜBİTAK-YTE) bir proje üzerinde değerlendirildi. Bu değerlendirmenin sonucu olarak model güdümlü mimari tabanlı test yöntemimiz test durumları oluşturmak ve koşturmak ve sistemin güvenilirliğini arttırmak için etkili bir yöntem olduğunu gördük.

Anahtar Kelimeler: Sistematik Literatür İnceleme, Yazılım Mimari Bakış Açıları, Mimari Tabanlı Test, Model Tabanlı Test

Acknowledgements

I would like to express my sincerest gratitude to my advisor Dr. Bedir Tekinerdoğan for giving me the chance to research on my topic. He drove me forward with his supervision, support, time and advises throughout my research.

I am thankful to Prof. Dr. Ali Hikmet Doğru and Assoc. Prof. Dr. Buğra Gedik for kindly accepting to be in the committee and also for giving their precious time to review and read this thesis.

I am grateful to The Scientific and Technological Research Council of Turkey Software Technologies Institute (STRCT-STI) for the support and permission of evaluating my approach on institute's project. I would also like to thank my colleagues for their support especially Mehmet, Ahmet and Ali.

I would like to thank to my soon to be wife Başak for her invaluable support and the joy she brings in my life with her endless love.

Last but not least, I would like to thank my mother Fatma, my father Müjdat and my brother Tekin Alp for being there for me when I need help and support me in every step of my life with their love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Background

Software testing is a process of investigating a software product to identify possible mismatches between expected and present requirements of the system [24]. Identified mismatches are categorized as bugs, errors and defects. Software bug is a static fault in the software system and activates a software error. Software error is a faulty state of the software system which is activated by the execution of software bug. Furthermore, software defect is a propagation of a software error which causes external, observable and incorrect behavior of the system. One of the main motivations of software testing is to ensure the correctness of a software system. Software is correct if and only if each valid input to the system produces an output according to system specifications. Therefore, software must be verified and validated according to specifications provided. Moreover, software testing requires executions of test cases which can detect possible bugs, errors and defects. In most cases, exhaustive testing is not possible due large number of possible operation sequences and inputs. As a result, selecting set of test cases which can detect possible flaws of the system is the key challenge in software testing. Model based testing addresses this challenge by automating the generation and execution of test cases using models based on system requirements and behavior. Recently, studies in model based testing adopt software architecture models to identify the defects related to systemic properties. Software architecture is a blueprint of the system which enables communication between stakeholders of the system and clear representation of abstract system model. Software representation of complex and large domains grounds software architecture modeling and proper documentation of software architecture to gain more importance [1].

International standards for software architecture are defined and adapted as the software architecture concept attracts more attention ([2], [23]). In a software system, software architecture provides high level specifications of the low level implementations in which the mapping between levels can be performed. Accordingly, one can use architecture specifications for verifying and validating a software system to ensure correctness of the system. This idea emerged architecture based testing concept for testing the architecture specifications in a software system. In this thesis we will present current techniques adopted in architecture based testing using systematic literature review protocol and novel approach evaluated on real industrial case study.

1.2. Problem Statement

Architecture based testing is a developing and promising research area in software testing. In architecture based testing software correctness is assessed with respect to given architectural specifications which are actually models representing software architecture of the system. Therefore, architecture based testing is inherently model driven approach. Accordingly, we will infer architecture based testing as model driven architecture based testing (MDABT) in this thesis. Furthermore, systematic literature review we applied on this domain enabled us to identify issues in current studies for MDABT. None of the studies we examined explicitly used architecture viewpoints in the definition software architecture models. Moreover, none of the studies presented in architecture based testing presents a systematic literature review for MDABT domain. At last, almost all studies evaluated suggested approaches on small scale examples such as client server applications.

Architecture viewpoint notion takes an important part in software architecture modeling. Software architecture defined by combination of different architecture views governed by viewpoints each mapping to different stakeholder concern. Therefore, in architecture based testing architecture view must be taken into account when the test cases are generated. This way each separate concern of the stakeholders can be tested as a specific architecture view.

As far as we know there is no study in which systematic literature review of the domain is performed. It is important to have a study to analyze and identify the studies in this domain so that new researchers can identify challenges and suggest new research areas.

Current literature in architecture based testing present approaches for the research area. In most of the studies the approach is evaluated on small scale examples. Moreover, most of the examples are created along with the approach to be performed on which implies to biased examples. It is critical to use unbiased case study to evaluate newly presented technique.

1.3. Contribution

The contribution of this thesis can be listed as follows:

- *A systematic review on current state of the art in model driven architecture based testing approaches*

We have carried out a systematic review on existing approaches for model driven architecture based testing. We have found 158 studies and selected 12 of these as primary studies which we analyzed in detail. The systematic review is novel and has not been defined before. The systematic review provides systematic overview of current state of the art and has resulted in important lessons learned about MDABT. This can be used both by researchers to identify the important challenges in MDABT and practitioners who can use the guidelines of the identified approaches in setting up an MDABT approach.

- *A novel systematic approach for MDABT*

After performing systematic literature review on architecture based testing domain, we have seen that although the definition of the architecture evolved none of the studies concentrated on architecture view. Software architecture is not single perspective entity rather a set of perspectives mapping to different stakeholder concerns. Each concern must be handled individually in testing so that complete testing on system using architecture can be achieved.

- *Tool support for MDABT*

Eclipse Epsilon IDE and its integrated tools have been used in the development of our approach implementation. Eclipse Epsilon offers a wide variety of tools, languages and support of all model driven development issues and centralizes these technologies in one framework. In our study we created an Eclipse Epsilon environment to generate test cases for our approach.

- *Evaluation of MDABT within a real industrial case context*

Systematic literature review we performed pointed out that most of the studies did not evaluate their suggested approaches on real industrial case contexts. In fact small scale examples are created to evaluate suggested approaches which are prone to be biased to suggested approach. Our approach has been successfully tested on a software systems infrastructure that is being actively used by 4500 users throughout the day since 2010.

1.4. Outline of the Thesis

The rest of thesis is organized as follows: Chapter 2 gives a preliminary about software architecture, MDABT and model based testing concepts. Note that in Chapter 3 more detailed information of MDABT will be delivered in which systematic literature review is presented. The generic process model for MDABT is obtained from the selected studies using Kitchenham's guideline for systematic literature review. Chapter 4 presents our approach on MDABT using architecture viewpoints. The process model we created for our approach and details of the implementation will be explained. Furthermore, Chapter 5 presents our case study from STRCT-STI in which we evaluated our approach. Evaluation results and result discussions of our approach are presented alongside with our case study. Chapter 6 presents the related work and at last Chapter 7 presents the conclusions and discussions.

Chapter 2

Preliminaries

In this chapter software architecture and architecture views will be presented alongside with model based testing (MBT) and architecture based testing (ABT). Software architecture is presented using IEEE 1471 standard [23]. This standard discusses that software architecture descriptions are inherently multi view entities and single view description of architecture cannot completely express stakeholders concerns. Additionally, architecture view definitions will be presented using Clements et al. work in [1]. Furthermore, software testing concept will be presented together with model based testing and architecture based testing concepts.

2.1. Software Architecture Modeling

Software architecture became a fundamental subpart of software engineering [1]. According to IEEE 1471 standard software architecture is defined as main part of a software intensive system expressed by set of components, components inter and intra relations [23]. It can be inferred that architecture of a system is not a monolithic concept rather it is complicated set of components and relations which are evolving with the system itself. Therefore, each component and component relation is very important from the different aspects of the system stakeholders from which the concept of architecture view emerged. In this section first we will give background information for software architecture and then architecture view types of Views and Beyond framework will be presented.

Software architecture is an entity exhibited by a system and expressed by architecture description. Figure 2.1 (adopted from [23]) shows the context of software architecture concept with respect to system perspective. System is situated in its environment and can be enterprise, system of systems, service or

software. System environment is explained as every interface that is interacting and affecting system. For instance environment can be the domain which system represents or it can be the server system resides. Moreover, stakeholders have interests in the system which can be called as system concerns. Stakeholders are any organization or people that have concerns and interests in system such as user, consumer, supplier... System must be present for an architecture concept to be provided. Systems have architecture or architectures which are expressed by architecture descriptions. Architecture description explains the architecture of the system utilizing different notation techniques and it is a "blueprint" between architects and stakeholders.

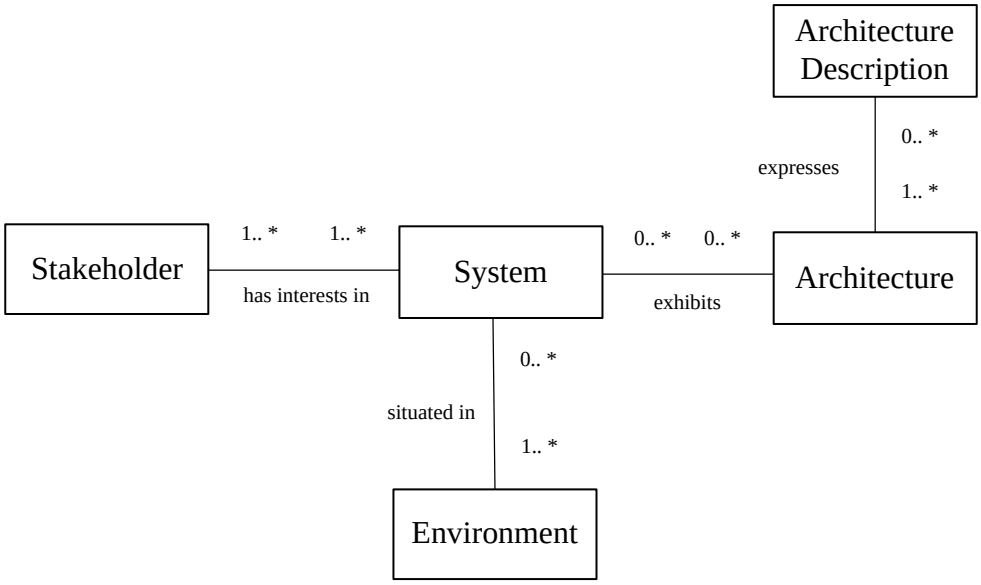

Figure 2.1. Context of software architecture

In Figure 2.2 (adopted from [23]) core architecture descriptions concept and its relations with other architecture concepts is shown. Architecture description expresses an architecture belonging to some system of interest. Moreover, it identifies system including system stakeholders and stakeholder concerns. Architecture description consists of architecture views and viewpoints where viewpoints are guidelines of mapped views. Architecture view is a perspective of a system addressing a specific concern of a system stakeholder.

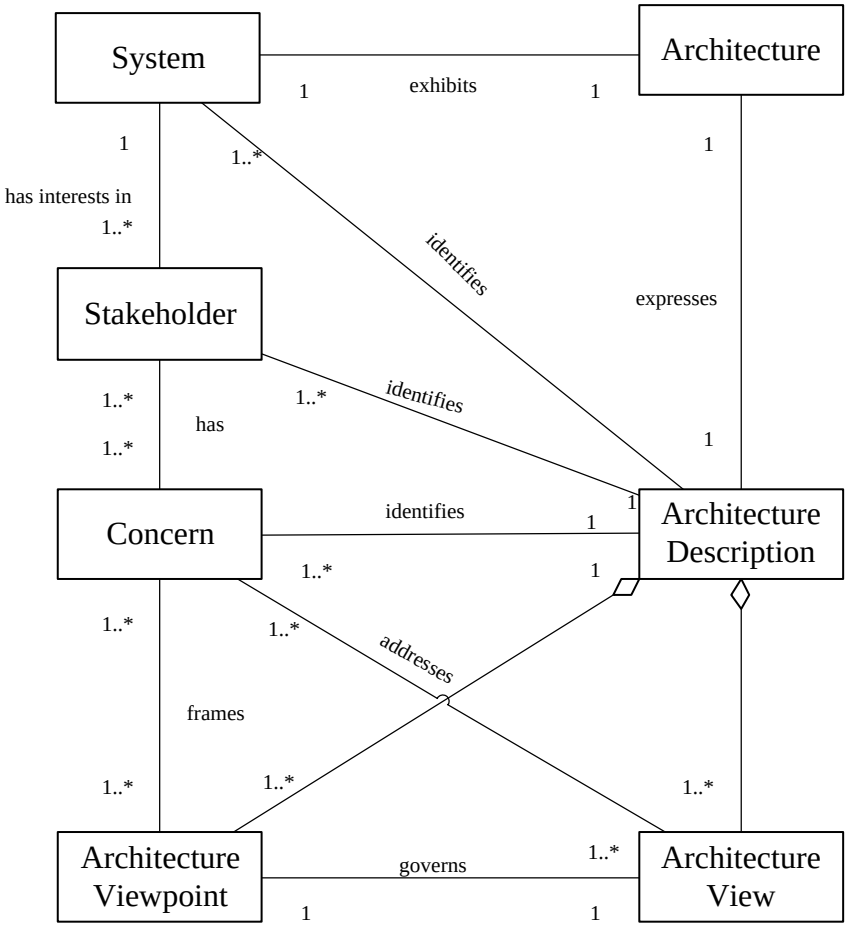

Figure 2.2. Core of software architecture description

In Figure 2.3 (adopted from [23]) architecture frameworks and its relations with other components are presented. Architecture framework provides a typical convention for presenting, analyzing and using architecture descriptions. There are several frameworks already practiced today which are not limited to MODAF, TOGAF, RM-ODP and Kruchten's 4+1 View Model. Architecture frameworks have a set of architecture viewpoints defined in the framework and identify stakeholder and their concerns.

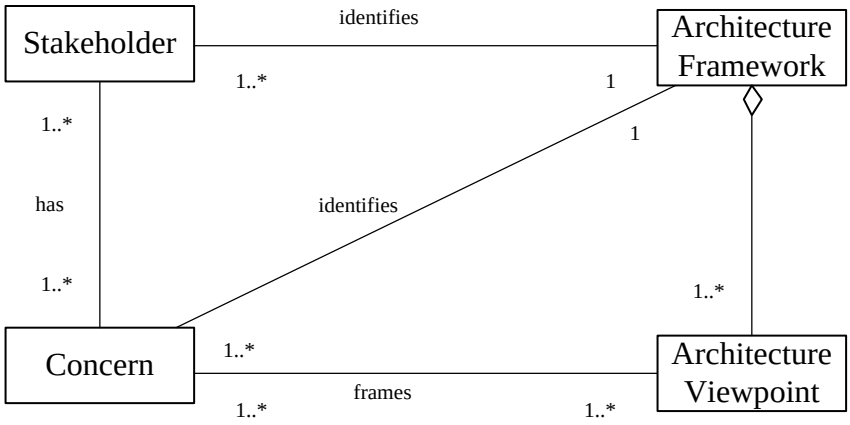

Figure 2.3. Architecture framework

Architecture is a complicated and large concept that cannot be explained from a single perspective [1].Therefore notion of architecture view is introduced for representing set of system components and inter and intra relations. An architecture view is related to a particular concern of a stakeholder. Different stakeholder yields different concern which yields set of views representing architecture of the system. Architectures described by multiple views is easy to understand, model, communicate with and analysis. Architecture modeling with architecture views supports separation of concerns where the concerns are identified by different stakeholders. Therefore, the notion of architecture view becomes more important to clearly express the architecture using different perspectives according to concern of the stakeholders. Architecture viewpoints dictate architecture view in which architecture viewpoint specifies details of architecture view and its structure. Moreover, viewpoint specifies what kind of information to be held in the architecture view model. Several architectural frameworks have been proposed in the current literature using architecture viewpoints. In this study we are interested in architecture views and viewpoints defined in Views and Beyond framework [1]. Views and Beyond framework utilizes three types of views module views, component and connector views and allocation views. In this thesis we will use the implementation of this framework implemented in Demirli et al. work [3].

2.2. Software Testing

Software testing is a process of investigating a software product to identify possible mismatches between expected and present requirements of the system [24]. Software testing can be done dynamically by executing the test cases or statistically by inspecting the system under test. Moreover, software testing

methods can be divided into two methods which are white box testing and black box testing. White box testing refers to testing the internal content of the system in which the tester must know the detail of the implementation. Black box testing on the other hand refers to testing the functionality of the system without knowing the internal structure. Software testing can be applied at different levels such as unit testing, integration testing, component interface testing and system testing. In unit testing the unit under test is isolated and the unit's functionality is tested. Integration tests verify the interaction between the units with respect to functional and non-functional requirements of the system. Moreover, component interface testing is applied for verifying the data transmission between the architectural design elements which can be component, units or subsystems. At last system testing is a typical acceptance testing of system in which the system components are integrated and possible scenarios are executed on the system under test.

2.2.1. Model Based Testing

According to Utting et al.[4] MBT is a type of testing that utilizes the information in model which is the intended behavior of the system and its environment. There are several motivations for to perform model based testing such as easy test maintenance, automated test design and enhancing test quality. In Figure 2.4 (adapted from [4]) process of model based testing is presented. Model of the system under test is constructed from the requirements of the system. Likewise, test selection criteria are formed by requirements, which is used for selecting test cases that detects faults, errors and possibly failures. Test case specifications are constructed from test selection criteria which are then used with system model to generate actual test cases. Test cases are executed at system under test and test results are analyzed by test verdict.

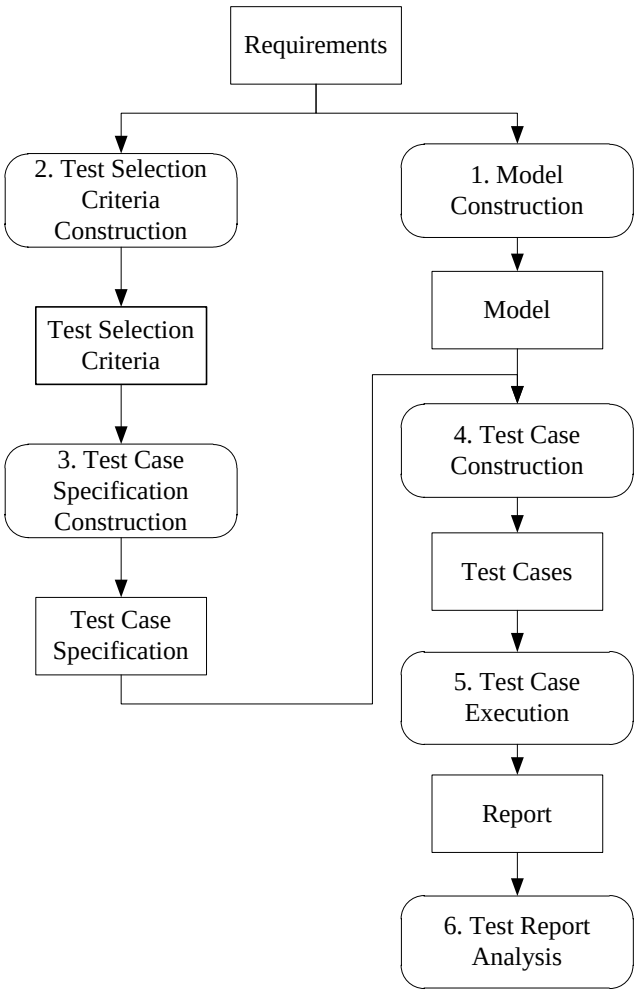

Figure 2.4. Process of model based testing

2.2.2. Architecture Based Testing

In architecture based testing system is test using the information presented in architecture model of the system. Current literature performs architecture based testing at two levels for different test concerns which are architecture and code level testing. Architecture-level testing and code-level testing performed using architectural relations and components specified in the architecture description. Architecture level testing carried out by testing architectural properties of the system. In this thesis we focus on architecture based testing at code level for architecture to code conformance concern for different views of the system. Different approaches for architecture based testing have been offered and these approaches are analyzed by us using systematic literature review technique.

Chapter 3

Model-Driven Architecture Based Testing: A Systematic Literature Review

In general, exhaustive testing is not possible or practical for most real programs due to the large number of possible inputs and sequences of operations. Because of the large set of possible tests only a selected set of tests can be executed within feasible time limits. As such, the key challenge of testing is how to select the tests that are most likely to expose failures in the system. Moreover, after the execution of each test, it must be decided whether the observed behavior of the system was a failure or not (the oracle problem). In the traditional test process the design of test cases and the oracles as well as the execution of the tests are performed manually. This manual process is time consuming and less tractable for the human tester. Model-based testing (MBT) relies on models of system requirements and behavior to automate the generation of the test cases and their execution. A model is usually an abstract, partial presentation of the desired behavior of a system under test (SUT). A model act as an oracle for the SUT since it defined the intended behavior. In addition the structure of the model can be exploited for the generation of test cases. Test cases derived from such a model are collectively known as an abstract test suite. Based on the abstract test suite a concrete test suite needs to be derived that is suitable for execution. Hereby, the elements in the abstract test suite are mapped to specific statements or method calls in the software to create the concrete test suite. The generated executable test cases often include an oracle component which assigns a pass/fail decision to each test. Because test suites are derived from models and not from

source code, model-based testing is usually seen as one form of black-box testing.

Model based testing can use different representations of the system to generate testing procedures for different aspects of the software systems. Example models include finite state machines (FSMs), Petri Nets, I/O automata, and Markov Chains. A recent particular trend in MBT is to adopt architecture models to provide automated support for the test process.

So far, many approaches have been introduced but no explicit effort has been undertaken to provide a systematic overview on the existing literature. In this study we adopt a systematic literature review (SLR) approach based on Kitchenham's guidelines.

The remainder of the chapter is organized as follows: Section 1 of the chapter describes the overall background on architecture-based testing, architecture modeling and systematic literature reviews. Section 2 discusses the overall protocol of the adopted in SLR. Section 3 presents the results of the adopted SLR protocols. Section 4 presents discussion over the SLR. At last section 5 presents the conclusion of this chapter.

3.1. Background

3.1.1. Architecture Based Testing

Software testing is process of verifying and validating software against its expected requirements. Testing is continuous process in software development life cycle and can be performed at different levels in the cycle. Software design phase is one of the levels where testing can be applied due to various benefits. Architecture based testing is a testing type exploiting the knowledge in the design phase to test the software system within different abstraction levels. One of the benefits using architecture based testing is to detect defects earlier at software development life cycle. This prevents the propagation of defect to other levels of software development life cycle such as implementation and integration of the system. In architecture based testing the software architecture implementation (SA) is validated and verified against the SA specifications provided. Below process model of model driven architecture based testing (MDABT) is given. This process model or "pattern" is extracted from the thoroughly analyzed studies that are involved in this literature review.

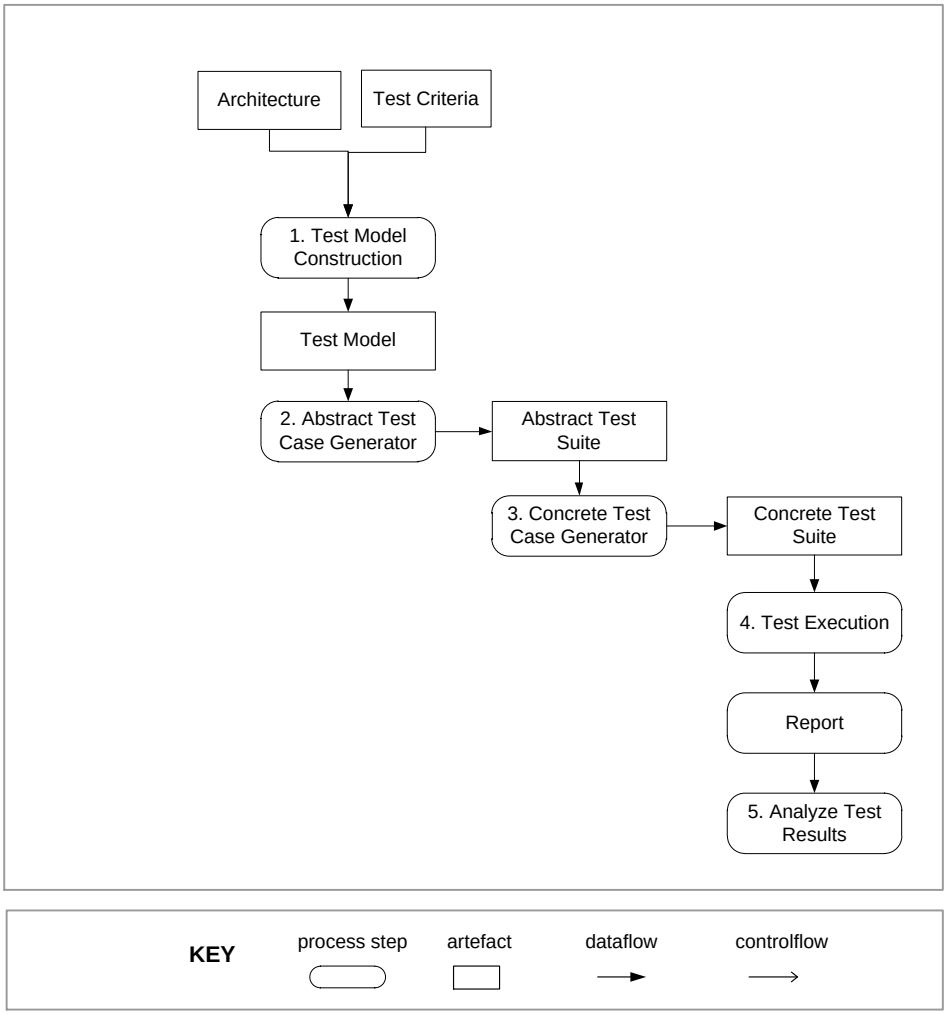

Figure 3.1. Process model of MDABT

Based on Figure 3.1 we can identify the following issues that are present for realizing MDABT:

- *Description of the Architecture*

In order to use the architecture for the purposes of MBT it should be properly described using a well-defined representation mechanism or language. The provided representation can be refined to other representations for purposes of analysis.

- *Description of Test Criteria*

Testing is carried out based on predefined testing goals and testing criteria. For example, the criteria might be based on coverage of graph paths. It is important to specify these criteria.

- *Generating Test Model based on Architecture*

Based on the architecture and the provided test criteria the required test model needs to be generated. The generation process can be carried out in different ways and may depend on the provided representations.

- *Test Case Generation based on Test Model*

Based on the provided test model test cases need to be generated. Different approaches might apply different generation approaches and adopt different representations for the test cases. Furthermore, test case can be defined in multiple steps and usually a distinction is made between abstract test cases and concrete test cases.

- *Test Execution*

Once the test cases have been derived these are executed on the real code or on the architecture of the system. The execution can be carried out in different ways.

- *Analysis of test results*

The final step of the process is the analysis of the test results which might be again represented in various ways. The analysis can be manual or automated.

Each approach will realize the MDABT process differently. Further some approaches might not apply all the steps that are described above, but focus on particular steps instead.

3.1.2. Systematic Literature Review

A systematic literature review is method of synthesis for identifying, evaluating and clarifying all filtered research with respect to specific research questions or research area [5]. Systematic literature review (SLR) is a secondary study taking primary studies as input for a synthesis. There are many motivations for taking on SLR approach such as summarizing the existent methods for research, finding out gaps in current techniques and discovering new research areas [5]. Summarizing the existent methods or techniques for specified research area will provide an overview and background of the domain. Overview and background information extracted can reveal limitations, benefits and pre-required conditions needed for methods in the research area. Moreover, SLR helps to find out the gaps or holes in the current research studies indicating areas for further examination. Furthermore, SLR studies can discover new research areas or

directions within the specified research areas. There are both advantages and disadvantages of applying SLR method. The disadvantage is that it consumes too much time. On the other hand, results of the SLR are more likely to be unbiased. SLR provides knowledge across different configurations and methods. Moreover, data in quantitative studies can be united by meta analytic techniques. The SLR method first applied successfully in the field of medicine for the need of evidence based research. Then it is adopted by other fields such as organic chemistry, education and psychology. Similarly, evidence-based software engineering emerged along with the guidelines for performing systematic literature reviews [6]. The main motivation of the evidence-based software engineering is to enhance the solutions for quality concerns as well as give primary knowledge to different stakeholder groups. In this chapter we aimed at identifying and evaluating the evidences and studies regarding the MDABT. As a result, a systematic literature review was an applicable and satisfactory research method for our research.

3.2. Research Method

As mentioned SLR is a method of synthesis identifying, clarifying and evaluating whole set of research studies answering specific set of research questions or related to a specific research area [5]. In this study we conducted the SLR for identifying and evaluating the existing evidences and studies in the MDABT area. Kitchenham and Charters [5] guideline of systematic literature review in software engineering is accompanied in this study. The further subsections of this section explain review protocol and steps directed in the guideline [5].

3.2.1. Review Protocol

Firstly, we have created a review protocol before accompanying a systematic literature review which can be seen in Figure 3.2. Review protocol identifies the approaches which will be utilized to accompany systematic literature review. Pre-defining review protocol will greatly reduce researcher bias.

To begin with research questions of the systematic literature review is identified. Then scope and strategy of our research study searching is defined. Search scope is a means of identifying platforms that studies are published and publication dates to be considered while picking out a set of possible studies. Search strategy on the other hand, is a means of identifying keywords for search queries which is a fundamental part of process for applying full scan on the research area. After

the definition of search strategy, definition inclusion and exclusion criteria are performed. In this study we have defined exclusion criteria to pick out studies which are not actually in our research area. Next step in our protocol is the definition of quality assessment where the criteria identification performed, in the previous step is detailed by assessing studies over set of attributes and picking the ones that are not satisfying the quality requirements. This step is then followed by the design of data extraction form for extracting information from studies answering the research questions. In order to design the data extraction form, we apply data extraction on sample studies for singling out the properties or attributes that will be extracted in data extraction form. At last, definition of data synthesis method is introduced for managing the extraction of attributes or properties from the studies.

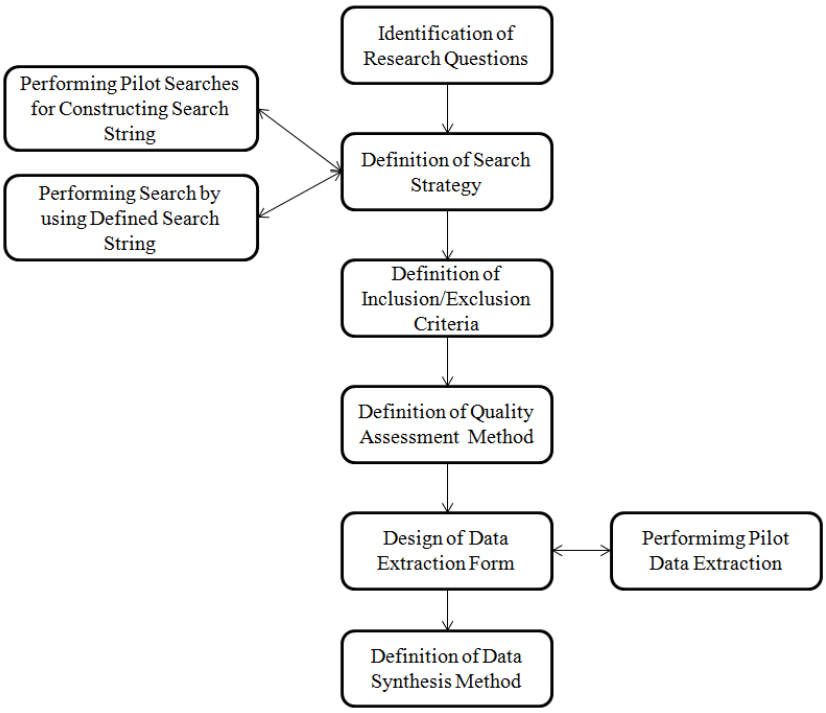

Figure 3.2. Review protocol

3.2.2. Research Questions

The fundamental step of the SLR is identification of specific and valid research questions. The selected studies after applying review protocol must answer all research questions defined. Research question must be valid in order to be answered by each study in the domain. We have identified the following research question which should be answered by the selected set of studies. In this study we are interested in conducting the following research questions which are...:

RQ.1: What are the addressed concerns for applying model-driven software architecture based testing?

RQ.2: What are the proposed solutions in architecture-based testing?

RQ.3: What are the existing research directions within architecture-based testing?

3.2.3. Search Strategy

One of the main motivations of SLR is to identify primary studies answering the specified research questions utilizing well-defined search strategy. In the subsections we will provide search scope and method details which are two subparts of our search strategy definition.

Scope

Our search scope includes two attributes which are publication date range and publication platforms. We have selected the range of dates between period of 2000 and April 2014 for our publication date range attribute. We have selected start date as 2000 because first paper having strong foundations on the topic with a proper case study explaining overall method is published at this date. We included reputable publication platforms which are listed below for identifying journal papers, conference papers and workshop papers.

- IEEE Xplore
- ACM Digital Library
- Wiley Interscience
- ScienceDirect
- Springer
- ISI Web of Knowledge

Search Method

To search a database we used two search methods, automatic search and manual search. Automatic search is performed by executing search strings on search engines of electronic data sources. Manual search is conducted by manually browsing journals, conference proceedings or other important sources. Since the selected databases include thousands of published papers, manual search becomes very time consuming. For this reason, we conducted automated search by using search string.

Search String

We have identified a search string for each publication platforms listed in our search scope for retrieving as much possibly related studies we can. Each platform has different features, attributes to query for primary studies we are interested in. We have defined queries for each platform using each platform search language. Search strings for each platform is located in Appendix A of the study. In Table 1 result of overall search process after querying for each platform can be seen.

Table 1. Overview of search results and study selection

Source	Number Of Included Studies After Applying Search Query	Number Of Included Studies After EC1-EC4 Applied	Number Of Included Studies After EC5-EC8 Applied
IEEE Xplore	50	21	3
ACM Digital Library	8	8	2
Wiley Interscience	30	9	0
Science Direct	56	14	4
Springer	4	3	2
ISI Web of Knowledge	10	6	1
Total	158	61	12

In the first column each platform listed in the search scope is shown. As can be seen in the second column 158 studies are retrieved after executing the search strings in the platforms. Third and fourth column of the table shows the filtered studies after applying study selection criteria explained in the following section. At the last step of the process 12 studies have been identified as the primary study for SLR.

3.2.4. Study Selection Criteria

Search query strings provide large range of primary studies over the domain so that we will not miss any related studies. Study selection criteria are defined so that studies that are not directly related to our domain are excluded. Defined criteria are manually applied on the set of selected studies. In the following part we provide exclusion criteria:

- EC 1: Papers in which the full text is unavailable.
- EC 2: Papers gathered as duplicate or similar at different platforms.
- EC 3: Papers are not written in English.
- EC 4: Papers do not relate to architecture based testing.

- EC 5: Papers do not explicitly discuss architecture based testing.
- EC 6: Papers which are experience and survey papers.
- EC 7: Papers don't validate the proposed study.
- EC 8: Papers that provide only a generals summary without a clear contribution.

3.2.5. Study Quality Assessment

We have defined a method for study quality assessment method for giving further detailed exclusion or inclusion criteria. The importance or validness of the study is examined by quality assessment method we provide. Our defined method is based on quality attributes which are checklist of properties that each study assessed by [5]. The checklist created by taking in account the properties that could bias study results. Defined quality assessment method maintains the summary quality checklist for quantitative studies and qualitative studies which is proposed on [5]. Table 2 provides our quality checklist for quality assessment method. Our main motivation in adopting quality checklist is to assess a study by an overall quality score. Therefore we utilized set of scores from three point scale which are yes (1), somewhat (0.5) and no (0). Results for each primary study filtered by study selection criteria are presented in Appendix B.

Table 2. Quality checklist

No	Question
Q1	Are the aims of the study is clearly stated?
Q2	Are the scope and context of the study clearly defined?
Q3	Is the proposed solution clearly explained and validated by an empirical study?
Q4	Are the variables used in the study likely to be valid and reliable?
Q5	Is the research process documented adequately?
Q6	Are all the study questions answered?
Q7	Are the negative findings presented?
Q8	Are the main findings stated clearly in terms of creditability, validity and reliability?
Q9	Do the conclusions relate to the aim of the purpose of study?
Q10	Does the report have implications in practice and results in research area for model-driven software architecture testing?

3.2.6. Data Extraction

Data extraction is performed by reading all 12 selected primary studies for answering each research question. Furthermore, data extraction form is designed for retrieving all the information to answer research questions and all the attributes for study quality assessment criteria. Data extraction form contains set

of attributes such as identification number of the study, date of data extraction year, publication year, authors of the study, platform of the publication, and type of the publication. Extraction of data purpose columns are inserted in to the form as well by study description and evaluation parts which can be seen in Appendix D.

3.2.7. Data Synthesis

Data synthesis is the most important part of the SLR process in which the extracted data from the primary studies is summarized and research questions are answered [5]. In this study we implemented both qualitative and quantitative synthesis on the extracted data that enables us to result the foundation of both perspectives. We examined if the qualitative results enable us to clarify any quantitative results as well. The results of the synthesis are provided in the next section for both perspectives of the selected studies.

3.3. Results

3.3.1. Overview of Reviewed Studies

This section of the study presents the publication year distribution and the publication platforms of the 12 selected primary studies. Figure 3.3 shows the publication year distribution of the selected primary studies.

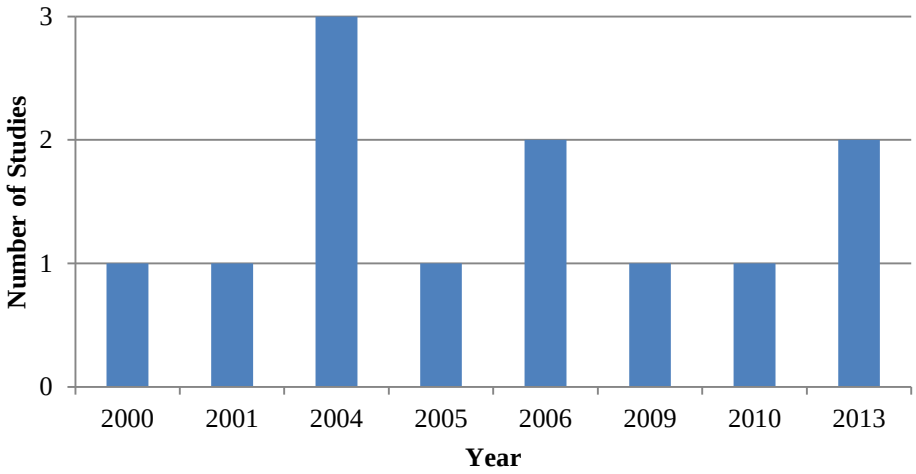

Figure 3.3. Year wise distribution of the primary studies

In Table 3 publication sources and channels of the selected studies are presented. This table also presents the publication type and distribution of studies over the attributes. We can infer that selected primary studies are published at very reputable publication sources such as IEEE, ScienceDirect, ACM and Springer.

Moreover, "Electronic Notes in Theoretical Computer Science" is one of the reputable publication channels having remarkable studies published related to theoretical computer science. Another reputable publication channel listed below is "Information and Software Technology" which is highly reputable for publication in the area of software engineering.

Table 3. Distribution of studies over publication source

Publication Channel	Publication Source	Type	Number of Studies
Information and Software Technology	ScienceDirect	Article	1
Formal Methods for Software Architectures	Springer	Chapter	1
Fundamental Approaches to Software Engineering	Springer	Chapter	1
Applying Formal Methods: Testing, Performance, and M/E-Commerce	Springer	Article	1
Software Engineering, 2000. Proceedings of the 2000 International Conference	ACM	Article	1
Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium	IEEE	Conference	1
Software Engineering, IEEE Transactions on (Volume:30 , Issue: 3)	IEEE	Article	1
Proceeding ROSATEA '06 Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis	ACM	Article	1
Electronic Notes in Theoretical Computer Science	ScienceDirect	Article	1
Information Technology: New Generations (ITNG), 2010 Seventh International Conference	IEEE	Conference	1
Information and Software Technology	ScienceDirect	Chapter	1
Journal of Systems and Software Volume 91	ScienceDirect	Article	1

3.3.2. Research Methods

Empirical studies having well-defined research methodologies are fundamental part for relying and validating the findings of the studies. Selected primary studies must state and utilize the research methodology. In Table 4 types of the research method applied in the selected primary studies are presented. Three types of research methods are identified during the review processes which are case study, experiment and short example. It can be identified from table that case study research method is widely used in the selected studies. Moreover, experiment and short examples are used in the selected studies as well.

Table 4. Distribution of studies over research methods

Research Method	Studies	Number	Percent
Case Study	A, C, D, E, F, H, K	7	58
Experiment	B, L	2	16
Short Example	G, J	2	16
None	I	1	10

3.3.3. Methodological Quality

This section provides quality results of the selected studies which are relevance, quality of reporting, rigor and credibility. Quality results are calculated from the quality checklist which was given in the previous sections and three point score range. First three question of the quality checklist forms the attributes of reporting quality. Moreover, last two questions are used for relevance quality. Fourth, fifth and sixth questions are used in the calculation of rigor quality. Seventh and eight questions are the assessment questions for credibility quality. In Appendix C, the result of the quality checklist is presented.

Figure 3.4 shows the reporting quality of the studies according to the first three questions of the quality checklist. It can be seen that 91% of the primary studies are perfect and 9% percent of the primary studies is very good. In this case perfect means that studies reporting quality calculated as three which is the full point. Good on the other hand means that study lost points in one of the questions and gathered two and half point.

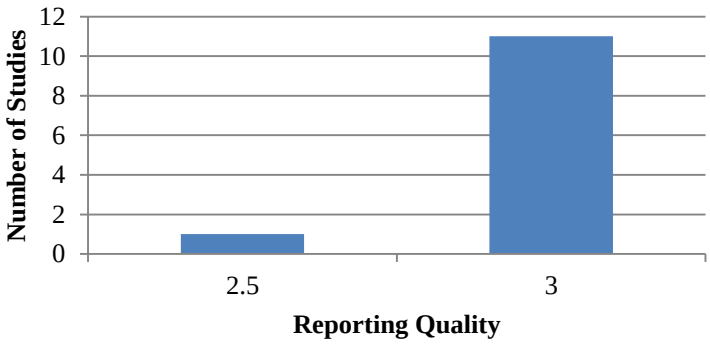

Figure 3.4. Reporting quality distribution of the primary studies

Rigor quality of the study refers to the trustiness of findings of the study. Figure 3.5 shows that 75% of the studies are perfect in terms of rigor quality. Moreover, 16 % of the studies assessed as very good. However, 9% of the studies ranked as good rigor quality.

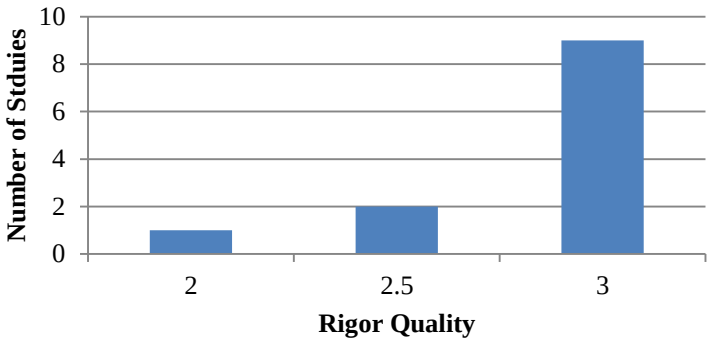

Figure 3.5. Rigor quality distribution of the primary studies

Another quality measure is the relevance quality of the primary studies. Figure 3.6 shows relevance quality scores calculated from ninth and tenth question of the quality checklist. It can be seen that 33% percent of the studies is directly relevant to MDABT. The rest of the studies are relevant to MDABT.

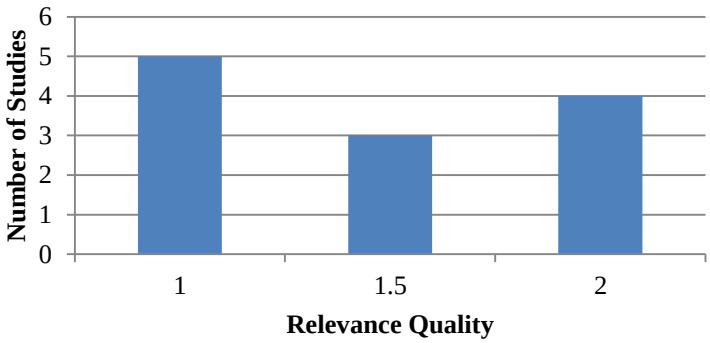

Figure 3.6. Relevance quality distribution of the primary studies

At last credibility quality of the studies are calculated by using the seventh and eight questions. Figure 3.7 presents the credibility quality score and distribution of the studies. It can be seen that 83 percent of the studies calculated as 1 point. Remaining 17% of the studies calculated as 0.5 point. According to our evaluation there is no primary study that has full credibility in terms of evidence. All studies are missing the statement of counter example for their suggested approaches.

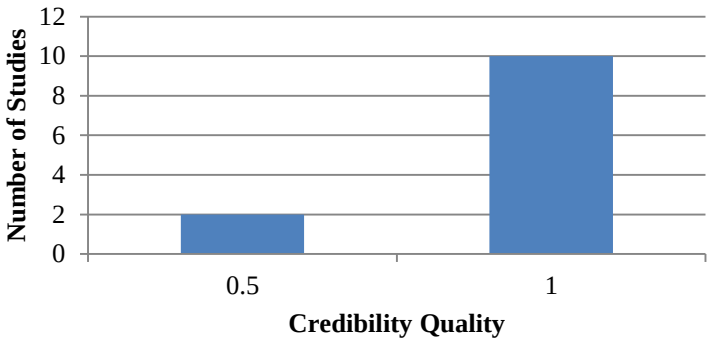

Figure 3.7. Credibility quality distribution of the primary studies

To sum up, summary of overall methodological quality scores of selected primary studies are given in Figure 3.8. Total quality is calculated by adding up reporting, relevance, rigor and credibility qualities. We consider 8.5-9 as high quality, 7.5-8 very good quality and 7 as good quality. It can be seen that 41% studies have high overall quality. Moreover, 41% of the studies have very good overall quality and 18% of the studies have good overall quality.

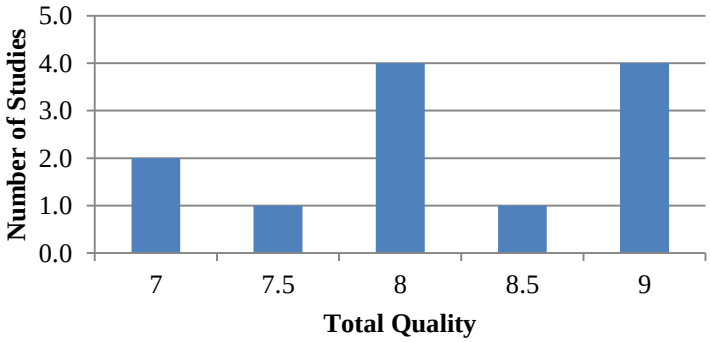

Figure 3.8. Total quality distribution of the primary studies

3.3.4. Systems Investigated

This section provides the results that are extracted from selected primary studies for answering the research question specified in the previous sections.

RQ1: What are the addressed concerns for applying model-driven software architecture based testing?

Target domain analysis of the 12 selected primary studies is performed and the results are shown in Figure 3.9. Figure presents addressed concerns over the distribution of the primary studies. There are 2 main concerns that are addressed which are functional concerns and code to architecture conformance. The functional concerns are functional requirements that are tested on architecture or code level. For instance architecture level functional requirement can be graph

representation of architecture cannot have tangling node whereas at the code level this can be the validation of the functional requirement.

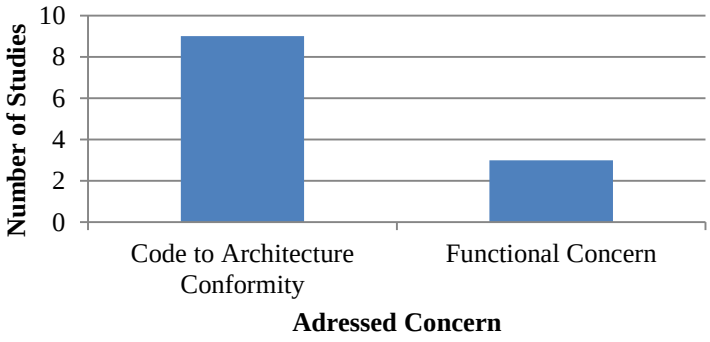

Figure 3.9. Addressed concern distribution of the primary studies

Furthermore, Table 5 shows the addressed concern of each study. As table present studies B, F and J addresses function concerns and other studies addresses code to architecture conformity concern.

Table 5. Addressed concern and study map

Addressed Concerns	Studies
Code to architecture conformity	A, C, D, E, G, H, I, K, L
Functional concern	B, F, J

In study A, authors are explaining how to derive test plans from software architecture (SA) specifications. Architecture based testing is performed for conforming implemented system with the specified SA. The main motivation of this paper is to use SA specifications for integration testing of the implemented system.

Study C is the continuation of the study A in which concerns and motivations are same A's. In this study authors explain how they fill the gap between the abstraction of SA and implementation using architectural and code level sequence diagrams.

In study B, authors define test criterion to use in SA based testing. The testing is done at architectural level to test functional properties of SA. The main motivation of this paper is to define formal testing criteria based on architectural relations.

In study D, authors explain how they provide systematic way to perform the refinement step. In this study specific architectural style is used which has mapping between SA and code based test cases to delivery completely systematic SA based approach. The testing is performed for code to architecture

conformance. The main motivation of this study is to exploit architectural style information to create mapping between component in SA and units in implementation.

In study E, authors uses both model checking and SA based testing approaches. The addressed concern of this study is code to architecture conformance and the main motivation is to select architecture level test cases using the output generated from model checking stage.

In study F, authors perform SA based testing at architectural level for testing specifications against functional properties of SA. The main motivation of this paper is to validate architectural units using object oriented models.

In study G, the addressed concern is code to architecture conformity. The main motivation of the paper is to perform validation at different abstraction levels of system under test using system goals.

In study H, authors used their previous works on architecture based testing for regression testing of software system. The addressed concern of code to architecture code conformity is regressively tested in this approach. The main motivation of this paper is to use architecture based testing for regression testing of systems.

In study I, both model checking of SA and architecture based testing approaches are experimented. The addressed concern of the study is code to architecture conformity. Moreover, the main motivation of this paper is to use Architecture Analysis and Design Language (AADL) specifications in verification of software system.

In study J, author addresses the concern of functional properties of SA. The testing is performed at architectural level. The main motivation of this paper is to use SA in model based testing to detect defects earlier in software lifecycle.

In study K, service oriented applications (SOA) are tested which is said to be more challenging than monolithic applications. The main motivation of this paper is to use SA based testing to solve observability and controllability problems that are created by message exchanges between the services that are hidden behind the front service of the system. The addressed concern of this study is code to architecture conformity.

In study L, authors use SA based testing approaches to variant-rich software systems. The main motivation of the study is to address the challenge of ensuring correctness of component implementations and interactions with any system variant. The addressed concern of this study is code to architecture

conformity. Transformation definition is executed by a transformation engine. It reads source model and outputs the target model. The transformation can be unidirectional or bidirectional based on the transformation definition.

RQ2: What are the existing solutions?

Study A

This study presents the adoption of chemical abstract machine (CHAM) specifications to represent software architecture and derive test cases from these specifications using coverage criteria. Test model is generated using CHAM specifications and coverage criteria which is label transition system (LTS). From this graph abstract labeled transition systems (ALTS) are obtained simply by applying obs function. Obs functions are functions that exclude unnecessary details for the selected view of the software architecture. Test cases are generated using the paths from ALTS graph. Each path can correspond to different concrete test cases. Test cases are generated manually by software architect. Test execution and test analysis are handled manually by software architect. In Figure 3.10 process model of this study can be seen.

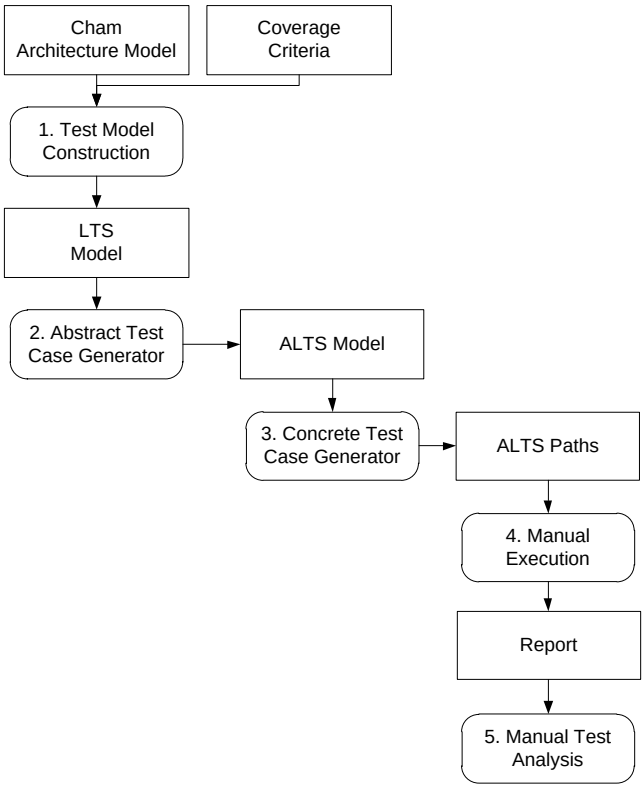

Figure 3.10. Process model applied in study A

Study B

This study presents testing at architecture level for conformance of SA properties which are pointed as test criterion they define. In this case SA is specified using Wright ADL and six test criteria are defined based on data flow reachability, control flow reachability, connectivity and concurrency. These criteria are used as functional properties of SA to be tested and verified. In this study test model is based on behavior graph (BG) and obtained by transforming Wright ADL specifications into BG using coverage criteria as the test criteria. From the BG test model test paths are generated using the tool they created called ABaTT. Each test path is manually transformed into test cases. The test cases and test case results are automatically handled. In Figure 3.11 process model of this study can be seen.

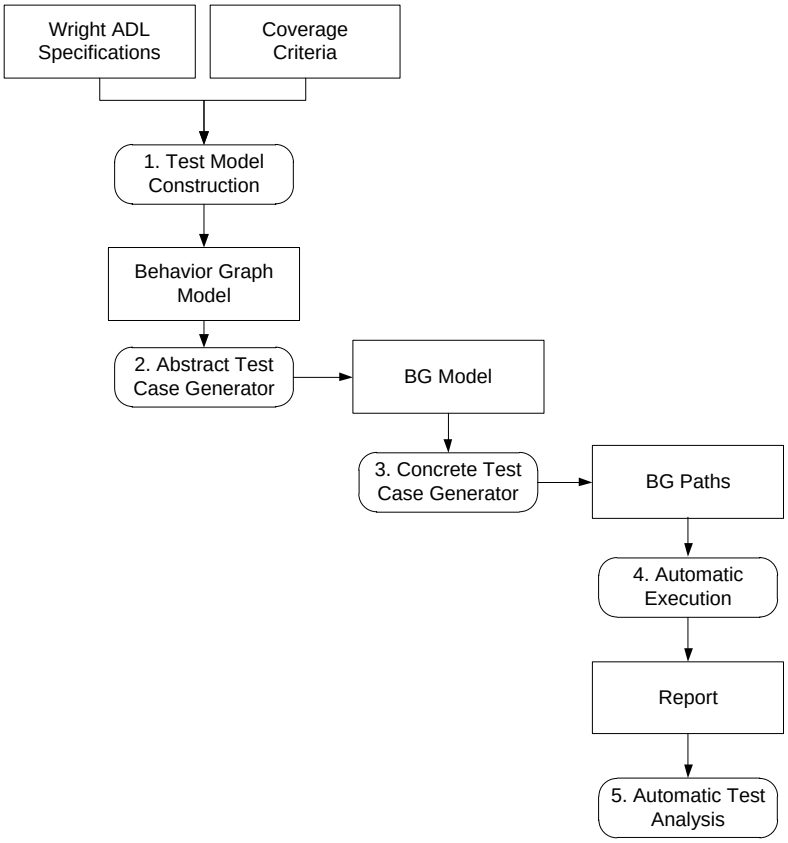

Figure 3.11. Process model applied in study B

Study C

This study presents the continuation work of previous authors in A by replacing their SA specification of CHAM model to FSP (Finite State Process) model. The reason to use FSP instead of CHAM model is stated as FSP algebra is easier to map to LTS graph. In this study authors explains the work of the software architect to generate test cases and execute them manually. This study presents the use UML Stereotyped Sequence Diagrams for filling the abstraction gap between the SA and implementation. For each architecture level sequence diagram, software architect defines code level sequence diagrams. Global sequence diagram is obtained by combining the code level sequence diagrams where it represents code scenarios that is implementing SA path that is extracted from ALTS graph. Software architect then runs the code to see if the created sequence diagram is implemented by the system.

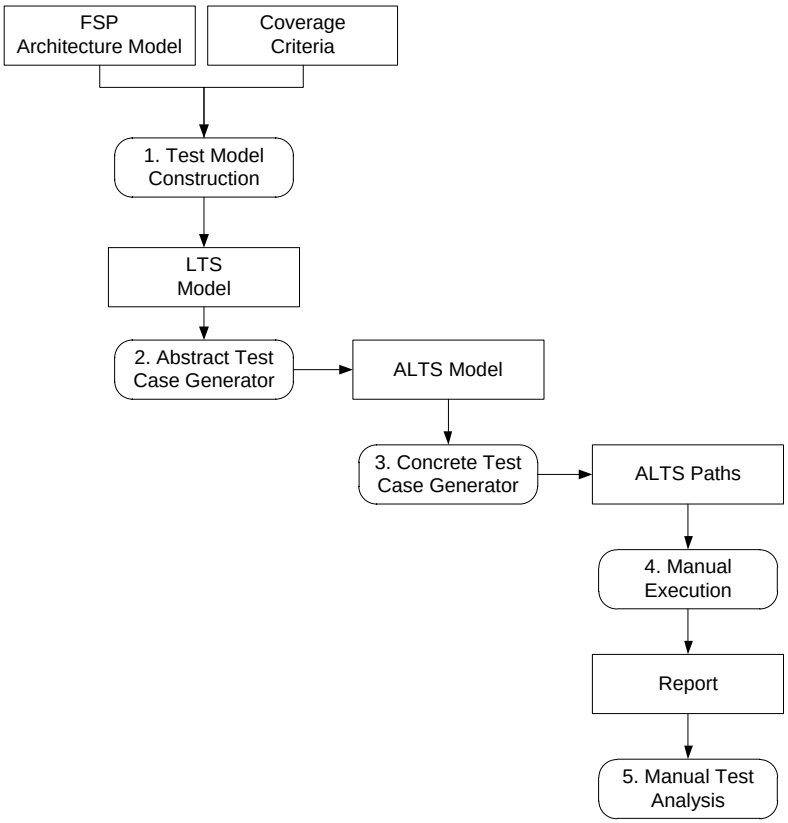

Figure 3.12. Process model applied in study C

Study D

This study presents the use of architecture specification for mapping the SA model to implementation. Different from the study C authors added automatic test case generation and analysis reporting. Figure 3.13 shows the process model extracted for study D.

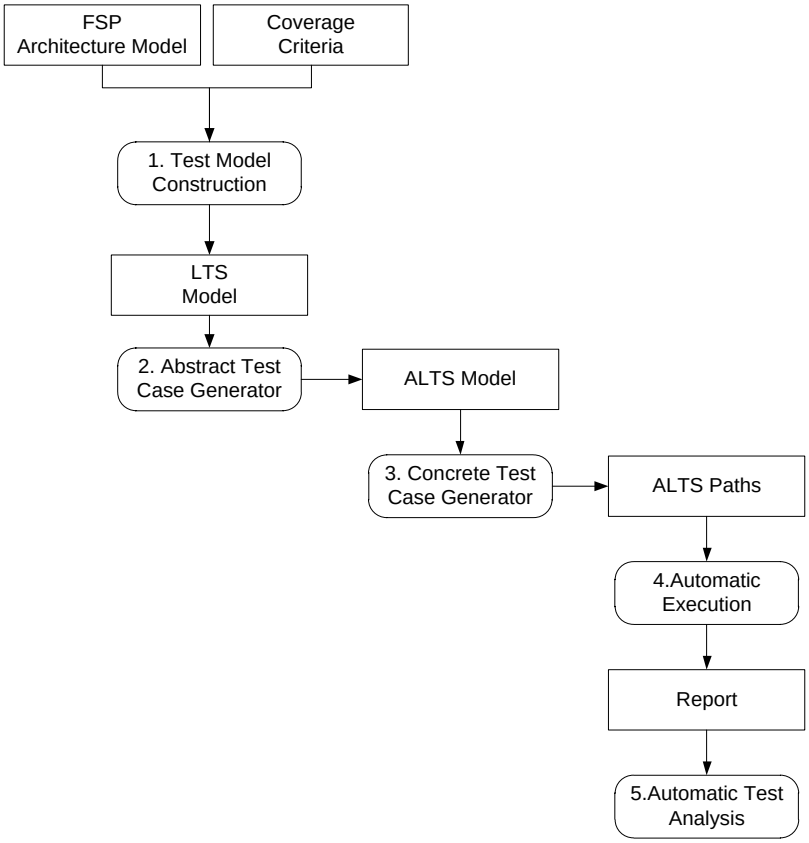

Figure 3.13. Process model applied in study D

Study E

This study uses the same methodology explained in study D. Moreover, they present model checking for software architecture using SPIN and CHARMY framework. SA is specified using CHARMY specifications rather than FSP model. Testing criteria chosen as coverage criteria which is selected as subsystem identification in CHARMY framework. The test model that was LTS now consists of Promela model, LTL Formulae and Buchi Automata models. Test cases that were ALTS paths are chosen as OK and NOK results that are obtained from model checking stage. OK and NOK results stands for successful and unsuccessful verifications obtained after applying model checking on SA. The results are used by test generator engine to create test cases automatically at

the SA level. The test execution and test analysis is handled manually. Figure 3.14 shows the process model for this study.

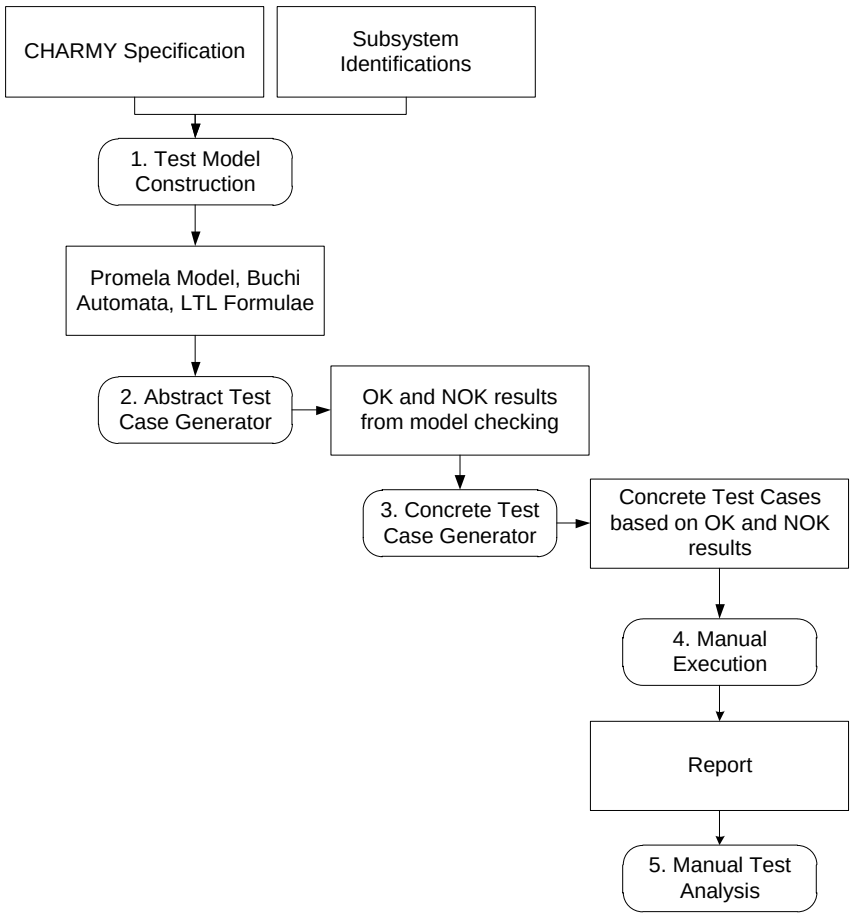

Figure 3.14. Process model applied in study E

Study F

This study presents SA specifications in UML state diagrams, UML sequence diagrams and UML component diagrams for transforming into test model based on LTS called Basic LOTOS (Language of Temporal Ordering Specifications). Basic LOTOS model is combined with test purposes to generate IOLTS model where test cases are generated using TGV tool. Figure 3.15 shows how this study maps to our process model.

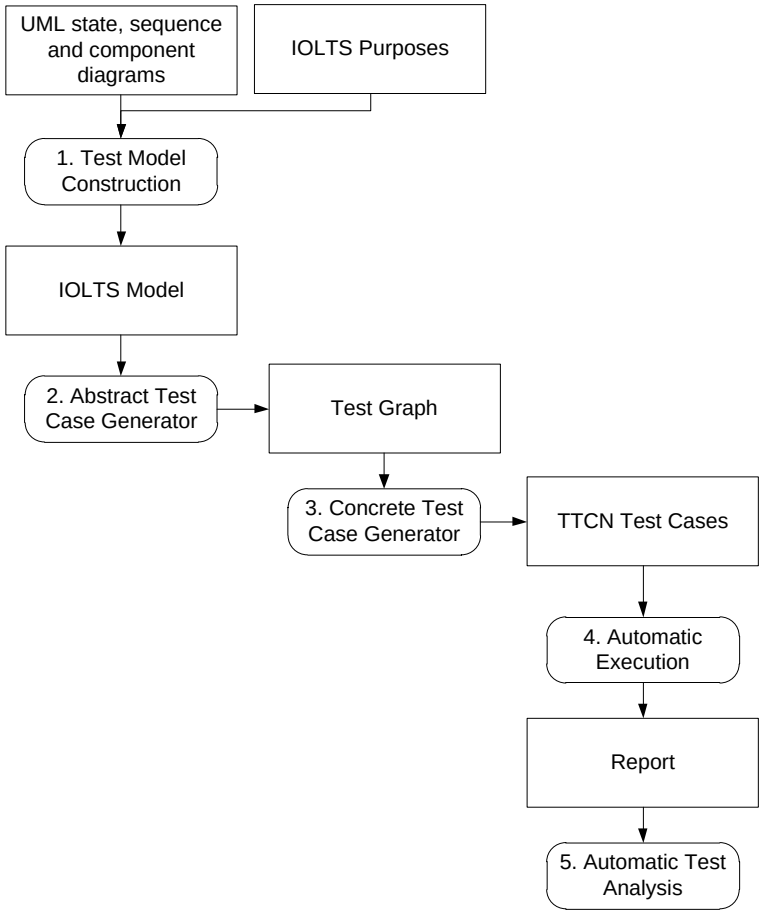

Figure 3.15. Process model applied in study F

Study G

System behavior is implemented by system level plan where system level plan consists of sequence of goals each describing the interactions between the components. Moreover, component level plan shows the sequence that component must achieve in that scenario. Together system level plan and component level plan provide specifications for SA. Test criteria is the scenario that is executed on the system described the interactions in system level plan and component level plan. System level plan and component level plan is used by JESS tool to annotate the implementation code. Then by the pre-compiler the annotations are transformed into codes. Test model is the annotated code itself. There is no test case generation the scenario is given in plans which consists of goals where assertions are defined. As the program executes the goals are emitted using rule based recognizer and plans are tried to be matched. Test execution is automated as the tests are executed during scenario running on the program. Test analysis details are automated as the program executes it matches the component level plan and if all the component level plans are matched

system level plan matches. If the system level plan matches in the executed program then the program passes the test. Figure 3.16 presents the process model of study G.

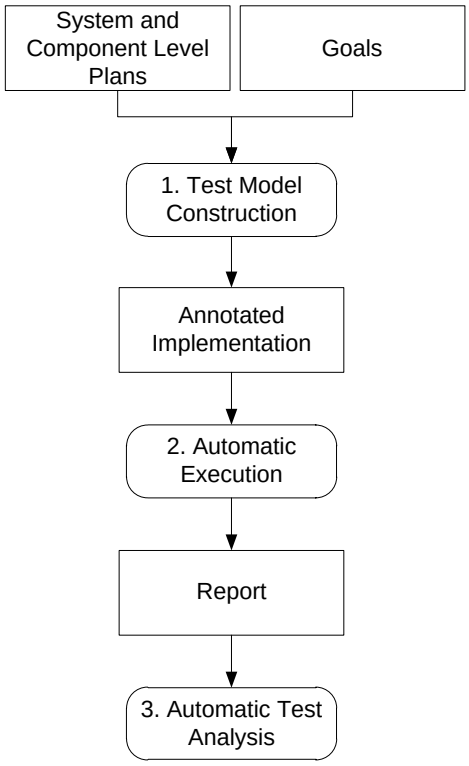

Figure 3.16. Process model applied in study G

Study H

This study uses the same process model in Figure 3.13 of study D.

Study I

This study presents the use of Architecture Analysis and Design Language (AADL) to specify SA which is transformed into Uppaal Model (timed automata) using the coverage criteria as the test criteria. Using the set of automata paths, consistency and completeness check is performed by applying model checking on the model. Moreover, automata paths are translated into concrete test cases using a mapping between architecture specification and implementation. At last, test execution and analysis is automatically handled. Figure 3.17 shows the process model for this study.

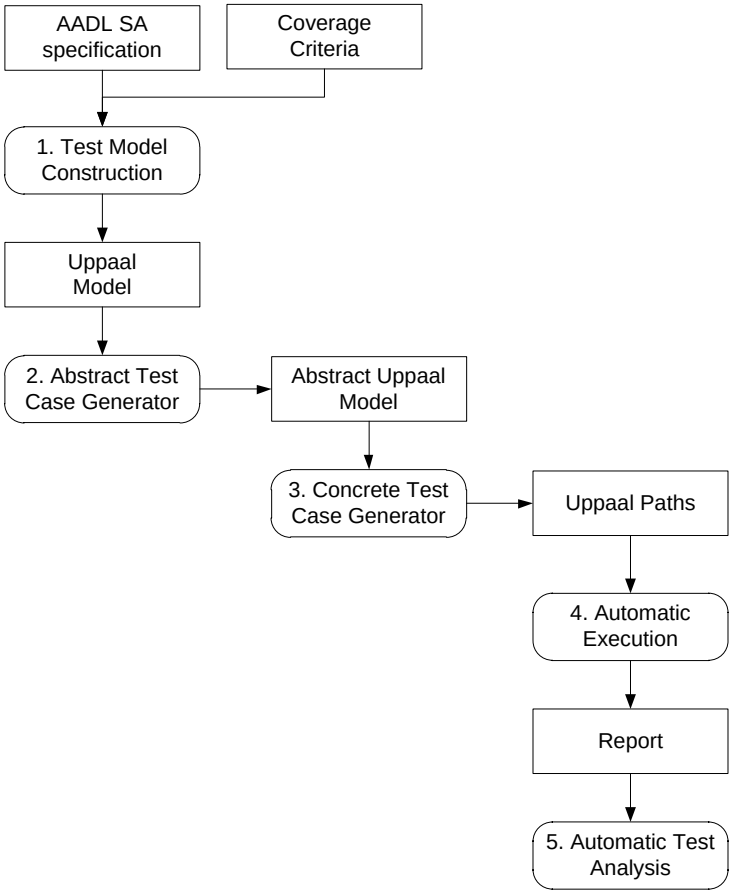

Figure 3.17. Process model applied in study I

Study J

This study presents the use of HPrTNs (a Petri Net) model to explain the behaviors of SA obtained by transforming Acme ADL specifications with coverage criteria as the test criteria. The HPrTNs model is divided into sub graphs which are abstract models depending on the model. Later using sub graphs architecturally significant path are extracted and test cases are generated from the extracted paths. Figure 3.18 shows the process model for this study.

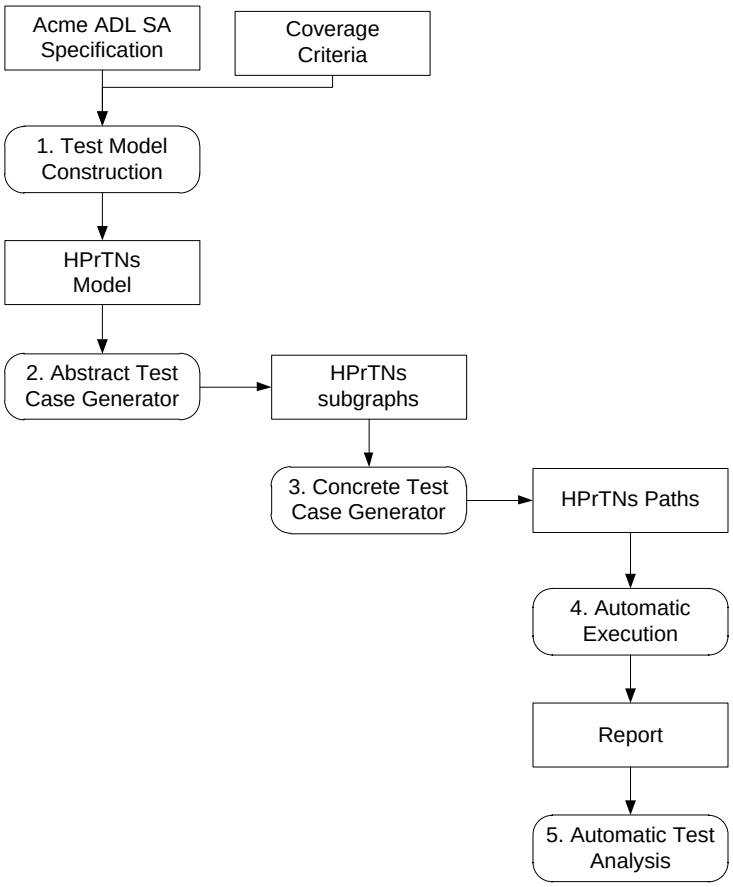

Figure 3.18. Process model applied in study J

Study K

This study presents the application of SA based testing methodology to service oriented applications in distributed systems. The service composition which is our architecture in this case is expressed using BPEL (Business Process Execution Language). In the SA specifications architecture topology and message exchange knowledge are included. The extended control flow graph (ECFG) test model is used to generate test cases by transforming BPEL specifications with coverage criteria into the test model. ECFG consists of control flow graphs (CFG) and from each CFG test paths are derived. Test cases are executed and results are analyzed automatically. Figure 3.19 shows the process model for this study.

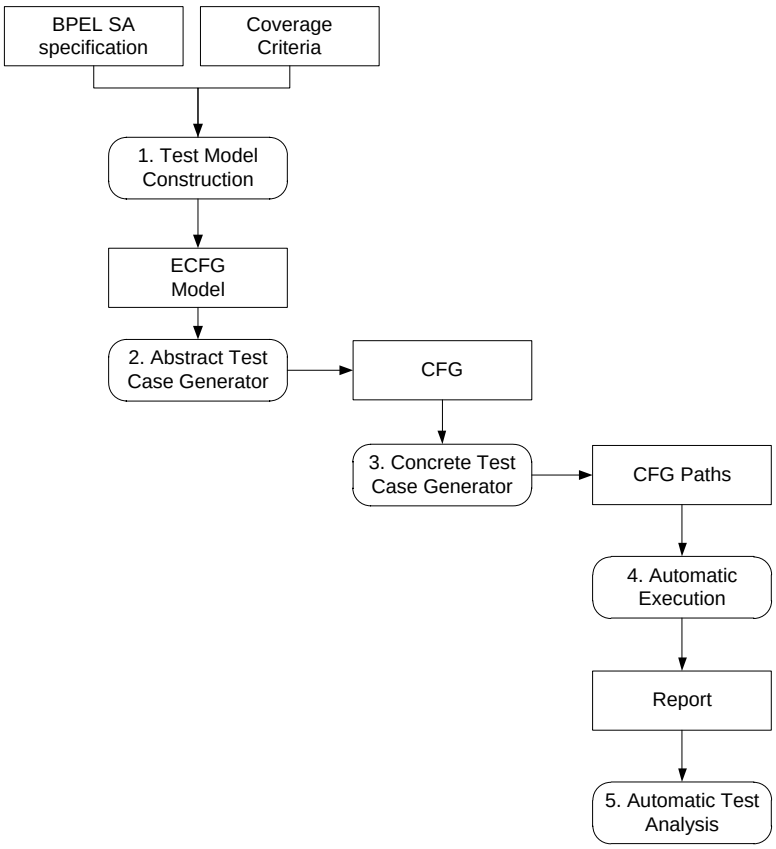

Figure 3.19. Process model applied in study K

Study L

This study presents specification of SA is using UML state, sequence and component diagrams. UML state machines are used for representing component behaviors of SA. UML message sequence diagrams are used for representing component interactions of SA. UML component diagrams are used for representing topological view of SA. All UML models are combined into one test model consists of state machine test model, message sequence chart test model and component test model. In this test model different views are obtained using signals as reducing function for state explosion problem. Test cases are generated from state machine test model are executed on those components which means that testing is done at architectural level. Test cases generated from the message sequence chart are executed in the system which implies that test cases are executed at code level system. Delta SM tool is used for constructing component state machine. DeltARX tool is used for constructing class diagram model. Moreover, IBM Rational Rhapsody ATG (Automatic Test Generation) tool is used for test case generation. Generated tests are based on test execution and analysis is automatically handled.

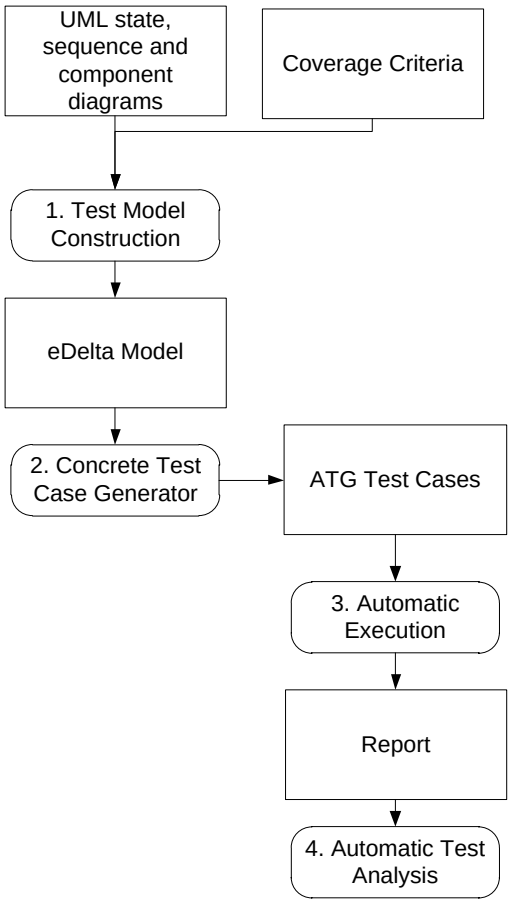

Figure 3.20. Process model applied in study L

Summary

As a conclusion, different studies yields different methodologies to apply architecture based testing on software system under test. As far as now, we have explained how each study maps to our reference process model. At each step of the process model we have seen different techniques applied by authors. Firstly, SA specification is important part of our process model and one needs to represent SA under test in some syntax for ongoing testing procedures. In Figure 3.21 the distribution of the SA specification models over primary studies can be seen. As seen from this graph FSP model and UML diagrams are most chosen SA specification type. FSP model is chosen in three different studies belonging to same author and each work is based on the previous work of the author. On the other hand UML diagrams, which are industry standard for modeling, are chosen by three different studies which belongs to different authors.

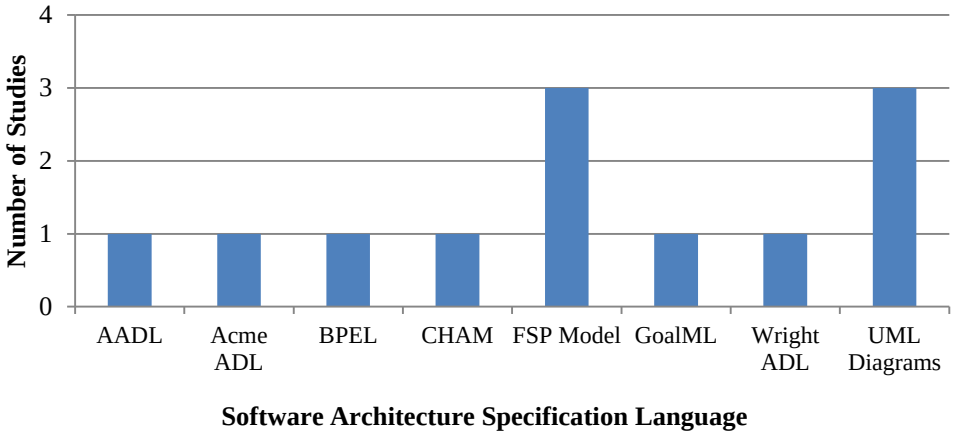

Figure 3.21. SA model type distribution over primary studies

Secondly, as in our process model and studies investigated every SA specification is transformed into a test model where test cases are generated from. In Figure 3.22 test model distribution over studies can be seen. Most of the studies use graph or automata based test models where test cases are generated from extracted paths. In one of the studies explicit test model is not used whereas annotated implementation can be considered as test model. From the following graph, we can infer that LTS is mostly used test model due to same author having four different studies on the subject. Other than that studies adopted different test models for their approaches.

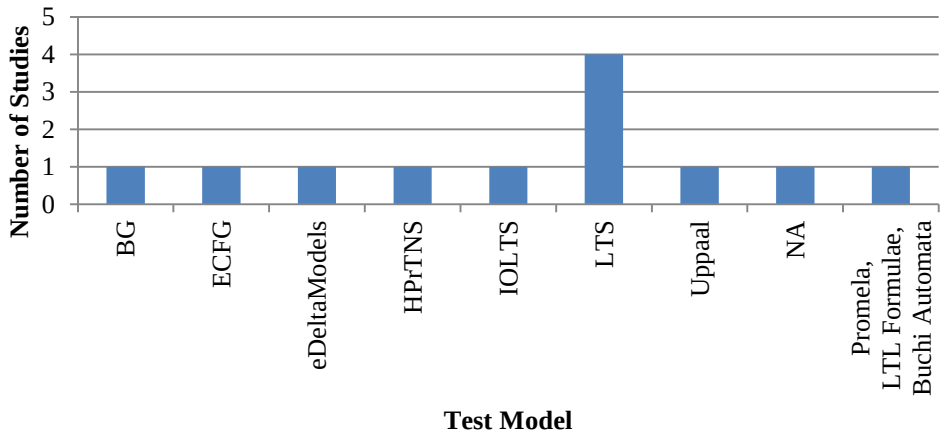

Figure 3.22. Test model distribution over primary studies

Thirdly, the test criteria are important property in refining the test model from SA specifications. In Figure 3.23 distribution of test criteria over studies can be seen. As the test models are based on graphs most of the studies used coverage criteria when generating test cases. Three of the studies used test purposes where test purposes are properties that are refined to be tested. For instance, in study G

test criteria is to match the component and system level plans of the executed system under test.

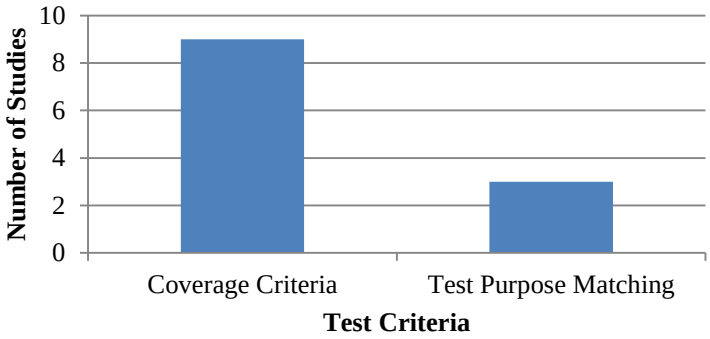

Figure 3.23. Test criteria distribution over primary studies

Moreover, test case generation type is another important property in model driven SA based testing. In order to have fully automated approach test case generation should be handled automatically. Manual test case generation is both time consuming and irrational in the context of model driven approaches. In Figure 3.24 test case generation type distributions over the studies can be seen. Most of the studies use automated test case generation where important aspect in these studies is that testing is carried in SA level rather than code level. In manual test case generated studies there are both SA level and code level testing is performed. Moreover, study that applied successfully using automated test case generation at SA level performed manual when testing at code level without using architecture style. However using the architecture style properties this stage can be automated as done in study D.

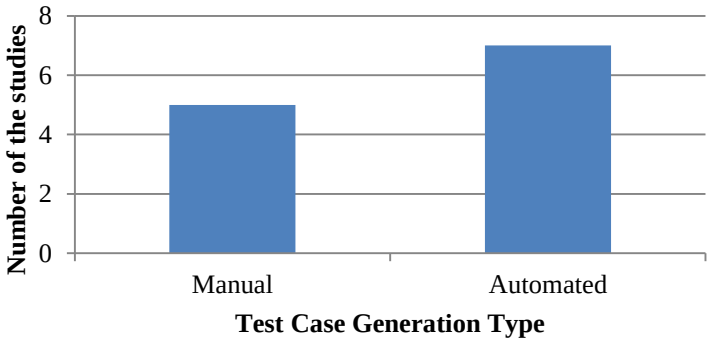

Figure 3.24. Test case generation type distribution over primary studies

At last, test analysis is the last part of our model where the results of the tests that are executed are analyzed whether the test passed or failed. In Figure 3.25 distribution of test analysis types over studies can be seen. In order to have fully automated approach, studies must adopt automated test analysis process. As in

test case generation, having automated test analysis process will save time and easy maintenance of test cases. From the following figure we can infer that most of the studies used automated analysis where the result of failing or passing is automatically determined. On the other hand, one third of the studies used manual test analysis processes.

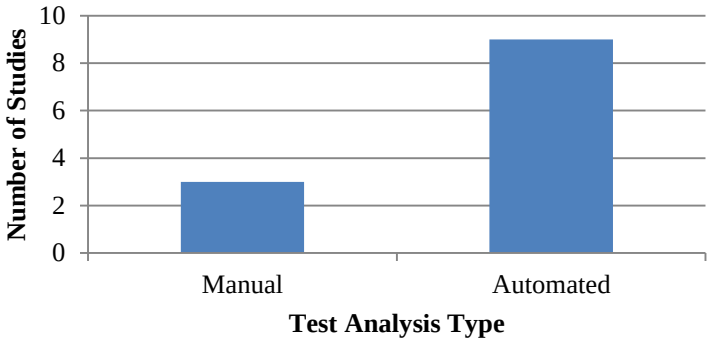

Figure 3.25. Test analysis type distribution over primary studies

RQ3: What are the existing research directions within model-driven software architecture based testing?

Need for executable models for representing architecture

For supporting model-based testing several authors have indicated the need for precise models that can be processed by model compilers. Most of the research in this domain is adopted from the developments in model-driven development in which software language engineering plays an important role. Different studies used different representation techniques for SA specifications. In the current literature authors states that a common SA model needs to be created to represent SA specifications.

Abstraction difference between architecture and code

The adoption of architecture models for supporting model-based testing provides both an abstract view of the system and the actual implementation of the system. Creating tests at SA level is simpler than creating tests at code level from SA specifications. There is a need for filling the gap between the SA and implemented system. One of the studies has proposed to use architecture style for filling the gap between the SA level and implementation level. Still there is a need to exploit architecture definitions at implementation level to generate test cases.

State explosion problem

In testing and model-based testing the generation of test cases easily leads to a combinatorial explosion and becomes less tractable for the human engineer. This appears also to be the case for model-driven architecture-based testing. Although the adoption of model-driven approaches supports the automated generation of test artifacts this intractability problem has also been pointed out by several authors. For example, when generating test cases from test model that is based on graph or automata full path coverage criteria causes numerous test cases. Moreover, some works have used model checking integrated with SA based testing where results from model checking are used for generating test cases. There is still more work needs to be done on this issue as we need to find more effective criteria while generating test cases.

Architecture views

As discussed in the background architecture needs to be modeled using multiple architecture views to represent the various stakeholder concerns. The approaches that we have considered however tend to focus on a single view of the architecture which is usually represented using graph formalisms. A few authors have indicated the need for representing views on the graphs which are more like queries on a single representation format. Yet, existing architecture viewpoints aim to provide a different perspective on the system including, for example, the module views, component-connector views and allocation views. Integrating architecture views, as such, will be a necessary enhancement to the existing architecture-based testing approaches.

Need for automating the process by tools and cost-benefit analysis

When using model driven techniques in some context it is important to automate processes and assessing methods performance by some metrics defined from the context. The automation of the several stages has been studied by different authors in their works. However, none of the work above has the complete autonomous system for MDABT based on our reference process model. Another important issue seems to be present in the current literature is not showing the cost-benefit analysis of the MDABT. There is exciting progress on this topic however none of the studies have systematic way to define benefits of the MDABT according to the cost of the methodology presented. Some authors have stated to work on this issue in their future works.

Need for applying MDABT on complex real system

It is very important to use empirical validations while assessing your methodology. In most of the studies proposed methodologies are assessed on simple case studies such as simple client server applications. On the other hand some studies used industry cases to apply their methodology which are more complex and already in use. Nevertheless, there is still need for applying MDABT on more complex industry cases.

3.4. Discussions

One of the primary dangers to legitimacy of SLR is the publication bias. Publication bias can be explained by researcher's trend to publish positive results than negative results of their studies. It is proposed in [5] to perform research protocol with research question to manage publication bias as we have done in our SLR. Later in the SLR, we identified search scope, search method and constructed search string for different publication platforms to query on the MDABT area. Important aspect of this step is that incompleteness caused by search keywords is another danger for legitimacy of SLR. We have constructed keyword list in a repeated manner by pre-performing sample searches in the domain. The construction of the keyword list is handled manually by adding new keywords if we cannot retrieve related studies to our field. As stated previously search strings constructed for different platforms is located in Appendix-A. Even though having the perfect set of search queries we can still not find related studies such as technical report, theses and company journals. In our study these neglected studies can have importance in terms of completeness and validation by strong case study. In our SLR we did not include such studies. Yet another danger to legitimacy of SLR can be seen the limited functional operations of publication platforms. Publication platforms have limited ability to perform complex queries on the database such as the length of the query and retrieving unrelated studies. As a result of this, study inclusion and exclusion criteria are defined and studies are filtered manually with respect to created criteria. After filtering and selecting the final set of primary studies we extracted data from the primary studies. We have constructed data extraction form to systematically extract data from the studies by reading each study according qualitative and quantitative properties.

3.5. Conclusion

In this study methodological details and results of the systematic literature review on model-driven architecture based testing is presented. As far as we know and investigate systematic literature review is not performed on this research field. Identification of the studies is performed for this field and further on analysis and synthesis of the studies since 2000 are performed. We first identified 158 studies from our query from various publication platforms. We then filtered 12 primary studies related to MDABT which are deeply examined and studied. In this SLR we analyzed the current techniques in MDABT and present the result of the synthesis performed on the selected studies. Moreover, we have discussed on the current literature inadequacy and further research directions for MDABT. As one of our research question is about the addressed concerns for studies in MDABT, we presented each concerns addressed by each study. Moreover, we have explained each study according to our process model and details of the selected studies are given. Current literature for MDABT has remarkable impact and ongoing advances on software testing. However, solutions proposed by current literature have limitations as well. Firstly, proposed solutions do not take architecture views into account in architecture model specification which makes it hard to test concerns of different stakeholders. Another limitation is that most of the studies use different models both for representing SA and test model. This makes it impossible to find and industry standard model such as UML or another common used model to be used by the other works influenced by the current studies. Additionally, most of the proposed solutions have low performance in terms of test case generation methods. The abstraction gap between the SA level and implementation level makes it very hard to test SA conformance to implementation level without human effort. As a result, the main argument is that can we provide a MDABT approach which removes or decreases these problems.

To sum up with this study can be seen as guideline for examining the current literature in the fields of MDABT. Results of this study can boost the improvement of MDABT approaches and can be utilized by new researchers developing new MDABT approaches.

Chapter 4

Model Driven Architecture Based Testing Using Architecture Viewpoints

In the previous chapter we have presented and deeply analyzed current suggested approaches for MDABT using SLR protocol. In this chapter, we present our approach MDABT using architecture viewpoints. First, we will present the generic process model for our approach in section one. Then the technology dependent process model which reflects our implemented approach will be presented in section two. Afterwards, software architecture views and test criteria for each architecture view will be explained in section three. Test model and transformation model will examine in section four for each viewpoint. At last in section five test results and execution will be explained.

4.1. Process

In Figure 4.1 process model for MDABT using architecture viewpoint is presented. In the following parts details of the process model will be explained thoroughly.

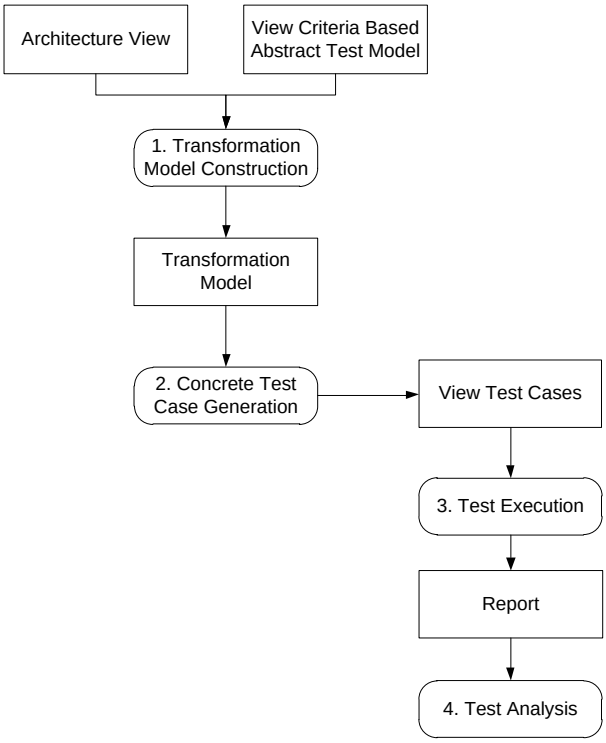

Figure 4.1. Process model of MDABT using architecture viewpoint

- *Architecture View*

Architecture view is the model representing the view of the architecture desired to generate test cases from. Architecture view models can be selected differently for testing purposes such as module views, component and connector views and allocation views. Different architecture views yields different concerns of stakeholders to be tested.

- *View Criteria Based Abstract Test Model*

View criteria based abstract test model is a static model based upon the selected criteria of architecture view for testing. Specific criteria will be defined for each architecture viewpoint and abstract test model will be constructed accordingly. Architecture view and abstract test model will be transformed into concrete test cases using transformation rules declared.

- *Transformation Model Construction*

Transformation model construction process is a static process where transformation model which contains the information of transformation rule to be applied for each architecture view. The construction is performed once for each architecture view. The output of this process is the transformation model

and the inputs are the architecture view and view criteria based abstract test model.

- *Transformation Model*

Transformation model is a rule model based on architecture view and view criteria based abstract test case model. Transformation model is static model created for each view once. Transformation model holds the transformation rule to be applied on inputs given. This model is executed for concrete test case generation.

- *Concrete Test Case Generation*

Concrete test case generation is a dynamic process where the transformation rule stated in transformation model is executed and concrete test cases are generated.

- *View Test Cases*

View Test Cases are concrete test cases based on the view under test. These test cases can be generated on different abstraction levels such as architecture level for testing architecture functional properties or code level for testing the conformance of implementation to architecture model.

- *Test Execution*

Test execution process is a dynamic process where the generated test cases are executed on component under test. The component in this process can be architecture of the system as well as implementation of the system. The result of this process is the test report.

- *Report*

Report is the output of the test execution process and is the input for the test analysis step. Test report contains the information about the result of test case executions (fail/pass).

- *Test Analysis*

Test analysis is the process where test execution report is taken as input results analysis is performed on. This process is responsible for the processes of test oracle.

4.2. Implementation

In the previous section we have provided a generic process for MDABT which is agnostic to the adopted tools. In principle the process can be defined using

different tool implementations. In this section we describe the MDABT environment that we have implemented for the generic process. The implementation is based on the Eclipse Epsilon environment that contains languages and tools for code generation, model to model transformation, model validation, comparison, migration and refactoring [7]. In the Eclipse Epsilon environment the creation of models is based on the Human-Usable Textual Notation (HUTN) each of which needs to conform to predefined metamodels. In Figure 4.2 the implemented process model based on the testing environment that we have defined is shown. In the following we describe each step in detail.

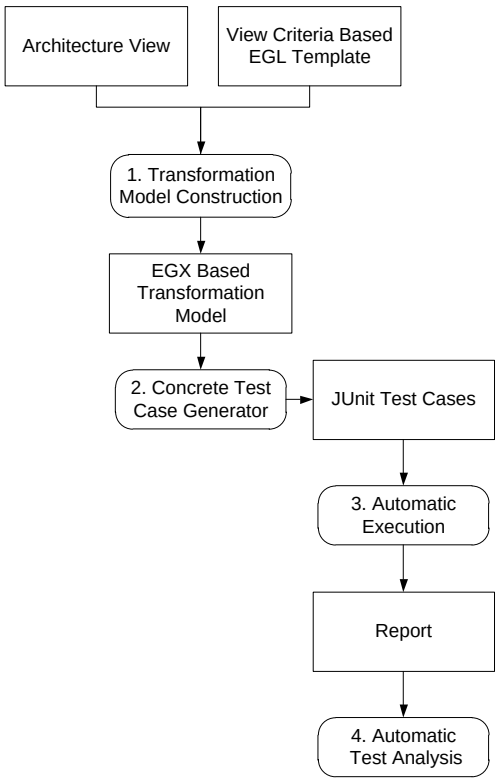

Figure 4.2. Process model of our approach

- *Save-Bench Architecture View Model*

Implemented system is using the metamodels of the software architecture views implemented at the work of Model Driven Engineering of Software Architecture Viewpoints [3]. The details of the view metamodels will be given in section four. HUTN is used to generate the view models that are used in test transformation model construction. Implemented view model in HUTN can be easily converted to view model using the integrated HUTN tool in Eclipse Epsilon environment.

- *View Criteria Based EGL Template*

View criteria based EGL templates are created for each architecture view under test. View criteria are defined for each view and used in the construction of EGL templates. These criteria decide what is to be tested in the test case for specific view. Details of the criteria definition will be presented in section three. Moreover, constructed templates are used when generating JUnit test cases. Basically transformation rule in the transformation model is applied on the template using the architecture view model.

- *EGX Based Transformation Model*

Transformation rule is embedded in this model. Transformation model is created for each architecture view used for once. Transformation model is executed and test cases for view under test are generated.

- *JUnit Test Cases*

Test case for each view is generated consisting of multiple test methods. The generated test cases depends on reflections library which is an open source java library scans classpaths, indexes the metadata, and allows you to query on your project [8]. Moreover, generated test cases depend on built-in JUnit and reflections libraries of Java.

4.3. Architecture Viewpoints & Architecture View Criteria

Clements et al. [1] define three different basic architectural styles including module, component-and-connector and allocation styles. Architectural styles are reoccurring forms across different systems implemented which deserves recognition [1]. Architecture views emerge when architecture styles are applied to a system. We have considered module and component and connector architecture styles in our work. Our main concern in testing which is architecture to code conformance is the main reason behind this decision. Other styles are not fully representing abstractions about the implementation. As a result of this decomposition, uses, generalization, layered and shared data views are selected. Furthermore, for each selected viewpoint the required architecture view criteria need to be defined as well. View criteria are specific for each viewpoint metamodel. In the following subsections we discuss the architecture viewpoint metamodel structure and possible view criteria for the viewpoints.

4.3.1. Decomposition Viewpoint

Decomposition viewpoint deals with concerns of partition of system responsibilities into modules and modules into submodules. It is a containment relation among modules and submodules. The implemented metamodel [3] for this viewpoint can be seen in Figure 4.3.

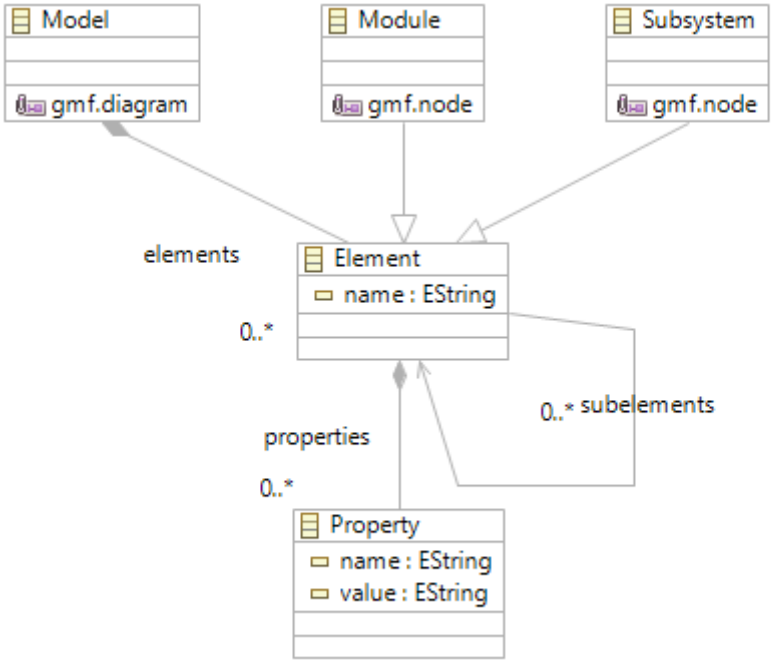

Figure 4.3. Decomposition viewpoint metamodel

- Model consists of elements.
- Element can be either Module or Subsystem.
- Element have properties which consists of name and value which enables to add necessary properties.
- Elements have subelements.

In the decomposition viewpoint we can identify the following view criteria:

1. Does every element in the view model appear in the code?
2. Does every subelement in the view model appear in the code?
3. Does every subelement exits under corresponding element in the code?

4.3.2. Uses Viewpoint

Uses viewpoint governs specialized depends-on relation (use relation) between modules. Use relation yields when module correctness depends on other

modules correctness. The implemented metamodel [3] for uses viewpoint can be seen in Figure 4.4.

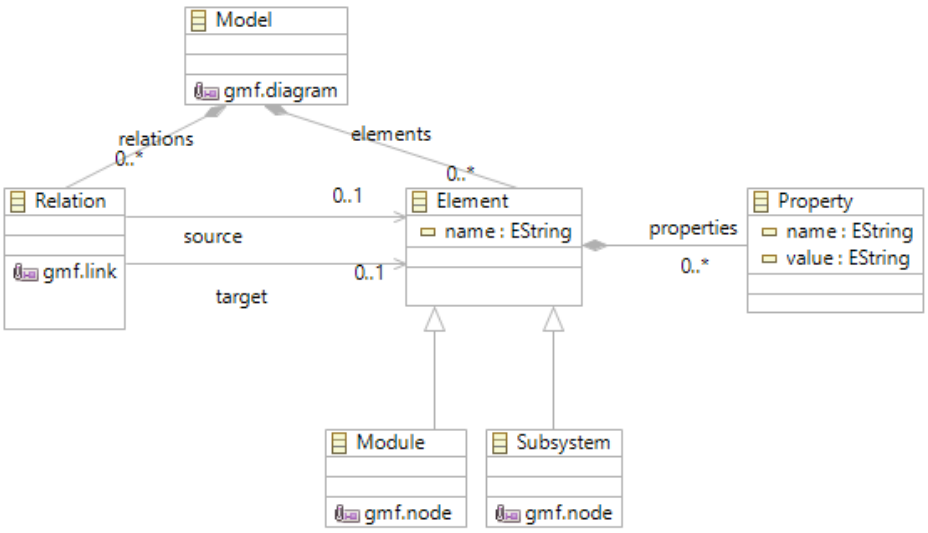

Figure 4.4. Uses viewpoint metamodel

- Model consists of relations and elements.
- Element can be either module or subsystem.
- Element have properties to bind dynamic properties to element.
- Relation have source and target element whereas source element uses target element.

In the uses viewpoint we can identify the following view criteria:

1. Does every uses relation in the view model appear in the code?
2. Does every source element appear in the code?
3. Does every target element appear in the code?

4.3.3. Generalization Viewpoint

Generalization viewpoint deals with is-a relation where an element generalizes another element either by implementation or inheritance. In the generalization relation parent element is more general element with respect to child element. Figure 4.5 shows the implemented metamodel [3] of generalization viewpoint.

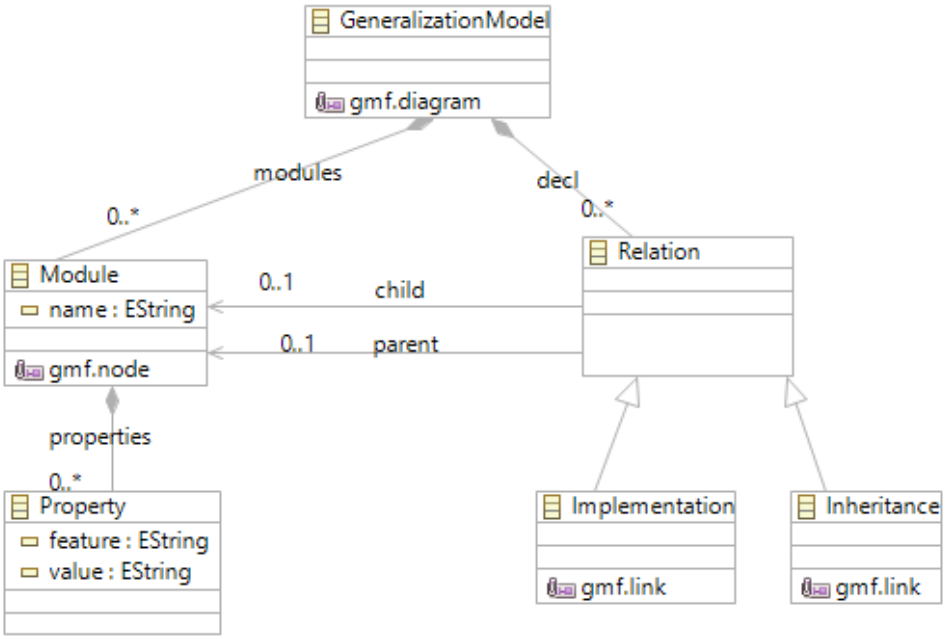

Figure 4.5. Generalization viewpoint metamodel

- Generalization model consists of modules and relation.
- Relation can be either implementation or inheritance relation.
- Modules have set of properties for dynamically adding information.
- Any relation has child module and parent module. Parent module is the module being generalized and child module is the module that generalizes.

In the generalization viewpoint we can identify the following view criteria:

1. Does each implementation or inheritance relation exist in the code?
2. Does each relation child and parent element is same in the code?
3. Does each relation child and parent element exist in the code?

4.3.4. Layered Viewpoint

Layered viewpoint reflects the division of modules into units where units are called layers. Each layers offers group of services that other layers uses. The uses relation in this viewpoint is either with restriction or without restriction. Uses relation with restriction forbids the use of higher level layers by lower level layers. On the other hand uses relation without restriction allows lower levels layers to use the service of higher level layers. Figure 4.6 shows the implemented metamodel [3] for layered viewpoint.

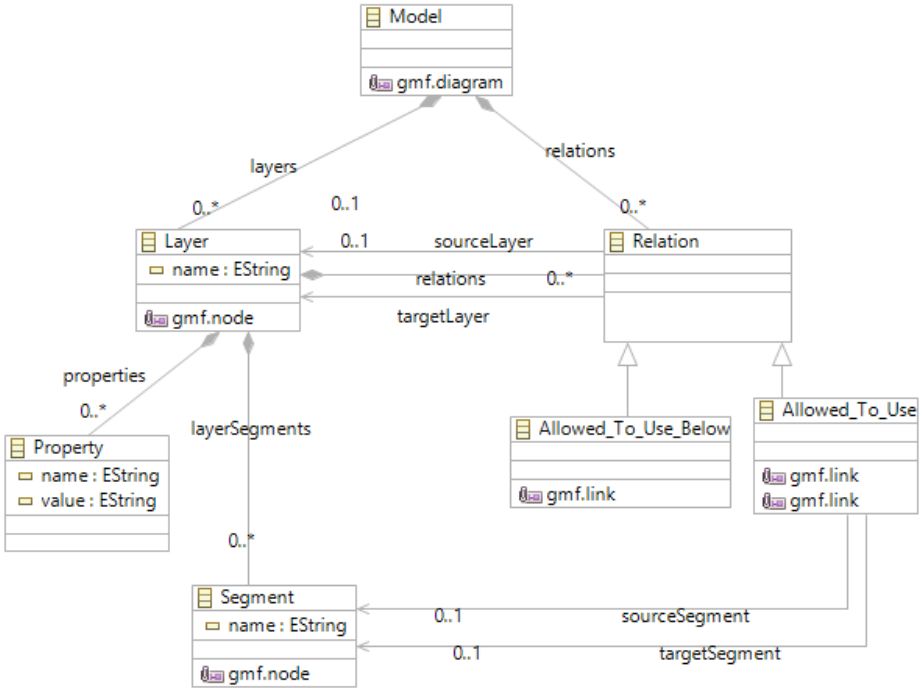

Figure 4.6. Layered viewpoint metamodel

- Model consists of layers and relations.
- Relation can be either allowed to use below or allowed to use relation.
- Layers have layer segments and layer relations.
- Relations have source layer and target layer where source layer is the layer using the target layer services.
- Layers have set of definable properties for attaching dynamic characteristic to layers.

In the layered viewpoint we can identify the following view criteria:

1. Does each allowed to use below relation in given model exists in the code?
2. Does each layer in the relation given in model exist in the code?

4.3.5. Shared Data Viewpoint

Shared data viewpoint stresses out the transmission of persistent data by the interaction of data accessors. The data or the repository has multiple accessors with different access right as read, write, or read and write. Figure 4.7 shows the implemented metamodel [3] of shared data viewpoint.

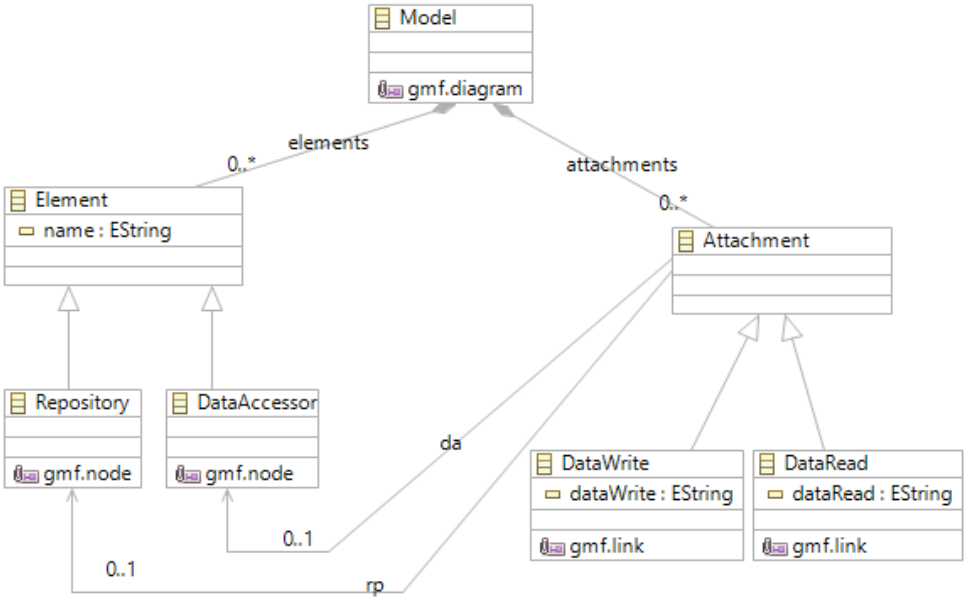

Figure 4.7. Shared data viewpoint metamodel

- Model consists of elements and attachments
- Elements can be either repository or data accessor
- Attachment can be either DataWrite or DataRead attachment

In the layered viewpoint we can identify the following view criteria:

1. Does each data accessor exist in the code?
2. Does each attachment of data accessor exist in the code?

4.4. Transformation Model Construction and Concrete Test Case Generator

In this section transformation model and view criteria based abstract test model of the implemented MDABT environment will be explained thoroughly for each architecture viewpoint. View criteria based abstract test model and transformation model is manually implemented for each architecture view for once. The view model and the view criteria based abstract test model are taken as inputs to the transformation model for generating concrete test cases which are JUnit test cases. Generated viewpoint test cases have one or more test methods depending on the architecture view model provided. Moreover, generated test cases can have helper methods if needed which are functions or operations that help the execution of another method. Rest of this section will present the implementation details for each architecture viewpoint. The full

implementation of the environment for each viewpoint is given in Appendix-E of this thesis.

4.4.1. Decomposition Viewpoint

Decomposition viewpoint test case is generated using abstract test model and decomposition view by executing corresponding transformation model. Decomposition viewpoint transformation model is presented in Figure 4.8. This model takes two inputs which are decomposition view model and decomposition abstract test model to generate a single test case file called 'TestDecomposition.java'. Generated test case consists of multiple test methods and helper methods.

```
rule Decomposition2JUnit
        transform decomposition : Model {
        // The EGL template to be invoked
        template : "Decomposition2JUnit.egl"
        // Output file
        target : "gen/TestDecomposition.java"
}
```

Figure 4.8. Decomposition viewpoint transformation model

In Figure 4.9 helper method for retrieving packages under given package name is presented. Method first finds all existing classes under given package name using reflections library. Then packages of the classes are extracted and returned to the caller method.

```
private List<Package> getSubPackages(String packageName) {
  List<Package> packageList = new ArrayList<Package>();
  List<ClassLoader> classLoadersList = Arrays.asList(ClasspathHelper.
     contextClassLoader(),ClasspathHelper.staticClassLoader());
  Reflections reflections = new Reflections(new ConfigurationBuilder()
    .setScanners(new SubTypesScanner(false),new ResourcesScanner ())
    .setUrls(ClasspathHelper.forClassLoader(classLoadersList
    .toArray(new ClassLoader[0]))).filterInputsBy(new FilterBuilder().
    include(FilterBuilder.prefix(packageName))));
   Set<Class<? extends Object>> allClasses= reflections.getSubTypesOf
     (Object.class);
  for (Class<? extends Object> clazz : allClasses) {
    if (!packageList.contains(clazz.getPackage())) {
       packageList.add(clazz.getPackage());
    }
  }
  return packageList;
}
```

Figure 4.9. Method for retrieving packages under given package name

Following helper method in Figure 4.10 searches the given package list for given package name and returns true if package exists in the package list. Otherwise, the method returns false.

```
private boolean isPackageExistsInGivenList(List<Package> packageList,
      String packageName) {
  for (Package pack : packageList) {
    if (pack.getName().equals(packageName)) {
        return true;
    }
    if (pack.getName().length() > packageName.length()) {
        if (pack.getName().substring(0, packageName.length())
             .equals(packageName)) {
            return true;
        }
    }
  }
  return false;
}
```

Figure 4.10. Method for searching given package name in package list

Figure 4.11 presents template test case method for decomposition viewpoint. In the first assertion the existence of package that is decomposed is verified. Second assertion verifies the existence of sub package of decomposed package. In the final assertion the existence of sub package positioned under decomposed package is verified. By performing all the stated verifications for each element and its subelement in decomposition view model we perform a complete testing with respect our previously defined view criteria. The EGL operation called 'testName' returns test method name according to Java coding conventions.

```
[%for (element in decomposition.elements){%]
        [%for (subelement in element.subelements){%]
@Test
public void test[%="".testName(element.name,subelement.name)%]() {
  String decomposedPackageName = "[%=element.name%]";
  String subPackageName = "[%=subelement.name%]";
  Assert.assertTrue(isPackageExistsInGivenList(
  getSubPackages(decomposedPackageName), decomposedPackageName));
  Assert.assertTrue(isPackageExistsInGivenList(
     getSubPackages(subPackageName), subPackageName));
  Assert.assertTrue(isPackageExistsInGivenList(
     getSubPackages(decomposedPackageName), subPackageName));
 }

        [%}%]
[%}%]
}
[%
function String testName(a:String, b:String):String{
  var a1:String=a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2).
     toUpperCase();
  var a2:String=a.substring(a.lastIndexOf('.')+2);
  var b1:String=b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2).
     toUpperCase();
  var b2:String = b.substring(b.lastIndexOf('.')+2);
  return a1+a2+"DecomposedOf"+b1+b2;
}
%]
```

Figure 4.11. Template test case method for decomposition viewpoint

4.4.2. Uses Viewpoint

Uses viewpoint test case is generated by executing uses viewpoint transformation model which takes abstract test model and uses view model as

inputs. Figure 4.12 shows the transformation model for uses viewpoint. By executing presented transformation model we generate 'TestUses.java' test case for uses viewpoint.

```
rule Use2JUnit
        transform uses : Model {
        // The EGL template to be invoked
        template : "Use2JUnit.egl"
        // Output file
        target : "gen/TestUses.java"
}
```

Figure 4.12. Uses viewpoint transformation model

Figure 4.13 presents the helper methods for retrieving classes that are directly in situated under given package name. This means that classes under subpackages of given package is not taken into consideration.

```
private Set<Class<? extends Object>> getClassesUnderPackage(String packageName
){
  List<ClassLoader> classLoadersList = new LinkedList<ClassLoader>();
  classLoadersList.add(ClasspathHelper.contextClassLoader());
  classLoadersList.add(ClasspathHelper.staticClassLoader());

  Reflections reflections = new Reflections(new ConfigurationBuilder()
        .setScanners(new SubTypesScanner(false),new ResourcesScanner())
        .setUrls(ClasspathHelper.forClassLoader(classLoadersList.toArray(new
        ClassLoader[0]))).filterInputsBy(new FilterBuilder().include
       (FilterBuilder.prefix(packageName))));

  Set<Class<Object>> allClasses=reflections.getSubTypesOf(Object.class);
  removeClassesThatAreNotDirectlyUnderGivenPackage(packageName,allClasses);
  return allClasses;
}
```

Figure 4.13. Helper method for retrieving direct classes under given package

Figure 4.14 presents a method to filter list of classes which are under given package and subpackages of package. Method filters out classes that are located under subpackage of package and returns only the direct classes located under given package.

```
private void removeClassesThatAreNotDirectlyUnderGivenPackage(String
             packageName ,Set<Class<? extends Object>> allClasses) {
   List<Class<?>> notDirectSubClasses = new ArrayList<Class<?>>();
   for (Class<?> clazz : allClasses) {
     if (!clazz.getPackage().getName().equals(packageName)) {
       notDirectSubClasses.add(clazz);
     }
   }
   allClasses.removeAll(notDirectSubClasses);
 }
```

Figure 4.14. Helper method for retrieving direct classes under given package

Figure 4.15 presents a helper method for deciding the uses relation between packages. Uses relation is defined as specialized version of depends-on relation in by Clements et al. In this thesis we detected uses relation of class A to class B

by checking whether class A holds the property of class B in form of either 1-1 or 1-many. Following helper method checks whether any class in source package uses any class in target package by iterating over each target package and source package classes and detecting uses relation. This method returns a map which contains the information of use relation to give meaningful failure messages for test execution. If the returned map is empty then it indicates source package classes does not use target package classes.

```
protected Map<String, String> doesSourceUseTarget(String sourcePackage,
        String targetPackage) {
    HashMap<String, String> usesMap = new HashMap<String, String>();
    Set<Class<? extends Object>> allUserClasses = getClassesUnderPackage
                (sourcePackage);
    Set<Class<? extends Object>> allUsedClasses = getClassesUnderPackage
                (targetPackage);
    for (Class<? extends Object> userClazz : allUserClasses) {
      for (Class<? extends Object> usedClazz : allUsedClasses) {
        Field[] fields = userClazz.getDeclaredFields();
          for (Field field : fields) {
            if (field.getType().equals(usedClazz)) {
              usesMap.put(userClazz.getName(), usedClazz.getName());
              return usesMap;
             }else if(field.getGenericType() instanceof ParameterizedType){
                 Type[] actualTypeArguments = ((ParameterizedType)
                      (field.getGenericType())).getActualTypeArguments();
                 for (Type type : actualTypeArguments) {
                   if (type.equals(usedClazz)) {
                     usesMap.put(userClazz.getName(),usedClazz.getName());
                     return usesMap;
                   }
                 }
          }else if(field.getGenericType() instanceof GenericArrayType){
              Type type = ((GenericArrayType) (field.getGenericType())).
                                  getGenericComponentType();
              if (type.equals(usedClazz)) {
                usesMap.put(userClazz.getName(),usedClazz.getName());
                return usesMap;
              }
          } else if (field.getType().isArray()) {
             Class<?> array = field.getType();
             if (array.getComponentType().equals(usedClazz)) {
                   usesMap.put(userClazz.getName(),usedClazz.getName());
                   return usesMap;
             }
          }
        }
      }
    }
    return usesMap;
}
```

Figure 4.15. Helper method for deciding uses relation

Figure 4.16 shows the template test case method for uses viewpoint. This method is generated for each uses relation specified in uses view model. First assertion of the method verifies the existence of user package. Second assertion verifies the existence of used package. Last assertion verifies the existence of uses relation between two packages by checking whether the map returned from helper method is empty or not. By performing stated three assertions we have successfully covered our uses viewpoint criteria stated in the previous section.

```
[% for (relation in uses.relations) { %]
@Test
public void test[%="".testName(relation.source.name, relation.target.name)%]()
{
        String source = "[%=relation.source.name%]";
        String target = "[%=relation.target.name%]";
        String errorMessage = source + " invalidates use relation to "+ target;
        assertTrue(isPackageExistsInGivenList(getSubPackages(source),source));
        assertTrue(isPackageExistsInGivenList(getSubPackages(target),target));
        Map<String, String> usesMap = doesSourceUseTarget(source, target);
        assertFalse(errorMessage, usesMap.isEmpty());
    }
[%}%]

}
[%
function String testName(a:String, b:String):String{
  var a1:String = a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2).
      toUpperCase();
  var a2:String = a.substring(a.lastIndexOf('.')+2);
  var b1:String = b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2).
      toUpperCase();
   var b2:String = b.substring(b.lastIndexOf('.')+2);
return a1+a2+"2"+b1+b2;
}
%]
```

Figure 4.16. Template test case method for uses viewpoint

4.4.3. Generalization Viewpoint

Generalization viewpoint test case is generated by executing generalization transformation model which is using generalization abstract model and generalization view model. Figure 4.17 shows transformation model for generalization viewpoint. Transformation model takes two inputs which are generalization view model and abstract test model and generates a JUnit test case file “TestGeneralization.java”. Test case file can contain multiple test methods depending on given generalization view model. Moreover, a helper method is used in test case as well.

```
rule Generalization2JUnit
        transform generalization : GeneralizationModel {
        // The EGL template to be invoked
        template : "Generalization2JUnit.egl"
        // Output file
        target : "gen/TestGeneralization.java"
}
```

Figure 4.17. Generalization viewpoint transformation model

Figure 4.18 shows the method for recursively retrieving every parent of the given class. All implemented classes are found and returned by this method at every inheritance level. Inheritance level for given class can change as it can be directly the parent of the class or the parent of the parents. Method finds all existing parent for inheritance by interface implementation or extension of another class.

```
private List<Class<?>> getGeneralizations(Class<?> classObject) {
        if (classObject == null) {
            return Arrays.asList();
        }
        List<Class<?>> generalizations = new ArrayList<Class<?>>();
        generalizations.add(classObject);
        generalizations.addAll(getGeneralizations(classObject.getSuperclass()));

        Class<?>[] superInterfaces = classObject.getInterfaces();
        for (int i = 0; i < superInterfaces.length; i++) {
            generalizations.addAll(getGeneralizations(superInterfaces[i]));
        }

        return generalizations;
  }
```

Figure 4.18. Helper method for retrieving every parent of given class

Figure 4.19 shows the template test method for generalization viewpoint. This test method created for each generalization relation in the model. Regarding to the type of the generalization where it can exist by extending a class or implementing an interface, test methods and failure messages are set. First assertion of the template test method verifies the existence of class that generalizes another component. Second assertion verifies the existence of class that is generalized by another class. Last assertion checks whether the given class extends or implements the parent class. By performing the stated assertions we cover all the generalization viewpoint criteria stated previously.

```
[%for (declaration in generalization.decl){%]
@Test
public void
test[%="".testName(declaration.child.name,declaration.parent.name)%]()
 throws ClassNotFoundException {
   String className = "[%=declaration.child.name%]";
   String inheritsFrom = "";
   String implementz = "";
   [%if(declaration.type().name== "Inheritance" ){%]
       inheritsFrom = "[%=declaration.parent.name%]";
   [%}else{%]
       implementz = "[%=declaration.parent.name%]";
   [%}%]
   Class<?> clazz = Class.forName(className);
   List<Class<?>> allGeneralizations = getGeneralizations(clazz);
   assertNotNull(clazz);
   if(inheritsFrom != ""){
   Class<?> inheritsFromClazz = Class.forName(inheritsFrom);
   String errorMessageExtension = clazz.getName()+ "does not extend"+
        inheritsFromClazz.getName();
   assertNotNull(inheritsFromClazz);
   assertTrue(errorMessageExtension,allGeneralizations.contains
     (inheritsFromClazz));
   }if(implementz != ""){
       Class<?> implementsClazz = Class.forName(implementz);
       String errorMessageImplements = clazz.getName()
       + " does not implement " + implementsClazz.getName();
       assertNotNull(implementsClazz);
       assertTrue(errorMessageImplements,
       allGeneralizations.contains(implementsClazz));
    }
  }
    [%}%]
 }
```

Figure 4.19. Template test method for generalization viewpoint

4.4.4. Layered Viewpoint

Layered viewpoint test case is generated using layered viewpoint transformation model. This model takes two inputs abstract test model and layered view model which is presented in Figure 4.20. Single JUnit test case file "TestLayered.java" is generated which can contain multiple test methods depending on the layered view model used.

```
rule Layered2JUnit
        transform layeredModel : Model {
        // The EGL template to be invoked
        template : "Layered2JUnit.egl"
        // Output file
        target : "gen/TestLayered.java"
}
```

Figure 4.20. Layered viewpoint transformation model

Figure 4.21 shows a template test method for the layered viewpoint. Same helper methods in uses viewpoint is used in the test case. Test method is generated for each allowed to use below relation. In the test method existence of source and target packages are verified. Then violation of layered property existence is checked by using the helper method in Figure 4.15. When calling this method source package and target package parameters are reversed for finding a uses relation from target package to source package.

```
[% for (relation in layeredModel.relations) { %]
        [%if(relation.type().name== "Allowed_To_Use_Below" ){%]
@Test
public void test[%="".testName(relation.sourceLayer.name,relation.targetLayer.
                    name)%](){
  String sourceLayer ="[%=relation.sourceLayer.name%]" ;
  String targetLayer = "[%=relation.targetLayer.name%]";
  Map<String,String> usesMap=doesSourceUseTarget(targetLayer,sourceLayer);
  Iterator<Entry<String, String>> iterator=usesMap.entrySet().iterator();
  String errorMessage = "";
  while (iterator.hasNext()) {
    Entry<String, String> entry = iterator.next();
    errorMessage += entry.getKey() +" breaks layered relation using "   +
      entry.getValue();
  }
  assertTrue(isPackageExistsInGivenList(getSubPackages(sourceLayer),
                                                    sourceLayer));
  assertTrue(isPackageExistsInGivenList(getSubPackages(targetLayer),
                                                     targetLayer));
  assertTrue(errorMessage, usesMap.isEmpty());
 }
     [%}%]
    [%}%]
}
[%
function String testName(a:String, b:String):String{
        var a1:String = a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2).
toUpperCase();
        var a2:String = a.substring(a.lastIndexOf('.')+2);
        var b1:String = b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2).
toUpperCase();
        var b2:String = b.substring(b.lastIndexOf('.')+2);
        return a1+a2+"2"+b1+b2;
}
%]
```

Figure 4.21. Template test method for layered viewpoint

4.4.5. Shared Data Viewpoint

Shared data viewpoint test case is generated by executing shared data viewpoint transformation model which takes shared data view model and abstract test model. Figure 4.22 shows the transformation model for shared data viewpoint. This transformation model generates single JUnit test case file called "TestSharedData.java". This test case class can contain multiple test methods depending on the provided shared data view model. Moreover, this test case contains helper method for test method to execute.

```
rule SharedData2JUnit
        transform sharedDataModel : Model {
        // The EGL template to be invoked
        template : "sharedData2JUnit.egl"
        // Output file
        target : "gen/TestSharedData.java"
}
```

Figure 4.22. Shared data viewpoint transformation model

Figure 4.23 shows a helper method for shared data viewpoint test case. This method returns whether the given method name exists in the given array of methods. This helper method is used for deciding if the given class structure contains a method given by its name.

```
private boolean isMethodExists(Method[] methods, String name) {
        for (Method method : methods) {
            if (method.getName().equals(name)) {
                return true;
            }
        }
        return false;
  }
```

Figure 4.23. Helper method for method existence checking

Figure 4.24 shows the detail of the template test method for shared data viewpoint. Method verifies the view criteria for shared data viewpoint for each attachment in share data model. According to the type of the attachment either data read or data write method name is extracted. Test first verifies the existence of the accessor class. Then it verifies if the given method name exists on the given data accessor class.

```
[% for (attachment in sharedDataModel.attachments) { %]
@Test
public void test[%if(attachment.type().name== "DataRead"){%][%="".testName(
attachment.da.name,attachment.dataRead)%][%}else{%][%="".testName(attachment.d
a.name,attachment.dataWrite)%][%}%]()throws ClassNotFoundException {
  String dataAccessorClassName = "[%=attachment.da.name%]";
  Class<?> accessorClass = Class.forName(dataAccessorClassName);
  assertNotNull(accessorClass);
  String readMethodName = "[%if(attachment.type().name== "DataRead")%]
                           [%=attachment.dataRead%]";
  String writeMethodName ="[%if(attachment.type().name == "DataWrite" )%]
                           [%=attachment.dataWrite%]";
  String failureMessage = accessorClass.getName();
  if (!readMethodName.equals("")) {
    failureMessage += "'s data read property is not satisfied";
    assertTrue(failureMessage,isMethodExists(accessorClass.getMethods(),
                                             readMethodName));
  }
  if (!writeMethodName.equals("")) {
     failureMessage += "'s data write read property is not satisfied";
     assertTrue(failureMessage, isMethodExists(accessorClass.getMethods(),
                                               writeMethodName));
  }
 }
[% } %]
}
[%
function String testName(a:String, b:String):String{
 var a1:String = a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2).
                                 toUpperCase();
 var a2:String = a.substring(a.lastIndexOf('.')+2);
 var b1:String = b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2).
                                 toUpperCase();
 var b2:String = b.substring(b.lastIndexOf('.')+2);
 return a1+a2+b1+b2;
}%]
```

Figure 4.24. Template test method for shared data viewpoint

4.5. Execution & Report

Test execution is performed using JUnit framework. JUnit is a framework based on xUnit architecture to write repeatable tests. Figure 4.25(adopted from [9]) shows the core classes of the xUnit test framework architecture.

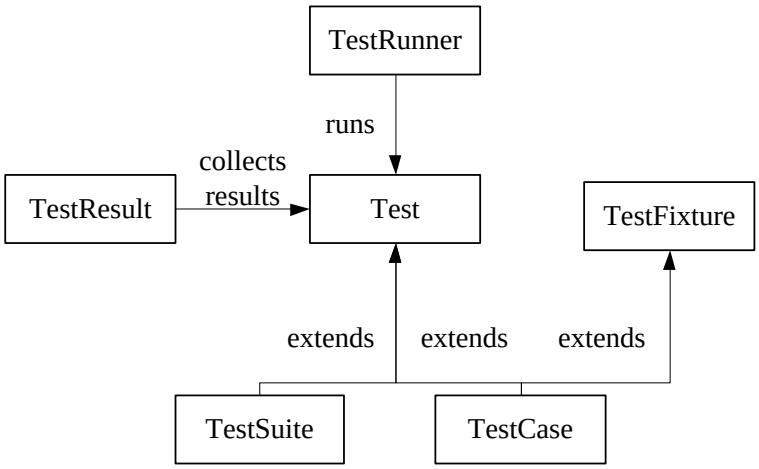

Figure 4.25. Core classes of xUnit test framework architecture

- Test

Test is an interface that contains the run method that takes TestResult instance as a parameter for running tests and collecting tests results.

- TestResult

Main purposes of running tests are obtaining test results where TestResult class serves for this purpose. When a tests run method executed the test result instance is passed to the method as a parameter for collecting the results. This class has several methods which are addError, addFailure, errorCount, failureCount and runCount.

- TestCase

Test case is the fundamental class of the xUnit architecture and parent of all unit tests. Class implements the run method in Test class which is the parent of the class.

- TestRunner

TestRunner is the class for reporting the details of the test executed and results. It has two methods run and main where run takes a Test instance as a parameter and main method indicates that the class is a runnable class.

- TestFixture

TestFixture is more complex test structure where test isolation is achieved. Test cannot run on manipulated object of the previously executed tests. As a result of this, TestFixture concept is introduced. Isolation is achieved by using setUp and tearDown methods running before each test in the test set.

- TestSuite

TestSuite is a class that holds the collection of TestCase instances. TestSuite has addTest interface where TestCase instance are inserted in TestSuite.

In our case we have generated five test cases for each viewpoint as explained in the previous section. Each generated test case has multiple test methods inside for verifying and validating the criteria we defined for each viewpoint. Concrete test cases generated that are in form of JUnit files which are java files indeed are executed by java virtual machine (JVM) using JUnit framework.

Jenkins CI is used for reporting of the test execution results. Jenkins CI is a continuous integration application implemented in Java. Jenkins provides automated test execution and reporting. Test reporting is handled by generating HTML reports are which are then sent to the specified user. Test reports present the result of the JUnit test case.

Chapter 5

Case Study

Case study used for our approach is e-government system within STRCT-STI Institute. The end users of the system are social assistance and solidarity foundations, general directorate of social assistance and citizens of the nation by using e-government portal. Different services are provided by the system such as conditional cash transfer, social assistance and solidarity foundation's service, general health insurance income test, decision support, widow assistance and etc... There are 31 million citizens registered to the system. Through the work hours in weekdays there are 4500 online users, 1001 social assistance and solidarity foundations that are interacting with the system. Averages of 4,833,341 database transactions are executed per day. Total of 732,430 lines of code are written in the client and server side of the system. It is a system integrated with seventeen other institutions communicating with 68 web services. As a result, we can infer that project X is a large scale software intensive system. During the development of the system, test driven development (TDD) technique is adopted. Unit tests and integration tests run after each new commit to the code base and the results are reported back to developers. Client side tests also run on nightly builds of the system and report back the results to developers as well. The system has three main structure the batch processes, client side and server side implementations. Both client and server side implementation depends on infrastructure packages. Our approach is applied on the infrastructure of the server side implementation on five viewpoints: decomposition, generalization, shared data, uses and layered. Each viewpoint model is generated based on the save bench viewpoints metamodels. The server infrastructure of the Project X consists of six thousand lines of codes.

The models are used by our testing environment and five test cases are generated each containing multiple test methods on the given viewpoint.

5.1. Architecture Design of Case Study

Real names of the components will not be given for confidentiality issues. In the following parts Project X infrastructure architecture will be explained using architecture views. Each architecture view details will be presented using the graphical representation of the view.

5.1.1. Shared Data Viewpoint

Project X has two database accessors named PM and QM where each can perform different operations. Moreover, one repository is present which is named DB. PM can perform both data read and data write operation on DB. However, QM can only perform data read operations on DB. Figure 5.1 shows the shared data view of the project.

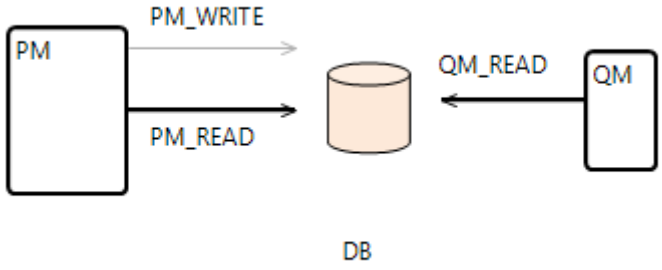

Figure 5.1. Shared data view of Project X infrastructure

5.1.2. Decomposition Viewpoint

Project X composed of one main package namely A. A is divided into subparts as B, C, D, E, G and F. Furthermore, B is decomposed into B1, B2 and B3. Likewise C is decomposed into C1 and C2. On the other hand, D is decomposed into three parts D1, D2 and D3 where D1 is decomposed into one more part called D11. Figure 5.2 shows the decomposition view of the Project X. Each label represents a package in project X. The hierarchy of the packages is shown via containment relation.

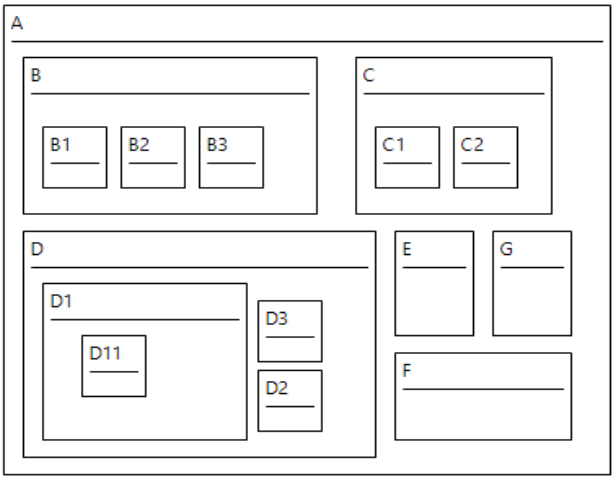

Figure 5.2. Decomposition view of Project X infrastructure

5.1.3. Uses Viewpoint

Diagram representing uses view of project x can be seen in Figure 5.3. B1 package uses itself, B2, B3 and D3 packages. B2 and B3 packages use itself and F package. Moreover, B package only uses D3 package. D3 package uses itself, E and D2 packages. E package uses D2 and D3 packages. As can be seen from the figure C uses F, C1 and C2 packages. On the other hand, F, C1 and C2 packages does not have any uses relation. D uses D11 where D11 uses D and itself. D2 package uses itself and D2 package. At last G package uses D3 and G package.

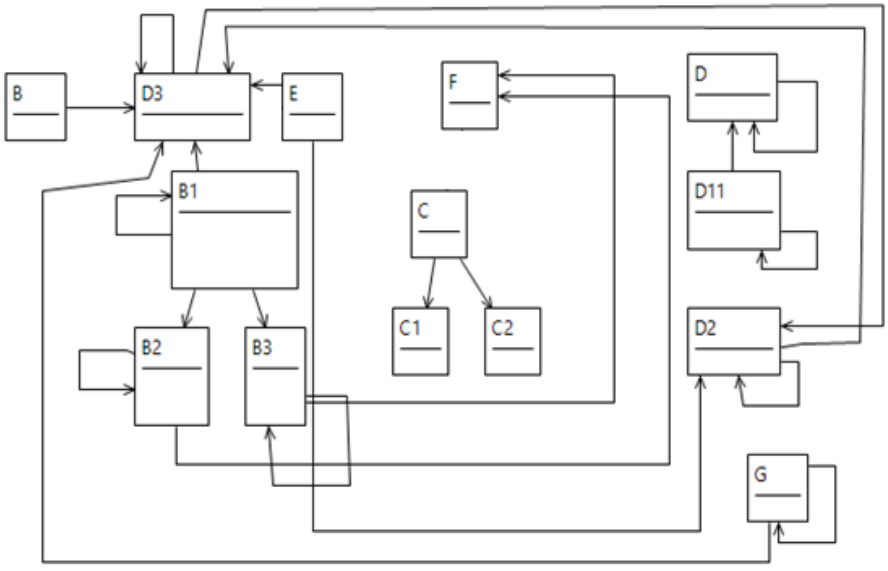

Figure 5.3. Uses view of Project X infrastructure

5.1.4. Layered Viewpoint

Project X has three main layered views as shown in Figure 5.4. B1 package is allowed to use B2 and B3 where B2 and B3 cannot use B1. D3 package is allowed to use E package and reverse cannot be applied. Moreover, package F is allowed to C where C is allowed to C1 and C2. C1 and C2 cannot use C as well as C cannot use F, which directly implies F cannot be used C1 and C2.

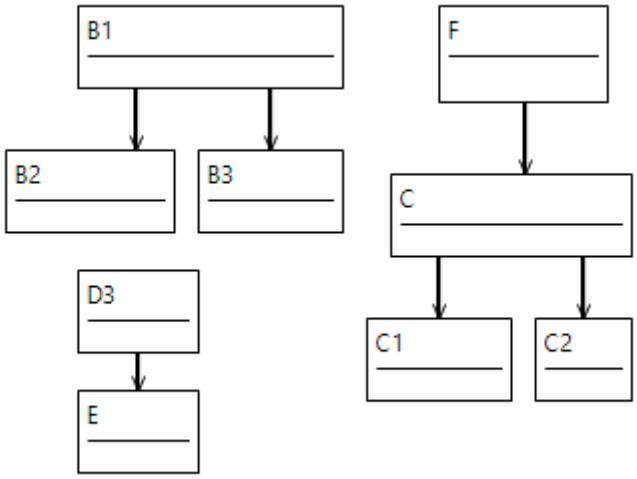

Figure 5.4. Layered view of Project X infrastructure

5.1.5. Generalization Viewpoint

In Figure 5.5 generalization view of package B is given in detail. Blue rounded squares are interfaces and black rounded squares are classes. In the figure, three types of generalization viewpoint are shown which are interface extensions, class extension and interface implementation. Class can extend another class and class can implement an interface. However, interface can only extend another interface. In the given view model all combinations of such relations are given. Moreover, in Figure 5.6 generalization view within C package can be seen. Various generalization relations can be seen in this model as well. Figure 5.7 shows the generalization view within D package. Differently, in Figure 5.8 and Figure 5.9 generalization relation between packages and within packages can be seen.

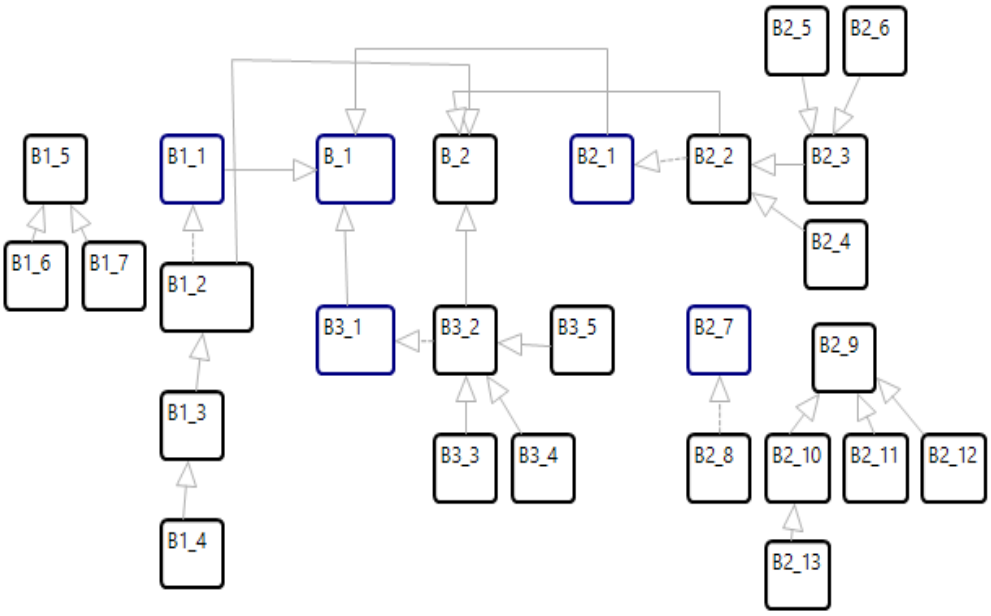

Figure 5.5. Generalization view of Project X infrastructure within package B

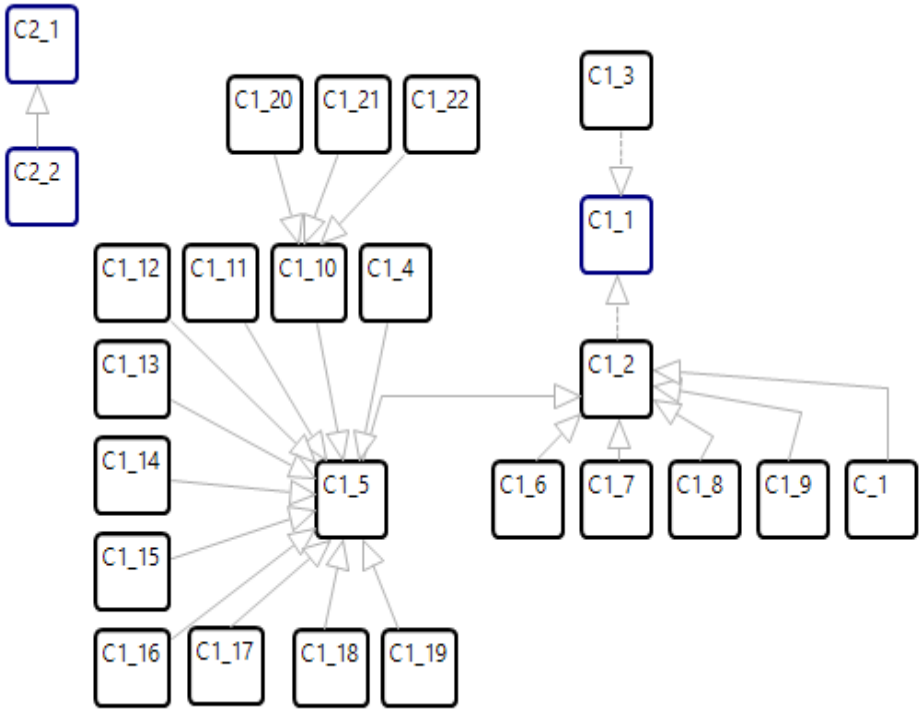

Figure 5.6. Generalization view of Project X infrastructure within package C

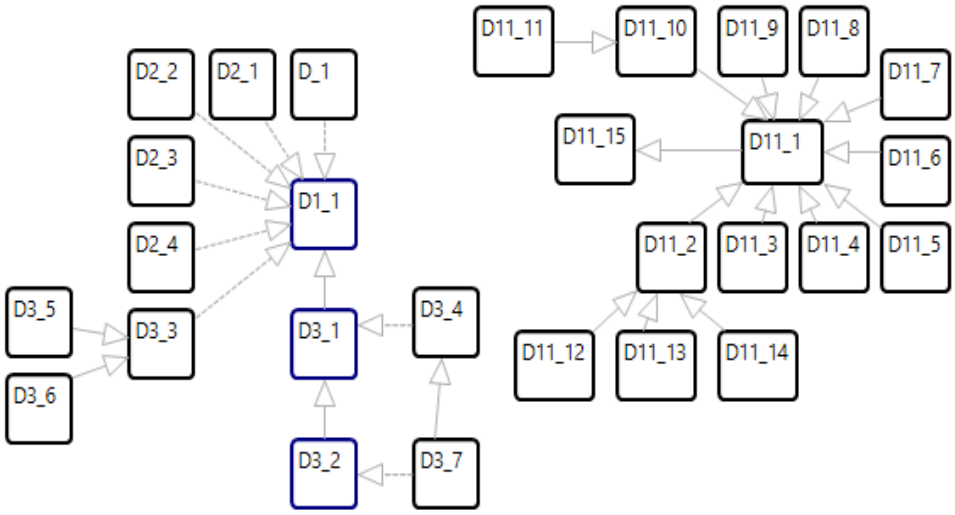

Figure 5.7. Generalization view of Project X infrastructure within package D

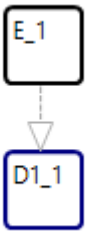

Figure 5.8. Generalization view of Project X infrastructure between package D and F

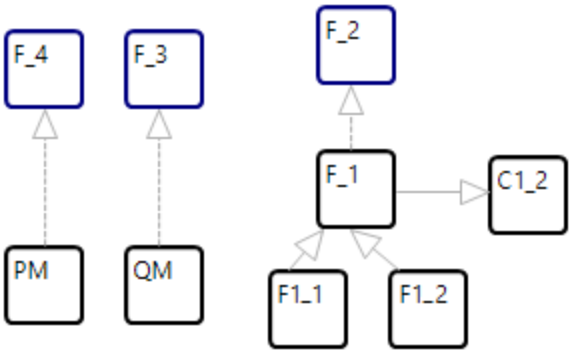

Figure 5.9. Generalization view of Project X infrastructure within package F

5.2. Validating the Test Execution Environment

We have applied fault-based testing techniques in order to validate our suggested approach before we evaluate our approach on presented case study. Software

testing is labeled as fault-based testing when it aims to demonstrate of absence of pre-defined faults [28]. Our motivation for applying fault based testing is to measure effectiveness of our generated test cases. In order to assess our test cases we created a mutant copy of our case study with injected faults. Given the implementation of the case study, we will introduce the following categories of faults:

- Absence relations
- Divergence relations

As we have discussed in section 5.1 each viewpoint includes its own set of criteria. As such absence and divergence relations will be based on these defined criteria. In the next sections we will present our fault injections for each viewpoint.

5.2.1. Shared Data Viewpoint

In shared data viewpoint we injected the following faults to the implementation of the case study:

- PM_WRITE and PM_READ relations are removed from the implementation of the case study. In this injection we expect to find the absence of relations given that these relations exist in our shared data view model.
- QM component is removed from the implementation. In this injection we expect to find the absence of QM component given that this component is present in our shared data view model.

Figure 5.10 presents the shared data view representation of mutant implementation. As can be seen there are four absences from the model presented in Figure 5.1 which are the absence of QM, QM_READ, PM_READ and PM_WRITE. We expect to find each absence in the test case we generated using model presented in Figure 5.1.

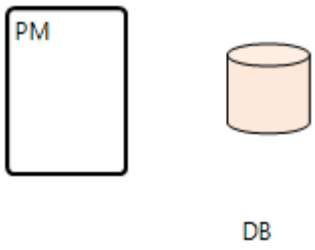

Figure 5.10. Shared data view representation of mutant implementation

5.2.2. Decomposition Viewpoint

In decomposition viewpoint we injected following faults to the implementation of the case study:

- Package F is removed from the content of package A. In this injection we expect to find the divergence of package hierarchy given that package F is located under package A in decomposition view model.
- Package E is removed from the case study. In this injection we expect to detect the absence of package E given that package E is presented in decomposition view model.

Figure 5.11 presents the decomposition view representation of mutant implementation. Packages F and E are removed from implementation. We expect to detect faults injected using the test case generated from the model presented in Figure 5.2.

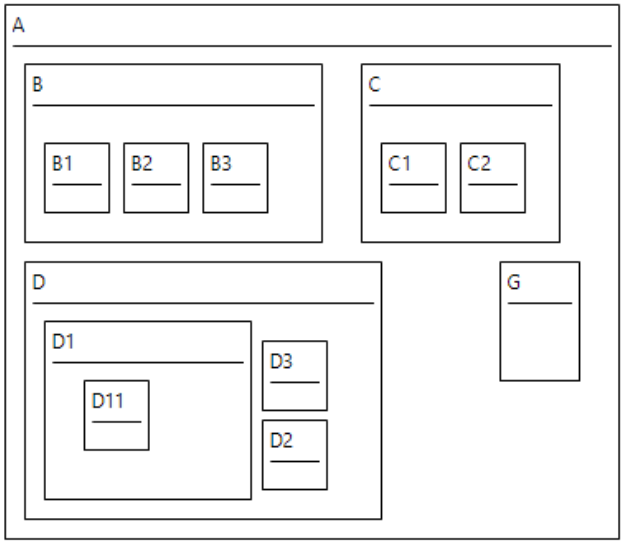

Figure 5.11. Decomposition view representation of mutant implementation

5.2.3. Uses Viewpoint

In uses viewpoint we injected the following faults to the implementation of the case study:

- Package C is removed from the case study implementation. In this fault injection we expect to find the absence of package C given that package C is presented in uses view model. The subpackages of C package are moved out of this package to under package A.
- Uses relation of B3 to F is removed from the case study implementation. In this fault injection we expect to find the absence of B3 to F uses relation given that removed relation is presented in uses view model.
- Uses relation of B to D3 is altered to B to D in the case study implementation. In this fault injection we expect to find the divergence of B's package relation from D3 to D given that uses view model contains the use relation of B to D3.

Figure 5.12 presents the uses view representation of mutant implementation. Stated faults are injected to the implementation and we expect to detect the absence of C, absence of relation B3 to F and divergence of relation B to D3.

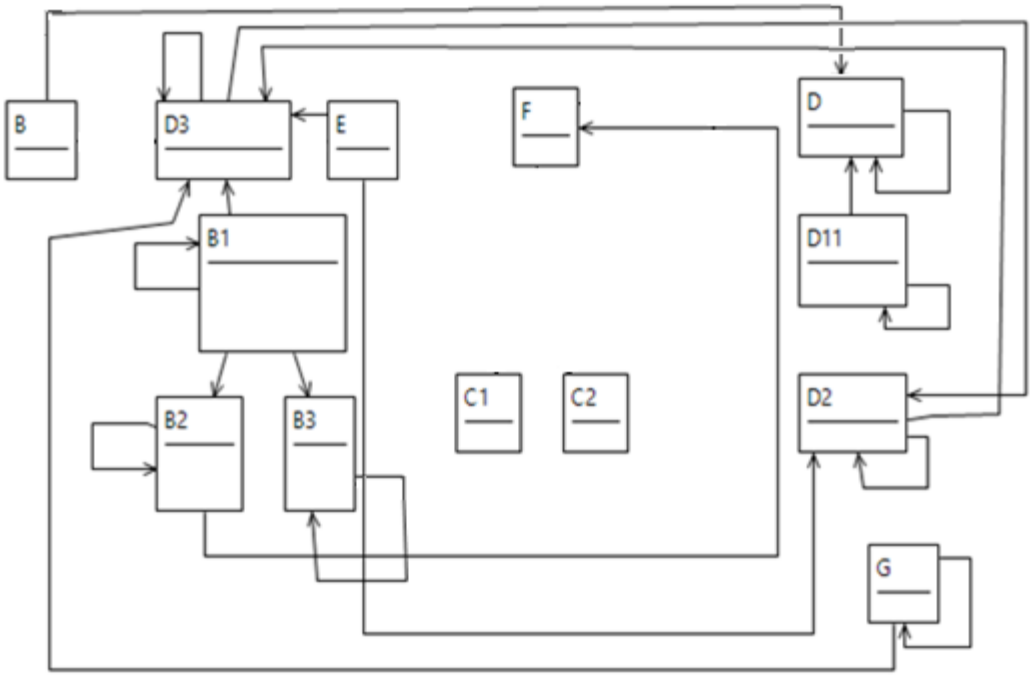

Figure 5.12. Uses view representation of mutant implementation

5.2.4. Layered Viewpoint

In layered viewpoint we injected the following faults to the implementation of the case study:

- Uses relation from package C to package F is introduced to the implementation of the case study. In this fault injection we expect to detect divergence in violation of layered property in allowed to use below relation of F to C.
- Package C1 is removed from the implementation of the case study. In this fault injection we expect to find the absence of package C1 given that layered view model contains the package C1.

Figure 5.13 presents the layered view of mutant implementation. We expect to detect layered property violation and absence of C1 package.

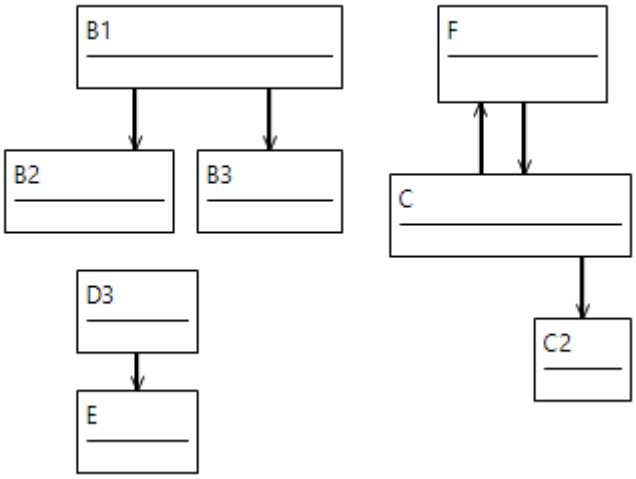

Figure 5.13. Layered view representation of mutant implementation

5.2.5. Generalization Viewpoint

In generalization viewpoint we injected the following faults to the implementation of the case study:

- F_2 component is removed from the implementation of the case study in order to detect the absence of the component given that generalization view model contains component F_2.

- Inheritance by extension relation between F_1 and F1_1 is removed from the implementation of the case study in order to detect the absence of generalization relation given that generalization view model contains such relation.
- Inheritance by implementation relation between F_1 and F_2 is removed from the implementation of the case study in order to detect the absence of the generalization relation given that generalization view model contains such relation.
- Inheritance by implementation relation between PM and F_4 altered to PM and F_3 in order to detect the divergence of the generalization relation given that generalization view model contains PM and F_4 generalization relation.
- Inheritance by extension relation between F1_1 and F_1 altered to F1_1 and C1_2 in order to detect the divergence of the generalization relation given that generalization view model contains F1_1 and F_1 generalization relation.

Figure 5.14 presents the generalization view of mutant implementation. We expect to detect all the faults injected with the case generated using generalization view model from Figure 5.9.

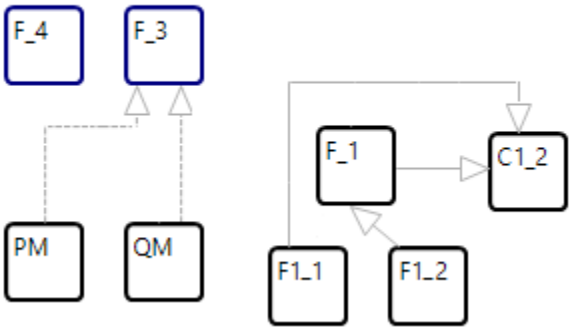

Figure 5.14. Generalization view representation of mutant implementation

5.2.6. Summary

As presented in previous sections we have injected faults to the implementation of the case study for each viewpoint with respect to defined view criteria. Our mutant copy of case study contained 15 faults injected. We have injected every possible fault with respect to architecture view criteria defined in section 5.1. Therefore, we ensure that our approach will detect every possible bug we expect

to detect. In this section we present the results of detected faults within the injected faults. Table 6 shows the fault-based testing results for each viewpoint. As can be seen from the table all the injected faults are detected successfully by our MDABT approach using architecture viewpoints.

Table 6. Fault-based testing results for each viewpoint

Viewpoint/Fault Detection	# of Faults Detected	# of Faults Not Detected
Shared Data Viewpoint	3	0
Decomposition Viewpoint	2	0
Uses Viewpoint	3	0
Layered Viewpoint	2	0
Generalization Viewpoint	5	0

5.3. Test Execution Results

In the following sections test execution results will be presented for each viewpoint. The results will be discussed and the meaning of the test cases results will be explained. Table 7 shows the summary of the test execution result for each viewpoint. It can be seen that tests for shared data, decomposition and generalization viewpoints have executed successfully. However, we found inconsistencies between implementation and architecture for uses and layered viewpoints. Further details will be given in following subsections for each viewpoint.

Table 7. Testing results for each viewpoint

Viewpoint/Test Result	# of Test Passed	# of Test Failed
Shared Data Viewpoint	3	0
Decomposition Viewpoint	15	0
Uses Viewpoint	21	1
Layered Viewpoint	5	1
Generalization Viewpoint	82	0

5.3.1. Shared Data Viewpoint

In shared data viewpoint all the tests within created file executed successfully. This result implies that two data accessors specified in our shared data view model exists in our code. The operation rights given in the model that they can perform on repositories conforms to our implementation. This test shows that code did not drift away from the architecture in shared data viewpoint.

5.3.2. Decomposition Viewpoint

All the tests in the created file for decomposition viewpoint test case executed successfully. This result implies that decomposition view implemented in code level conforms to our architecture model. Simply, code level implementation did not drift away from our architecture in terms of decomposition of the system.

5.3.3. Uses Viewpoint

In uses viewpoint one of the tests created within test case file failed while other tests executed successfully. In the failed test none of the classes within C package uses any of the classes in C2 package whereas uses view model shows use relation between C and C2 packages. This result shows the code has drifted away from architecture mainly on C package and its uses relations.

5.3.4. Layered Viewpoint

In layered viewpoint one of the tests created within test case file failed while other tests executed successfully. The reason for the failure of the single test within the test case is the allowed to use below relation between C and C1 packages where some class of C1 package breaks layered relation by using some class of C package. This result implies that code level implementation has drifted away from architecture of the system in for layered style in C.

5.3.5. Generalization Viewpoint

In generalization viewpoint all the tests created within test case file executed successfully. This result implies that all the generalization relation given in the generalization view model is implemented in the code level. Moreover, code level implementation did not drift away from architecture specified.

5.4. Discussion

Our MDABT approach for architecture to code conformance applied on huge industrial systems infrastructure. Details of the case study are given in the previous sections with statistical facts obtained from the system. Likewise, test execution results and test execution result meanings explained in previous section. Our approach show both the implementation drifting away from the given architecture model and code conformance to given architecture model. It is important to see that conformance relation is unidirectional where only components specified in the architecture view models are validated and verified

against the implementation level. Our approach cannot find conformance of components specified in the implementation level to architecture level. For instance, if we have an extra component in the implementation level that does not exist in architecture level our approach cannot decide the conformance of the specified component. Therefore, we assume that our architecture model which is our reference point for testing is both complete and correct.

Chapter 6

Related Work

Several systematic literature reviews have been published on model based testing. Arilo et al. [25] published a SLR on suggested model based testing approaches in 2007. Moreover, in [26] SLR on tool support for model based testing have been published. However, to our best knowledge no SLR on model driven architecture based testing have been proposed by any author. In this thesis we have presented a SLR on MDABT with 12 filtered primary studies. Moreover, as we have shown in the SLR in section 2 most MDABT approaches do not explicitly use architecture views. Two of the studies which use obs-functions to reduce the complexity of abstract test model use the view keyword. However, these studies do not address the architecture view definition in their suggested approaches.

Architecture based testing first presented at workshop of ISAW '96 Joint Proceedings of the Second International Software Architecture Workshop in 1996 by Bertolino et al [10] and Richardson et al. [11]. Since then many studies are presented on this domain by different researchers. In 2000 Bertolino et al. published a study [12] in which software architecture is specified by CHAM and transformed into a test model based on ALTS. Later Muccini et al. changed the architecture specifications from CHAM to FSP and used the same test model [14]. Jin et al. [13] defined test criteria for architecture based testing and performed testing at architectural level using Wright ADL as architecture specification and behavior graph as test model. This study presented generic test criteria for architecture based testing at architecture level. Moreover, in [15] and [16] authors exploited architecture style and model-checking results for using in architecture based testing respectively. For the first time the unified modeling language (UML) is used as architecture specification for architecture based

testing [17]. Test model for this study was the IOLTS and TGV tool is used for test case generation. Furthermore, different type of architecture based testing was applied in [18]. There is no explicit architecture specification or test model but testing was carried using event based technique. Simply the assertions are performed on the expected fired event and actual emitted event. Later Muccini et al. introduced the concept of architecture based regression testing [19] to the literature using previous work setup of architecture based testing process. In [20] authors use AADL to specify the architecture and automata as test model of the architecture based testing process model given in systematic literature review. Another different approach is adapted by [21] where ACME was used as architecture specification language and PetriNet was used as test model. At last in [22] again UML is utilized for describing the system architecture and test model they defined used for generating the concrete test cases. In this study different from other studies service oriented architecture was tested by the approach they suggested. Most of the studies described above used small scale examples for empirical evaluation of their approaches. In this thesis as we have shown in chapter 5 we have tested our approach on an ongoing complex industry project.

Chapter 7

Conclusion

As software architecture become more substantial standards have been defined for description of the software architecture. Using software architecture in software testing is a developing research area. Current literature offers greatly varying techniques for architecture based testing. Some of the studies test the validness of the architecture by defining architectural properties. Others exploit the information embedded in architecture to perform testing at the code level. Different approaches yields for different concerns for testing. In this work we presented model driven architecture based testing using architecture viewpoint. We have defined our architecture model in terms of architecture views that maps to different stakeholder concerns. By defining such architecture model we were able to systematically test each specific concern of the stakeholder separately at the code level. We generated a test case for each architecture view and executed tests and generated test results of the test cases. Moreover, current literature uses small scale examples for evaluating their approaches. However, we applied our approach on ongoing actively used industry project having 4500 active users throughout the day and millions of database transaction executions per day. Having applied our approach on such big scale system we have found architecture to code differences. Moreover, studies in the current literature all used different technologies for model development of their approach. In this thesis, we have used Eclipse Epsilon framework for implementing our approach. Eclipse Epsilon framework offers a great variety of tools, languages and plug-ins for implementing model based solutions.

At last we hope that our study paves the way for new research activities and contributes to the work of other researchers in this domain. We have

successfully implemented model driven architecture based testing using architecture viewpoints approach for architecture to code conformance concern. When analyzing the architecture with respect to the code we can distinguish among divergence and absence relations. In the current work we have focused on the absence relations that define the missing concerns in the code which are imposed by the architecture. The divergent properties, that are elements in the code not reflected in the architecture, have not been considered within the context of this thesis. We consider this as our future work. Another future work direction is the focus on the architecture based testing for testing the systemic properties at the architecture design level itself.

Bibliography

[1] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond. Second Edition. Addison-Wesley, 2010.

[2] [ISO/IEC 42010:2007] Recommended practice for architectural description of software-intensive systems (ISO/IEC 42010). (Identical to ANSI/IEEE Std1471–2000), July, 2014.

[3] B. Tekinerdogan, E. Demirli. "Evaluation Framework for Software Architecture Viewpoint Languages". in Proc. of Ninth International ACM Sigsoft Conference on the Quality of Software Architectures Conference (QoSA 2013), Vancouver, Canada, pp. 89-98,June 17-21, 2013.

[4] M. Utting, A. Pretschner, B. Legeard. A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability, 2011.

[5] B. Kitchenham, S. Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. (E. T. Report, Ed.)Engineering, 2(EBSE 2007-001), 1051, 2007 doi:10.1145/1134285.1134500

[6] B. Kitchenham, D. Budgen, O. P. Brereton, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering - A systematic literature review. Inf. Softw. Technol. 51, 1 (January 2009), 7-15. DOI=10.1016/j.infsof.2008.09.009

[7] Epsilon – model driven development IDE, http://eclipse.org/epsilon, last accessed on April, 2015.

[8] Reflections- reflections library built by Google in Java, https://code.google.com/p/reflections, last accessed on April, 2015.

[9] P. Hamill. Unit Test Frameworks. First Edition. O'Reilly, January, 2015.

[10] A. Bertolino,P. Inverardi."Architecture-based Software Testing". ISAW. Joint Proceedings of the Second International Software Architecture Workshop, 1996. pp.62-64.

[11] D.J. Richardson,A.L. Wolf."Software testing at the architectural level". ISAW. Joint Proceedings of the Second International Software Architecture Workshop, 1996. pp.68-71.

[12] A. Bertolino,F. Corradini,P. Inverardi,H. Muccini."Deriving Test Plans from Architectural Descriptions". Software Engineering.Proceedings of the 2000 International Conference, 2000. pp. 220-229.

[13] Z. Jin, J. Offutt."Deriving Tests From Software Architectures". Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium. pp. 160-171.

[14] H.Muccini,P. Inverardi, A. Bertolino. "Using software architecture for code testing", Software Engineering, IEEE Transactions on, vol.30, no.3, pp.160-171, March 2004.

[15] H. Muccini, M. Dias, D. J. Richardson. "Systematic Testing of Software Architectures in the C2 Style",Lecture Notes in Computer Science. pp. 295-309.2004.

[16] A. Bucchiarone, H. Muccini, P. Pelliccione, P. Pierini. "Model-Checking plus Testing from Software Architecture Analysis to Code Testing",Lecture Notes in Computer Science. pp. 351-365.2004.

[17] G. Scollo, S. Zecchini. "Architectural Unit Testing", Electronic Notes in Theoretical Computer Science, Volume 111, 1 January,pp. 27-52. 2005.

[18] K. Winbladh, A. T. Alspaugh,H. Ziv, D. J. Richardson."Architecture Based Testing Using Goals and Plans",Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis. pp. 64-68.2006.

[19] H. Muccini, M. Dias, D. J. Richardson."Software Architecture-Based Regression Testing",JSS, Special Edition on Architecting Dependable Systems. pp.1-18. 2006.

[20] A. Johnsen,P. Pettersson,K. Lundqvist. "An Architecture-Based Verification Technique for AADL Specifications",Lecture Notes in Computer Science. pp. 105-113.2009.

[21] H. Reza, S. Lande. "Model Based Testing Using Software Architecture",Information Technology: New Generations (ITNG), 2010 Seventh International Conference. pp. 188-193.

[22] C. Keum, S Kang, M. Kim."Architecture-Based Testing of Service-Oriented Applications In Distributed Systems",Information and Software Technology, Volume 55, Issue 7,pp. 1212-1223. July 2013.

[23] [ISO/IEC/IEEE 42010:2011] Systems and software engineering — Architecture description is an international standard for architecture descriptions of systems and software (ISO/IEC/IEEE 42010), May 2014.

[24] [ANSI/IEEE 1059:1993] Software Verification and Validation Plan (ANSI/IEEE 1059), May 2014.

[25] C. Arilo, N. Dias, S. Rajesh, V. Marlon, H. T. Guilherme. 2007. "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). pp. 31-16. 2007.

[26] M. Shafique, Y,Labiche. "A systematic review of model based testing tool support". Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04, 2010.

[27] M. Lochau, S. Lity, R. Lachmann, I. Schaefer, U. Goltz. "Delta-oriented Model-Based Integration Testing of Large-Scale Systems". Journal of Systems and Software. pp. 63-84.2014.

[28] L. J. Morell. A theory of fault-based testing. IEEE Transactions on Software Engineering, 16 (8): 844–857, 1990.

Appendix A - Search Strings

Electronic Database	Search String
IEEE Xplore	(("Document Title":"model based testing" OR "Document Title":"model based software testing" OR "Document Title":"model-based testing" OR "Document Title":"model-based software testing" OR "Document Title":"model driven testing" OR "Document Title":"model driven software testing" OR "Document Title":"model-driven testing" OR "Document Title":"model-driven software testing" OR "Document Title":"model based test" OR "Document Title":"model based software test" OR "Document Title":"model-based test" OR "Document Title":"model-based software test" OR "Document Title":"model driven test" OR "Document Title":"model driven software test" OR "Document Title":"model-driven test" OR "Document Title":"model-driven software test") AND ("Document Title":"architecture")) OR ("Document Title":"architecture based testing" OR "Document Title":"architecture based software testing" OR "Document Title":"architecture-based testing" OR "Document Title":"architecture-based software testing" OR "Document Title":"architecture driven testing" OR "Document Title":"architecture driven software testing" OR "Document Title":"architecture-driven testing" OR "Document Title":"architecture-driven software testing" OR "Document Title":"architecture based test" OR "Document Title":"architecture based software test" OR "Document Title":"architecture-based test" OR "Document Title":"architecture-based software test" OR "Document Title":"architecture driven test" OR "Document Title":"architecture driven software test" OR "Document Title":"architecture-driven test" OR "Document Title":"architecture-driven software test") OR ((("Abstract":"model based testing" OR "Abstract":"model based software testing" OR "Abstract":"model-based testing" OR "Abstract":"model-based software testing" OR "Abstract":"model driven testing" OR "Abstract":"model driven software testing" OR "Abstract":"model-driven testing" OR "Abstract":"model-driven software testing" OR "Abstract":"model based test" OR "Abstract":"model based software test" OR "Abstract":"model-based test" OR "Abstract":"model-based software test" OR "Abstract":"model driven test" OR "Abstract":"model driven software test" OR "Abstract":"model-driven test" OR "Abstract":"model-driven software test") AND ("Abstract":"architecture")) OR ("Abstract":"architecture based testing" OR "Abstract":"architecture based software testing" OR "Abstract":"architecture-based testing" OR "Abstract":"architecture-based software testing" OR "Abstract":"architecture driven testing" OR "Abstract":"architecture

	driven software testing" OR "Abstract":"architecture-driven testing" OR "Abstract":"architecture-driven software testing" OR "Abstract":"architecture based test" OR "Abstract":"architecture based software test" OR "Abstract":"architecture-based test" OR "Abstract":"architecture-based software test" OR "Abstract":"architecture driven test" OR "Abstract":"architecture driven software test" OR "Abstract":"architecture-driven test" OR "Abstract":"architecture-driven software test"
ACM Digital Library	((Title:"model based testing" OR Title:"model based software testing" OR Title:"model-based testing" OR Title:"model-based software testing" OR Title:"model driven testing" OR Title:"model driven software testing" OR Title:"model-driven testing" OR Title:"model-driven software testing" OR Title:"model based test" OR Title:"model based software test" OR Title:"model-based test" OR Title:"model-based software test" OR Title:"model driven test" OR Title:"model driven software test" OR Title:"model-driven test" OR Title:"model-driven software test") AND (Title:"architecture")) OR (Title:"architecture based testing" OR Title:"architecture based software testing" OR Title:"architecture-based testing" OR Title:"architecture-based software testing" OR Title:"architecture driven testing" OR Title:"architecture driven software testing" OR Title:"architecture-driven testing" OR Title:"architecture-driven software testing" OR Title:"architecture based test" OR Title:"architecture based software test" OR Title:"architecture-based test" OR Title:"architecture-based software test" OR Title:"architecture driven test" OR Title:"architecture driven software test" OR Title:"architecture-driven test" OR Title:"architecture-driven software test") OR ((Abstract:"model based testing" OR Abstract:"model based software testing" OR Abstract:"model-based testing" OR Abstract:"model-based software testing" OR Abstract:"model driven testing" OR Abstract:"model driven software testing" OR Abstract:"model-driven testing" OR Abstract:"model-driven software testing" OR Abstract:"model based test" OR Abstract:"model based software test" OR Abstract:"model-based test" OR Abstract:"model-based software test" OR Abstract:"model driven test" OR Abstract:"model driven software test" OR Abstract:"model-driven test" OR Abstract:"model-driven software test") AND (Abstract:"architecture")) OR (Abstract:"architecture based testing" OR Abstract:"architecture based software testing" OR Abstract:"architecture-based testing" OR Abstract:"architecture-based software testing" OR Abstract:"architecture driven testing" OR Abstract:"architecture driven software testing" OR Abstract:"architecture-driven testing" OR Abstract:"architecture-driven

	software testing" OR Abstract:"architecture based test" OR Abstract:"architecture based software test" OR Abstract:"architecture-based test" OR Abstract:"architecture-based software test" OR Abstract:"architecture driven test" OR Abstract:"architecture driven software test" OR Abstract:"architecture-driven test" OR Abstract:"architecture-driven software test")
Wiley Interscience	("model based testing" OR "model based software testing" OR "model-based testing" OR "model-based software testing" OR "model driven testing" OR "model driven software testing" OR "model-driven testing" OR "model-driven software testing" OR "model based test" OR "model based software test" OR "model-based test" OR "model-based software test" OR "model driven test" OR "model driven software test" OR "model-driven test" OR "model-driven software test") AND ("architecture based testing" OR "architecture based software testing" OR "architecture-based testing" OR "architecture-based software testing" OR "architecture driven testing" OR "architecture driven software testing" OR "architecture-driven testing" OR "architecture-driven software testing" OR "architecture based test" OR "architecture based soft-ware test" OR "architecture-based test" OR "architecture-based software test" OR "architecture driven test" OR "architecture driven software test" OR "architecture-driven test" OR "architecture-driven software test")
Science Direct	(((Title(model based testing) OR Title(model based software testing) OR Title(model-based testing) OR Title(model-based software testing) OR Title(model driven testing) OR Title(model driven software testing) OR Title(model-driven testing) OR Title(model-driven software testing) OR Title(model based test) OR Title(model based software test) OR Title(model driven test) OR Title(model driven software test) OR Title(model-driven test) OR Title(model-driven software test)) AND (Title(architecture))) OR (Title(architecture based testing) OR Title(architecture based software testing) OR Title(architecture-based testing) OR Title(architecture-based software testing) OR Title(architecture driven testing) OR Title(architecture driven software testing) OR Title(architecture-driven testing) OR Title(architecture-driven software testing) OR Title(architecture based test) OR Title(architecture based software test) OR Title(architecture-based test) OR Title(architecture-based software test) OR Title(architecture driven test) OR Title(architecture driven software test) OR Title(architecture-driven test) OR Title(architecture-driven software test))) OR (((Abstract(model based testing) OR Abstract(model based software testing) OR Abstract(model-based testing) OR Abstract(model-based software testing) OR Abstract(model driven testing) OR Abstract(model driven software testing) OR Abstract(model-driven testing) OR Abstract(model-driven software testing) OR Abstract(model based test) OR Abstract(model based software test) OR Abstract(model driven test) OR Abstract(model driven software test) OR

	Abstract(model-driven test) OR Abstract(model-driven software test)) AND (Abstract(architecture))) OR (Abstract(architecture based testing) OR Abstract(architecture based software testing) OR Abstract(architecture-based testing) OR Abstract(architecture-based software testing) OR Abstract(architecture driven testing) OR Abstract(architecture driven software testing) OR Abstract(architecture-driven testing) OR Abstract(architecture-driven software testing) OR Abstract(architecture based test) OR Abstract(architecture based software test) OR Abstract(architecture-based test) OR Abstract(architecture-based software test) OR Abstract(architecture driven test) OR Abstract(architecture driven software test) OR Abstract(architecture-driven test) OR Abstract(architecture-driven software test)))
Springer	("model based testing" OR "model based software testing" OR "model-based testing" OR "model-based software testing" OR "model driven testing" OR "model driven software testing" OR "model-driven testing" OR "model-driven software testing" OR "model based test" OR "model based software test" OR "model-based test" OR "model-based software test" OR "model driven test" OR "model driven software test" OR "model-driven test" OR "model-driven software test") AND ("architecture based testing" OR "architecture based software testing" OR "architecture-based testing" OR "architecture-based software testing" OR "architecture driven testing" OR "architecture driven software testing" OR "architecture-driven testing" OR "architecture-driven software testing" OR "architecture based test" OR "architecture based soft-ware test" OR "architecture-based test" OR "architecture-based software test" OR "architecture driven test" OR "architecture driven software test" OR "architecture-driven test" OR "architecture-driven software test")
ISI Web of Knowledge	(((TS="model based testing" OR TS="model based software testing" OR TS="model-based testing" OR TS="model-based software testing" OR TS="model driven testing" OR TS="model driven software testing" OR TS="model-driven testing" OR TS="model-driven software testing" OR TS="model based test" OR TS="model based software test" OR TS="model driven test" OR TS="model driven software test" OR TS="model-driven test" OR TS="model-driven software test") AND (TS="architecture")) OR (TS="architecture based testing" OR TS="architecture based software testing" OR TS="architecture-based testing" OR TS="architecture-based software testing" OR TS="architecture driven testing" OR TS="architecture driven software testing" OR TS="architecture-driven testing" OR TS="architecture-driven software testing" OR TS="architecture based test" OR TS="architecture based software test" OR TS="architecture-based test" OR TS="architecture-based software test" OR TS="architecture driven test" OR TS="architecture driven software test" OR TS="architecture-driven test" OR TS="architecture-driven software test"))

Appendix B – List of Primary Studies

A. A. Bertolino,F. Corradini,P. Inverardi,H. Muccini. "Deriving Test Plans from Architectural Descriptions". Software Engineering.Proceedings of the 2000 International Conference, 2000. pp. 220-229.

B. Z. Jin, J. Offutt."Deriving Tests From Software Architectures". Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium. pp. 160-171.

C. H. Muccini, P. Inverardi, A. Bertolino. "Using software architecture for code testing", Software Engineering, IEEE Transactions on , vol.30, no.3, pp.160,171, March 2004.

D. H. Muccini, M. Dias, D. J. Richardson. "Systematic Testing of Software Architectures in the C2 Style",Lecture Notes in Computer Science. pp. 295-309.2004.

E. A. Bucchiarone, H. Muccini, P. Pelliccione, P. Pierini. "Model-Checking plus Testing from Software Architecture Analysis to Code Testing",Lecture Notes in Computer Science. pp. 351-365. 2004.

F. G. Scollo, S. Zecchini. "Architectural Unit Testing", Electronic Notes in Theoretical Computer Science, Volume 111, 1 January,pp. 27-52. 2005.

G. K. Winbladh, A. T. Alspaugh, H. Ziv, D. J. Richardson."Architecture Based Testing Using Goals and Plans",Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis. pp. 64-68.2006.

H. H. Muccini, M. Dias, D. J. Richardson."Software Architecture-Based Regression Testing",JSS, Special Edition on Architecting Dependable Systems. pp.1-18. 2006.

İ. A. Johnsen, P. Pettersson, K. Lundqvist. "An Architecture-Based Verification Technique for AADL Specifications",Lecture Notes in Computer Science. pp. 105-113.2009.

J. H. Reza, S. Lande. "Model Based Testing Using Software Architecture",Information Technology: New Generations (ITNG), 2010 Seventh International Conference. pp. 188-193.

K. C. Keum, S Kang, M. Kim."Architecture-Based Testing of Service-Oriented Applications In Distributed Systems",Information and Software Technology, Volume 55, Issue 7,pp. 1212-1223. July 2013.

L. M.Lochau, S.Lity, R. Lachmann, I. Schaefer,U. Goltz. "Delta-oriented Model-Based Integration Testing of Large-Scale Systems". Journal of Systems and Software. pp. 63-84.2014.

Appendix C - Study Quality Assessment

	Quality of Reporting			Rigor			Credibility		Relevance		
Primary study	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Total
A	1	1	1	1	1	1	0	1	0.5	0.5	8
B	1	1	1	0,5	0,5	1	0	1	0,5	0,5	7
C	1	1	1	1	1	1	0	1	1	1	9
D	1	1	1	1	1	1	0	1	0,5	0,5	8
E	1	1	1	1	1	1	0	1	1	0,5	8,5
F	1	1	1	1	1	1	0	1	1	1	9
G	1	1	1	1	1	1	0	0,5	0,5	0,5	7,5
H	1	1	1	1	1	1	0	0,5	1	0,5	8
İ	1	1	1	1	1	1	0	1	1	1	9
J	1	0,5	1	0,5	1	1	0	1	0,5	0,5	7
K	1	1	1	1	1	0,5	0	1	1	0,5	8
L	1	1	1	1	1	1	0	1	1	1	9

Appendix D - Data Extraction Form

Study description	Extraction element	Contents
General Information		
1	ID	Unique id for the study
2	SLR Category	○ Include ○ Exclude
3	Title	Full title of the article
4	Date of Extraction	The date it is added into repository
5	Year	The publication year
6	Authors	
7	Repository	ACM, IEEE, ISI Web of Knowledge, Science Direct, Springer, Wiley Interscience
8	Type	○ Journal ○ Article ○ Book Chapter
Study Description		
9	Addressed Concern	○ Code to Architecture Conformity ○ Functional Concern
10	Test Criteria	○ Coverage Criteria ○ Test Purpose Matching
11	Software Architecture Description Language	CHAM, Wright ADL, FSP Model, UML State, Sequence, Component Diagrams, GoalML, AADL, Acme ADL, BPEL
12	Test Model	LTS , BG, Promela, LTL Formulae, Buchi Automata, IOLTS, Uppaal, HPrTNS, ECFG, eDeltaModels
13	Test Case Execution	○ Automatic ○ Manual
14	Test Case Generation	○ Automatic ○ Manual
15	Test Oracle	○ Automatic ○ Manual
16	Assessment Approach	○ Case Study ○ Experiment ○ Small Example
17	Findings	
18	Constraints / Limitations	
Evaluation		
19	Personal note	The opinions of the reviewer about the study
20	Additional note	Publication details
21	Quality Assessment	Detailed quality scores

Appendix E - Implementation Detail

GENERALIZATION VIEWPOINT

ABSTRACT TEST MODEL

```
package test;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertNotNull;
import java.util.ArrayList;
import java.util.List;

import org.junit.Test;

public class TestGeneralization {

private List<Class<?>> getGeneralizations(Class<?>classObject) {
  if (classObject == null) {
    returnArrays.asList();
  }
  List<Class<?>> generalizations = new ArrayList<Class<?>>();
  generalizations.add(classObject);
  generalizations.addAll(getGeneralizations(classObject.getSuperclass()));
  Class<?>[] superInterfaces = classObject.getInterfaces();
  for (int i = 0; i<superInterfaces.length; i++) {
    generalizations.addAll(getGeneralizations(superInterfaces[i]));
  }
  return generalizations;
}

[%for (declaration in generalization.decl){%]
@Test
public void test[%="".testName(declaration.child.name,declaration.parent.name)
                                    %]()throws ClassNotFoundException {
  String className = "[%=declaration.child.name%]";
  String inheritsFrom = "";
  String implementz = "";
  [%if(declaration.type().name== "Inheritance" ){%]
     inheritsFrom = "[%=declaration.parent.name%]";
  [%}else{%]
     implementz = "[%=declaration.parent.name%]";
  [%}%]
  Class<?> clazz = Class.forName(className);
  List<Class<?>> allGeneralizations = getGeneralizations(clazz);
  assertNotNull(clazz);
  if(inheritsFrom != ""){
    Class<?> inheritsFromClazz = Class.forName(inheritsFrom);
    String errorMessageExtension = clazz.getName() + "does not extend "
                                    + inheritsFromClazz.getName();
    assertNotNull(inheritsFromClazz);
    assertTrue(errorMessageExtension,allGeneralizations.contains
                                                    (inheritsFromClazz));
  }
  if(implementz != ""){
    Class<?> implementsClazz = Class.forName(implementz);
    String errorMessageImplements = clazz.getName()+ " does not implement " +
                                        implementsClazz.getName();
    assertNotNull(implementsClazz);
    assertTrue(errorMessageImplements,
    allGeneralizations.contains(implementsClazz));
  }
 }
    [%}%]
 }
[%
function String testName(a:String, b:String):String{
  var a1:String = a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2)
                                                    .toUpperCase();
  var a2:String = a.substring(a.lastIndexOf('.')+2);
  var b1:String = b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2)
                                                    .toUpperCase();
  var b2:String = b.substring(b.lastIndexOf('.')+2);
  return a1+a2+"Of"+b1+b2;
}%]
```

TRANSFORMATION TEST MODEL

```
rule Generalization2JUnit
        transform generalization : GeneralizationModel {
        // The EGL template to be invoked
        template :"Generalization2JUnit.egl"
        // Output file
        target :"gen/TestGeneralization.java"
}
```

DECOMPOSITION VIEWPOINT

ABSTRACT TEST MODEL

```
package test;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;

import org.junit.Assert;
import org.junit.Test;
import org.reflections.Reflections;
import org.reflections.scanners.ResourcesScanner;
import org.reflections.scanners.SubTypesScanner;
import org.reflections.util.ClasspathHelper;
import org.reflections.util.ConfigurationBuilder;
import org.reflections.util.FilterBuilder;

public class TestDecomposition {

  private List<Package> getSubPackages(String packageName) {
    List<Package>packageList = new ArrayList<Package>();
    List<ClassLoader>classLoadersList = new LinkedList<ClassLoader>();
    classLoadersList.add(ClasspathHelper.contextClassLoader());
    classLoadersList.add(ClasspathHelper.staticClassLoader());
    Reflections reflections = new Reflections(new ConfigurationBuilder()
               .setScanners(new SubTypesScanner(false),new ResourcesScanner())
               .setUrls(ClasspathHelper.forClassLoader(classLoadersList
               .toArray(new ClassLoader[0]))).filterInputsBy(new
               FilterBuilder().include(FilterBuilder.prefix(packageName))));

    Set<Class<? extends Object>>allClasses = reflections.getSubTypesOf
                                                    (Object.class);
    for (Class<? extends Object>clazz : allClasses) {
      if (!packageList.contains(clazz.getPackage())) {
         packageList.add(clazz.getPackage());
      }
    }
    return packageList;
  }

private Boolean isPackageExistsInGivenList(List<Package> packageList,
                                                    String packageName) {
  for (Package pack : packageList) {
    if (pack.getName().equals(packageName)) {
      return true;
    }
    if (pack.getName().length() >packageName.length()) {
     if (pack.getName().substring(0, packageName.length()).equals(packageName))
     {
       return true;
     }
    }
   }
  return false;
 }
 [%for (element in decomposition.elements){%]
       [%for (subelement in element.subelements){%]
@Test
public void test[%="".testName(element.name,subelement.name)%]() {
  String decomposedPackageName = "[%=element.name%]";
  String subPackageName = "[%=subelement.name%]";
  Assert.assertTrue(isPackageExistsInGivenList(
                getSubPackages(decomposedPackageName),decomposedPackageName));
  Assert.assertTrue(isPackageExistsInGivenList(
```

```
                getSubPackages(subPackageName), subPackageName));
  Assert.assertTrue(isPackageExistsInGivenList(
                getSubPackages(decomposedPackageName), subPackageName));
 }

   [%}%]
[%}%]
}
[%
function String testName(a:String, b:String):String{
  var a1:String = a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2).
                  toUpperCase();
  var a2:String = a.substring(a.lastIndexOf('.')+2);
  var b1:String = b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2).
                  toUpperCase();
  var b2:String = b.substring(b.lastIndexOf('.')+2);
  return a1+a2+"DecomposedOf"+b1+b2;
}

%]
```

TRANSFORMATION TEST MODEL

```
rule Decomposition2JUnit
        transform decomposition : Model {
        // The EGL template to be invoked
        template :"Decomposition2JUnit.egl"
        // Output file
        target :"gen/TestDecomposition.java"
}
```

SHARED DATA VIEWPOINT

```
package test;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertNotNull;
import java.lang.reflect.Method;

import org.junit.Test;
public class TestSharedData {

private Boolean isMethodExists(Method[] methods, String name) {
  for (Method method : methods) {
   if (method.getName().equals(name)) {
     return true;
   }
  }
  return false;
}
[% for (attachment in sharedDataModel.attachments) { %]
@Test
public void test[%if(attachment.type().name== "DataRead"){%][%="".testName(
attachment.da.name,attachment.dataRead)%][%}else{%][%="".testName(attachment.da
.name,attachment.dataWrite)%][%}%]()throws ClassNotFoundException {
  String dataAccessorClassName = "[%=attachment.da.name%]";
  Class<?> accessorClass = Class.forName(dataAccessorClassName);
  assertNotNull(accessorClass);
  String readMethodName = "[%if(attachment.type().name== "DataRead")%]
                           [%=attachment.dataRead%]";
  String writeMethodName ="[%if(attachment.type().name == "DataWrite" )%]
                          [%=attachment.dataWrite%]";
  String failureMessage = accessorClass.getName();
  if (!readMethodName.equals("")) {
    failureMessage += "'s data read property is not satisfied";
    assertTrue(failureMessage,isMethodExists(accessorClass.getMethods(),
                              readMethodName));
  }
  if (!writeMethodName.equals("")) {
    failureMessage += "'s data write read property is not satisfied";
    assertTrue(failureMessage, isMethodExists(accessorClass.getMethods(),
                              writeMethodName));
  }
 }
[% } %]

}
[%
```

```
function String testName(a:String, b:String):String{
  var a1:String = a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2).
                  toUpperCase();
  var a2:String = a.substring(a.lastIndexOf('.')+2);
  var b1:String = b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2).
                  toUpperCase();
  var b2:String = b.substring(b.lastIndexOf('.')+2);
  return a1+a2+b1+b2;
}
%]
```

TRANSFORMATION TEST MODEL

```
rule SharedData2JUnit
        transform sharedDataModel : Model {
        // The EGL template to be invoked
        template :"sharedData2JUnit.egl"
        // Output file
        target :"gen/TestSharedData.java"
}
```

USES VIEWPOINT

ABSTRACT TEST MODEL

```
package test;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import java.util.Map;

import org.junit.Test;

public class TestUses extends AbstractTestUseRelation {
[% for (relation in uses.relations) { %]
@Test
public void test[%="".testName(relation.source.name, relation.target.name)%]()
{
  String source = "[%=relation.source.name%]";
  String target = "[%=relation.target.name%]";
  String errorMessage = source + " invalidates use relation to "+ target;
  assertTrue(isPackageExistsInGivenList(getSubPackages(source),source));
  assertTrue(isPackageExistsInGivenList(getSubPackages(target),target));
  Map<String, String> usesMap = doesSourceUseTarget(source, target);
  assertFalse(errorMessage, usesMap.isEmpty());
}
[%}%]

}
[%
function String testName(a:String, b:String):String{
  var a1:String = a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2).
                  toUpperCase();
  var a2:String = a.substring(a.lastIndexOf('.')+2);
  var b1:String = b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2).
                  toUpperCase();
  var b2:String = b.substring(b.lastIndexOf('.')+2);
  return a1+a2+"2"+b1+b2;
}
%]
```

TRANSFORMATION TEST MODEL

```
rule Use2JUnit
        transform uses : Model {
        // The EGL template to be invoked
        template :"Use2JUnit.egl"
        // Output file
        target :"gen/TestUses.java"
}
```

LAYERED VIEWPOINT

ABSTRACT TEST MODEL

```
package test;

import static org.junit.Assert.assertTrue;

import java.util.Iterator;
```

```
import java.util.Map;
import java.util.Map.Entry;

import org.junit.Test;


public class TestLayered extends AbstractTestUseRelation {
[% for (relation in layeredModel.relations) { %]
  [%if(relation.type().name== "Allowed_To_Use_Below" ){%]
@Test
public void test[%="".testName(relation.sourceLayer.name,relation.targetLayer
                                                              .name)%]() {
  String sourceLayer ="[%=relation.sourceLayer.name%]" ;
  String targetLayer = "[%=relation.targetLayer.name%]";
  Map<String, String> usesMap = doesSourceUseTarget(targetLayer,sourceLayer);
  Iterator<Entry<String, String>> iterator = usesMap.entrySet().iterator();
  String errorMessage = "";
  while (iterator.hasNext()) {
    Entry<String, String> entry = iterator.next();
    errorMessage += entry.getKey() + " breaks layered relation using "
                          + entry.getValue();
  }
  assertTrue(isPackageExistsInGivenList(getSubPackages(sourceLayer),
                          sourceLayer));
  assertTrue(isPackageExistsInGivenList(getSubPackages(targetLayer),
                          targetLayer));
  assertTrue(errorMessage, usesMap.isEmpty());
 }
  [%}%]
[%}%]
}
[%
function String testName(a:String, b:String):String{
  var a1:String = a.substring(a.lastIndexOf('.')+1,a.lastIndexOf('.')+2).
                  toUpperCase();
  var a2:String = a.substring(a.lastIndexOf('.')+2);
  var b1:String = b.substring(b.lastIndexOf('.')+1,b.lastIndexOf('.')+2).
                  toUpperCase();
  var b2:String = b.substring(b.lastIndexOf('.')+2);
  return a1+a2+"2"+b1+b2;
}

%]
```

TRANSFORMATION TEST MODEL

```
rule Layered2JUnit
        transform layeredModel : Model {
        // The EGL template to be invoked
        template :"Layered2JUnit.egl"
        // Output file
        target :"gen/TestLayered.java"
}
```

AbstractTestUseRelation

```
package test;

import java.lang.reflect.Field;
import java.lang.reflect.GenericArrayType;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.reflections.Reflections;
import org.reflections.scanners.ResourcesScanner;
import org.reflections.scanners.SubTypesScanner;
import org.reflections.util.ClasspathHelper;
import org.reflections.util.ConfigurationBuilder;
import org.reflections.util.FilterBuilder;

public abstract class AbstractTestUseRelation {

 protected List<Package>getSubPackages(String packageName) {
  List<Package> packageList = new ArrayList<Package>();
  List<ClassLoader> classLoadersList = new LinkedList<ClassLoader>();
  classLoadersList.add(ClasspathHelper.contextClassLoader());
```

```
    classLoadersList.add(ClasspathHelper.staticClassLoader());
    Reflections reflections = new Reflections(new ConfigurationBuilder()
         .setScanners(new SubTypesScanner(false),new ResourcesScanner())
         .setUrls(ClasspathHelper.forClassLoader(classLoadersList
         .toArray(new ClassLoader[0]))).filterInputsBy(new FilterBuilder()
        .include(FilterBuilder.prefix(packageName))));
    Set<Class<? extends Object>>allClasses = reflections.getSubTypesOf
                                              (Object.class);
    for (Class<? extends Object> clazz : allClasses) {
      if (!packageList.contains(clazz.getPackage())) {
        packageList.add(clazz.getPackage());
      }
     }
    return packageList;
}

protected boolean isPackageExistsInGivenList(List<Package> packageList,
                                             String packageName) {
  for (Package pack : packageList) {
    if (pack.getName().equals(packageName)) {
       return true;
    }
   }
   return false;
  }

protected Map<String, String> doesSourceUseTarget(String sourcePackage,
                                                  String targetPackage) {
  HashMap<String, String> usesMap = new HashMap<String, String>();
  Set<Class<? extends Object>> allUserClasses = getClassesUnderPackage
                                                    (sourcePackage);
  Set<Class<? extends Object>>allUsedClasses = getClassesUnderPackage
                                                    (targetPackage);

  for (Class<? extends Object>userClazz : allUserClasses) {
   for (Class<? extends Object>usedClazz : allUsedClasses) {
     Field[] fields = userClazz.getDeclaredFields();
       for (Field field : fields) {
         if (field.getType().equals(usedClazz)) {
            usesMap.put(userClazz.getName(), usedClazz.getName());
            return usesMap;
          } else if (field.getGenericType() instanceof ParameterizedType) {
            Type[] actualTypeArguments = ((ParameterizedType) (field
                              .getGenericType())).getActualTypeArguments();
            for (Type type : actualTypeArguments) {
               if (type.equals(usedClazz)) {
                 usesMap.put(userClazz.getName(),usedClazz.getName());
                 return usesMap;
               }
             }
       } else if (field.getGenericType() instanceof GenericArrayType) {
         Type type = ((GenericArrayType) (field.getGenericType()))
                                                  .getGenericComponentType();
         if (type.equals(usedClazz)) {
            usesMap.put(userClazz.getName(),usedClazz.getName());
            return usesMap;
          }
       } else if (field.getType().isArray()) {
         Class<?>array = field.getType();
         if (array.getComponentType().equals(usedClazz)) {
           usesMap.put(userClazz.getName(), usedClazz.getName());
          return usesMap;
         }
       }
      }
     }
    }
   return usesMap;
}

private Set<Class<? extends Object>> getClassesUnderPackage(String packageName)
{
  List<ClassLoader>classLoadersList = newLinkedList<ClassLoader>();
  classLoadersList.add(ClasspathHelper.contextClassLoader());
  classLoadersList.add(ClasspathHelper.staticClassLoader());
  Reflections reflections = new Reflections(new ConfigurationBuilder().
              setScanners(new SubTypesScanner(false),new ResourcesScanner()).
              setUrls(ClasspathHelper.forClassLoader(classLoadersList
              toArray(new ClassLoader[0]))).filterInputsBy(new FilterBuilder()
```

```
            .include(FilterBuilder.prefix(packageName))));
  Set<Class<? extends Object>> allClasses = reflections.getSubTypesOf
                                                        (Object.class);
  removeClassesThatAreNotDirectlyUnderGivenPackage(packageName,allClasses);
  return allClasses;
}

private void removeClassesThatAreNotDirectlyUnderGivenPackage(String
                        packageName, Set<Class<? extends Object>>allClasses) {
  List<Class<?>> notDirectSubClasses = new ArrayList<Class<?>>();
  for (Class<?> clazz : allClasses) {
    if (!clazz.getPackage().getName().equals(packageName)) {
       notDirectSubClasses.add(clazz);
    }
   }
  allClasses.removeAll(notDirectSubClasses);
 }
}
```