Model-Driven Software Architecture Conformance Analysis

Burak Uzun

Propositions

1. Software architecture conformance analysis is a preferred verification technique for ensuring conformance of the software architecture and the code.
(this thesis)
2. A complete software architecture conformance analysis requires the adoption of architecture views.
(this thesis)
3. High energy consumption in blockchain technologies will impede its further growth.
4. Automation of repetitive tasks significantly reduces the effort needed for coping with complex problems.
5. Failed projects significantly decrease the economic well-being of countries and their residents.
6. Unregulated artificial general intelligence forms an existential risk for the human race.

Propositions belonging to the thesis, entitled
**“Model-Driven Software Architecture Conformance
Analysis”**

Burak Uzun

Wageningen, 27 August 2021

Model-Driven Software Architecture Conformance Analysis

Burak Uzun

Thesis committee

Promotor

Prof. Dr B. Tekinerdogan
Professor of Information Technology
Wageningen University & Research

Other members

Prof. Dr M. Akşit, University of Twente, Enschede
Prof. Dr M. Chaudron, Eindhoven University
Prof. Dr H. Vangheluwe, University of Antwerp, Belgium
Prof. Dr J. Pennings, Wageningen University & Research

This research was conducted under the auspices of the Wageningen School of Social Sciences (WASS)

Model-Driven Software Architecture Conformance Analysis

Burak Uzun

Thesis

submitted in fulfilment of the requirements for the degree of doctor
at Wageningen University
by the authority of the Rector Magnificus,
Prof. Dr A.P.J. Mol,
in the presence of the
Thesis Committee appointed by the Academic Board
to be defended in public
on Tuesday 27 August 2021
at 11:00 a.m. in the Aula.

B. Uzun
Model-Driven Software Architecture Conformance Analysis
215 pages.

PhD thesis, Wageningen University, Wageningen, NL (2021)
With references, and summary in English and Dutch

ISBN: 978-94-6395-765-6
DOI: 10.18174/544468

TABLE OF CONTENTS

LIST OF FIGURES	IX
LIST OF TABLES	XII
1 INTRODUCTION	13
1.1 BACKGROUND	15
1.2 OBJECTIVES AND RESEARCH QUESTIONS	16
1.3 CASE STUDIES	16
1.3.1 E-government Case	16
1.3.2 E-commerce Case	17
1.4 RESEARCH METHODOLOGY	17
1.4.1 Systematic Literature Review	19
1.4.2 Case Study Research	19
1.4.3 Survey Study	19
1.4.4 Domain Driven Analysis	19
1.4.5 Design Science Research	20
1.5 CONTRIBUTIONS	20
2 MODEL-DRIVEN ARCHITECTURE BASED TESTING: A SYSTEMATIC LITERATURE REVIEW	23
2.1 INTRODUCTION	25
2.2 BACKGROUND	26
2.2.1 Model Based Testing	26
2.2.2 Software Architecture Modeling	28
2.3 RESEARCH METHOD	28
2.3.1 Review Protocol	29
2.3.2 Research Questions	30
2.3.3 Search Strategy	30
2.3.4 Study Selection Criteria	31
2.3.5 Study Quality Assessment	31
2.3.6 Data Extraction	32
2.3.7 Data Synthesis	33
2.4 RESULTS	33
2.4.1 Overview of the Reviewed Studies	33
2.4.2 Research Methods	35
2.4.3 Methodological Quality	36
2.4.4 Systems Investigated	37
2.4.4.1 Summary	47
2.4.5 Threats to Validity	54
2.5 RELATED WORK	55
2.6 CONCLUSION	57
3 DESIGN OF VARIABLE BIG DATA ARCHITECTURES FOR E-GOVERNMENT DOMAIN	59
3.1 INTRODUCTION	61
3.2 BACKGROUND	62
3.2.1 Reference Architecture	62
3.2.2 Big Data Reference Architecture	63
3.3 METHOD FOR DERIVING E-GOVERNMENT SYSTEMS	64
3.4 DEVELOPMENT OF REFERENCE MODELS	66
3.4.1 Variability Model for Big Data Systems	66
3.4.2 Variability Model for E-Government Systems	67

3.4.3	<i>E-Government Reference Business Process Models</i>	68
3.4.4	<i>E-Government Big Data Reference Architecture</i>	69
3.5	DEVELOPMENT OF CONCRETE E-GOVERNMENT ARCHITECTURE	72
3.5.1	<i>Select Features for Big Data</i>	72
3.5.2	<i>Select Features for System E-Government System</i>	73
3.5.3	<i>Develop E-Government Application Business Process Model</i>	74
3.5.4	<i>Develop E-Government Big Data Concrete Architecture</i>	75
3.6	DISCUSSION	80
3.7	RELATED WORK	82
3.8	CONCLUSION	83
4	ARCHITECTURE CONFORMANCE ANALYSIS USING MODEL-BASED TESTING: A CASE STUDY APPROACH	85
4.1	INTRODUCTION	87
4.2	PRELIMINARIES	89
4.2.1	<i>Reflexion Modelling</i>	89
4.2.2	<i>Model Based Testing</i>	91
4.3	INDUSTRIAL CASE AND PROBLEM STATEMENT	92
4.4	CONFORMANCE CHECKING USING MODEL-DRIVEN ARCHITECTURE BASED TESTING	94
4.5	ACAMBT CONFORMANCE CHECKING TOOL FRAMEWORK	96
4.5.1	<i>Modelling Architecture Viewpoints</i>	96
4.5.2	<i>Modelling Architecture Views</i>	98
4.5.3	<i>Modeling Architecture Viewpoint Criteria</i>	98
4.5.4	<i>Creating Test Model for Architecture View</i>	100
4.5.5	<i>Creating Concrete Test Suite for Architecture View</i>	101
4.6	ARCHITECTURAL MODELS FOR THE E-GOVERNMENT SYSTEM	101
4.7	CASE STUDY DESIGN	105
4.7.1	<i>How effective is the adopted ACAMBT approach?</i>	107
4.7.1.1	Existing Architectural Compliance	107
4.7.1.2	Fault Injection-Based Testing	108
4.7.2	<i>How practical is the adopted ACAMBT approach?</i>	111
4.7.2.1	Survey Study Design	111
4.7.2.2	Data Collection	113
4.8	DISCUSSION	119
4.9	RELATED WORK	121
4.10	CONCLUSION	122
5	DOMAIN-DRIVEN ANALYSIS OF ARCHITECTURE RECONSTRUCTION METHODS	125
5.1	INTRODUCTION	127
5.2	PRELIMINARIES	128
5.2.1	<i>Software Architecture Reconstruction</i>	128
5.2.2	<i>Domain Analysis</i>	129
5.3	DOMAIN MODEL OF ARCHITECTURE RECONSTRUCTION METHODS	130
5.3.1	<i>Domain Scoping</i>	130
5.3.2	<i>Feature Model</i>	131
5.3.3	<i>Generic Business Process Model</i>	135
5.4	CONCRETE ARCHITECTURE RECONSTRUCTION METHOD	136
5.4.1	<i>Recovering software architecture from the names of source files</i>	136
5.4.2	<i>ART: An Architectural Reverse Engineering Environment</i>	138
5.5	RELATED WORK	139
5.6	DISCUSSION	139
5.7	CONCLUSION	140
6	VIEW-DRIVEN ARCHITECTURE RECONSTRUCTION APPROACH	141
6.1	INTRODUCTION	143

6.2	BACKGROUND	144
6.2.1	<i>Software Architecture Modelling</i>	144
6.2.2	<i>Software Architecture Reconstruction</i>	145
6.3	METAMODELING OF ARCHITECTURE VIEWPOINTS	146
6.3.1	<i>Decomposition View</i>	146
6.3.2	<i>Shared Data View</i>	146
6.3.3	<i>Uses View</i>	147
6.3.4	<i>Generalization View</i>	147
6.3.5	<i>Layered View</i>	147
6.4	VIEWPOINT ORIENTED SOFTWARE ARCHITECTURE RECONSTRUCTION TOOL	149
6.5	RELATED WORK	155
6.6	CONCLUSION	155
7	ARCHITECTURE VIEW DRIVEN CONFORMANCE ANALYSIS APPROACH FOR DETECTING DEVIATIONS IN THE CODE	157
7.1	INTRODUCTION	159
7.2	BACKGROUND	160
7.2.1	<i>Software Architecture Modelling</i>	160
7.2.2	<i>Software Architecture Conformance Analysis</i>	161
7.3	VIEWPOINT ORIENTED SOFTWARE ARCHITECTURE CONFORMANCE ANALYSIS	162
7.4	VIEWPOINT ORIENTED ARCHITECTURE CONFORMANCE ANALYSIS TOOL	163
7.4.1	<i>Implementation of Conformance Analysis for each Viewpoint</i>	163
7.4.2	<i>Results of the Conformance Analysis</i>	166
7.4.3	<i>Eclipse Tool</i>	167
7.5	CASE STUDY DESIGN	168
7.5.1	<i>Industrial Case Study: E-commerce</i>	169
7.5.1.1	<i>Shared Data View</i>	170
7.5.1.2	<i>Decomposition View</i>	170
7.5.1.3	<i>Uses View</i>	171
7.5.1.4	<i>Layered View</i>	171
7.5.1.5	<i>Generalization View</i>	172
7.5.2	<i>How effective is the adopted VOSACAM approach?</i>	172
7.5.2.1	<i>Shared Data View</i>	173
7.5.2.2	<i>Decomposition View</i>	173
7.5.2.3	<i>Uses View</i>	174
7.5.2.4	<i>Layered View</i>	174
7.5.2.5	<i>Generalization View</i>	175
7.5.2.6	<i>Summary</i>	176
7.6	DISCUSSIONS	176
7.7	RELATED WORK	177
7.8	CONCLUSION	180
8	GENERAL DISCUSSION	181
8.1	INTRODUCTION	183
8.2	RESEARCH QUESTIONS	183
8.2.1	<i>What are current solutions for model-driven architecture-based testing?</i>	183
8.2.2	<i>What are the limitations and benefits of code level model-based architecture conformance analysis using architecture viewpoints?</i>	184
8.2.3	<i>What are the characteristic features of architecture reconstruction methods?</i>	185
8.2.4	<i>What are the limitations and benefits of architecture level model-based architecture conformance analysis using architecture viewpoints?</i>	186
	APPENDICES	187
	APPENDIX-A SEARCH STRINGS	188
	APPENDIX-C STUDY QUALITY ASSESSMENT	193
	APPENDIX-D DATA EXTRACTION FORM	195

<i>APPENDIX-E PRIMARY STUDIES FOR SOFTWARE ARCHITECTURE RECONSTRUCTION METHODS.....</i>	<i>196</i>
REFERENCES.....	199
SUMMARY	207
SAMENVATTING.....	209
ACKNOWLEDGEMENTS.....	213
ABOUT THE AUTHOR.....	215

LIST OF FIGURES

<i>Figure 1-1: Research methodologies and corresponding contributions</i>	<i>18</i>
<i>Figure 2-1: Process of model based testing.....</i>	<i>27</i>
<i>Figure 2-2: The adopted review protocol</i>	<i>29</i>
<i>Figure 2-3: Year-wise distribution of primary studies.....</i>	<i>33</i>
<i>Figure 2-4: Addressed concern distribution of primary studies</i>	<i>38</i>
<i>Figure 2-5: Generic Process for Architecture-Based Testing</i>	<i>39</i>
<i>Figure 2-6: The Identified Architecture Description Approaches and the Frequency Over the Different Studies</i>	<i>48</i>
<i>Figure 2-7: The Identified Test Criteria</i>	<i>48</i>
<i>Figure 2-8: The Identified Test Model Representations</i>	<i>49</i>
<i>Figure 2-9: The Identified Test Model Construction</i>	<i>49</i>
<i>Figure 2-10: The Identified Abstract Test Suite.....</i>	<i>50</i>
<i>Figure 2-11: The Identified Abstract Test Case Generation Type.....</i>	<i>50</i>
<i>Figure 2-12: The Identified Concrete Test Suite.....</i>	<i>51</i>
<i>Figure 2-13: The Identified Concrete Test Case Generation Type.....</i>	<i>51</i>
<i>Figure 2-14: The Identified Test Case Execution</i>	<i>51</i>
<i>Figure 2-15: The Identified Test Analysis Type</i>	<i>52</i>
<i>Figure 3-1: Relation between reference architecture and concrete architectures</i>	<i>63</i>
<i>Figure 3-2: Big Data Reference Architecture (adapted from: [47])</i>	<i>64</i>
<i>Figure 3-3: Method for Developing E-Government Big Data System.....</i>	<i>65</i>
<i>Figure 3-4: Top-Level Feature Model for Big Data System (adapted from:[48])</i>	<i>66</i>
<i>Figure 3-5: Top-Level Feature Model for Big Data System (adapted from:[48])</i>	<i>67</i>
<i>Figure 3-6: BPMN Diagram for E-Government Systems</i>	<i>69</i>
<i>Figure 3-7: Decomposition View Reference Model for E-Government Big Data System</i>	<i>70</i>
<i>Figure 3-8: Layered View Reference Model for E-Government Big Data System</i>	<i>71</i>
<i>Figure 3-9: Deployment View Reference Model for E-Government Big Data Systems</i>	<i>72</i>
<i>Figure 3-10: BPMN Diagram for G2G Application.....</i>	<i>75</i>
<i>Figure 3-11: Decomposition View of the Application Architecture of Example E-Government System</i>	<i>78</i>
<i>Figure 3-12: Layered View of the Application Architecture of Example E-Government System</i>	<i>79</i>
<i>Figure 3-13: Deployment View of the Application Architecture of Example E-Government System</i>	<i>80</i>
<i>Figure 4-1: Activity diagram showing the steps in reflexion modeling for architecture-to-code conformance</i>	<i>90</i>
<i>Figure 4-2: Process of model-based testing</i>	<i>92</i>
<i>Figure 4-3: General Workflow of the G2C E-Government System</i>	<i>93</i>
<i>Figure 4-4: Example layered architecture of G2C E-Government System</i>	<i>94</i>
<i>Figure 4-5: Process model of architecture conformance analysis using model-based testing</i>	<i>95</i>
<i>Figure 4-6: Snapshot of the toolset that implements the approach</i>	<i>96</i>
<i>Figure 4-7: Metamodels of the selected viewpoints.....</i>	<i>97</i>
<i>Figure 4-8: Example HUTN model for the decomposition view.....</i>	<i>98</i>

Figure 4-9: EGL Template for conformance checking constraints of the decomposition viewpoint.....	100
Figure 4-10: EGX Model incorporating rules for generating test cases for the decomposition viewpoint.....	100
Figure 4-11: Example Junit Test Suite (part) for the Decomposition Viewpoint	101
Figure 4-12: Deployment view of the adopted project	102
Figure 4-13: Shared data view of adopted project infrastructure	102
Figure 4-14: Decomposition view of adopted project infrastructure	103
Figure 4-15: Uses view of the adopted project infrastructure.....	103
Figure 4-16: Layered view of adopted project infrastructure	104
Figure 4-17: Generalization view of adopted project infrastructure within package B.....	105
Figure 5-1: Conceptual Model for Architecture Reconstruction.....	128
Figure 5-2: Common structure of domain analysis methods.....	129
Figure 5-3: Feature Model for Architecture Reconstruction Methods.....	132
Figure 5-4: Generic business process model for Architecture Reconstruction	135
Figure 5-5: Relation between Reference SAR and concrete SAR.....	136
Figure 5-6: Business Process Model for Recovering software architecture from the names of source files.....	137
Figure 5-7: Business Process Model for ART: An Architectural Reverse Engineering Environment.....	138
Figure 6-1: Architecture Viewpoint Concepts and their relations	144
Figure 6-2: Conceptual Model for Architecture Reconstruction.....	145
Figure 6-3: Metamodels of the Selected Viewpoints.....	148
Figure 6-4: Architecture Reconstruction Library Details	149
Figure 6-5: Architecture Reconstruction API Usage.....	150
Figure 6-6: Example decomposition of source code and corresponding view model	151
Figure 6-7: Example generalization relations in source code and corresponding view model.....	152
Figure 6-8: Example shared data relations in source code and corresponding view model	153
Figure 6-9: Example uses relations in source code and corresponding view model	154
Figure 6-10: Example layered relations in source code and corresponding view model	154
Figure 7-1: Architecture Viewpoint Concepts and their relations	161
Figure 7-2: Workflow Model for VOSACAM.....	163
Figure 7-3: Decomposition Viewpoint ECL Code	165
Figure 7-4: Uses Viewpoint ECL Code.....	165
Figure 7-5: Generalization Viewpoint ECL Code	165
Figure 7-6: Layered Viewpoint ECL Code	166
Figure 7-7: Shared Data Viewpoint ECL Code.....	166
Figure 7-8: Conformance and Divergence Extracting ECL Code	167
Figure 7-9: Sample Output from Execution.....	167
Figure 7-10: Snapshot of the eclipse workbench showing the conformance analysis.....	168
Figure 7-11: Shared data view of adopted project infrastructure	170
Figure 7-12: Decomposition view of adopted project infrastructure	170
Figure 7-13: Uses view of adopted project infrastructure.....	171
Figure 7-14: Layered view of adopted project infrastructure	171
Figure 7-15: Generalization view of adopted project infrastructure	172

<i>Figure 7-16: Shared data conformance analysis result.....</i>	<i>173</i>
<i>Figure 7-17: Decomposition viewpoint conformance analysis result</i>	<i>174</i>
<i>Figure 7-18: Uses viewpoint conformance analysis result.....</i>	<i>174</i>
<i>Figure 7-19: Layered viewpoint conformance analysis result.....</i>	<i>175</i>
<i>Figure 7-20: Generalization viewpoint conformance analysis result.....</i>	<i>175</i>

LIST OF TABLES

<i>Table 1-1: Applied Research Methodologies for the identified research questions</i>	<i>17</i>
<i>Table 2-1: Overview of search results and study selection.....</i>	<i>31</i>
<i>Table 2-2: Adopted Quality Checklist.....</i>	<i>32</i>
<i>Table 2-3: Distribution of the studies over Publication Channel.....</i>	<i>33</i>
<i>Table 2-4: Distribution of studies over Research Method</i>	<i>35</i>
<i>Table 2-5: Overall quality of the primary studies.....</i>	<i>37</i>
<i>Table 2-6: Addressed Concerns and Studies.....</i>	<i>38</i>
<i>Table 2-7: Overview of the identified approaches</i>	<i>40</i>
<i>Table 2-8: Overview of the identified obstacles that has been discussed in the related primary studies</i>	<i>52</i>
<i>Table 3-1: Features of the Big Data System</i>	<i>73</i>
<i>Table 3-2: Features of the E-Government System</i>	<i>74</i>
<i>Table 3-3: Features of the Big Data System and the implied architecture design decisions ..</i>	<i>76</i>
<i>Table 3-4: Features of the E-Government System and the implied architecture design decisions</i>	<i>77</i>
<i>Table 4-1: Identified conformance checking constraints for the selected viewpoints</i>	<i>99</i>
<i>Table 4-2: Case Study Design.....</i>	<i>106</i>
<i>Table 4-3: Testing results of the real code for each viewpoint.....</i>	<i>107</i>
<i>Table 4-4: Example Fault-Injections for the adopted viewpoints.....</i>	<i>109</i>
<i>Table 4-5: Pseudo codes for the fault injections in the architecture views</i>	<i>110</i>
<i>Table 4-6: Fault-based testing results for each viewpoint.....</i>	<i>111</i>
<i>Table 4-7: Survey Study Design.....</i>	<i>112</i>
<i>Table 4-8: Overall Population Size and Selection</i>	<i>113</i>
<i>Table 4-9: Post-Survey Questionnaire.....</i>	<i>114</i>
<i>Table 4-10: Results of the Post-Survey Questionnaire</i>	<i>115</i>
<i>Table 4-11: Results of the Post-Survey Questionnaire represented as bar charts</i>	<i>116</i>
<i>Table 4-12: Results of the Open Questions in the Post-Survey Questionnaire.....</i>	<i>117</i>
<i>Table 4-13: Construct Validity.....</i>	<i>120</i>
<i>Table 5-1: Study Selection Criteria.....</i>	<i>130</i>
<i>Table 5-2: Overview of search results and study selection.....</i>	<i>131</i>
<i>Table 5-3: Identified feature composition constraints</i>	<i>134</i>
<i>Table 5-4: Feature Selection for Recovering software architecture from the names of source files.....</i>	<i>137</i>
<i>Table 5-5: Feature Selection for ART: An Architectural Reverse Engineering Environment</i>	<i>138</i>
<i>Table 6-1: Pseudocodes for reconstructing view models per architecture viewpoint</i>	<i>150</i>
<i>Table 7-1: Pseudocode for the conformance analysis</i>	<i>164</i>
<i>Table 7-2: Case Study Design.....</i>	<i>169</i>
<i>Table 7-3: Execution result of our approach.....</i>	<i>176</i>
<i>Table 7-4: Overview of existing architecture conformance analysis and architecture reconstruction method</i>	<i>179</i>

1

INTRODUCTION

1.1 BACKGROUND

A software system often includes a large number of elements interacting with each other in various ways to solve the concerns of the system's stakeholders. Every software system has an architecture that defines the gross-level structure of the system and consists of architectural components, relations and their interactions. A common practice to model software architectures is based on the notion of so-called architecture viewpoints, which define the templates and guidelines for the architecture from the perspective of one or more stakeholder concerns. Based on architecture viewpoints, architecture views can be derived for a particular system. Usually, the architecture of a system consists of multiple architecture views.

Since software systems constantly evolve, the software code can start to diverge from the documented software architecture. To tackle this problem, researchers proposed various architecture conformance analysis methods that aim to detect the differences between the code and the software architecture. These differences can be categorized into so-called absence and divergence relations. Absence relation refers to details that are present in intended architecture and missing in implemented architecture. Divergence relation refers to details that are present in implemented architecture and missing in intended architecture. Aside from divergence and absence relations, there is also a conformance relation which means implemented and intended architecture details are the same. Conformance analysis is performed on two levels which are code and architecture levels. Code-level conformance analysis employs intended architecture to generate runnable specifications that are executed on the code to reveal divergence or absence relations. Architecture level conformance analysis employs reconstructed architecture from implementation and compares intended architecture with reconstructed architecture.

Manually checking for divergence and absence relations in large-scale software systems is time-consuming and cumbersome; hence, most conformance analysis methods in the industry are either semi-automated or fully automated. Software testing can be used to detect the divergence and absence relations. Automation in software testing can be achieved by model-based testing techniques. Model-based testing is a software testing technique in which certain aspects of software system behaviours are defined in a model that is a basis for abstract tests. Concrete tests are derived from these abstract tests and executed against the software system under test.

The software architecture of systems contains vital information for both functional and non-functional requirements of system stakeholders. Software systems are designed and implemented with respect to their architectures in mind. Hence, architecture conformance analysis is important to ensure that the implemented system has not deviated from the intended architecture and thus the requirements.

Software architectures can be used when guiding software development, communicating between stakeholders, deciding on organizational structures and analysing software systems. Therefore, deviating from intended software architecture can create both organizational and system-wise flaws.

1.2 OBJECTIVES AND RESEARCH QUESTIONS

Software architecture conformance analysis is an important field of research and receiving more and more attention since the software systems are getting larger and more complex. Intended and implemented architectures deviate from each other during system development due to various reasons, which later on can lead to failed, overbudget or late delivered software projects. Therefore, different researchers proposed different methods for the software architecture conformance analysis. However, most of the proposed studies are lacking proper case study validation, automation and architecture view definitions. The main objective of this thesis is to develop automated model-driven conformance analysis methods to detect the absence and the divergence relations of software systems using architecture views. Moreover, we want to validate our proposed methods on industrial case studies as well. In this context we have defined the following research questions:

RQ1: What are the current solutions for model-driven architecture-based testing?

To tackle the architecture conformance analysis problem, we must analyse current solutions with respect to multiple perspectives. We systematically reviewed current solutions for implementation details and future research directions in this field.

RQ2: What are the limitations and benefits of code level model-based architecture conformance analysis using architecture viewpoints?

As mentioned earlier, architecture conformance analysis can be executed both at the code level and architecture level. In this thesis, we report on the implementation and evaluation of the proposed code-level architecture conformance analysis implementation.

RQ3: What are the characteristic features of architecture reconstruction methods?

To achieve architecture level-based conformance analysis, the software architecture of code must be derived from code implementation. In this thesis, we implemented a software architecture reconstruction method for architecture level conformance analysis.

RQ4: What are the limitations and benefits of architecture level model-based architecture conformance analysis using architecture viewpoints?

To answer this question, we implemented architecture level architecture conformance analysis. We also evaluated our study systematically using a case study protocol.

1.3 CASE STUDIES

To evaluate our proposed solutions for architecture conformance analysis, we used two different real-life industrial case studies belonging to e-government and e-commerce domains.

1.3.1 E-government Case

E-government is an acronym for the electrical government at which processes between government to governments, citizens, employees and businesses are facilitated through information and communication technology (ICT) solutions. There are five e-government system types which are citizen to government (C2G), government to government (G2G), government to citizen (G2C), government to employee (G2E) and government to business (G2B). E-government solutions aim to provide public services to its end users with efficiency and effectiveness. In this thesis, we adopted a service from G2C domain for our case study.

Certain applications are received from citizens either by themselves or by government clerks, which are then processed in the system to generate a decision output. Detailed architecture specification of this case is presented in chapter 4.

1.3.2 E-commerce Case

E-commerce is an acronym for electrical commerce at which products are traded over ICT solutions. Products in e-commerce can be goods, information or services. E-commerce can be divided into four categories, which are business to business (B2B), business to consumer (B2C), consumer to business (C2B) and consumer to consumer (C2C). B2B solutions aim to solve challenges that are present in electronically trading products or information between businesses. B2C solutions are utilized for businesses to offer their products directly to consumers over the internet. Contrary to B2C solutions, C2B solutions focus on consumers to offer their products to businesses. At last, C2C is like C2B where consumers trade directly with each other without involving businesses. In this thesis, we adopted a software system from B2C domain. In this system, business offers its product to consumers via the software system. Detailed architecture description for this system is provided in the respective chapter.

1.4 RESEARCH METHODOLOGY

In order to find answers to the stated research questions, we have conducted different types of research methodologies including:

- Systematic Literature Review (SLR)
- Case Study Research
- Survey Study
- Domain Analysis
- Design Science Research

Table 1-1 maps the adopted research methods for the identified research questions to the corresponding chapters of this thesis. More details for research methods are given in the next subsections.

Table 1-1: Applied Research Methodologies for the identified research questions

	Ch-2	Ch-3	Ch-4	Ch-5	Ch-6	Ch-7
Systematic Literature Review	RQ1					
Case Study Research			RQ2		RQ4	RQ4
Domain Analysis				RQ3		
Design Science Research		Case Study				
Survey Study			RQ2			

Figure 1-1 presents the research methodologies conducted in respective chapters of this thesis with contributions made per each research methodology. We started with conducting a systematic literature review on model-driven architecture-based testing domain for exploring challenges and current solutions. As a result, we created a generic process model for architecture-based testing and this model provided a basis for our case study researches. Later

we conducted design science research for big data e-government systems and created a business process model for deriving big data e-government systems architecture. Subsequently, we conducted case and survey study research for code level architecture conformance analysis tool implemented in chapter 4. Subsequently, we conducted domain-driven analysis on software architecture reconstruction and created an architecture reconstruction feature model. Lastly, we conducted one more case study research for the architecture level conformance analysis tool implemented in chapter 7. The conformance analysis and architecture reconstruction tools were contributions to this case study research. In the following we describe these steps in more detail.

Figure 1-1: Research methodologies and corresponding contributions

1.4.1 Systematic Literature Review

To answer RQ1 in chapter 2, we conducted a systematic literature review (SLR) or systematic review for short. SLR is a well-defined and rigorous method to identify, evaluate and interpret all relevant studies regarding a particular research question, topic area or phenomenon of interest [13][14]. The goal of an SLR is to give a fair, credible and unbiased evaluation of a research topic using a trustworthy, rigorous and auditable method. There are several reasons for undertaking a systematic literature review including summarizing the existing evidence concerning treatment or technology, identifying any gaps in current research in order to suggest areas for further investigation, providing a framework/background in order to appropriately position new research activities, examining the extent to which empirical evidence supports/contradicts theoretical hypotheses, or assisting in the generation of new hypotheses. Different approaches have been presented in the literature for conducting SLRs in different domains. We followed the complete guidelines for performing SLRs as proposed by Kitchenham et al. [14].

1.4.2 Case Study Research

In order to answer RQ2 and RQ4 in chapters 4, 6 and 7, we conducted a case study research. Our primary goal is to implement and evaluate architecture conformance analysis tools both on the code and the architecture level. We applied our case study researches based on guidelines provided by Runeson et al. [77]. There are five steps defined in this protocol which are: case study design, preparation for data collection, execution on a case, analysis and reporting of results. In order to conduct case study research, we have developed two architecture conformance analysis tools and one architecture reconstruction tool.

1.4.3 Survey Study

In order to answer RQ2 in chapter 4, we adopted a survey study based on survey study research guidelines [67][68][69][70]. Survey studies are quantitative and qualitative methods for gathering information and evaluating results about certain information and behaviours [108]. Generally, surveys are in the form of questionnaires which are series of questions asked by researchers to a set of participants. Survey studies are composed of several activities which are: defining measurable objectives, planning timelines and resources, designing survey studies, setting and validating data collection tools, selecting respondents, supervise data collection, analyse collected data and report results. In our study we conducted a survey study along with a case study research to evaluate our architecture conformance analysis tool in chapter 4.

1.4.4 Domain Driven Analysis

In order to answer RQ3 in chapter 5, we adopted a domain-driven analysis methodology. A well-known process for analysing and modelling the current state of the art of a particular domain is domain analysis. Domain analysis is the process of identifying and capturing domain knowledge about the problem domain with the purpose of making it reusable when creating new systems [41][46][53][84]. The aforementioned domain refers to “an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area”.

1.4.5 Design Science Research

In order to familiarize ourselves with the e-government case study used in chapter 4, design science research applied in chapter 3. There are three iterative cycles defined for design science research methodology: relevance, design and rigor [109]. The relevance cycle focuses on desired enhancements for an environment and eventually results in requirements and evaluating criteria [109]. The design cycle focuses on transforming requirements into design artefacts [109]. The rigor cycle focuses on realizing and evaluating design artefacts defined in the design cycle [109].

1.5 CONTRIBUTIONS

This thesis includes the following contributions:

Generic process model for architecture-based testing

In order to implement architecture conformance analysis tools, we applied a systematic literature review to analyse current solutions proposed by researchers. We filtered 31 primary studies out of 739 studies according to defined study selection criteria, which can be found in chapter 2. We analysed primary studies according to our research questions and synthesized a generic process model for architecture-based testing. We used this generic process model in later stages of work for deriving two different architecture conformance analysis tools.

Reference business process model for deriving big data e-government systems

In this thesis, we provide a business process model for deriving big data e-government systems. Since we adapted industrial big data e-government case study to evaluate our code level architecture conformance analysis tool, we analysed both e-government and big data domains. We synthesized a generic process model for deriving big data e-government architectures. Our process model takes variability models for big data and e-government systems along with e-government reference business process model and reference architecture to create a concrete big data e-government system architecture.

Code-level model-based architecture conformance analysis tool

We designed and implemented a code level architecture conformance analysis tool to check divergences between proposed architecture and resulting implementation. The tool takes the intended architecture as an argument and applies model-to-text-transformation to create runnable test cases on the system's implementation for different architecture viewpoints. Test cases are executed against the code and results are reported as possible divergence relation from intended architecture.

Architecture reconstruction feature model

To derive an architecture reconstruction feature model, we conducted a feature-based domain analysis. We used feature diagrams to model the common and variant features of architecture reconstruction methods. We filtered and analysed 17 studies out of 652 studies according to our study selection criteria to derive our feature diagrams, which can be found in chapter 5.

Architecture reconstruction tool

To implement architecture level conformance analysis tool, we need to reconstruct software architecture from code implementation. We designed and implemented a viewpoint-based architecture reconstruction tool based on our architecture reconstruction feature model.

Architecture level model-based architecture conformance analysis tool

We designed and implemented viewpoint-based architecture level conformance analysis. The tool takes intended and implemented software architectures as an input. Implemented software architecture is reconstructed from the code implementation and intended architecture should be present. Two models are compared to each other for detecting divergence and absence relations and results are reported for each relation and viewpoint.

2

MODEL-DRIVEN ARCHITECTURE BASED TESTING: A SYSTEMATIC LITERATURE REVIEW¹

¹ This chapter is based on following published paper:

Uzun, Burak, and Bedir Tekinerdogan. "Model-driven architecture based testing: A systematic literature review." *Information and Software Technology* 102 (2018): 30-48.

Abstract

Model-driven architecture-based testing (MDABT) adopts architectural models of a system under test and/or its environment to derive test artifacts. In the literature, different MDABT approaches have been provided together with the corresponding lessons results and lessons learned. The overall objective of this chapter is to identify the published concerns for applying MDABT, identify the proposed solutions, and describe the current research directions for MDABT. To this end we have provided a systematic literature review (SLR) that is conducted by a multi-phase study selection process using the published literature in major software engineering journals and conference proceedings. We reviewed 739 papers that are discovered using a well-planned review protocol, and 31 of them were assessed as primary studies related to our research questions. Based on the analysis of the data extraction process, we discuss the primary trends and approaches and present the identified obstacles. This study shows that although a generic process the approaches different in various ways with different goals, modelling abstractions and results. Further, based on the synthesis process in the SLR we can state that the potential of MDABT has not been fully exploited yet.

Keywords: model-based testing, software architecture, systematic review

2.1 INTRODUCTION

Software testing is a process of investigating a software product to identify possible mismatches between expected and present requirements of the system [1][16]. 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 the provided specifications. Moreover, software testing requires executions of test cases which can detect possible bugs, errors and defects.

In general, exhaustive testing is not practical or tractable for most real programs due to the large number of possible inputs and sequences of operations [22]. As a result, selecting the set of test cases which can detect possible flaws of the system is the key challenge in software testing [9][23].

Model based testing (MBT) addresses this challenge by automating the generation and execution of test cases using models based on system requirements and behaviour [15][22][29]. A number of benefits of MBT have been identified in the literature [23]. The main benefits of MBT include improved test coverage, reduced testing time, increased reliability, reusability of tests, increased confidence in the system, less human effort, reduction in cost, increased fault detection, and improved product quality. Altogether MBT is now considered a promising and effective approach for software testing.

MBT relies on models to automate the generation of the test cases and their execution [11]. A model is usually an abstract, partial presentation of the desired behaviour of a system under test (SUT). MBT 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 [11]. A recent trend in MBT is to adopt software architecture models to provide automated support for the test process [17] leading to the notion of model-driven architecture-based testing (MDABT). Software architecture is different from the other design representations since it provides a gross-level representation of the system at the higher abstraction level [5][26].

So far, various MDABT approaches have been introduced but no explicit effort has been undertaken to provide a systematic overview on the existing literature. Hence, the effectiveness of MDABT, the common approaches as a well as the variations, and the guidelines and lessons learned are actually scattered over different studies in the literature. In this chapter we aim to address the following research questions:

- (1) What are the addressed concerns for applying model-driven software architecture-based testing?
- (2) What are the proposed solutions in architecture-based testing?
- (3) What are the existing research directions within architecture-based testing?

To provide answers to these questions we have adopted a systematic literature review (SLR) approach based on Kitchenham's guidelines [13][14]. We reviewed 739 papers that are discovered using a well-planned review protocol, and 31 of them were assessed as primary studies related to our research questions. Based on the analysis of the data extraction process, we discuss the primary trends and approaches and present the identified obstacles.

For researchers, this SLR gives an overview of the reported MDABT together with an understanding of the common process and the lessons learned and guidelines for future studies. Practitioners may benefit from the SLR by identifying the strengths and weaknesses of the approaches as well as the remaining important challenges.

The remainder of the chapter is organized as follows: Section 2.2 of the chapter describes the overall background on architecture-based testing, architecture modelling and systematic literature reviews. Section 2.3 discusses the overall protocol of the adopted in SLR. Section 2.4 presents the results of the adopted SLR protocols. Section 2.5 presents the related work. Finally, section 2.6 presents the conclusion of this study.

2.2 BACKGROUND

2.2.1 Model Based Testing

Historically, models have had a long tradition in software engineering and have been widely used in software projects. The primary reason for modelling is usually defined as a means for communication, analysis or guiding the production process. Models are different in nature and quality. Mellor et al. make a distinction between three kinds of models, depending on their level of precision. A model can be considered as a Sketch, as a Blueprint, or as an Executable. A sketch has the level of precision and is typically an informal diagram that is used for creating ideas in the design process. A blueprint is a design with sufficient detail that can be handed over to a developer for realizing the system according to the blueprint. Unlike a sketch and a

blueprint, an executable model has everything required to produce the desired functionality of a single domain and can be interpreted by model compilers. In model-driven software development the concept of models can be considered as executable models as defined by the above characterization of Mellor et al. [8][18]. This is in contrast to model-based software development in which models are used as blueprints at the most. The language in which models are expressed is defined by meta-models. As such, a model is said to be an instance of a meta-model, or a model conforms to a meta-model. A meta-model itself is a model that conforms to a meta-meta-model, the language for defining meta-models.

Figure 2-1: Process of model based testing

Model-based testing builds on model-driven development in which models are used to automate the testing process. According to Utting et al. [28][29] MBT is a type of testing that utilizes the information in model which is the intended behaviour of the system and its environment. There are several motivations for performing model-based testing such as easy test maintenance, automated test design and enhancing test quality. In Figure 2-1 (adapted from [28]) the 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 analysed by test verdict.

2.2.2 *Software Architecture Modeling*

Software architecture is defined as the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [12]. Every system has a set of interested stakeholders with their specific concerns. Concerns can be functional or non-functional [19]. These stakeholders and their concerns are considered as the architectural drivers that shape the architecture [5][26]. A stakeholder is defined as an individual, team, or organization with interests in, or concerns relative to, a system. Each of the stakeholders' concerns impacts the early design decisions that the architect makes. A common practice is to model different architectural views for describing the architecture according to the stakeholders' concerns [6]. An architectural view is a representation of a set of system elements and relations associated with them to support a concern. Having multiple views helps to separate the concerns and as such support the modelling, understanding, communication and analysis of the software architecture for different stakeholders.

Architectural views conform to viewpoints that represent the conventions for constructing and using a view. An example viewpoint that is often used is the decomposition viewpoint that defines the specific guidelines and notations for representing the overall decomposition of the system. Another example is the deployment viewpoint that defines its own guidelines and modelling notations for mapping software modules and components to non-software elements such as hardware nodes. In the literature, initially a fixed set of viewpoints have been proposed to document the architecture. Because of the different concerns that need to be addressed for different systems, the current trend recognizes that the set of views should not be fixed but multiple viewpoints might be introduced instead. The ISO/IEC 42010 standard Recommended Practice for Architectural Description [12] indicates in an abstract sense that an architecture description consists of a set of views, each of which conforms to a viewpoint realizing the various concerns of the stakeholders. The Views and Beyond (V&B) approach as proposed by Clements et al. is another multi-view approach [5] that proposes the notion of architectural style similar to the notion of architectural viewpoint.

2.3 RESEARCH METHOD

In this section, we describe the SLR research methodology that we have adopted. First, we describe the adopted review protocol in Section 2.3.1 and then proceed with the execution of the steps of the protocol in the subsequent subsections.

2.3.1 Review Protocol

The adopted review protocol based on [14] is shown in Figure 2-2. According to the guidelines it includes the phases including planning the review, conducting the review and documenting the review.

Figure 2-2: The adopted review protocol

Planning the review results in the definition of the protocol. In essence we have defined this by studying the guidelines of Kitchenham et al. The conducting of review consists of five separate steps. First, we specified our research questions based on the objectives of this systematic review. Based on this we followed the selection of the studies. For this we defined the search scope and the search strategy. The search scope defines the time span and the venues that we looked at. In the search strategy, we devised the search strings that were formed after performing deductive pilot searches. A good search string brings the appropriate search results that will come to a successful conclusion in terms of sensitivity and precision rates. Once the search strategy was defined, we specified the study selection criteria that are used to determine which studies are included in, or excluded from, the systematic review. The selection criteria were piloted on a number of primary studies. We screened the primary studies at all phases on the basis of inclusion and exclusion criteria. Also, peer reviews were performed by the authors throughout the study selection process. The process followed with quality assessment in which the primary studies that resulted from the search process were screened based on quality assessment checklists and procedures. Once the final set of preliminary studies were defined the data extraction strategy was developed. For this we developed a data extraction form that was defined after a pilot study. In the final step the data synthesis process took place in which we have presented the extracted data and associated results. The review results were documented in the last phase, Document Review, of the overall process.

2.3.2 *Research Questions*

The fundamental step of the SLR is the identification of specific and valid research questions. The research questions drive the entire review process providing the basis for both the decision of the selection of primary studies, and the decision on what data must be extracted and how the data is synthesized to answer the questions. We have identified the following research questions:

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

RQ2: What are the proposed solutions in model-driven architecture-based testing?

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

2.3.3 *Search Strategy*

The search process begins by first defining a search strategy. For this the basic scope of the search strategy must be determined, which include the specific sources that will be searched and the search strings that will be used for automated searches and the sources that will be searched manually. Our search scope includes two attributes which are publication date range and publication platforms. We have selected studies between 2000 and April 2017 for our publication date range attribute. The start date of 2000 has been selected because the first paper having strong foundations on the topic was published at this date. For the publication platforms we included IEEE Xplore, ACM Digital Library, Wiley, Science Direct, Springer, and ISI Web of Knowledge. To search a database, we used both automated search and manual search. Automated 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.

We have identified a search string for each publication platform listed in our search scope for retrieving the relevant studies. Each platform has different features, attributes to query for primary studies we are interested in. Hence, we have defined queries for each platform using each the platform search language. The created queries define the intersection of the papers that has software architecture testing and model-based testing in the publications title or abstract. The search strings for each platform are described in APPENDIX-A Search Strings. Table 2-1 presents the results of the overall search process. The first column provides the searched venues. In the second column of the table, 739 studies are retrieved after executing the search strings in the platforms. The third and fourth columns show the filtered studies after applying the study selection criteria as explained in the following subsection. In the last step of the process 31 studies have been identified as primary studies to be used for detailed data analysis and synthesis.

Table 2-1: 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	64	47	12
ACM Digital Library	155	13	5
Wiley Interscience	67	0	0
Science Direct	57	15	5
Springer	155	10	6
ISI Web of Knowledge	76	35	3
Total	739	120	31

2.3.4 Study Selection Criteria

The search query strings have been defined so that we will not miss any related studies. On the other hand, study selection criteria have been defined so that irrelevant studies are excluded. We have applied the following exclusions 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 do not provide a process model for architecture-based testing

2.3.5 Study Quality Assessment

Once we have identified the primary studies, we have also assessed the quality of each study. For this we had to decide on the criteria against which quality will be assessed. Further we had to establish the procedure for applying the criteria. The criteria have been expressed as a checklist as shown in Table 2-2.

The procedure for applying the quality criteria is specified in a way that aims, as far as is possible, to ensure the reliability of the assessment. We have applied the following procedure:

1. For each paper, a reviewer was nominated randomly as data extractor/quality assessor or data checker.
2. The data extractor/quality assessor read the paper and completed a form.
3. The checker read the paper and checked the form
4. Disagreements were resolved by discussion among the researchers.

We utilized a three-point scale including yes (1), somewhat (0.5) and no (0). The results for each primary study filtered by study selection criteria is presented in APPENDIX-B List of Primary Studies.

Table 2-2: Adopted Quality Checklist

No	Question
Q1	Are the aims of the study 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?

2.3.6 Data Extraction

Data extraction is performed by reading all the 31 selected primary studies for answering each research question. Furthermore, a data extraction form is designed for retrieving all the information to answer the research questions and all the attributes for study quality assessment criteria. The data extraction form contains the 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. The extraction of data purpose columns is inserted in to the form as well by study description and evaluation parts which can be seen in APPENDIX-D DATA EXTRACTION FORM.

2.3.7 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. In this study, we implemented both qualitative and quantitative synthesis on the extracted. 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.

2.4 RESULTS

2.4.1 Overview of the Reviewed Studies

This section of the study presents the publication year distribution and the publication platforms of the 31 selected primary studies. Figure 2-3 shows the publication year distribution of the selected primary studies.

Figure 2-3: Year-wise distribution of primary studies

Table 2-3 presents the publication sources and channels of the selected studies together with the publication type and distribution of studies over the attributes. From the table we can infer that selected primary studies are published in various reputable publication sources such as IEEE, ScienceDirect, ACM and Springer.

Table 2-3: Distribution of the studies over Publication Channel

Publication Channel	Publication Source	Type	Number of Studies
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

Fundamental Approaches to Software Engineering	Springer	Article	1
Applying Formal Methods: Testing, Performance, and M/E-Commerce	Springer	Conference	1
Electronic Notes in Theoretical Computer Science	ScienceDirect	Article	1
Proceeding ROSATEA '06 Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis	ACM	Workshop	1
JSS, Special Edition on Architecting Dependable Systems	ScienceDirect	Article	1
Software Architecture	WebOfKnowledge	Chapter	1
2016 IEEE International Conference on Software Testing, Verification and Validation	IEEE	Conference	1
The 7th International Conference on Ambient Systems, Networks and Technologies	ScienceDirect	Conference	1
Information Technology: New Generations (ITNG), 2010 Seventh International Conference	IEEE	Conference	1
Information and Software Technology Volume 55, Issue 7	ScienceDirect	Chapter	1
Journal of Systems and Software Volume 91	ScienceDirect	Article	1
ICSE '01 Proceedings of the 23rd International Conference on Software Engineering	ACM	Conference	1
AST '07 Proceedings of the Second International Workshop on Automation of Software Test	ACM	Workshop	1
ISEC '08 Proceedings of the 1st India software engineering conference	ACM	Conference	1
Software Technologies (ICSOF), 2015 10th International Joint Conference	IEEE	Conference	1
2010 Forum on Specification & Design Languages (FDL 2010)	IEEE	Conference	1
Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference	IEEE	Conference	1
Applied Machine Intelligence and Informatics (SAMI), 2013 IEEE 11th International Symposium	IEEE	Conference	1
Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference	IEEE	Conference	1
Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium	IEEE	Conference	1

Service Science and Innovation (ICSSI), 2013 Fifth International Conference	IEEE	Conference	1
Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops	IEEE	Conference	1
Lecture Notes in Computer Science	Springer	Chapter	1
Communications in Computer and Information Science	Springer	Chapter	1
Software Testing, Verification and Validation Workshops (ICSTW)	WebOfKnowledge	Workshop	1
Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference	WebOfKnowledge	Conference	1
Lecture Notes in Computer Science	Springer	Chapter	1
IFIP Advances in Information and Communication Technology	Springer	Conference	1

2.4.2 Research Methods

In essence, every primary study will have its own research method together with the adopted empirical validations. Table 2-4 presents the adopted types of the research method that is applied in the selected primary studies.

Table 2-4: Distribution of studies over Research Method

Research Method	Studies	Number	Percent
Case Study	A, C, D, E, F, H, M, K, O, R, W, Y, DD, EE	14	45
Experiment	B, L, U	3	10
Small Example	G, J, N, P, Q, S, T, V, X, Z, AA, BB, CC	13	42
None	I	1	3

Three types of research methods are identified during the review process which are case study, experiment and small example. From the table, we can conclude that case study research method is the most frequently adopted approach in the selected studies. On the other hand, one of the studies does not validate the proposed approach via any research method.

2.4.3 Methodological Quality

This section provides the quality of the selected primary studies based on the attributes of relevance, quality of reporting, rigor and credibility [14]. The values for the quality attributes are derived from the quality checklist which was given in the previous section. The first three questions of the quality checklist relate to the reporting quality attribute, the fourth, fifth and sixth questions are used for the calculation of rigor quality, the seventh and eighth questions are the assessment questions for credibility quality, and finally the last two questions are used for relevance quality. In APPENDIX-C Study Quality Assessment the result of the quality checklist is presented. Table 2-5 shows the qualities for studies in each category.

Table 2-5a shows the reporting quality of the studies according to the first three questions of the quality checklist. It can be seen that almost all of the primary studies have the highest score while one of the primary studies is close to the highest score. Rigor quality of the study refers to the trustiness of findings of the study. Table 2-5b shows that 61% of the studies have the highest score in terms of rigor quality. Moreover, 32 % of the studies have been assessed as very good. However, 6% of the studies ranked as good rigor quality. Another quality measure is the relevance quality of the primary studies.

Table 2-5c shows relevance quality scores calculated from ninth and tenth question of the quality checklist. It is observed that, 39% of the studies is directly and %42 of the studies is mostly relevant to MDABT, where %19 of the studies half relevant. The credibility quality of the studies is calculated by using the seventh and eighth questions. Table 2-5d presents the credibility quality score and distribution of the studies. It can be seen that 87% of the studies calculated as 1 point. Remaining 13% 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.

The summary of the overall methodological quality scores of selected primary studies is given in Table 2-5e. Total quality is calculated by adding up all the quality attributes, which are reporting, relevance, rigor and credibility. The values for the quality attributes are derived from the quality checklist as shown in Table 2-2 (including 10 questions). For example, for the reporting quality attribute three questions will be used for each of which a value of 0, 0.5 or 1 can be given. Hence in total, reporting quality can have a value of 3.0 as a maximum. The similar procedure has been applied to the other 3 quality attributes. The X-scale of the figures in Table 2-5 represent the frequency for the indicated values. For example, for the reporting quality 1 study got in total 2.5 points while the other 30 studies got a value of 3. For the overall quality we consider 8.5-9 as high quality, 7.5-8 very good quality and 7 as good quality. It can be seen that 55% studies have high overall quality. Further, 35% of the studies have very good overall quality and 10% of the studies have good overall quality.

Table 2-5: Overall quality of the primary studies

2.4.4 Systems Investigated

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

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

From the analysis of the primary studies, we could derive that in essence two basic concerns are addressed in the application of model-driven software architecture-based testing. These include checking the internal architecture consistency, and code to architecture conformance. The distribution of the primary studies over these concerns are shown in Figure 2-4. It appears thus that the majority of the motivation for MDABT is the conformance checking of the code with the architecture. Hereby, typically the architectural relations and constraints are derived and test cases are generated accordingly. These test cases are then applied on the real code to identify the possible differences. It should be noted that architecture conformance analysis is one of the key approaches for analysing the so-called architecture drift problem [20][27]. When

discussing the results of RQ2 we will elaborate on the addressed concern per study and the corresponding details.

Figure 2-4: Addressed concern distribution of primary studies

Table 2-6: Addressed Concerns and Studies

Addressed Concerns	Studies
Code to architecture conformity	A, C, D, E, G, H, I, M, N, K, L, O, P, Q, R, S, T, U, V, W, X, Y, Z, AA, BB, CC, DD
Internal Architectural Consistency Checking	B, F, J, EE

RQ2: What are the proposed solutions for model-driven architecture-based testing?

Architecture based testing (ABT) is a testing approach exploiting architectural models for testing the software system. The generic process for ABT is shown in Figure 2-5. This process model or "pattern" is extracted from the thoroughly analysed studies that is involved in this literature review. In the following we will first explain the generic process and then discuss each of the approaches that are instantiations of this generic process. Based on the figure 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 modelling approach. The provided model 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 in a well-defined format.

- 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 cases 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.

Figure 2-5: Generic Process for Architecture-Based Testing

The process in Figure 2-5 is based on a general model-based testing approach. In this study we do not focus on models in general but focus on architectural models for supporting model-based testing. In principle, different ABT approaches can be derived from the same generic process. In the following we discuss the proposed solutions that we could derive from each primary study. For this, we adopt and instantiate the reference process model as discussed in section 2. Table 2-7 summarizes the results of the above findings. In the following we describe each study in detail.

Table 2-7: Overview of the identified approaches

Study	Criteria	Architecture Model	Test Model Construction	Test Model	Abstract Test Case Generation	Abstract Test Suite	Concrete Test Case Generation	Concrete Test Suite	Test Case Execution	Test Oracle
A	CC	CHAM	M	LTS	M	ALTS	M	ALTS Paths	M	M
B	CC	Wright ADL	A	BG	NA	NA	M	BG Paths	A	A
C	CC	FSP Model	A	LTS	M	ALTS	M	ALTS Paths	M	M
D	CC	FSP Model	A	LTS	M	ALTS	M	ALTS Paths	A	A
E	TPM	UML	A	Promela	M	NA	M	Promela Paths	M	M
F	TPM	UML	A	IOLTS	A	Test Graph	A	Tree and Tabular Combined Notation	A	A
G	TPM	GoalML	A	Annotated Code	NA	NA	M	Annotated Code	A	A
H	CC	FSP Model	A	LTS	A	ALTS	A	ALTS Paths	A	A
I	CC	AADL	A	Uppaal	NA	NA	A	Uppaal Paths	A	A
M	CC	UML	A	Test Graph	A	Test Graph Paths	A	Cucumber Scenarios	A	A
N	CC	UML	A	U2TP	NA	NA	A	JUnit	A	A
J	CC	Acme ADL	A	PetriNet	A	HPrtNS	A	Petri Net Paths	A	A
K	CC	BPEL	A	ECFG	A	CFG	A	CFG Paths	A	A
L	CC	UML	A	eDelta Models	A	xUnit	A	JUnit	A	A

O	CC	LTS	A	ALTS	NA	NA	M	ALTS Paths	A	A
P	CC	UML	A	xUnit	NA	NA	A	JUnit	A	A
Q	CC	Wright ADL	M	ACFG	A	ACDG	A	ACDG Paths	A	A
R	CC	UML	A	PetriNet	NA	NA	A	Petri Net Paths	A	A
S	CC	UML	A	UTP	NA	NA	A	Cunit	A	A
T	CC	MSC	A	MSC	NA	NA	A	MSC Test Cases	M	M
U	CC	UML	A	UTP	NA	NA	A	CPPUnit	A	A
V	CC	CDM	A	DG	NA	NA	A	TUM	A	A
W	CC	UML	A	UTP	NA	NA	A	TTCN-3	A	A
X	CC	EFSM	A	EFSM -SeTM	NA	NA	A	EFSM -SeTM Paths	A	A
Y	CC	UML	A	Activity Diagram	NA	NA	A	Activity Diagram Paths	A	A
Z	CC	UML	A	PetriNet	NA	NA	A	Petri Net Paths	A	A
AA	CC	UML	A	UTP	A	xUnit	A	JUnit	A	A
BB	CC	UML	A	VDM-SL	NA	NA	A	VDM	A	A
CC	CC	WSDL-S	A	ESG	NA	NA	A	ESG Paths	A	A
DD	CC	PrT	A	PetriNet	A	Transition Tree	A	JUnit	A	A
EE	CC	UML	A	DERCS	NA	NA	A	DERCS	A	A

CC: Coverage Criteria; TPM: Test Purpose Matching; A: Automatic; M: Manual; NA: Not Applicable

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. The testing is performed for checking the conformance of the implemented system with the specified SA. The main motivation of this paper is to use SA specifications for integration testing of the implemented system. Test model is generated using CHAM specifications and coverage criteria which is a transition graph called Labelled Transition System (LTS). From this graph, abstract labelled transition systems (ALTS) are obtained simply by applying obs function. Obs functions are functions that excludes 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 many concrete test cases. Test cases are generated manually by software architect. Test execution and test analysis are handled manually by software architect.

Study B

In this study the testing is done at the architecture design level to test functional properties of SA. The key concern is to check the internal architecture consistency. Hereby 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 Behaviour 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.

Study C&D&H

The key concern of these studies is architecture conformance checking. These studies present the ongoing work of previous authors in A by replacing their SA specification of CHAM model with a Finite State Process (FSP) model. The provided reason to use FSP instead of CHAM model is that FSP algebra is easier to map to LTS graph. Hereby, the work of the software architect is explained to generate test cases and execute them manually. UML Stereotyped Sequence Diagrams are used for filling the abstraction gap between the SA and implementation. For each architecture level sequence diagram, the 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.

Study E

This study uses the same methodology explained in study D. The key concern is the conformance checking of the code to the architecture. A model checking tool is presented for software architecture using Checking Architectural Model Consistency (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, linear logical temporal (LTL) Formulae and Buchi Automata. The results are used by test generator engine to create test cases automatically at the SA level. The test execution and test analysis are handled manually.

Study F

The main motivation of this study is to validate architectural units using object-oriented models. The 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 Language of Temporal Ordering Specifications (LOTOS). Basic LOTOS model is combined with test purposes to generate Input/Output Labelled Transition Systems (IOLTS) model where

test cases are generated using Test Generation using Verification techniques (TGV) tool. Test cases generation and test result analysis is automated in this study

Study G

In this study the addressed concern is again 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. Hereby, system behaviour is implemented by a system level plan that consists of sequence of goals each describing the interactions between the components. In addition, component level plan shows the sequence that component must achieve in that scenario. The system level plan and component level plan together provide the specifications for SA. 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.

Study I

The addressed concern of the study is code to architecture conformity. The approach uses the Architecture Analysis and Design Language (AADL) to specify SA which is then 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. 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.

Study J

The main motivation of this paper is to use SA in model-based testing to detect defects earlier in software lifecycle. Testing is performed at architectural level. The study presents the use of Hierarchical Predicate Transitions Nets (HPrTNs) model to explain the behaviours 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. Test execution and analysis is performed automatically.

Study K

The addressed concern of this study is code to architecture conformity. 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 Business Process Execution Language (BPEL). In the SA specifications architecture topology and message exchange knowledge is 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 analysed automatically.

Study L

The addressed concern of this study is code to architecture conformity. This study presents specification of SA is using UML state, sequence and component diagrams. UML state machines are used for representing component behaviours 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. 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. Test execution and analysis is automatically handled.

Study M

The study addresses the concern of conforming given architecture to implementation. In this study, a tool for generating behaviour driven tests using behavioural UML diagrams are proposed. Software architecture is defined in terms of different behavioural UML diagrams such as use case, interaction, activity diagrams and using full graph coverage as coverage criteria tool creates so called model test graph. Each path of the test graph represents a concrete behavioural test case scenario which are executed automatically via another tool called Cucumber. Testers then writes mappings for each generated Cucumber scenarios and execute tests.

Study N

This study concerns the correctness of the code with respect to the architecture. An approach is proposed for model driven automated tests generation using UML sequence diagrams as software architecture. UML sequence diagrams together with coverage criteria is used for creating test models as U2TP sequence diagrams. The generated platform specific models are then transformed into test cases which are then executed and analysed on small example for validating the proposed approach.

Study O

The study aims to check the conformance of the code with respect to the architecture. Hereby an LTS is defined as an architecture model and coverage criteria as test criteria for creating a test model ALTS. There is no abstract test model for this approach each path of ALTS model is transformed in to concrete test case where executed and analysed automatically.

Study P

This study aims to check the code to architecture conformity. UML sequence diagrams are used to represent SA where the proposed approach automatically creates xUnit model as a test model for generating concrete test cases based on xUnit such as junit, cunit, cppunit etc. For demonstration purposes, the approach generates junit test cases which can be automatically executed and results are analysed.

Study Q

This study introduces a new algorithm for slicing of architecture and checking the conformance of architecture to code. The authors represent SA using Wright ADL. Test model of Architecture Control Flow Graph (ACFG) is manually created using coverage criteria. This

model is automatically transformed into abstract test model of Architecture Control Dependence Graph where each path of this abstract test model is then mapped to concrete test cases automatically. Generated test cases can be executed and the results analysed automatically.

Study R

In this study UML models are used to generate test cases to ensure the conformity of distributed systems architecture with respect to each other based on scenarios modelled in sequence diagrams. SA model is transformed into Timed Event-Driven Colored Petri Nets (TEDCPN) using coverage criteria for all models. Each path of the test model is then transformed into concrete test cases which are executed and analysed automatically.

Study S

This study S addresses aims to check the conformity of code to the given architecture to catch the defects at earlier lifecycle of development in real-time critical embedded systems. The main motivation of the paper is to elevate a newly proposed testing architecture for embedded systems. The SA is presented using UML state diagrams and coverage criteria is selected as test model creation criteria. Test model of type UML testing profile (UTP) models are generated. Concrete test cases are generated using UTP models in the form of cUnit. Generated test cases can be automatically executed and analysed.

Study T

This study S addresses aims to check the conformity of code to the given architecture. This study greatly differs from the others by the model that is used for SA model and test model which are basically the same model type which is message sequence charts (MSC). MSC is a specialized type of sequence diagrams. Coverage criteria is used for generating MSC test model where each path along the chart represents a concrete test case. Concrete test cases are manually executed and analysed.

Study U

In this study test cases are generated for checking the conformance of architecture to code for critical properties. In this study, SA is represented using UML models. UTP test model is generated using SA model along with coverage criteria. UTP test models are then translated into CPPUnit concrete test cases which can be automatically executed and analysed.

Study V

This study represents the dependencies between components and checks the dependency relations at the implementation level for conformance to architecture. SA is represented using Component Dependency Model (CDM) which is then transformed into dependency graph (DG) using coverage criteria. Time Usage Models (TUM) are generated from DG models as concrete test cases.

Study W

In this study test cases are generated for conformance checking of code level implementation to architecture models for software-intensive mission-critical systems. SA is represented using UML models and along with coverage criteria UTP test models are generated. Test models are

transformed into Testing and Test Control Notation (TTCN-3) test cases which are automatically executed and analysed.

Study X

This study aims to check the conformance of web service components and the given architecture. SA is represented using Extended Finite State Machine (EFSM). Test model of EFSM-Sequence Test Model (EFSM-SeTM) is generated using coverage criteria. EFSM-SeTM is a model in which EFSM is extended by sequence diagram.

Study Y

In this study the authors check for the conformance of the camera software application for mobile phones with respect to given UML architecture of the software. SA is represented using UML models. Along with coverage criteria activity diagram test model is generated. Each path of the diagram corresponds to a concrete test case to be executed. Test execution and result analysis is performed automatically.

Study Z

In this study, the authors generate tests cases in the form of petri net paths to check the conformance of implemented system with respect to architecture. SA is represented using UML class, collaboration and state diagrams. Along with the coverage criteria these architecture models are transformed into Concurrent Object-Oriented Petri Nets (CO-OPN) test models. Each path on CO-OPN model corresponds to a concrete test case to be executed on system under test. Test execution and result analysis are performed automatically in the proposed approach.

Study AA

In study AA, the authors generate test cases to conform the implemented system with respect to UML models of software. SA is represented using UML sequence diagrams. Architecture model and coverage criteria as the test criteria generates UTP test model. Abstract test cases of xUnit based models are generated from UTP test models. JUnit test cases are automatically derived from xUnit models in which test execution and result analysis can be performed automatically.

Study BB

This study presents an approach to check the conformity between provided architecture models and implemented systems of distributed systems. SA is represented using UML sequence diagrams. Test model of type Vienna Development Method Specification Language (VDM-SL) is generated using architecture model and coverage criteria. Concrete test cases are generated using the specifications provided in the test model. Test case execution and result analysis is performed automatically.

Study CC

In this study test cases are generated for services deployed in cloud computing environment with provided service as a software platform. The key concern is again conformance checking. Web service architectures are represented using Web Service Semantics (WSDL-S) which is an extension of Web Service Description Language (WSDL). Event Sequence Graphs (ESG)

are generated using WSDL-S model and coverage criteria. Each path on ESG is transformed into concrete test cases which can be executed automatically. Test case execution results are then automatically derived.

Study DD

In this study test cases are generated to check the conformance of implementation and given models. SA is represented using Predicate/Transition (PrT) nets. Proposed approach automatically generates test models based on coloured Petri Nets using coverage criteria and SA models. Abstract test case in form of Transition Trees are generated which are then transformed into JUnit test cases. JUnit test cases are executed automatically and results are analysed.

Study EE

In this study EE, an approach is presented to generate test cases to validate and verify the correctness of designed embedded system architecture. SA is represented using behavioural UML diagrams which are: use case models, interaction diagrams, sequence diagrams, collaboration diagrams and state diagrams. Distributed Embedded Real-time Compact Specification models are generated as test models with coverage criteria and provided architecture models. Concrete test cases are generated from the test models are same of type and can be executed automatically. Results of the test case executions are automatically analysed.

2.4.4.1 Summary

So far, we have shown the different MDABT approaches which we explained as instantiations of the generic reference model as shown in Figure 2-5. On one hand we can state that all these approaches share the similar approach since these could all be discussed based on the reference process. On the other hand, each of these approaches were different for the different parts of the reference process. The reference process helped not only to understand each of these processes but also was a useful instrument to compare the proposed approaches. Although MDABT can be applied to any domain, some domains had a clear interest including embedded systems, real-time systems, cloud services, distributed systems, mission-critical systems and large-scale variant-rich systems.

Obviously, the SA specification appeared to be an important part of our process model and different approaches applied different modelling approaches. Figure 2-6 shows the distribution of the various SA specifications over the approaches from the primary studies. As it can be observed from the figure, UML is the most preferred approach in the literature. FSP model follows UML as the second most frequently used approach. The main reason that UML is the most preferred approach could be attributed due to the fact that it is a popular approach in system modelling. Hence, the learning curve for UML is lower and the willingness to adopt UML models for testing is probably higher.

Figure 2-6: The Identified Architecture Description Approaches and the Frequency Over the Different Studies

Based on the reference process, the second differing aspect in the approaches is the adopted testing criteria in which most of the studies adopts coverage criteria for testing purposes. A small proportion of the studies use test purposes that are exclusively defined for the approaches presented. Figure 2-7 shows the distribution of testing criteria over the selected studies. Coverage criteria applies for graph-based test models in which all the paths of the graph denote a test case. Based on this we can conclude that most of the studies adapted graph-based approaches and test criteria.

Figure 2-7: The Identified Test Criteria

Another important aspect of the process model is test models, which are used to generate test cases. Figure 2-8 shows the test model distribution over the various approaches. LTS, PetriNet, and UTP appeared to be the most preferred test models in the literature. Most of the studies use graph-based or automata-based test models where test cases are generated from extracted paths from nodes to other nodes. In one of the studies SA is specified using a domain specific language GoalML and no test models are used. Hereby, the tested properties are annotated in the implementation itself and during the execution of the system the assertions are emitted.

Figure 2-8: The Identified Test Model Representations

Figure 2-9 shows the distribution of the identified test model construction approaches. Most of the studies apply automated generation of test models, but in two of the studies this process is executed manually. Manual test model generation is a both effort and resource intensive process which can be manageable for small scale applications. However, for large-scale applications this does not appear to scale well.

Figure 2-9: The Identified Test Model Construction

Another important aspect of the reference process model is the abstract and concrete test suite generations. Figure 2-10 shows the distribution of the approaches for abstract test model types for the given studies. From the figure we can see that most of the studies do not have any abstract test suites as represented by the label “NA” (not applicable). Except two of the studies, all the studies use graph-based notation (mostly ALTS) for abstract test suites. Hence, we can infer that graph-based solutions are popular among the primary studies.

Figure 2-10: The Identified Abstract Test Suite

Figure 2-11 shows the distribution of the abstract test case generation approaches. The majority of the studies (19) did not provided an abstract test case generation approach as indicated by the label “NA”. For the ones that did apply it, we can see that an automated generation process is mostly adopted. As in the test model creation, manual construction of abstract test case generation is usually not preferred since it is effort and resource consuming.

Figure 2-11: The Identified Abstract Test Case Generation Type

Figure 2-12 shows the distribution of the adoption of concrete test suite types. Here, we can we can see that ALTS and JUunit test cases are the most frequently used approaches. From the overall perspective, graph paths based notations for concrete test cases are dominating the studies but in later studies we see that xUnit based concrete test suites are preferred.

Figure 2-12: The Identified Concrete Test Suite

Figure 2-13 shows the distribution of the concrete test suite generation methods. Here we can observe that automated approaches are most frequently used to create concrete test suites from abstract test suites. However, there are still manual approaches present for the concrete test case generation.

Figure 2-13: The Identified Concrete Test Case Generation Type

In Figure 2-14, the distribution of test case execution over the studies can be seen. Most of the studies again use automated test case generation whereby testing is carried out at the SA level rather than code level. In manual test case development, we could observe both SA level and code level testing. Manual test case execution is simply executing the concrete test case on the code or SA, which will take lots of effort for large test suites.

Figure 2-14: The Identified Test Case Execution

Figure 2-15 shows the distribution of test analysis types, which considers the analysis of the executed tests. Most of the studies used automated analysis where the result of failing or passing is automatically determined. In manual cases reporting of each test case execution will be handled manually and for large test suites it is unfeasible for most cases.

Figure 2-15: The Identified Test Analysis Type

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

After we have characterized each model-driven architecture-based testing approach we will now discuss the existing research directions. The research directions are on the hand derived from the identified concerns and solution directions for applying MDABT, on the other hand these have been sometimes provided independently. For RQ1 we have identified the two key concerns including (1) code to architecture conformity and (2) internal architecture consistency checking. For RQ2 we have identified 31 approaches each of which had different properties. Table 2-8 presents the overview of the identified research directions. In the following we describe the indicated research directions in the primary studies.

Table 2-8: Overview of the identified obstacles that has been discussed in the related primary studies

Research Direction	Discussed in primary study
Need for executable models for representing architecture	A, B, C, D, E, F, H, J, L, P, Q, R, S, T, U, V, W, Y, Z, BB, CC, DD, EE
Abstraction difference between architecture and code	A, B, C, D, E, F, G, H, J, O, S, AA
State explosion problem	A, C, D, F, H, M, K, L, O, Q, S, V, X, Y, Z, DD
Architecture views	C, D, E, O
Automating the process by tools and cost-benefit analysis	A, B, C, D, F, G, , H, I, M, N, L, O, P, Q, R, S, T, U, V, W, X, Y, Z, AA, BB, CC, DD, EE
Applying MDABT on complex real systems	A, B, C, D, F, G, H, I, M, N, J,K L, O, R, W, X, AA

- 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. This has been indicated for both the concerns of code to architecture conformity and internal architecture consistency checking (RQ1).

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 solutions have been proposed (RQ2) in the studies using different representation techniques for SA specifications. For supporting model-driven architecture-based testing it is important that the architecture models are at the level of executable models. This typically requires that the models are expressed in some domain specific language.

- Abstraction difference between architecture and code

This research direction was mainly identified in the studies that focus on conformance checking of the code with respect to the architecture (RQ1). As stated before, these were the majority of the studies. 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 the SA level appeared to be simpler than creating tests for code level that are derived from SA specifications. Based on the provided solutions (RQ2) it appeared indeed that a challenge was to fill the gap between the SA and the implemented system. Although several solutions have been provided such as architectural styles, this problem has been identified in many studies as one of the key challenges.

- 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, and has been addressed in both the studies that concern conformance checking and internal architecture consistency (RQ1). Although the adoption of the presented model-driven solutions (RQ2) 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 studies have used model checking integrated with SA based testing where the results from model checking are used for generating test cases. Yet, this challenge of state explosion problem seems to be recurring and there seem still to be many challenges for finding more effective criteria in the test generation process.

- Architecture views

As discussed in the background, architecture needs to be modelled using multiple architecture views to represent the various stakeholder concerns [5][19]. This is again applicable both for conformance checking and internal architecture consistency analysis (RQ1). For conformance

checking it appeared to be important to identify the proper architecture views on which the code would be checked. For internal architecture consistency it seemed to be important to check the consistency among the views. The presented solution approaches (RQ2), however, largely 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. These approaches however do not adopt the current notion of viewpoints that adopt different perspectives on the architecture based on different stakeholders. Integrating architecture views, as such, will be a necessary enhancement to the existing architecture-based testing approaches.

- Need for automating the process and its cost-benefit analysis

Both conformance checking and internal architecture consistency processes become less tractable and require automation for large scale systems (RQ1). While analysing the provided solutions (RQ2) we could identify that the MDABT process as we have presented is largely followed. Further we can conclude that automation has been provided for these steps. Yet, no single study has presented an approach that completely automates the overall process. This has also been identified and discussed by the authors of these studies. Further automation of the overall process seems to be a recurring future work. Another important issue that seems to be present in the current literature is the lack of justification and the cost-benefit analysis of the MDABT. Some progress can be observed in this context, however none of the studies have adopted a systematic approach to evaluate and justify the benefits of the automation process. This has been also explicitly addressed as a challenge by many authors.

- Need for applying MDABT on complex real systems

It is very important to use empirical validations while assessing your methodology. In most of the studies the proposed solution approaches (RQ2) 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 a need for applying MDABT on more complex industry cases to justify the effectiveness of these approaches. This applies for both the concerns of conformance checking and internal architecture consistency (RQ1).

2.4.5 *Threats to Validity*

Every SLR is susceptible to various validity threats, which have to be addressed for the results to be acceptable. In the following we describe the possible validity threats and discuss the adopted risk mitigation strategies.

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 rather than negative results of their studies. It is proposed in [13] to perform research protocol with research question to manage publication bias as we have done in this study. Later in the study we identified search scope, search method and constructed search string for different publication platforms to query on the

MDABT area. Incompleteness caused by search keywords is another danger for legitimacy of SLR. To mitigate this risk, we have constructed keyword list in a repeated manner by pre-performing sample searches in the domain. The construction of the keyword list was handled manually by adding new keywords if we were not able to retrieve related studies to our field. As stated before, the constructed search strings for different platforms are located in Appendix-A.

Even though we have carefully designed the search queries there is still a threat that 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 data-base 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.

2.5 RELATED WORK

Souza et al. [24] provide a mapping study to characterize the state of the art on how software architecture and software testing have been explored in a complementary way. The systematic mapping study identified 27 primary studies that have explored the use of information related to software architecture to support the testing activity. The identified approaches have covered the integration, unit, and regression phases, and also considered functional and structural techniques. The basic conclusion of the paper is that there are still a range of open research topics to be explored, in particular, tool support. In contrast to this mapping study we have provided a systematic literature review and have analysed the primary studies in detail. We have also provided an explicit and detailed list of the addressed concerns, the proposed solutions and the research directions.

Rafi et al. [23] provide a systematic literature review and survey on benefits and limitations of automated software testing. The SLR selects 25 studies and conduct data extraction for both benefits and limitations of automated software testing. The data extraction results are on its turn used to conduct a practitioners' survey for determining whether the benefits and limitations are of relevance for practitioners. The survey has been performed with 115 practitioners. The authors conclude that the main benefits of test automation are reusability, repeatability, effort saved in test executions, and improvement of the test coverage. Our systematic literature review is also in the scope of automated software testing, but we have in focused directly on architecture-driven model-based testing. For this we have provided more dedicated insight which is useful for paving the way for further research in this domain.

Neto et al. [21] perform a systematic review on model-based testing approaches. They focus on 78 papers to show where MBT approaches have been applied, the characteristics and the limitations of MBT. They analysis the selected studies in terms of application scope of MBT approaches, level of automation, tools to support MBT, models used for test case generation, test coverage criteria, behaviour model limitations, cost and complexity of application of MBT

approach. In addition to these, they also discuss the issues regarding MBT approaches and limitation of MBT approaches. This study is similar to our study in that it considers model-based testing in general. However, it does not explicitly consider the adoption of architecture models for generating test cases.

Gurbuz et al. [10] have focused on the adoption of model-based testing for safety-critical systems. Detecting the potential faults is crucial for these systems since a failure or malfunction may result in death or serious injuries to people, equipment or environment. Again, hereby one of the key challenges is the derivation of test cases that can identify the potential faults, and MBT can be useful in this domain as well. The paper provides a systematic review to identify the state-of-the-art model-based testing for software safety. In the mapping study 604 papers have been identified and 32 of them have been selected as primary studies to analyse the benefits of MBT for software safety. The study shows that although MBT can provide important benefits for software safety testing still a number of challenges must be overcome to support reliable model-based testing approach for safety. The study of Gurbuz et al. have a specific focus, that is, model-based testing for software safety. We focused on architecture-driven model-based testing and did not consider a specific quality factor.

Felderer et al. [7] provide a taxonomy for model-based security testing approaches. The taxonomy is based on a comprehensive analysis of existing classification schemes for model-based testing and security testing. The study has identified 119 publications on model-based security testing and classified these according to the defined filter and evidence criteria. Likewise, the article provides an overview of the state of the art in model-based security testing and discusses promising research directions with regard to security properties, coverage criteria and the feasibility and return on investment of model-based security testing. Here again, the focus is on adopting model-based testing for a particular quality concern, which was not our focus.

Besides of testing, software architecture has also been used for model-checking. Model checking is a formal verification technique that aims for the automated analysis of a systems' behaviour with respect to selected properties. It takes as input a system model and a property specification [3][4]. The output is "true", or false with a counter-example when an error is detected. Zhang et al. [30] provide a classification and comparison of model checking software architecture techniques. The authors describe the main activities in a model checking software architecture process. Then, a classification and comparison framework are provided which is subsequently used to compare model checking software architecture techniques. Similar to software testing, model checking is one of the verification approaches. The study did not apply a systematic literature review but the outcome of the study could be considered complementary to our study in that it focuses directly on the adoption of architectural descriptions in the verification process.

To sum up, we can state that none of the above studies have adopted architecture descriptions in the model-based testing process. On the other hand, there are interesting complementary issues such as the focus on particular quality concerns and the adoption of model-checking.

2.6 CONCLUSION

In general, exhaustive testing is not practical or tractable for most real programs due to the large number of possible inputs and sequences of operations. As a result, selecting the set of test cases which can detect possible flaws of the system is the key challenge in software testing. MBT aims to automate the generation and execution of test cases using models based on system requirements and behaviour. MBT provides several important advantages such as improved test coverage and fault detection, reduced testing time, and reduction in cost. So far, various different models have been adopted to derive the test cases in MBT. In this study, we have focused on adopting architecture models to provide automated support for the test process leading to the notion of model-driven architecture-based testing (MDABT).

Different from other models such as finite state machines, which are typically at the detailed design level, software architecture provides a gross-level representation of the system at the higher abstraction level. To depict the state-of-the-art on MDABT, to identify the current solutions as well the open research challenges, we have carried out a systematic literature review on MDABT. We reviewed 731 papers that are discovered using a well-planned review protocol, and 31 of them were assessed as primary studies related to our research questions. Considering the different studies and the background on model-based testing in general we derived a reference process model to depict model-based testing. Based on our study we can derive several important lessons.

First of all, as we have derived from the primary studies the basic motivation for adopting MDABT seems to be the conformance checking of the code with the architecture. Hereby, typically the architectural relations and constraints are derived and test cases are generated, which are then applied on the real code. The motivation for checking the internal consistency of an architecture using MDABT was identified in a few primary studies. Although MDABT can be applied to any domain, some domains had a clear interest including embedded systems, real-time systems, cloud services, distributed systems, mission-critical systems and large-scale variant-rich systems.

Regarding the provided solutions we can state that all the presented approaches in the primary studies could be considered as instantiations of the reference process model. This shows that there is a consensus on the generic model-based testing process. The approaches differ in the description of the architecture models, description of test criteria, the generation process of the test models, the test execution and the analysis of the results. We have discussed each of these elements in detail. Besides of the various usages of the process elements, some approaches in the literature also relax the control flow that is defined in the generic process model. This is typically done by deriving test cases directly from a concrete test suite and thus ignoring the generation of an abstract test suite.

Further, although each approach appears to be interesting and effective for a given scope several obstacles still need to be solved. We have discussed these obstacles as the need for executable models for representing architecture, coping with the abstraction difference between architecture and code, the combinatorial state explosion problem, the lack of architecture views to consider multiple different concerns, the need for automating the process by tools, the cost-benefit analysis, and the need for applying MDABT on complex real systems to provide evidence of the approaches with respect to conventional approaches. Some of these obstacles

are an inherent problem also in the broader domain of MBT but these all seem to be relevant obstacles for MDABT in particular.

Finally, from our study we can state that the potential of MDABT has not been fully exploited yet. Given the reference process model that we have provided and the insight in the various different approaches, we can derive many more different possible MDABT processes than the 31 approaches that we could derive from the literature. It would be worth-while to explore these to optimize the benefits for MDABT in the future research. Similar to automated software testing in general, the MDABT approach could be used also for different quality concerns such as safety and security. Given the many benefits of MBT it will be worthwhile to elaborate on these in further research and likewise pave the way for more effective automated testing in practice.

We believe that the results of this study can be used by researchers as an input to advance the research on MDABT and derive even more effective solutions that can be applied in practice. Our own future research will be in alignment with the results of this study and consider the various obstacles to derive novel MDABT solutions. Furthermore, we aim to apply MDABT for large scale industrial case studies.

3

DESIGN OF VARIABLE BIG DATA ARCHITECTURES FOR E-GOVERNMENT DOMAIN¹

¹ This chapter is based on following published paper:

Tekinerdogan, Bedir, and Burak Uzun. "Design of Variable Big Data Architectures for E-Government Domain." *Software Engineering for Variability Intensive Systems: Foundations and Applications* (2019): 251.

Abstract

Like many application domains Big Data has become a very important driver for innovation and growth for e-government which aims to automate the public services to citizens. In general, e-government systems are often characterized as big data systems in which data storage and processing is one of the business-critical concerns. However, different features are required for different e-government systems, and likewise the corresponding big data architectures will need to be different as well. In this chapter, we present a systematic approach for the design of various big data architectures within the e-government domain. In this context, we adopt a big data reference architecture with a variability model for big data systems and e-government systems. We discuss the design decisions, the experiences and the lessons learned for deriving application architectures for different e-government systems.

Keywords: Big Data, E-Government, Reference Architecture, Variability Management

3.1 INTRODUCTION

Big Data has become a very important driver for innovation and growth for various industries such as health, administration, agriculture, defence, and education. The term Big Data usually refers to data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process data within a tolerable elapsed time. The realization of Big Data relies on several disruptive technologies such as Cloud Computing, Internet of Things and Data Analytics. Usually, big data systems represent major, long-term investments requiring considerable financial commitments and massive scale software and system deployments. Hence, it is important to identify the required features for developing the proper big data system. In practice, the characteristics of big data systems may vary in terms of the application domain and as such require different properties

One of the important domains in which big data is applied are e-government systems [1][37][44][49]. E-government is a general term describing the use of Information and Communication Technology (ICT) to facilitate the operation of government to provide better public services to citizens and businesses. The target of the services can be different and as such e-government includes different models including government-to-government (G2G), government-to-business (G2B), and government-to-citizen (G2C). E-government systems are often characterized as big data systems in which data storage and processing is one of the relevant issues. However, different features are required for different e-government systems, and likewise the corresponding big data architectures will need to be different as well. In general, designing big data systems for the specific requirements of the corresponding e-government system is not trivial.

In the literature we can identify several reference architectures for Big Data systems, but these are usually abstract and do not provide the required steps for deriving the specific big data architecture. Moreover, the features of e-government systems themselves are also not fixed and

can vary based on the different contexts. To support and guide the architect in deriving the proper big data architecture for e-government systems we present a systematic approach. The approach aims to develop the common and variant features of e-government systems and the required reference architecture. To this end, we have adopted a reference architecture for big data systems. Further, we discuss the design decisions for deriving the application architectures from this reference architecture, the experiences and the lessons learned. The results of our study can be of benefit for both practitioners who wish to develop big data e-government systems and researchers who are interested in challenges in these domains.

The remainder of the chapter is organized as follows. In section 3.2 we present the background on reference architectures and related to this big data reference architecture. In section 3.3 we present the overview of the method for deriving e-government systems from a reference architecture. Sections 3.4 presents the steps for developing reference models. Section 0 presents the steps for deriving a concrete e-government big data architecture. Section 3.6 presents the discussion, section 3.7 the related work, and finally section 3.8 concludes the chapter.

3.2 BACKGROUND

3.2.1 *Reference Architecture*

Every software system has a software architecture, whether it is complex or simple. A software architecture is an abstract representation that identifies the gross-level structure of the system and is important for supporting the communication among stakeholders, for guiding the design decisions, and for analysis of the overall system [26][53]. A stakeholder is defined as an individual, team, or organization with interests in, or concerns relative to, the system.

The architecture can be designed for a single system, but when we are dealing with a family or domain of systems, we can define a reference architecture. One of the key motivations for adopting reference architectures is to support reuse and productivity. Several definitions of reference architecture exist in the literature. Angelov et al [32] define a reference architecture as a generic architecture for a class of information systems that is used as a foundation for the design of concrete architectures from this class. Software Engineering Institute (SEI) defines the reference architecture as “a reference model mapped onto software elements that implements the functionality defined in the reference model”; and the reference model is described as “a division of functionality into elements together with the data flow among those elements” [5]. In general, a reference architecture presents the architectural best practices by various means such as standards, design patterns, and can be employed by software architects as a base from the beginning of the project to the end of it. While reference architectures are generic and focus on the family of systems, a concrete architecture or application architecture represents the architecture of a single system. Hereby, a concrete architecture is an instantiation of the reference architecture that defines the boundaries and constraints for the implementation and is used to analyse risks, balance trade-offs, plan the implementation project and allocate tasks [42].

Developing reference architectures is not trivial and requires a thorough understanding of the domain and the systems that it needs to cover. In general, a reference architecture can be

developed either from scratch whereby no concrete system exist yet, or be derived from the knowledge and experiences accumulated in designing concrete architectures in the past. Each concrete architecture is specific and shaped based on the requirements of the stakeholder concerns and the related requirements.

A reference architecture can have a descriptive role and/or prescriptive role. A reference architecture is descriptive in the sense that it captures the essence of existing architectures. It is prescriptive when it is used to guide the development of concrete architectures. Figure 3-1 depicts the relations between reference architecture and concrete architectures [42]. In this chapter we adopt both roles of the reference architecture. We will develop reference architectures for e-government systems and Big Data systems, and define the approach deriving concrete e-government architecture.

Figure 3-1: Relation between reference architecture and concrete architectures

3.2.2 Big Data Reference Architecture

Obviously, an appropriate big data architecture design will play a fundamental role to meet the big data processing needs. Several reference architectures are now being proposed to support the design of Big Data systems [33][34][35][39][40][43][45][48]. In this chapter, we adopt the functional architecture as described in [47] and as shown in Figure 3-2. The reference architecture distinguishes 6 key modules including Data Extraction, Data Loading and Pre-processing, Data Processing, Data Analysis, Data Loading & Transformation, and Interfacing & Visualization. The reference architecture can be used to describe many different big data systems. We will use it to describe e-government big data systems.

Figure 3-2: Big Data Reference Architecture (adapted from: [47])

3.3 METHOD FOR DERIVING E-GOVERNMENT SYSTEMS

When developing e-government systems we must take into account multiple different concerns. Although different e-government systems adopt different concerns and likewise will have a different architecture we can still identify a large part of commonality that can be reused. Hence, we describe a reuse-based approach in which we will make use of predefined reference architectures from which we will derive the concrete e-government architecture. Besides of e-government system concerns we will also explicitly distinguish big data concerns.

The method for deriving a concrete e-government big data system is shown in Figure 3-3. The method is represented using BPMN and consists of two key activities including the development of reference models, and the final concrete architecture from these reference models. In the top-level activity Development of Reference Models four different reference models are developed including the variability model for big data systems, the variability model for e-government systems, the reference business process models and the e-government big data reference architecture. During this activity no particular e-government system is developed, but only reference models are developed that could be reused.

In the activity Development of Concrete Application, the four reference models are reused to develop the concrete models including the selection of concrete features for the required big data system, the selection of the e-government system, the development of the concrete business process model, and finally the development of the concrete e-government big data system.

Figure 3-3: Method for Developing E-Government Big Data System

3.4 DEVELOPMENT OF REFERENCE MODELS

In this section we present the reference models that will be later reused to derive the e-government systems. Subsection 3.4.1 presents the variability model for big data systems, subsection 3.4.2 the variability model for e-government systems, subsection 3.4.3 the reference business process models and subsection 3.4.4 the e-government big data reference architecture.

3.4.1 Variability Model for Big Data Systems

The variability model for big data systems is represented as a feature diagram [41] and is shown in Figure 3-4. This has been adopted from our earlier work on a feature-based analysis of big data systems [48].

Figure 3-4: Top-Level Feature Model for Big Data System (adapted from:[48])

The feature diagram includes a root node representing the domain or system that includes features representing the essential characteristics or externally visible properties of the system. Features may have sub-features which as such can lead to a hierarchical tree. Features can be mandatory or variant. Variant features are usually represented as optional or alternative features. Optional features can be selected or not, whereby alternative features require the selection of one of the defined features. A feature configuration is a set of features which describes an alternative model. A feature constraint further restricts the possible selections of features to define configurations. Since the variability model represents the domain of big data systems, we term this as a family feature diagram that can be used to derive concrete feature diagrams representing the required features for a concrete big data system.

3.4.2 Variability Model for E-Government Systems

Figure 3-5 represents the variability model for the e-government domain. The model has been derived from a set of selected chapters on e-government systems [1][37][44][49].

Figure 3-5: Top-Level Feature Model for Big Data System (adapted from:[48])

The variability model has been again represented as a feature diagram. Based on the sources that we have selected e-government systems are characterized using the top-level features Style, Security, Fault Management, Network, Data Management and Requirements. Each of these features have on their turn sub-features which define the variability. E-government style feature consists of Government-to-Government (G2G), Government-to-Customer (G2C) and Government-to-Business (G2B) and all e-government systems must conform to one of these styles. Every e-government system must have security feature which includes the sub-features logon authentication, authorization, validation and e-signature. Every e-government system must have at least one of these sub-features. Likewise, authentication sub-feature is decomposed into logon authentication, network access authentication, remote authentication and single sign. Any e-government system must be authenticated according to one of these authentication types. On the other hand, validation feature is decomposed into client-side and server-side validations but these two sub-feature types are optional in an e-government system. Every e-government system must manage faults that occur during the operation of the system. Fault management is divided into three sub-features which are fault handling, auditing and logging. It can be seen from the figure that fault handling is a mandatory feature, but auditing and logging features are optional in a e-government system. E-government network can be open network in which any computer connected to internet can have access or can be located in a closed network(intranet) such as VPN (Virtual Private Network). Data management is a crucial feature for e-government systems as the data stored in these systems are often used for reporting. As the e-government systems are big data systems, the generated data of the system must be archived regularly so that no data loss can occur. E-government reporting consists of list reporting, interactive analysis, ad-hoc querying and dashboards. At last, e-government systems are built from requirements from different stakeholders in which each stakeholder concern is addressed. These concerns must consist business rules and visualization of the system. Moreover, localization of the system can represent a concern for stakeholders.

3.4.3 E-Government Reference Business Process Models

One of the aspects that differentiate e-government systems is the adoption of different workflows. Figure 3-6 shows the generic workflow for e-government domain using the business process modelling notation (BPMN). The model represents the workflows for different e-government styles including G2G (government to government), G2C (government to citizen) and G2B (government to business).

There are three participants in the figure including government clerk, e-government system, and requester which can be further divided into business, government clerk or citizen. In the overall process, first a requester fills the e-forms and sends it to the system to process the request. The system processes the request and it either completes its task or sends the request to government clerks for approval. System requests a service from the government clerk by sending a request for evaluation of request. Government clerk evaluates the request and sends the result back to system and completes its task. The workflow represents the generic steps and can be customized in different ways.

Figure 3-6: BPMN Diagram for E-Government Systems

3.4.4 E-Government Big Data Reference Architecture

We provide the reference architecture for e-government big data system with respect to three viewpoints including decomposition viewpoint, layered viewpoint and deployment viewpoint [5].

The decomposition viewpoint defines the partitioning of the system into modules and sub-modules. Hereby, the only architectural relation that is used is the decomposition relation. In Figure 3-7 the decomposition view reference model for e-government big data system is presented. The figure shows that the system is decomposed into five main modules which are security, networking, fault management, functional requirements and data management modules. Security module is further divided into authentication, authorization, validation and e-signature modules. Furthermore, networking is another important aspect of the system as e-government system can have internal and external communications during the operation of the system. Networking module is decomposed into cooperability, communication and network security submodules in which communication submodules is further divided into sync and async communication modules. Fault management module is decomposed into three submodules which are logging, auditing and fault handling. Functional requirements address the concerns of stakeholders and as shown in the figure it consists of localization, business logic, visualization interface and reporting. As stated before, data is invaluable for any e-

government system and must be managed delicately through the e-government system lifecycle. Data management module is divided into seven submodules which are analysis, processing, extraction, pre-processing, archiving, loading and transformation. Processing, extraction, analysis and pre-processing is further divided into submodules as shown in the decomposition view.

Figure 3-7: Decomposition View Reference Model for E-Government Big Data System

Layered viewpoint reflects the decomposition of the system as a set of layers. Each layer offers a cohesive set of services that are provided to the upper layers. In a layered architecture every layer can access the next lower layer, and no call-backs are allowed, nor bridging of layers. Figure 3-8 shows the layered view for the reference model of big data e-government systems. Hereby, the system is divided into four main layers which are client, gateway, service and persistence layer. Client layer represents interaction layer of the users of the system where it has two different types either client applications or webservices at the same level. Client layer uses gateway layer which does the load balancing and serves as endpoint for applications to communicate with. Gateway layer uses the service layer to send service requests from different applications at client layer. Service layer is decomposed into three layers: authentication authorization, validation and business logic layer. After processing of service requests, service layer can retrieve information from persistence layer or simply store the necessary information from the request. Persistence layer has different data sources to populate and retrieve from as media, relational data bases, no-SQL data bases, logs, document and archives. Repository layer has different layers for persistence of data and retrieval of data. Data sent from the service layer to be persisted passed into respective data source according to data type. Data retrieval is divided into five layers which are data extraction, data loading and pre-processing, data processing, data analysis and data loading and transformation.

Figure 3-8: Layered View Reference Model for E-Government Big Data System

Deployment viewpoint defines the mapping of software modules to hardware nodes. Figure 3-9 shows the reference deployment view model for e-government big data systems. It has three main nodes which are clients of different types, server and data. Clients interact only with server node and can be webservices integrated into these systems services or client applications that interact regularly with system through services that are publicly defined. Server node holds seven main inner artifacts that can be deployed to server such as security, validation, service for downloading interface, logging, gateway, business logic and data management. Server artifacts interact with both client and data nodes. Data node holds different data sources to be

served on demand to data management node. The data sources are shown as relational database, no-SQL database, logs, archives, documents and media.

Figure 3-9: Deployment View Reference Model for E-Government Big Data Systems

3.5 DEVELOPMENT OF CONCRETE E-GOVERNMENT ARCHITECTURE

In this section we will describe the concrete architecture for the case study. To this aim we will first select features from big data variability model and e-government variability model for our application. We will then develop the business process model and present the concrete architecture with respect to the earlier stated three viewpoints.

3.5.1 Select Features for Big Data

For instantiating the big data reference architectures, we first need to identify and describe the features of the system. This means in essence the selection of features in the earlier defined family feature diagram. The selected features are listed in Table 3-1. The left column represents the important features of big data systems based on the features as shown in Figure 3-4. The right column represents the features that are important for our example system. For the given case study, data can be of different types including structured, semi-structured and unstructured. The system adopts all the three types of data. The majority of the data is structured data which is stored in the relational database. Besides of these semi-structured data is also processed such as xml files. Finally, unstructured data comes in the form of graphics and media that is mainly used for visualization. Data loading and transfer methods are developed for different data types.

To support the real-time statistics queries the system adopts batch processing of data which is updated periodically from day to weeks, and sometimes months depending on different requirements. Real-time stream processing is also adopted, for web service integrations. Streaming data from web-services are processed and stored in the system for further operations. Due to the privacy issues, security is an important concern in the overall system. Several security protocols including data access mechanisms are adopted during both the storage, pre-processing, processing and visualization of the data. Reporting and dashboarding of the data is important for users to gather valuable data information in presentable format.

Table 3-1: Features of the Big Data System

Feature	Selected Features of Big Data System
<i>Data</i>	Mobility; Structure: Structured, Semi-Structured, Unstructured; Security; Modality: Textual, Graphic, Media
<i>Information Management</i>	Data Load; Data Transfer
<i>Data Processing</i>	Batch Processing; Stream Processing; Pre-Processing: Compression
<i>Interface and Visualization</i>	Reporting and Dashboarding: User Interface, Application Interface
<i>Security</i>	Trusted: Data over VPN, data access control

3.5.2 Select Features for System E-Government System

Besides of selecting the features for the required big data system we also need to select the features for the e-government concerns. These e-government features are listed in Table 11.2. The left columns present important features for e-government systems based on the feature diagram in Figure 3-5. The right column presents the features that are selected for the example e-government system. The selected e-government system style is G2G. The systems security is crucial as important operations are being held by the system and system data holds private information for different organizations. To this aim, we select network access authentication to control user who are able reach the systems network and also log-on authentication for entrance to the system. Moreover, authorization should be done both by data level and roles that the logged in user have. Fault management is done by two features fault handling and logging. System network is closed, using a VPN. Data archiving is important as it is a big data system and data size can slow down real-time operation of the system, but archives are used in batch processing for different statistics to be taken from the system. Three types of reporting are

selected list reporting, dashboard and interactive analysis. The system must meet requirements of the organization for different business processes and must have client interface for users of different role to interact with the system.

Table 3-2: Features of the E-Government System

Feature	Features of E-Government System
<i>Style</i>	Government to Government
<i>Security</i>	Log-on Authentication, Network Access Authentication, Authorization, Client-Side Validation, Server-Side Validation, E-Signature
<i>Fault Management</i>	Fault Handling, Logging
<i>Network</i>	Closed
<i>Data Management</i>	Data Archiving, List Reporting, Dashboard, Interactive Analysis
<i>Requirements</i>	Client Interface, Business Rules

3.5.3 Develop E-Government Application Business Process Model

After selecting the big data and e-government system we need to define the required workflow. Figure 3-10 shows the BPMN diagram for government to citizen e-government system which is required for the selected case study. There are three actors which are requester government clerks, e-government system and approver government clerks. Requester government clerk makes requests for retrieving client-side application from system. System sends response which holds e-forms for client-side application and government clerk fills out e-forms and sends a request to the system. System returns response for this request and it either completes the task directly or sends an evaluation request to the approver government clerk asking for service. Approver government clerk receives a request from the system and evaluates the request and sends the response to the system. As it can be seen in the figure, the business process model largely follows and reuses the earlier defined reference business process model of Figure 3-6.

Figure 3-10: BPMN Diagram for G2G Application

3.5.4 Develop E-Government Big Data Concrete Architecture

After selecting the required big data system features, the e-government features, and the workflow, we can derive the concrete architecture. For this we will reuse the earlier defined reference architecture that provided the decomposition view, the layered view, and the deployment view.

In our approach the selection of the features in the variability models implies a design decision for the selection of the modules in the architecture views. These design rules could be defined as follows:

IF feature $\langle f \rangle$ selected THEN select architecture component $\langle c \rangle$ in architecture view

We can also represent the design decisions in a table format. Likewise, in Table 3-3 and Table 3-4 we show the implied architecture design decisions for the selected features for each architecture view. Each selection of a feature represents thus an architecture design decision. The rationale for these design decisions, that is, the selection of the corresponding features is driven by the project requirements and constraints. Both the selection of the features for big data systems and the e-government system will have an impact on the eventual architecture. Features that do not have a direct implication on the architecture have been indicated as not applicable (NA).

Table 3-3: Features of the Big Data System and the implied architecture design decisions

Feature	Features of Big Data System	Architecture Design Decision		
		Decomposition	Layered	Deployment
<i>Data</i>	Mobility; Structure: Structured, Semi-Structured, Unstructured; Security; Modality: Textual, Graphic, Media	Select modules Extraction, Data Archiving	Select Media, Relational DB, Logs, Archives, Documents layers	Select Media, Relational DB, Logs, Archives, Documents at Server Node
<i>Information Management</i>	Data Load; Data Transfer	Select the modules Data Loading, Data Transformation	Select Data Loading, Data Transformation layers	Select Data Management at Server Node
<i>Data Processing</i>	Batch Processing; Stream Processing; Pre-Processing: Compression	Select Batch Processing, Stream Processing, Data Compression modules	Select Data Processing, Data Loading, Data Preprocessing Layers.	Select Data Management at Server Node
<i>Interface and Visualization</i>	Reporting and Dashboarding: User Interface, Application Interface	Select Visualization and Reporting modules	Select Client Layer together with its Client Applications, Web Services Layers	Select Web Services and Client Nodes
<i>Security</i>	Trusted: Data over VPN, data access control	Select Network Security Module	NA	NA

Table 3-4: Features of the E-Government System and the implied architecture design decisions

Feature	Features of Big Data System	Architecture Design Decision		
		Decomposition	Layered	Deployment
<i>Style</i>	Government to Government	NA	NA	NA
<i>Security</i>	Log-on Authentication, Network Access Authentication, Authorization, Client-Side Validation, Server-Side Validation, E-Signature	Select modules Authentication, Authorization, Validation, E-Signature	Select layers Authentication, Authorization, Validation in the Service Layer	Select Security and Validation Modules at the Server Node
<i>Fault Management</i>	Fault Handling, Logging	Select modules Fault Management module with Fault Handling and Logging modules	NA	Select Logging to Server Node
<i>Network</i>	Closed	Select module Network Security	NA	NA
<i>Data Management</i>	Data Archiving, List Reporting, Dashboard, Interactive Analysis	Select modules Data Archiving, Visualization Interface, Reporting	Select layers Client Applications, Persistence, Webservice	Select Client Node
<i>Requirements</i>	Client Interface, Business Rules	Select modules Functional Requirements Module, Business Logic, Reporting	Select Business Logic Layer	Select Business Logic to Server Node. Select Interface module to Server Node. Select Client Node

As a result of the adopted design decisions, Figure 3-11 shows the decomposition view of the application architecture of example e-government system. It has five modules which are security, networking, functional requirements, fault management and data management. Security is decomposed into four more modules which are authentication, authorization, validation and e-signature. Networking modules has only network security submodule. Fault management module is decomposed into logging and fault-handling modules. Functional requirements module is decomposed into business logic, visualization interface and reporting submodules. At last data management module is decomposed into pre-processing, processing, extraction, archiving, data loading and transformation.

Figure 3-11: Decomposition View of the Application Architecture of Example E-Government System

Figure 3-12 shows the layered view of the application architecture of example e-government system. The first layer is client layer where it consists of client applications and webservices. It interacts with service layer which consists of authentication, authorization, validation and business logic layer. Service layer than unidirectionally interacts with repository to use the services offered by the persistence layer. Persistence layer is decomposed of different data sources such as relational database, media, logs, documents and archives. Data is persisted into these data sources by service layer or different systems. Data is also queried from this layer by service layer. In this process data is first extracted and pre-processing is applied which is followed up by loading the data. Loaded data is then processed and transformed into structure recognizable by the service layer.

Figure 3-12: Layered View of the Application Architecture of Example E-Government System

Figure 3-13 shows the deployment view of the application architecture of example e-government system. There are three main artifacts shown in figure: client applications, server application, and data sources. Client applications that interact with the server application can be webservices or client-side interfaces. Server application has data management, logging, business logic, security, validation, and interface artifacts deployed on it. The deployed artifacts operate with each other internally while data management also operates with data sources to manage the differently structured data. There are structured, semi-structured, and unstructured data sources available which are relational database, logs, archives, documents, and media.

Figure 3-13: Deployment View of the Application Architecture of Example E-Government System

3.6 DISCUSSION

We have provided a reuse-based approach for developing big data architectures of e-government systems. For this we have adopted a reference architecture based on the functional architecture as described in [47]. This reference architecture appeared to be quite useful together with the feature diagrams for big data systems and e-government systems that we have developed. In the literature several different reference architectures can be identified. One of the popular reference architectures is the so-called Lambda architecture, which is a big data architecture that is designed to satisfy the needs for a robust system that is fault-tolerant, both against hardware failures and human mistakes [43]. Hereby it takes advantage of both batch- and stream-processing methods. In essence, the architecture consists of three layers including batch processing layer, speed (or real-time) processing layer, and serving layer. The batch processing layer has two functions: (1) managing the master dataset (an immutable, append-only set of raw data), and (2) to pre-compute the batch views. The master data set is stored using a distributed processing system that can handle very large quantities of data. The batch views are generated by processing all available data. As such, any errors can be fixed by recomputing based on the complete data set, and subsequently updating existing views. The speed layer processes data streams in real time and deals with recent data only. In essence, there are two basic functions of the speed layer: (1) storing the real-time views and (2) processing the incoming data stream so as to update those views. It compensates for the high latency of the batch layer to enable up-to-date results for queries. The speed layer's view is not as accurate

and complete as the ones eventually produced by the batch layer, but they are available almost immediately after data is received. The serving layer indexes the batch views so that they can be queried in low-latency, ad-hoc way. The query merges result from the batch and speed layers to respond to ad-hoc queries by returning precomputed views or building views from the processed data. If we would have adopted the Lambda architecture instead of the functional architecture, the overall process would not differ but the feature diagrams for big data systems had to be aligned for this.

Architecture design is critical for the success of a project and it is therefore important to evaluate the architecture with respect to the stakeholders' concerns [5][26]. A proper architecture evaluation will help to predict the quality of the system before it is built, thereby reducing unnecessary maintenance costs. Different architecture evaluation approaches have been proposed in the literature. In essence, each architecture evaluation approach takes as input the stakeholder concerns, the environment issues and the architecture description. Based on these inputs, the evaluation results in an architecture evaluation report, which is used to adapt the architecture. In the method that we have provided feature models are used to capture the required features. The focus hereby was on functional features and less on non-functional or quality concerns. The approach is however agnostic to any architecture evaluation method and can be easily integrated with the existing architecture evaluation approaches.

The method that is proposed in this chapter can be categorized as a reuse-based approach in which we focus on a family of systems or products. In this context the method borrows the ideas of software product line engineering that distinguishes between the activities of domain engineering and application engineering. A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [36][54]. Software product line engineering (SPLE) is a systematic and comprehensive process that aims to develop and maintain product lines. SPLE aims to provide pro-active, pre-planned reuse at a large granularity (domain and product level) to develop applications from a core asset base. In the domain engineering activity core assets are developed for reuse purposes. Core assets can include any work product in the development life cycle. We have focused on the architecture design. In principle we could elaborate the approach and define a product line engineering approach for e-government systems. The results in this chapter pave the way for this idea since the architecture design is also the core asset of SPLE that defines the scope of the products in the product family.

Several mitigation strategies have been carried out to cope with the potential threats to this study. Construct validity refers to what extent the operational measures that are studied really represent what the researchers have in mind and what is investigated according to the research questions [55]. Our key focus was on developing models that could be reused to derive concrete models and in particular the architecture. For developing the reference models, we have applied a thorough domain analysis. The domain scope could be further broadened by selecting additional papers, which would further shape the models. Internal Validity relates to a causal relationship between treatment and the outcome. In our study there is a causal relationship between the selected features and the architecture design decisions. We have shortly discussed the implication of the selection of features to the architecture design decisions. External Validity concerns the ability to generalize the results of the study. Although the reference models could

be further refined in case the corresponding domain evolves, the overall approach is generic and will be persistent. Hence, we believe that from this perspective the result of the study, a reusable and generic method has been provided.

3.7 RELATED WORK

In the literature, we can identify different approaches for defining reference architectures. Galster and Avgeriou [38] propose a methodology to define empirically-grounded reference architectures. The approach consists of the following steps: Decision on type of reference architecture, selection of design strategy, empirical acquisition of data, construction of reference architecture, enabling the reference architecture variability and finally evaluation of the reference architecture.

Design rationale has been studied in different disciplines including engineering design in AI, human computer interaction and software engineering. Various surveys have been published that compare different systems that capture and use design rationale [46][52]. These studies have shown that design rationale is considered important by practitioners, but it is rarely captured in practice. In our earlier work, we have provided a systematic approach for feature-based approach for adapting architectures. and the corresponding tool environment ArchiRationale for supporting software architecture adaptation [52]. The approach takes as input an existing architecture and captures the design rationale for adapting the architecture for a given quality concern. For this we define a feature model that includes the possible set of architectural tactics to realize the quality concern. The presented approach captures the rationale for deciding on feature selections and for selecting the corresponding architecture design alternatives. ArchiRationale customizes and integrates the Eclipse plugin tools XFeature, ArchStudio and XQuery to provide tool support for capturing, storing and accessing the design rationale. We illustrate the approach for adapting a software architecture for fault tolerance.

In our earlier work we have defined an approach for supporting the decision making on the selection of platforms within the context of model-driven architecture [50]. Hereby, we have adopted the notion of platform independent model (PIM) and Platform Specific Model (PSM) and stated that although the MDA approach provides a conceptual framework for separating the two models and as such adapting the platforms, it does not provide the decision support for selecting the proper platform. The similar problem applies to the adoption of big data systems in e-government systems. In fact, e-government can be considered as a platform independent model whereas the big data system can be considered as the platform specific model and infrastructure. In case of multiple different big data systems, we could integrate our earlier approach with the approach for deriving application architecture that we have discussed in this chapter.

Quality assurance is one of the important issues when developing e-government systems [25]. For supporting quality early on, several software architecture analysis approaches have been introduced. The goal of these approaches is to assess whether or not a given architecture design satisfies desired concerns including quality requirements. We were able to derive the architecture based on the reference architectures. For further detailed analysis we could use the existing architecture analysis approaches. We consider this as part of our future work in which we will investigate the impact of quality concerns on big data architectures.

In our recent work we have provided a feature driven survey of big data systems [48]. To this end, we proposed a domain engineering approach in which a family feature model, a reference architecture and the corresponding design rules are identified. The family feature model is derived based on a domain analysis of big data systems and represents the common and variant features. The reference architecture represents the generic structure for various application architectures of big data systems. Finally, the design rules define the reusable design heuristics for designing an application architecture based on the selection of features of the family feature model and the reference architecture. We have illustrated our approach for deriving the big data architectures of different well-known big data systems [48].

3.8 CONCLUSION

In this chapter we have presented BITA*, a framework for aligning business processes and IT systems of multiple collaborating organisations. We identified three types of alignment concerns including business process to business process (BP2BP), IT to IT (IT2IT) and business process to IT (BP2IT) alignment concerns, and provided the corresponding three alignment design viewpoints.

In this chapter we have focused on supporting the software architecture design of e-government systems. For this we have adopted reference architectures and variability modelling. The adoption of reference architecture for deriving architectures is well-known in the literature [42][51]. In fact, for the design of an architecture either an existing reference architecture can be used, or a reference architecture can be designed first. For our study, we used the reference architecture for Big Data systems and developed the reference architecture for e-government systems. The approach that we have provided is generic and the study could be replicated by using different reference architectures for both big data systems and e-government systems.

For deriving the application architecture, we first analysed the stakeholders, the workflow models and the required features for e-government systems using big data. Based on the selected features we instantiated the reference architecture. Hereby, we implicitly made design decisions to derive the architecture that meets the selected features. One of the key issues in architecture design is the design rationale management, that is, the reasons why and how design decisions are made and applied. In this chapter we have guided the decision making through the reference models. The particular design decisions are dependent on the required e-government systems.

In our approach we have adopted architecture views to model the architecture for different stakeholder concerns [5]. We have used the decomposition view, layered view and deployment. These viewpoints appeared to be essential in designing e-government systems. If needed other viewpoints could be integrated in the approach. The overall process that is developing a reference architecture for e-government system together with the commonality and variability modelling, and the workflow models will however be the same.

An e-government system must take into account multiple different concerns. We have adopted a reuse-based approach in which we modelled the common and variant features of an e-government system together with its big data and workflow properties. This supported not only the quick development of the required e-government architecture but in this way, we also

ensured that all the required properties were identified. As stated before, the approach itself is generic. The reference models could be further enhanced to broaden the scope of the domain and herewith the architecture.

ARCHITECTURE CONFORMANCE ANALYSIS USING MODEL-BASED TESTING: A CASE STUDY APPROACH ¹

¹ This chapter is based on following published papers:

Uzun, Burak, and Bedir Tekinerdogan. "Model Driven Architecture based Testing Tool based on Architecture Views." MODELSWARD. 2018.

Uzun, Burak, and Bedir Tekinerdogan. "Architecture conformance analysis using model-based testing: A case study approach." Software: Practice and Experience 49.3 (2019): 423-448.

Abstract

The architectural drift problem defines the discrepancy between the architecture description and the code. Deviations of the code from the architecture can occur if architectural constraints as defined in the architectural models are not implemented in the code. For large-scale systems manually checking the consistency of the architecture with the code is not trivial and cumbersome. The overall objective of this chapter is to propose and analyse the effectiveness and practicality of an architecture conformance analysis approach using model-based testing (ACAMBT) approach for checking the consistency between architectural models and the code. Hereby, we consider the case whereby the architecture is assumed correct, and the code needs to align with the architecture. We propose a model-based testing (MBT) that uses architectural views to automatically derive test cases for checking the architectural constraints in the code. We have evaluated the approach and the corresponding toolset for a real industrial case study using a systematic case study protocol. Further we have adopted exhaustive fault-injection techniques to detect the constraint violations. The evaluation of the approach on real code showed that deviations with the architectural constraints could be easily detected in the code. We can conclude that ACAMBT is effective for identifying inconsistencies between the architecture views and the code for the defined view constraints. Our survey study with practitioners showed that adopting the ACAMBT approach is practical and easy to use. The approach as such can be considered as a complimentary tool to the existing testing and reflexion modelling approaches.

Keywords: Model-based testing, software architecture, architecture conformance analysis, case study

4.1 INTRODUCTION

Software systems are rarely static and need to evolve over time due to bug fixes or new requirements. This situation causes the so-called architectural drift problem which defines the discrepancy between the architecture description and the resulting implementation [56]. A popular approach for coping with this problem is reflexion modelling which usually compares an abstract model of the code with the architecture model to identify the differences [60][20]. Identifying the discrepancies between the code and the architecture in this manner can be cumbersome for large scale software system. Hence, this form of conformance checking would benefit from automation to detect the flaws due to the inconsistencies between the architecture and the code.

Software testing is the traditional approach for the verification of the code to identify possible mismatches between expected and present requirements of the system [2][74]. Software is correct if and only if each valid input to the system produces an output according to the system specifications. In general, exhaustive testing is not practical or tractable for most real programs due to the large number of possible inputs and sequences of operations [74]. 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 (MBT) addresses this challenge by automating the generation and execution of test cases using models based on system requirements and behaviour [10][28]. MBT relies on models to automate the generation of the test cases and their execution. A model is usually an abstract, partial presentation of the desired behaviour of a system under test (SUT). MBT 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 trend in MBT is to adopt software architecture models to provide automated support for the test process leading to the notion of model-driven architecture-based testing. Software architecture is different from the other design representations since it provides a gross-level representation of the system at higher abstraction levels [5][12][26].

Model-driven architecture-based testing can be used for various purposes in software testing [59][64][75][30]. In this chapter, we propose a model-driven architecture-based testing approach for checking the consistency between the architecture and the code (ACAMBT). When considering architectural consistency checking in principle three different types of scenarios can be distinguished (1) the architecture is assumed correct and code needs to conform to architecture (2) the code is considered correct, and architecture needs to be aligned to the code (3) neither the architecture nor the code is considered definitive and both can be adapted to realize consistency. Based on the industrial context our study presents an approach in which we make the assumption that the architecture is correct and that it is required to check whether the provided code conforms to the architecture. The assumption that the architecture is correct while the code needs to be aligned with the architecture has also been addressed in various different studies [50][59][65][17][75][80].

In this chapter, we report on our experiences using such an ACAMBT approach and evaluate it with respect to effectiveness and practicality within a real industrial context. To this end, we apply a systematic case study approach using guidelines provided by Runeson and Höst [77] for a large-scale software company in the context of a smart e-government system. We have also applied a survey study using guidelines by Kitchenham's et al [66][67][68][69][70] to check the practicality of the adopted approach.

The outline of the chapter is organized as follows: section 4.2 provides the background on reflexion modelling, and model-based testing. Section 4.3 presents an example industrial case and the problem statement. Section 4.4 presents the approach for conformance checking using model driven architecture-based testing. Section 4.5 presents the corresponding tool that implements the approach. Section 0 describes the architecture of the system that the industrial case study is performed on, a big data e-government system. Section 4.7 discusses the case

study design along with the survey study design adopting the ACAMBT approach. Section 4.8 provides the discussion and elaborates on the results, section 4.9 the related work, and finally, section 4.10 presents the conclusion.

4.2 PRELIMINARIES

4.2.1 *Reflexion Modelling*

Each software architecture design is the result of a broad set of design decisions and their justifications, that is, the design rationale [52]. To capture and communicate these architecture decisions a proper documentation of the software architecture is needed. The architecture documentation usually includes a set of architecture views which are developed based on the corresponding architecture viewpoint. Each architecture viewpoint and associated, instantiated architecture view is important for one or more stakeholders and their concerns. The software architecture documentation as such can be used to serve as a guideline for the subsequent life cycle activities, to realize and control the design decisions.

Unfortunately, software systems are rarely static and need to evolve over time due to bug fixes or new requirements. This situation causes the so-called architectural drift problem which defines the discrepancy between the architecture description and the resulting implementation [60][71][72][20][76][19]. This architectural drift can be even introduced during the initial implementation of the architecture due to lack of knowledge about the architecture or stringent time-to-market constraints. The existence of discrepancies between the architecture description and its implementation is not a merely theoretical issue but may directly lead to performance, security issues as well as increased maintenance time and cost, because the original design goals have been lost. In the extreme cases, a system's architecture may even deteriorate to a degree where further development is not feasible, leading to a complete system re-implementation.

The architecture and the code can be inconsistent in different ways for different views. If the relations that are present in the architecture are also found in the implementation, then this is convergent relation. In case the architecture relation is not present in the implementation then this is called an absence relation. Finally, if the implementation includes relation that is not present in the architecture, then this is called divergence relation. Architectural violations are due to absence or divergence relations.

A successful technique that is used for architecture consistency checking is the reflexion modelling approach, as proposed in the software architecture design community [71][20]. A reflexion model is the model produced by the technique which allows comparison of the architectural design with a derived, abstracted model of the code's dependencies. Typically, a reflexion model highlights the differences between the code and the architecture and, as such, defines the extent of the architectural drift problem. Based on the reflexion model either the architecture design, the code or the abstracted model of the code (the mapping) can be adapted to keep the consistency among both artefacts.

Figure 4-1: Activity diagram showing the steps in reflexion modeling for architecture-to-code conformance

Figure 4-1 shows the common workflow representing the reflexion modelling approach. In principle, a reflexion model allows a software developer to view the structure of a system's source through a chosen high-level (often architectural) view. To check the consistency between the architecture model and the code, an abstract model of the code is derived. In the activity diagram it is shown that this is carried out by the developer. Sometimes this activity can also be carried out by the architect (who could be a developer as well). This abstract model contains elements such as packages, classes, methods and variables of the source, grouped together as the 'architectural elements' that might be included in architectural model. The two models are then compared to each other by parsing of the system and assessing the actual relations between the architectural elements in the abstract model. The results of the comparison are presented to the user through a Reflexion Model. The reflexion model explicitly represents the convergence, the divergence, and the absence relations. If multiple architecture views are adopted then typically for each architecture view a separate reflexion model can be created.

By analysing the reflexion model, the architecture, the code or the mapping rules can be altered. Usually architecture conformance analysis approaches that apply reflexion modelling include tools for modelling the architecture, modelling the mappings, deriving the abstract model from the source code, checking the consistency, and generating the resulting reflexion model.

The architecture of a system is usually modelled using multiple architectural views that represents the system with respect to a particular concern. Because multiple concerns need to be taken into account when designing a system, multiple architecture views can help to separate the concerns and, as such, support the modelling, understanding, communication and analysis of the software architecture for different stakeholders. The existence of multiple architecture

views requires that conformance address all these views, in particular the views that relate to implementation concerns.

4.2.2 *Model Based Testing*

Similar to architecture modelling, the primary reason for modelling in general is usually defined as a means for communication, analysis or guiding the production process. Usually models are characterized based on their level of precision, and can be considered as a Sketch, a Blueprint, or an Executable [18]. An executable model is a model that has everything required to produce the desired functionality of a single domain. Executable models are more precise than sketches or blueprints and can be interpreted by model compilers. In model-driven software development the concept of models can be considered as executable models. This is in contrast to model-based software development in which models are used as blueprints at the most. The language in which models are expressed is defined by meta-models. As such, a model is said to be an instance of a meta-model, or a model conforms to a meta-model.

Model-driven development has been used in different phases in the lifecycle including software testing. 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 human testers. Model-Based Testing (MBT) relies on models of system requirements and behaviour to automate the generation of the test cases and their execution [28]. There are several motivations to perform MBT such as easy test maintenance, automated test design and enhancing test quality. Figure 4-2 presents the common workflow for MBT.

Based on the Test Requirements and the Test Plan a Test Model is constructed. A model is usually an abstract, partial presentation of the desired behaviour of a system under test (SUT). The test model is used to generate test cases that together form the Abstract Test Suite. The test cases in the abstract test suite lack the detail needed by the SUT and as such are not directly executable. Hence, in the third step, the abstract test suite is transformed to a concrete or executable test suite. This is typically done using a transformation tool, which translates each abstract test case to an executable test case. An advantage of the separation between abstract test suite and concrete test suite is the platform and language independence of the abstract test cases. The same abstract test case can be reused in different test execution environments. In the fourth step, the concrete test cases are executed on the SUT. The test execution will result in a report that contains the outcome of the execution of the test cases. In the final, fifth step, these results are analysed and if needed corrective actions are taken: for each test that reports a failure, the cause of the failure is determined and the program (or model) is corrected.

Figure 4-2: Process of model-based testing

4.3 INDUSTRIAL CASE AND PROBLEM STATEMENT

In this section we will describe the industrial case that is used to both illustrate the approach (next section) and to validate the approach. For confidentiality, the name of the private company is not provided. The case study concerns an e-government system describing the use of ICT to facilitate the operation of government to provide better public services to citizens and businesses. In general, e-government includes different models including government-to-government (G2G), government-to-business (G2B), and government-to-citizen (G2C). Figure

4-3 shows the general workflow of the case study system. Citizens can request services through an internet browser, which is mapped to e-forms that trigger process requests. The process requests are handled by an intermediate Mid-Office that sends requests to the corresponding government clerk. The required data is derived from the data administration database.

In our case we are dealing with a G2C e-government system in which the end users of the system are government clerks that provide services to citizens. There are around six thousand online users on average throughout the week. Users generate structured data conforming to business rules defined for smart e-government system. The system which utilizes the relational database solutions handles more than 7 million database transactions per day. Batch processes run nightly on the real-time data to create batch views and the required reports and statistics.

Figure 4-3: General Workflow of the G2C E-Government System

To guide the development of the system according to stakeholder concerns, a software architecture design is provided. For maintenance and consistency purposes it is important that the code of the e-government system follows the corresponding architecture. In the following sections we will elaborate on the architecture models. Figure 4-4 shows, as an example, the layered architecture that is often used in e-government systems.

Figure 4-4: Example layered architecture of G2C E-Government System

As shown in the figure the architecture includes the three layers: Client Layer, Service Layer, and Data Storage Layer. The Client Layer includes the module for e-forms, webservices and the human computer interaction, the service layer includes the modules for the validation, authorization and e-government business logic, and finally the data storage layer includes the functionality for storing and accessing the persistent data.

For consistency it is required that the code conforms to the architectural constraints. This implies that the defined layers and the relations among the layers are not violated in the code. This is required for the layered view of the architecture. In case of multiple different architecture views, checking the inconsistencies requires that conformance analysis must be carried out for the code and each of the selected views. Likewise, the code should reflect the architectural constraints in each of the views. For small scale systems with a few architecture views this might be carried out manually. Yet the conformance analysis can soon become cumbersome and require further automation for larger systems.

4.4 CONFORMANCE CHECKING USING MODEL-DRIVEN ARCHITECTURE BASED TESTING

For verifying whether the code is consistent with the architecture views we propose an architecture conformance analysis using model-based testing approach (ACAMBT) in which we adopt architecture views for deriving the test cases for conformance checking of the code with respect to architectural view constraints. Figure 4-5 presents the generic workflow for ACAMBT. In the following we discuss the process in more detail.

Figure 4-5: Process model of architecture conformance analysis using model-based testing

Similar to the earlier reflexion modelling process as shown in Figure 4-1 the first step consists of developing the architecture views. The architecture views are the models that will be used to eventually generate the required test cases for conformance checking for each architecture view constraints. Different architecture view models can be selected including module views, component and connector views and allocation views [3]. Figure 4-4 shows, for example, the layered view of the adopted case study. The activity Create View Criteria defines the architectural constraints for the corresponding architecture viewpoint. Every viewpoint will impose different constraints for the conformance with the code. For example, for the given layered view of the e-government system, the Layered View criteria implies that layers exist in the code, and the layered property constraints are respected. The latter means that modules in Client Layer can only access, modules in the Service Layer, and modules in the Service Layer are only allowed to access modules in the Data Storage Layer. Similarly, for example, conformance checking based on the decomposition view with the code will require checking whether the code has the same decomposition as defined in the architecture decomposition view. Conformance checking based on the uses view will require checking the architectural uses relations in the code. Hence, specific criteria will be defined for each architecture viewpoint.

Once the architecture views and the corresponding view criteria have been defined, in the next activity Create Test Model for Architecture View a test model for the view will be developed that will be used to generate the concrete test suite. The concrete test suite will include test cases dedicated for the corresponding view to check the potential inconsistencies in the code. The concrete test suite as such will be executed on the code. In case of inconsistencies the architect or the developer will be notified. For example, based on the Layered View constraint that the modules in the Client Layer can access only modules in the Service Layer, a test case needs to be created that checks whether this is the case in the code. Similarly, other test cases need to be defined for the criteria of the corresponding views.

4.5 ACAMBT CONFORMANCE CHECKING TOOL FRAMEWORK

In the previous section, we have provided a generic process for ACAMBT which is agnostic to the adopted tools. In principle, the process can be defined using different tool implementations. In this section, we describe the ACAMBT Checking Tool framework 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 [58]. A snapshot of the tool framework is shown in Figure 4-6. In the following subsections we explain the activities of the overall process and the adopted tools.

Figure 4-6: Snapshot of the toolset that implements the approach

4.5.1 Modelling Architecture Viewpoints

For modelling the architecture viewpoints, we have first defined the corresponding metamodels for the viewpoints and then mapped these to domain specific languages using the Eclipse Ecore model [6][61][10]. The metamodels of the selected viewpoints are shown in Figure 4-7.

Decomposition Viewpoint Metamodel

Shared Data Viewpoint Metamodel

Uses Viewpoint Metamodel

Generalization Viewpoint Metamodel

Layered Viewpoint Metamodel

Figure 4-7: Metamodels of the selected viewpoints

These metamodels are represented as Ecore models in the Eclipse IDE. Each of these metamodels have been developed after a thorough domain analysis on the corresponding viewpoints. For this we have analysed the viewpoints as discussed by Clements et al.[5].

4.5.2 Modelling Architecture Views

For modelling the architecture views for a particular case, we use the Human-Usable Textual Notation (HUTN) which is an OMG standard for storing models in a human understandable format. An example of the decomposition view is shown in Figure 4-8. In it, a system is illustrated that consists of three sub-modules X, Y and Z. The module X consists further of sub-modules X1 and X2, while the module Y consists of sub-modules Y1, Y2, Y3, and Z is not further decomposed. Architecture view models can also be modelled via Ecore diagram interface.

```
decomposition {  
  Model "decomposition"{  
    elements :  
      Module "S"{  
        name : "s"  
        subelements : Module "X", Module "Y", Module "Z"  
      }, Module "X"{  
        name : "s.x"  
        subelements : Module "X1", Module "X2"  
      }, Module "X1"{  
        name : "s.x.x1"  
      }, Module "X2"{  
        name : "s.x.x2"  
      }, Module "Y"{  
        name : "s.y"  
        subelements : Module "Y1", Module "Y2", Module "Y3"  
      }, Module "Y1"{  
        name : "s.y.y1"  
      }, Module "Y2"{  
        name : "s.y.y2"  
      }, Module "Y3"{  
        name : "s.y.y3"  
      }, Module "Z"{  
        name : "s.z"  
      }  
    }  
  }  
}
```

Figure 4-8: Example HUTN model for the decomposition view

4.5.3 Modeling Architecture Viewpoint Criteria

As stated before, each architecture viewpoint will impose a different set of constraints for conformance checking. Table 4-1 lists the set of constraints for the selected set of viewpoints.

Table 4-1: Identified conformance checking constraints for the selected viewpoints

Viewpoint	Constraints
Decomposition	<ul style="list-style-type: none"> • Does every element in the view model appear in the code? • Does every sub-element in the view model appear in the code? • Does every sub-element exist under the corresponding element in the code?
Uses	<ul style="list-style-type: none"> • Does every uses relation in the view model appear in the code? • Does every source element appear in the code? • Does every target element appear in the code?
Generalization	<ul style="list-style-type: none"> • Does each implementation or inheritance relation exist in the code? • Does each relation child and parent element exist in the code? • Are each relation's child and parent element in the model have same child's and parent's in the code?
Layered	<ul style="list-style-type: none"> • Does each layer in the relation given in model exist in the code? • Does each allowed to use below relation in given model exists in the code? • Does each layer only use functions in the next lower layer?
Shared Data	<ul style="list-style-type: none"> • Does each data accessor exist in the code? • Does each attachment of data accessor exist in the code?

To specify these criteria for architecture viewpoints we have applied the Epsilon Generation Language (EGL) [61] which is a template-based model-to-text language for generating code, documentation and other textual artefacts from models. View criteria are defined for each view and used in the construction of EGL templates. An example of an EGL template that implements the constraints for the decomposition viewpoint is shown in Figure 4-9. The EGL template checks the three constraints for the decomposition viewpoint as shown in Table 4-1. As it can be seen from the EGL template each constraint is implemented as an assertion. Since we have selected 5 viewpoints, we have thus 5 EGL templates where each EGL template implementing the constraints of the corresponding viewpoints.

```

public class TestDecomposition {
    [%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));
            }
        [%}%]
    [%}%]
}

```

Figure 4-9: EGL Template for conformance checking constraints of the decomposition viewpoint

4.5.4 Creating Test Model for Architecture View

So far, the architecture views have been defined as HUTN models and the corresponding constraints as EGL templates. These two inputs will be used to generate the concrete test suite for each view. For this we use the Epsilon Generation Runner (EGX), which is a runner for EGL templates that can parameterize the defined templates in which transformation rule is defined. An example of the EGX model for the decomposition view is shown in Figure 4-10. The model takes as input decomposition which represent the decomposition view and executes the template Decomposition2Unit.egl template of Figure 4-9. The output is written to the file “gen/TestDecomposition.java” and defines the concrete JUnit test suite for the decomposition view.

```

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

```

Figure 4-10: EGX Model incorporating rules for generating test cases for the decomposition viewpoint

4.5.5 Creating Concrete Test Suite for Architecture View

The execution of EGX file generates a single JUnit test case which can be executed automatically. An example concrete test suite for the decomposition view is shown in Figure 4-11.

```
public class TestDecomposition {

    @Test
    public void testSDecomposedOfX() {
        String decomposedPackageName = "s";
        String subPackageName = "s.x";
        Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(decomposedPackageName), decomposedPackageName));
        Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(subPackageName), subPackageName));
        Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(decomposedPackageName), subPackageName));
    }

    @Test
    public void testSDecomposedOfY() {
        String decomposedPackageName = "s";
        String subPackageName = "s.y";
        Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(decomposedPackageName), decomposedPackageName));
        Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(subPackageName), subPackageName));
        Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(decomposedPackageName), subPackageName));
    }

    ...
}
```

Figure 4-11: Example Junit Test Suite (part) for the Decomposition Viewpoint

4.6 ARCHITECTURAL MODELS FOR THE E-GOVERNMENT SYSTEM

To support the communication with the stakeholders, guide the overall development, and analyse the system, architecture documentation has been provided that includes descriptions of several architecture views. Figure 4-12 shows, for example, the deployment view of the system. There are integrated web-services and client applications that are connected to the server side of the application. Our approach is applied on the infrastructure of the server-side implementation.

Figure 4-12: Deployment view of the adopted project

Figure 4-13 shows the shared data view of the project. The system has two database accessors named PM and QM whereby 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 4-13: Shared data view of adopted project infrastructure

The decomposition view of the system is shown in Figure 4-14. The decomposition view is similar to that shown in the figure but, for confidentiality, the names of the modules have been omitted. The adopted project is composed of one main package A. A is further divided into sub-modules B, C, D, E, F and G. Furthermore, B is decomposed into B1, B2 and B3. Likewise, C

is decomposed into C1 and C2. D is decomposed into three parts D1, D2 and D3 where D1 is decomposed into one more part called D11.

Figure 4-14: Decomposition view of adopted project infrastructure

The uses view of the system is shown in Figure 4-15. Here the arrows represent a uses relation that defines the dependency of a module to the used module.

Figure 4-15: Uses view of the adopted project infrastructure

Figure 4-16 shows the layered view of the system. The adopted project has three main layered views. B1 is allowed to use B2 and B3, D3 is allowed to use E, F is allowed to use C, C is allowed to use C1 and C2.

Figure 4-16: Layered view of adopted project infrastructure

In the system we have also defined the generalization view to show the generalization specialization relations among the modules in the system. Figure 4-17 shows, for example, the generalization view of package B. The rounded squares are either interfaces or classes. In the figure, three types of generalizations are shown including 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. Similar to module B we have also defined generalization views for other relevant packages which are not shown due to space constraints. In the figure blue nodes signify interfaces and black nodes signify classes. Moreover, the dashed arrows are implementation relation and straight lines are extension relations. The naming convention for the nodes is that nodes with just one number are roots and nodes with other numbers concatenated onto that number are children.

4.7 CASE STUDY DESIGN

To validate our ACAMBT approach we have adopted the case study empirical evaluation protocol as discussed by Runeson and Höst [77]. The protocol consists of the following steps: (1) case study design (2) preparation for data collection (3) execution with data collection on the studied case (4) analysis of collected data (5) reporting. Table 4-2 presents the case study design steps for the selected cases study.

To validate our ACAMBT approach we have adopted the case study empirical evaluation protocol as discussed by Runeson and Höst [77]. The protocol consists of the following steps: (1) case study design (2) preparation for data collection (3) execution with data collection on the studied case (4) analysis of collected data (5) reporting. Table 4-2 presents the case study design steps for the selected cases study.

Table 4-2: Case Study Design

Case Study Design Activity	Case Study
Goal	Assessing the effectiveness and practicality of adapted ACAMBT approach
Research Questions	RQ1: How effective is the adopted ACAMBT approach? RQ2: How practical is the adopted ACAMBT approach?
Background and source	Software Engineers (1 st degree) Meetings and interviews (1 st degree) Technical documents and reports (2 nd degree) Source Code (2 nd degree) Official documents (3 rd degree)
Data Collection	Direct data collection through semi-structured interviews, meetings and survey (mix of open and closed questions) Independent data collection based on document analysis (the papers and technical reports) Indirect data collection based on source code analysis.
Data Analysis	Qualitative Data Analysis using Bar Charts Quantitative Data Analysis using Tables

The case study design approach is in the category of the applied research type and as such the primary purpose is to understand the impact of the adoption of ACAMBT for conformance analysis within the real industrial context. Two research questions have been defined. RQ1 relates to how effective the approach is for conformance analysis, while RQ2 aims to assess the practicality of the approach. The effectiveness of the proposed approach is calculated by the number of inconsistencies discovered in the implementation of the system. To this end, we have applied the approach on real code to detect the inconsistencies with the architectural constraints. Further we have also adopted fault injection techniques to detect potential bugs that could

appear in the code [3][74][80][81]. For assessing the practicality of approach, we have applied surveys with the team leaders and software engineers and presented the results of the survey. As shown in Table 4-2 our 1st degree information sources are software developers, meetings and interviews. We also analysed technical reports, technical documents along with the source code as 2nd degree information source. Finally, we analysed official documents provided by the company to other stakeholders as 3rd degree information source. Table 4-2 also shows the data collection methods from the information sources. We have conducted semi-structured interviews, meeting and surveys with the software developers. Then, we independently analysed technical reports and official documents. Finally, the results of the previous steps have been reported.

4.7.1 How effective is the adopted ACAMBT approach?

4.7.1.1 Existing Architectural Compliance

To assess the effectiveness of the adopted ACAMBT approach we performed data collections for constructing architecture models for the viewpoints which will be an input to adapted approach. For this, we analysed paper and technical reports for the projects architecture models and references which are maintained throughout the project development. The architecture was documented separately in several reports. We performed semi-structured interviews with technical leads and server-side software architects. These analyses led to the architecture presented in previous sections. Subsequently, we analysed the project's source code, and finally we derived the abstract architecture models of the current code that would be checked against the earlier defined architectural models. In alignment with the steps of the approach we used these derived architecture models to generate test cases for each viewpoint. The generated test cases have been executed on the implemented system and the results have been recorded.

Table 4-3 shows the overall summary of the test execution result for each viewpoint. As it can be observed from the table, depending on the defined view the number of test cases differs. In total 126 test cases were defined and only 2 tests failed. The tests for shared data, decomposition and generalization viewpoints have executed successfully. However, we found inconsistencies between implementation and architecture for uses and layered viewpoints.

Table 4-3: Testing results of the real code for each viewpoint

Viewpoint/Test Result	Total possible tests	# of Test Passed	# of Test Failed
Shared Data Viewpoint	3	3	0
Decomposition Viewpoint	15	15	0
Uses Viewpoint	22	21	1

Layered Viewpoint	6	5	1
Generalization Viewpoint	82	82	0
Total	128	126	2

4.7.1.2 Fault Injection-Based Testing

Besides testing the real existing code as-is, we have also applied fault-based testing techniques in order to validate our adapted approach. Software testing is labelled as fault-based testing when it aims to demonstrate of absence of pre-defined faults [73][74]. In order to assess our test cases, we created a mutant copy of our case study with relevant injected faults. A mutant copy of a program is a copy of the program seeded with mutations which are structural changes/bugs [23]. We have run the generated test cases on the mutant copy and aimed to kill all mutants, which implies that at least one test case catches the bug. Killing all the mutants also implies that the test case is effective at finding real life defects. We introduced the following categories of faults as our mutation operations:

1. Absence relations: removal of expected relation
2. Divergence relations: altering a relation

As we have discussed before, each viewpoint includes its own set of criteria, and likewise the absence and divergence relations will be based on these defined criteria. Different strategies exist in mutation testing which are based on first order mutants and high order mutants. Jia et al introduced higher order mutation testing in which mutation operators are applied more than once [63]. In our case, this concept is inapplicable as our operators are conflicting with each other as one cannot remove and alter the same relation. Therefore, we applied first order mutation testing.

Since the number of architectural elements is manageable, we applied exhaustive testing. Hence, we applied fault injection for checking the absence and the divergence of each architectural element in each view. Examples of fault injection for the earlier defined architecture views are shown in Table 4-4 . The pseudo code for generating faults in the code based on the elements in the architecture views is shown in Table 4-5.

Table 4-4: Example Fault-Injections for the adopted viewpoints

Viewpoint	Fault Injections
Shared Data Viewpoint	<ul style="list-style-type: none"> PM_WRITE and PM_READ relations are removed from the implementation of the case study. QM component is removed from the implementation.
Decomposition Viewpoint	<ul style="list-style-type: none"> Package F is removed from the content of package A in the implementation. Package E is removed from the implementation.
Uses Viewpoint	<ul style="list-style-type: none"> Package C is removed from the implementation. Uses relation of B3 to F is removed from the implementation. Uses relation of E to D2 is removed from the implementation.
Layered Viewpoint	<ul style="list-style-type: none"> Uses relation B1 to B2 is removed from the implementation (hence a violation of the expected allowed-to-use-relation) Uses relation from package C to package F is introduced to the implementation of the case study (and hence a violation of layered property that does not allow call backs).
Generalization Viewpoint	<ul style="list-style-type: none"> B1_6 component is removed from the implementation Inheritance by extension relation between B_1 and B3_1 is removed from the implementation Inheritance by implementation relation between B2_7 and B2_8 is removed from the implementation.

Table 4-5: Pseudo codes for the fault injections in the architecture views

for each attachment in sharedData.attachments remove attachment.da from code endfor for each attachment in sharedData.attachments if attachment.type == DataRead remove attachment.dataRead from code endif if attachment.type == DataWrite remove attachment.dataWrite from code endif endfor	<i>for each element in decomposition.elements</i> for each subelement in element.subelement change subelement hierarchy in code endfor endfor <i>for each element in decomposition.elements</i> for each subelement in element.subelement remove subelement from the code endfor endfor
for each relation in uses.relations remove relation from code endfor for each relation in uses.relations alter relation in code endfor for each relation in uses.relations remove relation.source from code endfor for each relation in uses.relations remove relation.target from code endfor	<i>for each relation in layered.relations</i> remove relation.source from code endfor <i>for each relation in layered.relations</i> remove relation.target from code endfor <i>for each relation in layered.relations</i> remove relation from code endfor <i>for each relation in layered.relations</i> alter relation in code so that relation.target uses relation.source endfor
for each declaration in generalization.declarations remove declaration.parent from code endfor for each declaration in generalization.declarations remove declaration.child from code endfor for each declaration in generalization.declarations remove declaration.inheritance from code endfor for each declaration in generalization.declarations if declaration.type == Extension alter declaration.child to implement declaration.parent in code endif if declaration.type == Implementation alter declaration.child to extend declaration.parent in code endif endfor	

After injecting the faults, we applied the conformance analysis approach and checked the results. We have injected every possible fault with respect to the architecture view criteria defined, to ensure that our approach will detect every possible bug that we expect to detect (ones that are presented in architecture model but not presented in code level implementation). Table 4-6 shows the fault-based testing results for each viewpoint. As it can be seen from the table all the injected faults are detected successfully by our ACAMBT approach. After testing the real code and this fault-based testing approach we can state that the approach is thus effective.

Table 4-6: Fault-based testing results for each viewpoint

Viewpoint/Fault Detection	# of Faults Detected	# of Faults Not Detected
Shared Data Viewpoint	5	0
Decomposition Viewpoint	32	0
Uses Viewpoint	78	0
Layered Viewpoint	16	0
Generalization Viewpoint	77	0

4.7.2 How practical is the adopted ACAMBT approach?

Within an industrial context it is important that the adopted approach and the corresponding tool is also practical in use. To validate this, we have applied designed and conducted a survey study based on the protocol and the guidelines of Kitchenham's et al [66][67][68][69][70]. We discuss this in the following sub-sections.

4.7.2.1 Survey Study Design

Table 4-7 shows the description of the survey design. For the survey form we decided to select an experimental design as we conducted experiments using the adapted approach with survey takers.

In experimental studies five different study types are defined including concurrent control studies in which participants can be randomly assigned to groups, concurrent control studies in which participants are not randomly assigned to groups, self-control studies, historical studies, and studies using a combination of techniques. Self-control studies are a type of experimental survey form where both pre- and post- treatment measures are used for obtaining information. For our case, self-control studies were feasible since we wanted to assess the practicality of the adapted approach with comparison to past practices in the company for architecture-based testing, if any. The data requirement for this survey study is the research question for our case study which aims to assess the practicality of adapted ACAMBT approach. The survey study was conducted in the company and our population size was small. There are 59 employees in the company, and the employee background varies from technical to non-technical. Different

projects are being implemented within the company. In each project, employees from different backgrounds work on different aspects of the project. Therefore, we defined our population exclusion criteria according to those stated above. The population of the company consists of managers, assistants, non-technical staff, software developers and software development team leaders. The defined exclusion criteria include (a) employees who do not work in adopted project (b) employees who do not have computer science background and (c) employees are not involved in software development.

Table 4-7: Survey Study Design

Survey Study Activities	Survey Study
Survey Form	Experimental – Self-control Studies
Data Requirements	Practicality of adapted ACAMBT approach
Population	Managers, Assistants, HR Staff, Other Non-Technical Staff, Software Developers and Software Development Team Leads
Population Selection Criteria	Exclusion Criteria: EC1: Employees who do not work in the adopted project EC2: Employees who are not involved in software development EC3: Employees who do not have computer science background
Sampling Technique	Non-Probabilistic Sampling – Convenience Sampling
Sample Size	7
Data Collection Technique	Interview

Table 4-8 shows the overall population and population selection after each exclusion criteria application. At the remaining sample, we applied convenience sampling, which is one of the non-probabilistic sampling methods, if any of the resulting population unavailable for survey taking. The sampling technique established sample size is identified as 7 in which two software development team leads and five software developers reside. As convenience sampling is applied, we had high response rates and meaningful answers from the volunteer survey takers. Team leads have a good insight of modelling and analysing software architecture and had worked on different projects. Four out of the five software developers were senior level developers and also had a good insight in designing software architectures. The remaining developer was working on the current project for two years and had an average knowledge about software architecture.

Table 4-8: Overall Population Size and Selection

Population	Total Population Size	Population Size After Applying EC1	Population Size After Applying EC2	Population Size After Applying EC3	Population After Sampling
Company	59	18	13	9	7

4.7.2.2 Data Collection

Self-control studies depend on pre- and post- studies and are intervened by exposure to the approach between studies.

For the data collection the following protocol was followed:

First a meeting was scheduled with the survey takers for the initial interview. The goal of this interview was to capture the initial thoughts and experience on ACAMBT adoption.

In the second step, we gave a short presentation about the goal of the adapted approach.

In the third step engineers applied the ACAMBT approach with the researcher on the government system, with the models created by the researchers.

In the fourth step, engineers took a survey to assess the practicality of the approach. To collect data according to pre- and post- questionnaires defined below, we used online free survey tools.

For the pre-survey, we have defined two questions regarding the participant's initial knowledge and willingness to adopt the provided approach. The questions were the following:

What do you know about architecture-based testing?

Do you apply architecture-based testing techniques?

The survey was conducted on seven subjects according to the exclusion criterion applied on working staff on adopted project's company. With the help of pre-survey, the subjects were classified according to their knowledge level on architecture-based testing. According to the answers of the subjects, it was found that five of the subjects were not familiar with the architecture-base testing concept, while two of them have a general idea about the concept. However, none of the subjects had detailed knowledge about the concept. Moreover, the subjects have never experienced the application of architecture-based testing techniques. The survey took much more time than it was estimated, since none of the subjects were familiar with the concept. On the other hand, all of them were very eager to contribute to the study and positively welcomed the suggested approach.

Table 4-9 shows the design of the post-survey questionnaire. The questions were grouped as input, execution, output practicality and general. For each question group, we have defined three purposeful and concrete questions. As suggested by Kitchenham's the questions are not contradicting and positively constructed as it is harder to understand negative questions.

Further, the survey has both open and closed questions. Input, execution and output practicality groups answers are hard coded as strongly agree, agree, disagree and strongly disagree as suggested in [68]. The last group includes open questions for which answers could be given. These answers to these questions were recorded and later on processed. The interpretation of the answers was carried out together by both the researchers. The results of the questions Q1 to Q9 of the survey are given in Table 4-10 and visualized as bar charts in Table 4-11. The answers to questions Q10 to Q12 are shown in Table 4-12.

Table 4-9: Post-Survey Questionnaire

		Strongly Agree	Agree	Disagree	Strongly Disagree
INPUT PRACTICALITY	Q1. Input architecture models are easy to understand				
	Q2. Input architecture models are easy to create				
	Q3. Input architecture models are easy to validate				
EXECUTION PRACTICALITY	Q4. Approach is easy to use				
	Q5. Approach is understandable				
	Q6. Approach is useful and can be applied in practice				
OUTPUT PRACTICALITY	Q7. Approach can detect defects				
	Q8. Approach generates meaningful test cases				
	Q9. Approach generates test cases quickly				
GENERAL	Q10. What are the advantages of adapted ACAMBT approach?				
	Q11. What are the disadvantages of adapted ACAMBT approach?				
	Q12. What are the interesting points in adapted ACAMBT approach?				

Table 4-10: Results of the Post-Survey Questionnaire

Subject No	Q1 Answers	Q2 Answers	Q3 Answers	Q4 Answers	Q5 Answers	Q6 Answers	Q7 Answers	Q8 Answers	Q9 Answers
1	Agree	Agree	Disagree	Strongly Agree	Agree	Strongly Agree	Strongly Agree	Agree	Agree
2	Disagree	Disagree	Strongly Disagree	Agree	Agree	Agree	Agree	Strongly Agree	Agree
3	Agree	Agree	Agree	Disagree	Agree	Strongly Agree	Agree	Agree	Agree
4	Agree	Disagree	Agree	Strongly Agree	Disagree	Agree	Strongly Agree	Strongly Agree	Agree
5	Agree	Agree	Agree	Strongly Agree	Agree	Strongly Agree	Agree	Strongly Agree	Agree
6	Disagree	Disagree	Disagree	Disagree	Disagree	Disagree	Agree	Agree	Agree
7	Agree	Agree	Agree	Agree	Strongly Agree	Agree	Agree	Agree	Agree

From Table 4-11 we can see that the answers for the questions for practicality are in general positive. For the questions 1 to 3 relating to input practicality the majority of the participants agreed that the architecture models were understandable, and the validation of the architecture models was indeed practical. The creation of architecture models was also perceived somewhat practical, while three participants found it less practical, four participants did find it practical. It should be noted that those who found it practical were also the more experienced employees. Questions 4 to 6 relate to execution practicality, and from Table 4-11 we can derive that the

majority of the participants indicated that the approach is easy to use, understandable and useful. Questions 7 to 9 relate to the output practicality, and from Table 4-11 we can derive that all participants indicated that the approach can effectively detects defects, can generate meaningful tests, and generate these tests quickly.

Table 4-11: Results of the Post-Survey Questionnaire represented as bar charts

Table 4-12: Results of the Open Questions in the Post-Survey Questionnaire

Subject No	Q10 – Advantages: Answers	Q11 – Disadvantages: Answers	Q12 – Interesting: Answers
1	It is fast and accurate. It can actually detect defects	Need to convert already existing UML models to another model to generate test cases	Defining models for different architecture viewpoints so that you can test them individually
2	It is useful. It detects bugs and shows the difference between architecture and implementation	UML models are not accepted. It cannot be accessed by its own executable program.	Viewpoints are modelled so that each can be separately tested. We can conform the architecture on the documents to implemented code.
3	It is easy to apply. It can be integrated into any test environment as it simply uses JUnit library.	Needs programming background to apply. Need to update software architecture model in case of design change.	Uses standard java libraries and reflection libraries to conform the architecture and the implemented code itself.
4	It generates meaningful test cases in software design perspective. We did not have any tool or methodology to cover this concern as the code itself changes constantly.	Cannot use existing architecture models, needs overwork to translate UML models to approach's viewpoint based models.	It is a new methodology that I'm facing and it has its cost and benefits as UML models are not accepted but before the delivery of product the design document can be verified
5	Generated test cases cover for architecture conformance issues in detail as it uses viewpoints in isolation.	It uses Eclipse environment but it can have its own executable. You have to open Eclipse each time you want to generate.	I have never used a tool in Eclipse Epsilon environment.
6	Conforming architecture to code in every phase of development is crucial, if a defect in early stages of development is not detected it will cause a lot of damage.	Does not accept predefined and well-known models for software design.	Input architecture models are semi-structured format such as JSON or xml and can be edited by simple notepad program.
7	Tests the architecture against code is crucial. The generated files are actually executing on code as simple test cases. Fast and understandable approach.	Extra work need to be done to maintain approach based architecture models. Changing formal model and approach model can be costly.	Can be improved by adapting new viewpoints to the approach. Adapting new viewpoints are very easy as to create model and write translation templates for generation. Approach takes the rest of the work.

What are the advantages of adapted ACAMBT approach?

The advantages of the approach can be categorized according to the results of the survey. As the first and more common advantage according to the results, subjects highlight the accuracy and functionality of the system. They are satisfied with defect detection functionality of the system. They agreed on that the system detects bugs and shows the difference between architecture and implementation. They found the generated test cases meaningful from a software-design perspective. Additionally, it is considered as an advantage for the system that generated test cases cover the architecture conformance issues in detail as it uses viewpoints in isolation. Moreover, the system tests the architecture against code, and conforms the code in every phase of development which is interpreted by the subjects as an asset of the system. The subjects are emphasizing the opportunity cost of an early-detected defect.

In terms of usability, the subjects find the system easy to understand and easy to use. In terms of accessibility, the subjects highlight that the system can be integrated into any test environment as it simply uses JUnit library. In terms of availability, subjects are satisfied with the speed of the approach when it is compared to traditional methods. Additionally, they found the system original while emphasizing the lack of such a system in the company. They accent the importance of such a system especially in an environment when changes to the system occur frequently.

What are the disadvantages of adapted ACAMBT approach?

In general, the subjects are concentrated on two main disadvantages of the system. The most mentioned disadvantage of the system is that the system is not compatible with UML models. The subjects use UML models in the adopted project they work on. Therefore, the lack of integration with UML models forces the subjects to rework their models to convert them into the intended viewpoint-based models by the approach, which means additional time and cost to the subject. Additionally, the subjects underlined the configuration management challenges of the tested program. Since the approach cannot work with UML models, it brings the necessity of using two-or-more models in parallel, which makes configuration management of the code open to individual errors and inadvertencies. As the second disadvantage, the subjects pointed out that the approach would not work without an executable program. The program uses the Eclipse environment, which makes the execution process of the program complex and forces the subjects to have a programming background.

What are the interesting points in adapted ACAMBT approach?

The subjects mentioned as a positive aspect that the program makes it possible that different viewpoints of the architecture can be separately tested and can be concentrated on. With the help of this functionality, the architecture can be conformed on the design documents without waiting for the entire implementation phase of the code. Thus, defects of the program can be detected at earlier phases of the adopted project, and design can be updated accordingly without extra effort. The program is open to development because it is easy to integrate new viewpoints to the program. As the suggested approach uses open-source standard java libraries for testing, it is easy to configure and run on the system. Finally, the ease of editing and simplicity of viewpoint models were remarked upon by subjects.

Summary

When three aspects of the practicality are analysed, output practicality is the strongest feature of the approach according to the survey results. The approach has some features which are open to development in terms of input practicality, while the approach is good enough in terms of execution practicality.

4.8 DISCUSSION

Our ACAMBT approach has been evaluated both with respect to effectiveness and practicality. To assess the effectiveness of our approach, the architecture and the code of a large industrial case has been used for testing architecture-code conformance. We can state that the approach was effective in checking code conformance to the specified architecture. It executes successfully on a large system and presents plausible results.

To assess the practicality of our approach we have conducted a survey study according to protocols we defined with respect to Kitchenham's et al [66]. In our survey study, we have conducted both a pre-operation survey and post operation survey where operation denotes the execution of the approach with participants and detailing the mentality and logic behind the suggested approach. In our pre-survey, we have seen that participants never used any ACAMBT tools, on the other hand two of the participants appeared to be familiar with the concept. Then in our post-survey, we have assessed the practicality of our approach for three categories including input, execution and output practicalities.

Regarding input practicality we have assessed whether the architecture is easy to understand, to create and to validate. Model formation, that is, proper architecture modelling is essential for the success of the approach. Although some effort is needed for architecture modelling, the creation and understanding of architecture models were largely found easy, the validation of the architecture appeared to be more difficult. In general, based on our survey study with the practitioners in the project it appeared that the approach was found practical and that the tool would be used as part of the overall testing effort.

Similar to any case study research our study also has some validity threats. Construct validity refers to demonstration of measuring what the research has in mind and what the study suggests. Our approach focused on testing the drifting of the code from the architecture. That is, architecture view models are validated and verified against the implementation level, but we do not check whether the code is completely drifting away from the architecture. For example, if a developer adds a relation in the code base and that relation is not in the architectural model, the relation will not be identified as an inconsistency. In essence, we consider that our architecture model which is our reference point for testing is both complete and correct. We do aim to focus on architecture drift analysis from both directions in our future research.

Although each of the participant had a mature experience in design, development and testing, the focus on testing architectural constraints in the code was not something they had done before explicitly. For coping with this issue, we organized a meeting where we have first presented the notion of model-based testing, conformance analysis and deriving test cases from architectural descriptions. Since the focus was still on testing, the concepts could be easily grasped. This

might as such not have a serious impact on the overall outcome of the study. Still, the potential bias of the participants could be taken into account to soften this claim.

For the practicality criteria, we have applied survey study before and after applying the approach. In the survey study, designed to assess the practicality of the suggested approach, we singled out construct validities which are presented in Table 4-13.

Table 4-13: Construct Validity

Threats	Counter Measures
Wrong selection of interviewed persons.	To eliminate this threat, we have defined exclusion criteria for sample population and applied Non-Probabilistic Sampling and Convenience Sampling with respect to Kitchenham's et al [11].
Wrong interpretation of questions by interviewed personnel.	Survey questions are prepared using the guidelines defined in Kitchenham's et al [11]. We both provided open and categorized questions.
Number of participants	The total population was analyzed according to the guidelines defined in Kitchenham's et al [11]. We searched within the whole company to identify the participants related to the topics of this study. For this we discussed with higher level managers and analyzed the existing projects. In the end, only participants who could provide relevant insight were selected.
Difference between measured and what was intended to be measured	We have applied fault-based testing techniques to prove that our approach can find the newly presented defects on defect free environment and then we applied our approach.

Internal validity refers to casual relation between treatment and outcome. While assessing the effectiveness of the suggested approach, we have applied formal fault-based testing and reported the results in an isolated environment that no external variable can affect. Additionally, the practicality of the approach is shown by a survey study in which we have provided both open and closed questions to discover the participants explicitly-held and implicitly-held perceptions. Another internal validity threat is the lack of experience in architecture conformance approaches of the participants. Finally, another issue that should be mentioned is the possibility of experimenter effects/the Hawthorne effect. This implies that the participants

could have behaved differently due to the researcher's presence. This Hawthorne effect could have had an impact on different phases in the case study including the modelling of the architecture, the testing process, and even the answers to the questions in the survey study. To reduce the effect of this Hawthorne effect, additional studies could be carried out.

External validity refers to concern of generalizing the results of a scientific study. In our case study, we have applied our suggested approach on a smart e-government project with a large number of transactions. The approach has been applied to one case study with 7 participants and this implies a threat to external validity of the approach. However, we consider the case study as representative for the challenges that many different systems have to deal with when coping with architecture consistency. What really matters in our context is not the number of case studies but the size and complexity of the systems. In our approach we have adopted five different viewpoints, defined the metamodels and domain specific languages for these, implemented the constraints and generated the test cases. The approach itself is generic and could be also used for different viewpoints. As stated before, the system that is being developed can be a large-scale complex system and hence, we could state that we have adopted a sufficiently representative system to support the external validity. Obviously, in practice even larger systems could be identified, but the properties of architecture code consistency will be similar. In addition, to enhance the external validity, although we have one case study, we actually carried out two different experiments including the one with testing the real code, and the other using an exhaustive fault-based testing approach. In our future research we will focus on further applications of the approach.

4.9 RELATED WORK

Neto et al. [21] provides the results of a systematic review on model-based testing approaches. In the literature review 78 primary studies have been selected. The review shows where MBT approaches have been applied, presents the characteristics and limitations of MBT. The selected studies are analysed in terms of application scope of MBT approaches, level of automation, tool support for MBT, models used for test case generation, test coverage criteria, behaviour model limitations, cost and complexity of application of MBT approach. In addition to these, they also discuss the issues regarding MBT approaches and the limitation of MBT approaches. Based on this characterization, our approach is general purpose and can be applied to different application domains. Automation includes the modelling of architecture, test case generation, test case execution and analysis. We have provided tool support that supports these levels of automation. The models used for test case generation include executable models of architecture views and the code. The test coverage criteria relate to the defined architecture constraints. We primarily check static architectural constraints and do not consider behavioural constraints. The cost and complexity of the MBT approach was considered manageable based on the output of the survey study.

Besides testing, software architecture has also been used for model-checking. Model checking is a formal verification technique that aims for the automated analysis of a systems' behaviour with respect to selected properties. It takes as input a system model and a property specification [4]. The output is "true", or false with a counter-example when an error is detected. Zhang et al. [30] provide a classification and comparison of model checking software architecture techniques. The authors describe the main activities in a model checking software architecture

process. Then, a classification and comparison framework are provided which is subsequently used to compare model checking software architecture techniques.

Different approaches are presented for ACAMBT in the current literature. In one of the early studies [59], the authors present a ACAMBT approach using chemical abstract machine (CHAM) for software architecture representation and labelled transition systems (LTS) for test model representation. Jin et al. [64] define software architecture using Wright architectural design language (ADL) and presents testing criteria based on data flow reachability, control flow reachability, connectivity and concurrency. Behaviour graphs are generated from the architecture descriptions and tests are executed on this graph with respect to defined criteria. Muccini et al. [17] presents a continuation study for [59] in which CHAM model is replaced by finite state process (FSP) model and same process of test model of type LTS is generated. Johnsen et al. [65] presents a ACAMBT approach based on architecture analysis and design language (AADL), and Uppaal Model (timed automata). AADL is used for representing architecture and timed automata is used as a test model for generating test cases. Reza et al. [75] presents use of petri nets in ACAMBT in which software architecture is described in Acme ADL, and test model is generated as Hierarchical Predicate Transitions Nets (HPTrNs). Javed et al. [62] authors use UML sequence diagrams to model the dynamic behaviors of software architecture and transform it into xUnit test model from which different types of test cases can be generated such as jUnit, cUnit, cppUnit and etc. All of these studies use different kinds of software architecture description and test models, but none of the studies have taken software architecture viewpoints into consideration. Another way of detecting nonconformance issues is stated in study [57], in which is traceability between the implementation and code is maintained as the software is being produced. Tera et al. [79], authors propose a recommendation system for repairing the inconsistencies found between architecture and code.

4.10 CONCLUSION

Model-based testing is a well-known approach for automating the generation and execution of test cases using models based on system requirements and behaviour. Our approach is one of the few approaches that adopts architecture models to derive test cases. A unique part of our approach is further the generation and usage of the test cases for checking conformance of the architecture with the code for architectural constraints. For preparing the model-based testing for architecture models we first had to define the metamodels and the corresponding domain specific languages. We could do this without serious problems for all the viewpoints that we considered. The approach for mapping the architecture viewpoints to the domain specific languages is general and can be also applied to other viewpoints. Our approach builds on and refines the generic MBT process. We have implemented the corresponding toolset with which architecture views can be specified, the generators can be developed, and the test cases generated and tested. As a result, both the approach and the corresponding toolset provides an additional practical toolset for the testing and the architecture code conformance.

We have adopted a real industrial case study and evaluated our approach based on the guidelines of a systematic case study design protocol that we planned and executed carefully. We evaluated the effectiveness of our approach both on the real architecture and the code. Further we also evaluated the practicality of the approach using a survey study with the real practitioners. From the overall evaluation we can conclude that the approach and the corresponding toolset is both

effective and practical. In our future work we aim to further enhance the toolset, apply it for other case studies, develop test cases for other viewpoints, and focus on the integration of the tool with the conventional development tools.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their valuable and constructive comments that greatly helped to improve the quality of the chapter.

5

DOMAIN-DRIVEN ANALYSIS OF ARCHITECTURE RECONSTRUCTION METHODS¹

¹This chapter is based on following published paper:

Uzun, Burak, and Bedir Tekinerdogan. "Domain-driven analysis of architecture reconstruction methods." *Model Management and Analytics for Large Scale Systems*. Academic Press, 2020. 67-84.

Abstract

Software architecture reconstruction (SAR) is a process which aims to obtain the architecture information of any system using various sources. These sources contain documentations, logs, codes and stakeholder's concern. Various automated and manual approaches have been proposed in the literature for SAR. Manual architecture reconstruction is usually time consuming, costly, error prone and complex process. Hence different studies in the literature addresses these problems by proposing automatized methods and tools for SAR. In this chapter we present a systematic characterization of the published architecture reconstruction methods. We apply a domain analysis process in which we first define the generic domain model of architecture reconstruction. The model is represented as a set of key terms, a generic business process model and a feature diagram that represents the common and variant features of architecture reconstruction. We also present the method for deriving concrete architecture reconstruction methods from the generic domain model. We illustrate our approach for deriving two different concrete architecture reconstruction methods.

Keywords: Domain Analysis, Feature Modelling, Architecture Reconstruction Methods

5.1 INTRODUCTION

One of the key artefacts in the software development life cycle process is the architecture design that represents the gross-level structure of the system. Architecture model is important for supporting the communication among stakeholders, for analysis of the design decisions, and for guiding the organizational processes [26]. Software architecture can be prescriptive in which it defines how detailed design and code should be structured. Software architecture can also be descriptive in which it reflects the structure of current design and code artefacts [32]. Very often the architecture of a system is not existing or needs to be reconstructed due to the code that has evolved. In this context SAR methods can be used to reconstruct and obtain the architecture information of the system using various sources such as documentations, logs, code, etc. [86].

Various different approaches have been proposed in the literature for architecture reconstruction. Since manual handling of the architecture reconstruction process is usually time consuming and costly, automatized methods and tools are proposed by different studies in the literature. In this chapter we present a domain-driven survey of the published architecture reconstruction methods. For this we apply a domain analysis process in which we first define the generic domain model of architecture reconstruction. The domain model is represented as a set of key terms, a generic business process model, and a feature diagram that represents the common and variant features of architecture reconstruction. We also present the method for deriving concrete architecture reconstruction methods from the generic domain model. We illustrate our approach for deriving two different concrete architecture reconstruction methods.

The remainder of the chapter is organized as follows. In Section 5.2, we discuss the background for architecture reconstruction and feature driven domain analysis by presenting the conceptual model for it. Section 5.3 presents the domain model of software architecture reconstruction methods. Section 5.4 presents the derived feature and business process models for two case

studies. Section 5.5 presents the related work and section 5.6 presents the discussion. Section 5.7 concludes the chapter.

5.2 PRELIMINARIES

5.2.1 Software Architecture Reconstruction

Software architecture reconstruction is a reverse engineering process which aims to obtain the architecture information of any system using existing other source artefacts [86]. Figure 5-1 shows a conceptual model for architecture reconstruction.

Figure 5-1: Conceptual Model for Architecture Reconstruction

An architecture reconstruction process uses different source artefacts which are input SAR process such as code, detailed design, logs, documentation and stakeholder's concern. Two basic motivations for SAR can be identified, that is, missing or incomplete architecture documentation, or an architecture drift due to evolution of the requirements. Since manual handling of architecture reconstruction is usually time consuming and costly, fully or semi automatized methods and tools are implemented. The result of the SAR process is an architecture documentation that includes a description of a set of architecture views for addressing stakeholder concerns [104]. An architecture view conforms to an architecture viewpoint. Architecture views can be visual or textual [6].

5.2.2 Domain Analysis

Several domain analysis processes have been defined in the literature but in general these include the steps for (1) domain scoping, and (2) domain modelling. Figure 5-2 shows the overall domain analysis process that we will adopt.

Figure 5-2: Common structure of domain analysis methods

In domain scoping it is explicitly described what will be considered in the domain and likewise the scope will be determined. This will include also the selection of the primary studies that will be used to analyse the domain. In the domain modelling process, usually a commonality and variability analysis process are carried out to derive a domain model. With the commonality and variability analysis process the identified primary studies are simultaneously analysed to identify common and variant features of a domain [51]. A feature model is a model that defines features and their dependencies. A feature is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate between. Feature models are usually represented in feature diagram (or tables). A feature diagram is a tree with the root representing a concept (e.g., a software system), and its descendent nodes are features. Various

relationships between a parent feature and its child features (or sub-features) are defined including mandatory, optional, or, and alternative. A feature configuration is a selected set of features which describes an instance of the feature diagram. Cross-tree constraints further restrict the possible selections of features to define configurations. The most common cross-tree constraints are requiring and excludes.

5.3 DOMAIN MODEL OF ARCHITECTURE RECONSTRUCTION METHODS

In this section we present the results of the domain analysis process for SAR methods. This process resulted in the domain model which consists of a feature diagram and a generic business process model. In section 5.3.1, we present the adopted scope of the domain that we have used to derive the domain model. Section 5.3.2 presents the feature diagram for SAR methods. Finally, section 5.3.3 presents the generic business process model.

5.3.1 Domain Scoping

As discussed before, the domain scoping defines what will be considered in the domain and likewise the scope will be determined. Our key concern in this study is architecture reconstruction and in particular we focus on the process and the corresponding tools. In the scoping process we have used different reputable and most cited libraries including IEEE Explore, ACM, Science Direct, Springer, WebOfKnowledge and Wiley. In these platforms we have searched for papers related software architecture reconstruction. We have created search strings including the terms “software architecture reconstruction” and “software architecture recovery” in abstract of papers. After a careful analysis of the selected papers according to exclusion criteria we have derived the list of papers as shown in APPENDIX-E Primary Studies. We have thoroughly read these papers and applied the commonality and analysis for deriving the common and variant concepts related to SAR. The result of the data extraction process was described using a domain which is explained in the next sub-section. Table 5-1 provides our study exclusion criteria for study selection process. We further filtered out the query results according to these defined criteria. Table 5-2 presents the number of study selections for after applying search queries and exclusion criteria. 17 studies were singled out of 652 studies after applying the exclusion criteria.

Table 5-1: Study Selection Criteria

EC1	Papers in which the full text is not available
EC2	Papers that are not in English
EC3	Papers that are duplicate
EC4	Papers that are published before 1998
EC5	Papers that do not relate to software architecture reconstruction
EC6	Papers that do not have a proposed tool
EC7	Papers that are experience or survey papers

Table 5-2: Overview of search results and study selection

Source	# of Included Studies After Applying Search Queries	# of Included Studies After Exclusion Criteria (EC1-EC4)	# of Included Studies After Exclusion Criteria (EC5-EC7)
IEEE Xplore	406	25	8
ACM Digital Library	25	7	0
Wiley Interscience	5	3	3
Science Direct	121	5	2
Springer	8	4	4
ISI Web of Knowledge	87	6	0
Total	652	110	17

5.3.2 Feature Model

Figure 5-3 shows the feature model of SAR Methods that we have derived from the domain analysis process.

An SAR method consists of the mandatory features of goal, source artefacts, adopted method, process steps, architecture model and one optional feature of architecture model.

Goal feature defines the aim of the SAR and why the SAR is actually applied. The goal of the SAR method can be architecture redesign, system modernization, refactoring and architecture nonconformity identification. Architecture redesign is a process for enhancing, adapting or revising the current architecture. Architecture redesign can be applied when underlying system architecture does not perform as desired against the requirements currently or in future. Refactoring is a software development process where the low-level components are rearranged without changing the functionality of the program code. Refactoring is heavily used paradigm in industry which makes the code more readable, adaptable to changes in the further requirements. System modernization is a software process in which the current systems architecture and used technology stack cannot overwhelm the current requirements and maintaining current state of the systems. At last, architecture nonconformity identification is a process that aims to find differences between intended and implemented architecture. These differences can be crucial and can lead to bugs, defects or errors. One or more of the defined goals can be valid for a SAR method.

Figure 5-3: Feature Model for Architecture Reconstruction Methods

Source artefacts are artefacts that are used to (re)construct an architecture. Code, logs, documentations and stakeholders' concerns are source artefacts for SAR methods. SAR methods can use one or more of these artefacts to extract architectural elements which can be represented by architectural models. Code is an executable that is written according to a programming language. Code of systems are implemented with respect to their specifications in architecture designs. Logs are output data from systems which might give insight about system architecture. Documentation show textual information about systems which can be architecture designs, functional and non-functional requirements. Stakeholders are individuals, groups, people or organizations that have an interest in the developed system. Stakeholder's concerns shape how the architecture is designed and implemented. SAR method can have one or more of these source artefacts to reconstruct system's architecture.

Adopted method represents techniques applied for reconstructing system architecture. Various different methods have been implemented and proposed in the literature for SAR. These include clustering, partitioning, data mining, querying or scripting. These methods can coexist together or can be used by itself. Clustering methods consist of grouping some of related data together. Partitioning is dividing data into parts that may or may not be related each other. In SAR data mining techniques are adopted in which patterns are detected in large sets source codes, logs and documentation. Some of the SAR methods also utilizes querying on graphs in which the architecture is represented as graph-based models. Scripting software architectures in their own domain specific languages is another method adopted by SAR methods.

Process steps defines phases used in a SAR method. SAR methods usually consist of a series of different processes including pre-processing, architecture extraction, architecture analysis, architecture visualization and suggestions. Pre-processing is an optional step in which the source artefact might be prepared for architecture extraction step. This can include reformatting, removing unnecessary libraries, and removing comments. Architecture extraction performs on source artefacts and extracts the valuable data from these artefacts and hence it is a mandatory step in all SAR methods. In architecture analysis step architecture components are formed using the output of extraction step. Architecture visualization step presents the reconstructed architecture in user friendly manner which can be graphical or textual. Visualization of the reconstructed architecture is not always present; thus, this feature is optional. Another optional feature in process steps is suggestions. Suggestions such as design pattern application, architecture pattern application or code refactoring can be advised after the architecture is constructed.

Architecture model representation defines the representation used for describing the architecture. The adopted SAR methods in the primary studies use different representation including Unified Modelling Language (UML), XML Metadata Interchange (XMI), Meta-Object Facility (MOF) or graph representations. UML is standardized model to represent the software architecture which contains diagrams for developers to develop, construct and document the architectures. XMI is a metadata for exchanging information using extensible mark-up languages. Software architectures can be represented and visualized using XMI based models. MOF is standard for model driven engineering proposed by Object Management Group. Graph based models can also represent the software architecture in different forms such as Bayesian networks, Markov chains and state diagrams.

Architecture view feature represents which view is to be reconstructed using SAR method. Architecture view is a mandatory feature in SAR methods for representing what part of the architecture will be constructed. Module view, allocation view and component & connector views can be reconstructed in the architecture analysing step. Module view represents how the system is structured as set of code units. Allocation view represents how the system relates the structures that are not software in systems running environment. Component & Connector view shows how the system is structured in terms of components and its relations among each other.

Using the feature diagram, we can generate many different architecture reconstruction methods by selecting the desired features. However, not all configurations might be possible in practice. Hence, cross-tree constraints are used to eliminate the invalid configurations. We have seen that if the architecture model selection feature is not chosen then the architecture is not visualized. As architecture is not represented in any manner it is impossible them to visualize. In addition, if there is a model selection feature is present it is not mandatory to see architecture visualization feature. Code source artefact must be selected if the goal of the SAR method is refactoring or architecture nonconformity identification. Hence both of these goals are operated at code level, code is needed. Refactoring or architecture nonconformity identification may or may not be selected if the code is selected as source artefact. Stakeholders source artefact must be chosen if system modernization goal is selected as a feature. System modernization goal feature may or may not be selected if stakeholder source artefact is selected. Querying feature must be selected as adopted method if the architecture model is graph based and vice versa of this constraint does not apply. Pre-processing step must be selected from process steps if log or documentation source artefacts are selected and vice versa of this constraint does not apply. Code source artefact feature must be selected if the code refactoring or design pattern application feature of process suggestions is selected and vice versa of this relation does not apply. Table 5-3 shows the summary for identified feature composition constraints.

Table 5-3: Identified feature composition constraints

<i>Process Steps. Architecture Visualization requires Architecture Model</i>
<i>Goal. Refactoring requires Source Artefact. Code Source</i>
<i>Goal. System Modernization requires Source Artefact. Stakeholders</i>
<i>Adopted Method. Querying requires Architecture Model. Graph</i>
<i>Source Artefact. Log requires Process Steps. Pre-processing</i>
<i>Source Artefact. Documentation requires Process Steps. Pre-processing</i>
<i>Goal. Design Pattern Application requires Source Artefact. Code Source</i>
<i>Goal. Architecture Nonconformity Identification requires Source Artefact. Code Source</i>

5.3.3 Generic Business Process Model

Architecture reconstruction is in essence a process activity that includes several steps linked in different ways. Based on the identified studies as discussed in the previous sub-section Figure 5-4 provides the generic business process model for architecture reconstruction methods.

Figure 5-4: Generic business process model for Architecture Reconstruction

The first step for an architecture reconstruction begins with a definition of a problem statement explaining what the proposed method aims to solve. According to the problem statement the required architectural information which consists of architecture views is selected. Next step is to select a method to solve the stated problem according to the needs defined in the previous step. These methods can utilize techniques from clustering, partitioning, data mining or querying methods. Reconstruction methods might utilize a model for solving the problem or might need pre-processing the source artefacts before the extraction of architectural elements. Models are used for representing both the abstractions in the code and reconstructed architectures. These models can be graph, MOF, UML or XMI based. These four steps define the SAR process design describing the planning of the proposed SAR method.

Pre-processing can be done in several ways such as reformatting and refactoring code, removing undesired libraries and dependencies and providing intermediate models to a workflow. The successful completion of architectural element extraction is followed by the analysing of the extracted elements for forming the architectural information intended in information selection phase. The process follows by two optional steps including visualizing the reconstructed architecture and suggest improvements on the reconstructed architecture respectively. In the

visualization step the architecture is shown using the selected architecture representation. In the suggestion step the process provides guidelines for the architecture reconstruction.

5.4 CONCRETE ARCHITECTURE RECONSTRUCTION METHOD

The domain model that has been defined in the earlier section represents both the common and variant aspects of architecture reconstruction processes. We have provided a family feature model and a generic process model that has been described after a study to multiple methods. To develop a concrete architecture reconstruction method, family feature model and generic process model can be reused and adapted with respect to the required features. Figure 5-5 illustrates this process.

Figure 5-5: Relation between Reference SAR and concrete SAR

Similar to reference models in general the approach can be used in either a prescriptive way or descriptive way. In the prescriptive way the family feature model and the generic process can be reused to prescribe the common elements Moreover, it can indicate the elements that might be selected as variant elements. Alternatively, in the descriptive way, the approach can be used to describe a given SAR method using the provided family feature model and the generic business process model. In the following sub-sections, we show two different examples of concrete architecture reconstruction methods that are derived from the provided domain model. The first example relates to the architecture reconstruction method as discussed by Anqueutil et al [82]. This is explained in section 5.4.1. The second example describes the Art method as discussed by Fiutem et al. [90]. This is explained in section 5.4.2.

5.4.1 Recovering software architecture from the names of source files

Table 5-4 shows the features for the architecture reconstruction method of Anqueutil et al [82]. As shown in the table the goal of this method is to identify the nonconformance between the

intended architecture and implemented architecture. Only the code source artefact is used in architecture extraction step. This method adapts clustering and partitioning method and does not generate any models. In architecture analysis step the component & connector view of the architecture is created using the extracted information from the architecture extraction step.

Table 5-4: Feature Selection for Recovering software architecture from the names of source files

Feature	Selected Features of SAR Method
Goal	Architecture nonconformity identification
Source Artefacts	Code
Adopted Method	Clustering & Partitioning
Process Steps	Architecture Extraction Architecture Analysis
Architecture Model	Not used
Architecture View	Component & Connectors View

Figure 5-6 shows the concrete business process model for this SAR method that is derived from the generic business process model of Figure 5-4. The first step is to state the problem and then selecting the which information will be extracted from the architecture. Subsequently, the reconstruction technique to be applied is selected and the steps so far actually designed the SAR method. Then the architectural elements are extracted and analysed for reconstructing the architecture.

Figure 5-6: Business Process Model for Recovering software architecture from the names of source files

5.4.2 ART: An Architectural Reverse Engineering Environment

Table 5-5 shows the features for the architecture reconstruction method of Fiutem et al. [90]. The goal of this SAR method is to identify nonconformity between the architecture and code. Code of the system is used as source artefact and data mining techniques are adopted in this method. Process steps consists of pre-processing, architecture analysis, architecture extraction and architecture visualization. The graph-based model is used for representing the architecture throughout the process. The module and component & connectors view of the architecture is reconstructed. The method proposes nonsuggestive architectural recovery tool based on system, module, task and data architectures views for software architecture. Lexical parsers were used to model the architecture from the code into abstract syntax trees (AST). The user constructs a file before the architecture extraction phase. After the architecture elements are analysed, they form a graph which is visualized through a user interface.

Table 5-5: Feature Selection for ART: An Architectural Reverse Engineering Environment

Feature	Selected Features of SAR Method
Goal	Architecture nonconformity identification
Source Artefacts	Code
Adopted Method	Data Mining
Process Steps	Pre-processing Architecture Extraction Architecture Analysis Architecture Visualization
Architecture Model	Graph
Architecture View	Module View, Component & Connector View

Figure 5-7 shows the business process model for this study. In this method, first the problem definition. Information, technique and model is selected according to the defined problem. Pre-processing is applied on the code and architectural elements are extracted and analysed. Different from the previous method architecture visualized in this method.

Figure 5-7: Business Process Model for ART: An Architectural Reverse Engineering Environment

5.5 RELATED WORK

In the literature we have found similar studies but none of the studies provides a feature model and generic business process model for software architecture reconstruction. Duenas et al. [88] presents experiences over a currently adopted architectural recovery tools. In this study, architecture reconstruction process is modelled in an iterative manner. Also, authors compare different approaches for tackling architecture construction problems such as statistical, cluster recognition, interface discovery and model enrichments.

Garcia et al. [92] compares six software architecture recovery techniques accuracies for each technique. The objective of the study is to find a suitable and reliable fully automated method for reconstruction. Moreover, Gorton et al. [93] compares the architecture reconstruction as Garcia et al. for just-in-time architecture reconstruction. They construct different architectural views using five tools and output of these tools were compared in terms of accuracy rates on the same commercial system.

Guaman et al. [94], proposes a reference process framework for architecture reconstruction in which architecture reconstruction is divided into four main parts pre-processing, extraction, analysing and visualization. In this study the complete framework is presented with a proposed adoption of machine learning concepts for recommendation and improvements after method is trained with source code, documentation and software architecture.

Koschke [97], summarizes the current state of the art for architecture reconstruction process with architecture viewpoints in mind. Study divides architecture reconstruction process into two phase which are reconstruction design and reconstruction execution. These two phases are further decomposed into subphases accordingly. This study reveals that SAR methods are mostly carried out on module viewpoints. Component and connector viewpoints are also addressed by some studies in the literature. Author also suggests that there are very few studies that documents the complete SAR method and its application along with the cost benefit analysis.

Zahid et al. [107], presents a survey study for the solutions and evolutions in SAR method. Study suggests that knowledge-based clustering methods are most commonly used for automated architecture reconstruction. The studies selected for this survey mainly concentrates on component and connector architecture views of architecture. Different clusters are formed from this view and architecture is modelled accordingly. This survey also reveals that the presented solutions in studies discusses precision levels of the reconstructed architecture with respect to original architecture. Due to this fact authors suggests that none of the proposed solutions have reconstructed architecture precisely.

5.6 DISCUSSION

In this chapter we have adopted a domain analysis process to identify the common and variant features of SAR methods. Although we have carefully performed the domain analysis process a number of validity threats to the study can be identified. First of all, for the development of the domain model we have not opted for a heavyweight systematic literature review process. However, we adopted a domain analysis process in which we selected a sufficient set of papers that we believed to cover the adopted features. Of course, a more in-depth systematic literature review could have resulted in more papers which could have an impact on the feature model.

However, we can also state that the papers that we have selected for our study appeared to be repeating the similar set of common and variant features. This showed the convergence of the domain model and hence increased the confidence of the validity of feature model.

The family feature model for the SAR methods defines the common and variant features that can be used to describe individual SAR methods. With the feature model we indeed derive many different alternative SAR methods, describe existing SAR methods, and support the identification of novel SAR methods. We have illustrated how we could use the feature model to describe two different SAR methods. Although possible, in this study our goal was not to provide a survey of existing SAR methods. We aim to provide this in our future work.

Similar to the representation of common and variant features we have also explicitly described the reference model for SAR methods. Here again we could identify the common and variant steps of SAR methods. The combination and usage of feature models and process models appeared to be very useful to represent the SAR methods. However, similar to the construction validity threats for the feature model, process models could be perhaps more refined if an in-depth systematic literature review was provided. From our experience the process models also appeared to converge and hence we believe that we have largely captured the currently adopted SAR process activities.

5.7 CONCLUSION

Software projects often need to evolve or usually do not have a well-described software architecture. For these situations it is worthwhile to extract the information from the existing artefacts to derive the architecture. In the literature several architecture reconstruction methods have been proposed albeit fragmented over different studies. The objective of this study was to provide a more comprehensive view and analysis of these methods. For this we have applied a domain-driven approach in which we applied the domain scoping and domain modelling steps. With the domain scoping process, we have identified 17 relevant primary studies that we analysed in detail to derive the domain model. The domain model has been presented as a set of common terms of software architecture reconstruction, a generic business process model, and a feature diagram. The domain model represents a family of SAR methods and can be used to derive a concrete SAR method. We have used the domain model to derive two different SAR methods. From the domain analysis process, we concluded that the topic of SAR has been considered for a longer time but is still an important question that is further developing. The domain model that we have provided can be used to provide insight in the overall methods and pave the way for further research. In our future work we aim to enhance our work on architecture reconstruction and derive a comprehensive SAR method.

6

VIEW-DRIVEN ARCHITECTURE RECONSTRUCTION APPROACH¹

¹This chapter is based on following submitted paper:

Uzun, Burak, and Bedir Tekinerdogan. "Detecting Deviations in the Code using Architecture View-Based Drift Analysis" Software Quality Journal.

Abstract

Architecture of any software system can be reconstructed from a system implementation for various reason. Architecture conformance checking is one of these reasons in which we are interested for this thesis. Reconstructing architecture models from the code is cumbersome for large scale systems and therefore needs to be automatized to be feasible. This chapter proposes architecture reconstruction method based on architecture viewpoint definitions. This approach is automatically deriving EMF based architecture view models from the code for desired architecture viewpoints. A model-driven development approach is adopted in which architecture views are represented as specifications of domain-specific languages. The code on its turn is analysed, and the architectural view specifications are reconstructed. To illustrate the approach, we provided details of our approach and sample reconstruction example. Also, we use our approach in next chapter as a part of our architecture conformance analysis tool.

Keywords: Software Architecture Reconstruction, Model-Driven Development, Reverse Engineering, Software Architecture

6.1 INTRODUCTION

Software architecture design represents the gross level structure of the system and defines the systemic design decisions. The architecture design, together with the rationale of the design decision, is described in the architecture documentation that can be used as a guideline for the corresponding implementation. A well-documented architecture is crucial for supporting the communication among stakeholders, for guiding and analysis of the design decisions, and for guiding the organizational processes[26].

Software architecture is typically modelled using so-called *architecture views* that represent the system from one or more stakeholders' perspectives. By separating the architecture views, the ubiquitous notion of the separation of concerns principle is applied, thereby supporting the understandability, maintainability and complexity management of the overall system. Hence to provide a proper architecture conformance analysis, it is important to check whether the guidelines of all the relevant architecture views have been addressed in the code.

Software architecture reconstruction is a reverse engineering process used for deriving the current state of software architecture from the implementation. There are various methods proposed by different researchers around the world as presented in chapter 5. Software architecture reconstruction methods takes system artefacts as source and outputs an architectural definition.

The remainder of the chapter is organized as follows. Section 6.2 provides the background on software architecture modelling and software architecture reconstruction. Section 6.3 presents

the derived metamodeling of architecture viewpoints. Section 6.4 describes the architecture viewpoint-oriented software architecture reconstruction method in detail. Section 6.5 presents related work, and finally, section 6.6 concludes the chapter.

6.2 BACKGROUND

6.2.1 Software Architecture Modelling

A common practice for describing the architecture according to the stakeholders' concerns is to model different architectural views [5][6][26]. An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern. Usually, multiple architectural views are needed to separate the concerns and as such support the modelling, understanding, communication and analysis of the software architecture for different stakeholders. Architectural views conform to viewpoints that represent the conventions for constructing and using a view. Having multiple views helps to separate the concerns and as such support the modelling, understanding, communication and analysis of the software architecture for different stakeholders. A recent approach for modelling software architecture based on viewpoints is the Views and Beyond (V&B) approach. The V&B approach defines the following view categories: *Module view* category that is used for documenting a system's principal units of implementation. *Component and Connector* category that is used for documenting the system's units of execution. *Deployment View* category that is used to document the relationships between a system's software and its development and execution environments. Viewpoints are defined as styles which are used to define views. Although the V&B approach has defined a predefined set of architectural styles it is also possible to define new styles for particular concerns.

Figure 6-1: Architecture Viewpoint Concepts and their relations

6.2.2 Software Architecture Reconstruction

Software architecture reconstruction (SAR) is a reverse engineering process in which the architectural structure of a software system is extracted from system entities such as code, log, documentation. Figure 6-2 shows a conceptual model for architecture reconstruction. SAR is often needed to derive a missing or incomplete architecture documentation or to identify and manage architecture drift. SAR methods can be applied to derive a single abstract model or extract architecture views of the system [111][112]. Hereby, since manual handling of the architecture reconstruction process is usually time-consuming and costly, automatized methods and tools are proposed. The result of the SAR process is architecture documentation that includes a description of a set of architecture views for addressing stakeholder concerns.

Figure 6-2: Conceptual Model for Architecture Reconstruction

6.3 METAMODELING OF ARCHITECTURE VIEWPOINTS

In order to model the architecture viewpoints, we have first defined the corresponding metamodels for the viewpoints and then mapped these to domain-specific languages using the Eclipse Ecore model [110][26]. The metamodels of the selected viewpoints are shown Figure 6-3. These metamodels are represented as Ecore models in the Eclipse IDE. Each of these metamodels has been developed after a thorough domain analysis on the corresponding viewpoints. For this, we have analysed the viewpoints as discussed by Clements et al. Hereby we have focused on the module viewpoints that define the architecture structure based on implementation units, that is, modules. As can be seen from the figure, we have used the following five views: decomposition view, shared data view, uses view, generalization view and layered view. Architecture models are defined as instances of these metamodels. Similarly, architectures extracted from the code (next section) is also defined as specifications of these models.

6.3.1 *Decomposition View*

Due to improvements in software development technology and the need for more complex requirements, software industry is building larger and more complex software systems. One way to handle this complexity is well adapted design paradigm called divide and conquer. Divide and conquer paradigm is basically breaking down complex problems into much smaller and manageable problems so that original problem can be solved with ease. One of the first actions taken by software architects while designing software systems is to derive decomposition view of the software system. Decomposition view visualizes the partition of code across different modules and submodules in a software system [5]. Decomposing a software system into smaller and cohesive parts is a great example of dividing and conquering paradigm. Figure 6-3 presents metamodel for decomposition viewpoint in which model aggregates elements that are in type of module and subsystem. Element has a list property called subelements which references other elements for keeping submodules of a module.

6.3.2 *Shared Data View*

Due to research in fields like big data and cloud engineering, database systems and their usages are getting more and more common. Nowadays lots of software intensive systems are integrated with some kind persistent data stores which can be relational databases, filesystems, messaging queues and nonrelational databases. Since presentation of interaction patterns between software systems and persistence data sources are important aspect of software system design. Shared data view presents data sources along with their accessors in which accessors may have read or write interactions with these data sources. Shared data view is useful when data sources have multiple data accessors both reading and modifying the shared data [5]. Also, some additional data can be presented along with shared data view models such as restrictions for connections, access control authorizations, synchronization mechanisms and data properties. Shared data view model aggregates elements and attachments. Elements for shared data viewpoint can be repository and data accessor. These elements have two different attachments which are data write and data read relations.

6.3.3 *Uses View*

Generally, software systems are decomposed into little pieces due to complexity in software systems which makes it more manageable and understandable for all involving stakeholders. Most of the time these decomposed pieces are depending on each other for performing. Uses view presents usage dependencies between these pieces which are called modules. Usage dependencies occurs whenever module's correctness depends on another module's correctness [5]. Uses view is insightful since the modules presented are the subset of the modules presented in decomposition view and it presents special type of relation between these modules. Also documenting uses view helps with incremental development and deployment [5]. Uses view model aggregates of relations and elements. Elements are modules or subsystems and relations hold two properties which are source and target elements.

6.3.4 *Generalization View*

By the nature of domain software system is modelling, it can be broken into pieces where some part of the system is more specialized version of another part. These specializations denote not just for difference but also commonality between parts. Generalization view presents special type of relation between modules called "is-a" relation [5]. This view is useful when extension and evolution of modules [5]. Modules in this case are classes or interfaces and generalization view presents inheritance or implementation relations between these elements. Inheritance relation exists between class and class or interface and interface elements. Moreover, implementation relation exists between class and interface elements. Generalization view model aggregates of modules and relations. Relations can be either implementation or inheritance and has two properties which are child and parent of that relation.

6.3.5 *Layered View*

Software architecture evolved from copy and paste architecture to layered monolith and then to microservices. Both layered monolith and microservices divides software architecture units into layers of division where layers interact with each other for their correctness. Layered view is specialized version of uses view where layers which are cohesive group of modules can interact in one-way direction [5]. Layered view model aggregates of layers and relations. Relations can be either allowed to use below relation or allowed to use relation and hold two properties which are source and target layer.

Decomposition Viewpoint Metamodel

Shared Data Viewpoint Metamodel

Uses Viewpoint Metamodel

Generalization Viewpoint Metamodel

Layered Viewpoint Metamodel

Figure 6-3: Metamodels of the Selected Viewpoints

6.4 VIEWPOINT ORIENTED SOFTWARE ARCHITECTURE RECONSTRUCTION TOOL

We have developed a software architecture reconstruction tool that can extract architecture models from the code. The extracted architecture models represent the selected architecture views of the systems. We have created a library which can be plugged into projects that are programmed using Java as the programming language. The library depends on scripting on the source code using Java reflection features. We have utilized an open-source project for Java runtime metadata analysis to fetch classes from the code under test [113].

The developed architecture reconstruction method derives architecture view models from the code in four steps: pre-processing, extraction, analysis and model. We first pre-process the code by annotating code parts for repositories and its data accessors. Architecture elements are then extracted and mapped to architectural components. The resulting architecture is represented as Human-Usable Textual Notation (HUTN), which conforms to the Meta Object Facility (MOF) from the Object Management Group (OMG) for storing models in a human understandable format. Figure 6-4 presents the part of the architecture reconstruction library. As can be seen from the figure, the `ArchitectureReconstructor` class has five fields that are responsible for extracting five different viewpoints. The results of the extractors are view models, which are then translated to HUTN models in `reconstruct` method.

```
public class ArchitectureReconstructor {
    private GeneralizationViewpointExtractor generalizationViewpointExtractor;
    private DecompositionViewpointExtractor decompositionViewpointExtractor;
    private UsesViewpointExtractor usesViewpointExtractor;
    private SharedDataViewpointExtractor sharedDataViewpointExtractor;
    private LayeredViewpointExtractor layeredViewpointExtractor;
    private HutnWriter hutnWriter;

    public ArchitectureReconstructor() {
        this.generalizationViewpointExtractor = new GeneralizationViewpointExtractor();
        this.decompositionViewpointExtractor = new DecompositionViewpointExtractor();
        this.usesViewpointExtractor = new UsesViewpointExtractor();
        this.sharedDataViewpointExtractor = new SharedDataViewpointExtractor();
        this.layeredViewpointExtractor = new LayeredViewpointExtractor(usesViewpointExtractor);
        hutnWriter = new HutnWriter();
    }

    public void reconstruct(String path) {
        Generalization generalizationView = generalizationViewpointExtractor.extract(path);
        Decomposition decompositionView = decompositionViewpointExtractor.extract(path);
        Uses usesView = usesViewpointExtractor.extract(path);
        SharedData sharedDataView = sharedDataViewpointExtractor.extract(path);
        LayeredViewpoint layeredView = layeredViewpointExtractor.extract(path);
        hutnWriter.write(generalizationView, decompositionView, usesView, sharedDataView, layeredView);
    }
}
```

Figure 6-4: Architecture Reconstruction Library Details

Figure 6-5 presents the simple usage of the library API. Architecture reconstructor object needs to be initialized and reconstruct method with the path of code under test should be executed.

```
ArchitectureReconstructor reconstructor = new ArchitectureReconstructor();
reconstructor.reconstruct ( path: "case");
```

Figure 6-5: Architecture Reconstruction API Usage

Table 6-1 reports the pseudocodes for the architecture reconstruction code of the selected 5 viewpoints. We have defined an algorithm for each viewpoint and implemented them to reconstruct the corresponding architecture view model.

Table 6-1: Pseudocodes for reconstructing view models per architecture viewpoint

Viewpoint	Pseudocode
Decomposition	<ul style="list-style-type: none"> Find all classes for given path For each class <ul style="list-style-type: none"> Add package name to set Tokenize the package name by “.” and add them to set For each package name <ul style="list-style-type: none"> Find subelements that begins with package name Create a decomposition module object with name and subelements
Uses	<ul style="list-style-type: none"> Find all classes for given path For each class <ul style="list-style-type: none"> Create uses module Create uses relation for each declared field type Create uses relation for each method parameter types Create uses relation for each method return types
Generalization	<ul style="list-style-type: none"> Find all classes for given path For each class <ul style="list-style-type: none"> Create generalization module Create inheritance relation for each parent class Create implementation relation for each implemented interface
Layered	<ul style="list-style-type: none"> Find all uses relations Filter uses relations where no circular dependency exists
Shared Data	<ul style="list-style-type: none"> Manually annotate data accessor classes with @Repo Manually annotate data accessor methods with @Read and @Write Create repositories for each repository used by data accessors Create data accessors for each class annotated with @Repo Create data read for each method annotated with @Read Create data write for each method annotated with @Write

Figure 6-6 shows, for example, the reconstruction of an architecture decomposition view from the code. Hereby, a system is illustrated that consists of two sub-modules *pack1* and *pack2*. The module *pack1* consists further of sub-modules *pack11* and *pack12*, while the module *pack11* consists of sub-module *pack111*. In a similar sense, the reconstructed architecture code views are easily extracted.

Figure 6-6: Example decomposition of source code and corresponding view model

Figure 6-7 presents the reconstruction of generalization view from the code. Source code consists of three classes and two interfaces which are *A*, *T*, *Y*, *I1* and *I2* respectively. Also, we can see generalization relations between these classes as follows: class *A* implements interface *I1*, interface *I1* extends interface *I2*, class *T* implements interface *I2*, class *Y* extends class *T* and

implements interface *I2*. As can be seen from the figure, the reconstructed generalization view contains both architectural elements and relations between these elements.

```

public class A implements I1
public interface I1 extends I2.
public interface I2
public class T implements I2
public class Y extends T implements I1

modules :
    Module "foo2.I1" {
        name : "foo2.I1"
    },
    Module "foo2.T" {
        name : "foo2.T"
    },
    Module "foo2.I2" {
        name : "foo2.I2"
    },
    Module "foo2.Y" {
        name : "foo2.Y"
    },
    Module "foo2.A" {
        name : "foo2.A"
    }

decl :
    Implementation {
        child : Module "foo2.T"
        parent : Module "foo2.I2"
    },
    Inheritance {
        child : Module "foo2.Y"
        parent : Module "foo2.T"
    },
    Inheritance {
        child : Module "foo2.I1"
        parent : Module "foo2.I2"
    },
    Implementation {
        child : Module "foo2.Y"
        parent : Module "foo2.I1"
    },
    Implementation {
        child : Module "foo2.A"
        parent : Module "foo2.I1"
    }

```

Figure 6-7: Example generalization relations in source code and corresponding view model

Figure 6-8 presents shared data view in source code and corresponding reconstructed architecture. As can be seen from the figure, source code must be annotated with *@Repo*, *@Write* and *@Read* annotations. In this example, we have a *Repository* class that has four methods for performing operations on the *db* storage. There are two elements in the reconstructed architecture which are: *db* as repository and *Repository* as data accessor. Moreover, four relations between the repository and data accessor are also identified. Add, update and delete methods are annotated with *@Write* annotation and the reconstructed shared data view model contains *DataWrite* relation between *db* and *Repository*. Query method on the other hand is annotated with *@Read* and the reconstructed shared data view model contains *DataRead* relation between *db* and *Repository*.

```

@Repo(name = "db")
public interface Repository{
    @Write Object add(Object entity);
    @Write Object update(Object entity);
    @Write Object delete(Object entity);
    @Read List<Object> query(String queryDTO);
}

elements :
    Repository "db" {
        name : "db"
    },
    DataAccessor "sd.Repository" {
        name : "sd.Repository"
    }

attachments :
    DataRead {
        dataRead : "query"
        da : DataAccessor "sd.Repository"
        rp : Repository "db"
    },
    DataWrite {
        dataWrite : "add"
        da : DataAccessor "sd.Repository"
        rp : Repository "db"
    },
    DataWrite {
        dataWrite : "update"
        da : DataAccessor "sd.Repository"
        rp : Repository "db"
    },
    DataWrite {
        dataWrite : "delete"
        da : DataAccessor "sd.Repository"
        rp : Repository "db"
    }

```

Figure 6-8: Example shared data relations in source code and corresponding view model

Figure 6-9 presents a sample source code with a reconstructed uses view model. As can be seen from the figure, we have four classes under packages that have class names. Our reconstruction method extracts the packages that are involved in uses relations. In this case, we extracted four packages which are: *uses.config*, *uses.controller*, *uses.repository* and *uses.configuration*. It can be seen from the source code that there is a uses relation between *Controller* class and *Service* class due to *Controller* class having an instance of *Service* class as field definition. Moreover, there is a uses relation between *Service* class and *Repository* class due to *Service* class constructor has *Repository* class instance as a parameter. At last, there is a uses relation between *Configuration* class and *Repository* class due to *Configuration* class having a method with a return type of *Repository* class.

```

public class Controller {
    private Service service;
}
public class Service {
    public Service(Repository repository) {}
}

public class Configuration {
    public Repository getRepository() {
        return new Repository();
    }
}
public class Repository {
}

elements :
    Module "uses.config" {
        name : "uses.config"
    },
    Module "uses.controller" {
        name : "uses.controller"
    },
    Module "uses.repository" {
        name : "uses.repository"
    },
    Module "uses.service" {
        name : "uses.service"
    }

relations :
    Relation {
        source : Module "uses.controller"
        target : Module "uses.service"
    },
    Relation {
        source : Module "uses.service"
        target : Module "uses.repository"
    },
    Relation {
        source : Module "uses.config"
        target : Module "uses.repository"
    }

```

Figure 6-9: Example uses relations in source code and corresponding view model

Figure 6-10 presents an example layered relation in source code and its reconstructed layered view model. In this case, we have a similar setup that we had in Figure 6-9 with a difference of a uses relation between *Repository* class and *Controller* class. We introduced a cycle by adding this new relation to *Repository* class and hence violated the layered view properties between *Controller*, *Service* and *Repository*. As can be seen from the figure, we extracted two layers which are: *layered.repository* and *layered.config*. Moreover, we identified a layered uses relation between *layered.config* and *layered.repository*.

```

public class Controller {
    private Service service;
}
public class Service {
    public Service(Repository repository) {}
}

public class Configuration {
    public Repository getRepository() {
        return new Repository();
    }
}
public class Repository {
    public void add(Controller controller) {}
}

layers :
    Layer "layered.repository" {
        name : "layered.repository"
    },
    Layer "layered.config" {
        name : "layered.config"
    }

relations :
    Allowed_To_Use_Below {
        sourceLayer : Layer "layered.config"
        targetLayer : Layer "layered.repository"
    }

```

Figure 6-10: Example layered relations in source code and corresponding view model

6.5 RELATED WORK

Architecture reconstruction has been addressed in several studies in the literature. In [85] a software architecture reconstruction method based on architecture viewpoints is presented. The method uses natural language processing and string parsing to reconstruct UML architecture models. In [110] an architecture approach is presented that uses source files names in clustering algorithms for reconstructing architecture models from the result of the applied algorithm. In [83] a method is presented similar to [85] which is based on architecture viewpoints using clustering algorithms and generating customer architecture models that are conforming to metamodel defined in the study. Anquetil et al [82] proposed an approach based on file names to recover software architecture using clustering algorithms. At the first phase of the study, a set of possible abbreviations that can be present in filenames are defined and then source code is clustered into partitions with respect to this setup. Favre et al [89] proposed a metamodel-driven architecture reconstruction method in which global metamodel is defined for a software system project. Architectural elements are extracted into forms of metaware specifications and executions which are then inputted in metaware evaluation tool. Fontana et al [91] proposed a method that both detects design patterns and recovers software architecture. In this approach, code is translated into abstract syntax tree by an information detection engine. Later, xml files that contain architecture and design patterns are extracted from abstract syntax tree and visualized to the user. Granchelli et al [95] propose a software architecture recovery tool for microservice architecture. There inputs are flowing into this tool which are: source codes, IP addresses of running docker containers and logs of the running microservices. Architectural elements are extracted and grouped into logical clusters which are then visualized as well.

6.6 CONCLUSION

In this chapter, we presented our view-driven architecture reconstruction method. We used our previously defined feature model to implement our approach. We first presented the metamodel definitions for architecture viewpoints which are decomposition, shared data, uses, generalization and layered viewpoints. Later, we defined our reconstruction criteria per viewpoint and give snippets from the library that we created. We implemented our library using Java reflections features which analyses source code in runtime and extracts architecture view models defined in HUTN format. In the next chapter, the developed approach will be used to reconstruct a software architecture from an industrial case study and developed architecture level conformance analysis will be applied.

ARCHITECTURE VIEW DRIVEN CONFORMANCE ANALYSIS APPROACH FOR DETECTING DEVIATIONS IN THE CODE¹

¹This chapter is based on following submitted paper:

Uzun, Burak, and Bedir Tekinerdogan. "Detecting Deviations in the Code using Architecture View-Based Drift Analysis" Software Quality Journal.

Abstract

Architectural drift implies the diverging of the implemented code from the architecture design of the system. Manually checking the consistency between the implemented code and architecture can be intractable and cumbersome for large scale systems. This chapter proposes a holistic, automated conformance analysis approach that explicitly focuses on the adoption of architecture views. The approach builds on, complements, and enhances existing architecture conformance analysis methods that do not adopt a holistic approach or fail to address the architecture viewpoints. A model-driven development approach is adopted in which architecture views are represented as specifications of domain-specific languages. The code on its turn is analysed, and the architectural view specifications are reconstructed, which are then automatically checked with the corresponding architecture models. To illustrate the approach, we have applied a systematic case study research for a conformance analysis of the business to customer (B2C) system within a large-scale software company. The case study research showed that divergences and absences of architectural elements could be detected in a cost-effective manner with the proposed approach.

Keywords: Software Architecture Reconstruction, Model-Driven Development, Architecture Conformance Analysis, Architecture Drift

7.1 INTRODUCTION

Software architecture design represents the gross level structure of the system and defines the systemic design decisions. The architecture design, together with the rationale of the design decision, is described in the architecture documentation that can be used as a guideline for the corresponding implementation. A well-documented architecture is crucial for supporting the communication among stakeholders, for guiding and analysis of the design decisions, and for guiding the organizational processes[26].

Unfortunately, software systems are rarely static and need to be adapted due to bug fixes or new requirements. In case the code and/or the architecture is adapted separately, this leads to the so-called *architectural drift problem*, which defines the discrepancy between the architecture description and the resulting implementation [110]. Architectural drift can occur even during the initial implementation of the architecture due to lack of knowledge about the architecture or stringent time-to-market constraints. This architectural drift may directly lead to increased maintenance time and cost because the important systemic design decisions are not followed and lost. On its turn, this eventually leads to a system whereby the difference between the code and the system's architecture is so large that a complete system re-implementation is required.

Manually checking the consistency between the implemented code and architecture can be cumbersome and intractable for large scale systems. Hence automated conformance analysis has been proposed to automatically check the discrepancy between architecture and code. For supporting the conformance analysis, usually a model of the code is reconstructed, which is then compared to a model of the architecture, after which the discrepancies are highlighted. Three different discrepancies can be distinguished, including (1) implemented code has missing

architectural elements (absence) (2) implemented code has extra defined architectural elements (divergence) (3) implemented code has same architectural elements as architecture (convergence).

Software architecture is typically modelled using so-called *architecture views* that represent the system from one or more stakeholders' perspectives. By separating the architecture views, the ubiquitous notion of the separation of concerns principle is applied, thereby supporting the understandability, maintainability and complexity management of the overall system. Hence to provide a proper architecture conformance analysis, it is important to check whether the guidelines of all the relevant architecture views have been addressed in the code.

The approach builds on, complements, and enhances existing architecture conformance analysis methods that do not adopt a holistic approach or fail to address the architecture viewpoints. With this, a model-driven development approach is adopted in which architecture views are represented as specifications of domain-specific languages. The code on its turn is analysed, and the architectural view specifications are reconstructed, which are then automatically checked with the corresponding architecture models. For the architecture framework we have used selected viewpoints from the views and beyond approach[5].

To illustrate the approach, we have applied a systematic case study research for a conformance analysis of the business to customer (B2C) system within a large-scale software company. The case study research showed that divergences and absences of architectural elements could be detected in a cost-effective manner with the proposed approach.

The remainder of the chapter is organized as follows. Section 7.2 provides the background on software architecture modelling and architecture conformance analysis. Section 7.3 presents the proposed architecture conformance analysis method. Section 0 presents the implementation and the corresponding tool of the presented conformance analysis method. Section 7.5 presents the case study research to illustrate and validate the approach, section 7.6 presents discussions. Section 7.7 presents the related work, and finally, section 0 concludes the chapter.

7.2 BACKGROUND

Before we describe the overall method, we first describe the key elements that are integrated in the overall process, including software architecture modelling (Section 7.2.1) and architecture conformance analysis (Section 0).

7.2.1 Software Architecture Modelling

A common practice for describing the architecture according to the stakeholders' concerns is to model different architectural views [5][6][26]. An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern. Usually, multiple architectural views are needed to separate the concerns and as such support the modelling, understanding, communication and analysis of the software architecture for different stakeholders. Architectural views conform to viewpoints that represent the

conventions for constructing and using a view. Having multiple views helps to separate the concerns and as such support the modelling, understanding, communication and analysis of the software architecture for different stakeholders. A recent approach for modelling software architecture based on viewpoints is the Views and Beyond (V&B) approach. The V&B approach defines the following view categories: *Module view* category that is used for documenting a system's principal units of implementation. *Component and Connector* category that is used for documenting the system's units of execution. *Deployment View* category that is used to document the relationships between a system's software and its development and execution environments. Viewpoints are defined as styles which are used to define views. Although the V&B approach has defined a predefined set of architectural styles it is also possible to define new styles for particular concerns.

Figure 7-1: Architecture Viewpoint Concepts and their relations

7.2.2 Software Architecture Conformance Analysis

Within the architecture driven development context, it is important that the code is consistent with the architecture (and vice versa). In case the architecture elements and the corresponding design decisions are not correctly reflected in the code, then we can identify this as an architecture drift. The notion of drift implies also the dynamic behaviour of the problem due to the bugs that are introduced in the code or the need to adapt the code for changing requirements. When comparing the architecture elements with the code then we can identify three different scenarios. If the relations that are present in the architecture are also found in the implementation, then this is *convergent* relation. In case the architecture relation is not present in the implementation then this is called an *absence* relation. Absence relations occur of course during the initial development of the system in which the architecture is defined but the implementation is not ready yet. As such, in the early phases of the development these absence relations might be a lesser concern. Finally, if the implementation includes relation that is not present in the architecture, then this is called *divergence* relation. Architectural violations are due to absence or divergence relations.

An often-used architecture consistency approach is the reflexion modelling approach as proposed by Murphy et al.[20]. In principle, a reflexion model allows a software developer to view the structure of a system's source through a chosen high-level (often architectural) view.

To check the consistency between the architecture model and the code, an abstract model of the code is derived. The two models are then compared to each other with respect to earlier defined mapping rules between the code and the implementation. The results of the comparison are presented to the user through a *Reflexion Model*. Usually, architecture conformance analysis approaches that apply reflexion modelling include tools for modelling the architecture, modelling the mappings, deriving the abstract model from the source code, the consistency analysis checker, and the generator of the resulting reflexion model.

7.3 VIEWPOINT ORIENTED SOFTWARE ARCHITECTURE CONFORMANCE ANALYSIS

In this section, we present our viewpoint-oriented software architecture conformance analysis method (VOSACAM). Figure 7-2 presents the workflow model that defines the steps in VOSACAM. Four different swimlanes are defined including the different roles: software architect, software developer, conformance tool and reconstruction tool. The left part of the figure shows the role software architect who creates the architecture view models for the system. The right part of the figure shows the role of the software developer who implements the code with respect to the provided architecture view models. Both processes continue throughout the lifecycle of software system and thus, in due time architecture drift can occur. In many projects, the architecture drift is checked manually, but as stated before, this does not scale with the increasing size and complexity of software projects. Hence automated conformance analysis is required, which are realized by a dedicated tool, as represented by the swimlane, Software Architecture Conformance Tool. The tool expects two inputs, architecture models based on the various views, and the architecture models based on the implementation. Both inputs, that is, the original architecture model specification, as well as the extracted architecture models from the code, reflects architecture views. The software architecture reconstruction process extracts architectural elements from the code, which was developed by software developers. The extracted architectural elements are then analysed and formed into architecture components which are then derived into architecture view models. The models derived from architecture reconstruction and software architect are compared against each other to find absence, divergence relations. It is important to note that flows under conformance and reconstruction tools are automated. We have implemented our tool for five architecture viewpoints which are: decomposition, shared data, uses, generalization and layered. The next sections of this study provide metamodels for used architecture viewpoints and the viewpoint-based approaches for architecture reconstruction and architecture conformance analysis.

Figure 7-2: Workflow Model for VOSACAM

7.4 VIEWPOINT ORIENTED ARCHITECTURE CONFORMANCE ANALYSIS TOOL

In the previous chapter, we have introduced the method to reconstruct the software architecture models from the code. At this stage, we have both the reconstructed and original architecture ready for conformance analysis. We used Epsilon Comparison Language (ECL) to compare different architecture models that conform to the same metamodel of the corresponding viewpoint. ECL is rule driven domain specific language for comparing different or same type of models [4]. The following subsections elaborate on the implementation of the conformance analysis. Section 7.4.1 describes the implementation of the conformance analysis for each of the five viewpoints. Section 7.4.2 describes the results of the execution of the code.

7.4.1 Implementation of Conformance Analysis for each Viewpoint

The conformance analysis pseudocode for each viewpoint is primarily based on the elements and structure of the metamodels that we have presented in 6.3. In Table 7-1 we first report the corresponding derived pseudocodes for each architecture viewpoint. Each of the pseudocodes checks whether the required architecture elements from the corresponding viewpoints are similar in both the original architecture view and the reconstructed architecture view. The pseudocodes of Table 7-1 have all been implemented using ECL.

Table 7-1: Pseudocode for the conformance analysis

Viewpoint	Comparison Pseudocode
Decomposition	<ul style="list-style-type: none"> • For each module <ul style="list-style-type: none"> ○ Module names must match ○ Modules subelements must match
Uses	<ul style="list-style-type: none"> • For each module <ul style="list-style-type: none"> ○ Modules name must match • For each relation <ul style="list-style-type: none"> ○ Relations source module and target module must match
Generalization	<ul style="list-style-type: none"> • For each module <ul style="list-style-type: none"> ○ Modules name must match • For each implementation and inheritance relation <ul style="list-style-type: none"> ○ Child and parent module of the relation must match
Layered	<ul style="list-style-type: none"> • For each module <ul style="list-style-type: none"> ○ Modules name must match • For each allowed to use below relation <ul style="list-style-type: none"> ○ Relations source module and target module must match
Shared Data	<ul style="list-style-type: none"> • For each repository <ul style="list-style-type: none"> ○ Repository name must match • For each data accessor <ul style="list-style-type: none"> ○ Data accessor name must match • For each data read <ul style="list-style-type: none"> ○ Data read repository must match ○ Data read accessors must match ○ Data read qualifier must match • For each data write <ul style="list-style-type: none"> ○ Data write repository must match ○ Data write accessors must match ○ Data write qualifier must match

Figure 7-3 shows the ECL code for comparing decomposition views as described in Table 7-1. A rule is defined to match the modules in the original and reconstructed decomposition view of an architecture. Matching is performed between original architecture(l) and reconstructed architecture(r). In alignment with the pseudocode and the metamodel, the rule compares the two models by checking whether the name of the modules as well as the subelements equality match.


```

rule ModuleRule
  match l : original!Module with r : reconstructed!Module {
    compare : l.name = r.name and l.subelements.matches(r.subelements)
  }

```

Figure 7-3: Decomposition Viewpoint ECL Code

Figure 7-4 presents the ECL code for comparing reconstructed and original uses views of an architecture, which consists of two comparison rules for modules and relation elements. Reconstructed and original uses views of an architecture are compared by checking the equality in the module's name and relation's source and target modules.

```

rule ModuleRule
  match l : original!Module with r : reconstructed!Module {
    compare : l.name = r.name
  }
rule RelationRule
  match l : original!Relation with r : reconstructed!Relation {
    compare : l.source.matches(r.source) and l.target.matches(r.target)
  }

```

Figure 7-4: Uses Viewpoint ECL Code

Figure 7-5 presents the ECL code for comparing reconstructed and original generalization views of an architecture. The comparison is performed on three elements of generalization viewpoint: module, implementation relation and inheritance relation. Module comparison checks for the equality between original and reconstructed views for the name property of modules. Both implementation and inheritance comparison check the equality of child and parent in corresponding relations.

```

rule ModuleRule
  match l : original!Module with r : reconstructed!Module {
    compare : l.name = r.name
  }

rule ImplementationRule
  match l : original!Implementation with r : reconstructed!Implementation {
    compare : l.child.matches(r.child) and l.parent.matches(r.parent)
  }

rule InheritanceRule
  match l : original!Inheritance with r : reconstructed!Inheritance {
    compare : l.child.matches(r.child) and l.parent.matches(r.parent)
  }

```

Figure 7-5: Generalization Viewpoint ECL Code

Figure 7-6 presents the ECL code for layered views. Comparisons are like the comparisons for the uses viewpoint. The only difference is the naming of the elements, whereas layer maps to the module and allowed to use below maps to relation rule. Layer comparison checks for the name equality and allowed to use below comparison checks matching the source and target layers of these restricted relations.

```
rule LayerRule
  match l : original!Layer with r : reconstructed!Layer {
    compare : l.name = r.name
  }

rule AllowedToUseBelowRule
  match l : original!Allowed_To_Use_Below with r : reconstructed!Allowed_To_Use_Below {
    compare : l.sourceLayer.matches(r.sourceLayer) and l.targetLayer.matches(r.targetLayer)
  }
```

Figure 7-6: Layered Viewpoint ECL Code

Figure 7-7 presents the ECL code for shared data views. The set of rules as shown in the figure compares the reconstructed and original shared data views of an architecture. Two elements repository and data accessors are compared their name equality. Data read and write rules are compared by matching their data accessors, repositories and respective properties.

```
rule RepositoryRule
  match l : original!Repository with r : reconstructed!Repository {
    compare : l.name = r.name
  }

rule DataAccessorRule
  match l : original!DataAccessor with r : reconstructed!DataAccessor {
    compare : l.name = r.name
  }

rule DataReadRule
  match l : original!DataRead with r : reconstructed!DataRead {
    compare : l.dataRead = r.dataRead and l.da.matches(r.da) and l.rp.matches(r.rp)
  }

rule DataWriteRule
  match l : original!DataWrite with r : reconstructed!DataWrite {
    compare : l.dataWrite = r.dataWrite and l.da.matches(r.da) and l.rp.matches(r.rp)
  }
```

Figure 7-7: Shared Data Viewpoint ECL Code

7.4.2 Results of the Conformance Analysis

The result of the execution of each ECL rule implementations in the previous sub-section is an array object that holds information about matches (convergence, absence and divergence) between the compared two view models. Matches contains information about what are being compared and if there are discrepancies or not. This object alone does not add much value, so we needed to extract matched elements (convergence), elements present in reconstructed view but not in original view (divergence) and elements present in original view but not present in reconstructed view (absence). Figure 7-8 presents a code piece to extract this information from the mentioned data array. The first for loop statement extracts convergence relations between the original and reconstructed architecture. The second for loop statement extracts both divergence and absence relations between the original and reconstructed architecture.

```

var matchedElements = new Set;
var originalUnmatchedElements = new Set;
var reconstructedUnmatchedElements = new Set;
for (matched in matchTrace.getMatches()) {
    if(matched.matching){
        matchedElements.add(matched.left.extract());
    }
}
for (matched in matchTrace.getMatches()){
    if(not matched.matching){
        var originalUnmatched = matched.left.extract();
        var reconstructedUnmatched = matched.right.extract();
        if(not matchedElements.contains(originalUnmatched)){
            originalUnmatchedElements.add(originalUnmatched);
        }
        if(not matchedElements.contains(reconstructedUnmatched)){
            reconstructedUnmatchedElements.add(reconstructedUnmatched);
        }
    }
}
}

```

Figure 7-8: Conformance and Divergence Extracting ECL Code

Figure 7-9 presents a sample output from decomposition ECL code execution. In this execution we used the sample model from Figure 6-6 as an original decomposition view. We introduced a new module named K with its subelements K1 and K2 beneath module X and removed module Z completely from the view. In the figure, we can see that untouched module of X1, X2, Y, Y1, Y2 and Y3 are matched between the original and reconstructed view. However, modules X, Z and S did not match the modules in the reconstructed decomposition view. Moreover, modules S, X, K, K1 and K2 of reconstructed decomposition view did not match the ones in the original architecture.

```

[epsilon.ecl - DecompositionModelComparator.ecl] Matched Elements
[epsilon.ecl - DecompositionModelComparator.ecl] 1. Module s.x.x2
[epsilon.ecl - DecompositionModelComparator.ecl] 2. Module s.x.x1
[epsilon.ecl - DecompositionModelComparator.ecl] 3. Module s.y
[epsilon.ecl - DecompositionModelComparator.ecl] 4. Module s.y.y1
[epsilon.ecl - DecompositionModelComparator.ecl] 5. Module s.y.y2
[epsilon.ecl - DecompositionModelComparator.ecl] 6. Module s.y.y3
[epsilon.ecl - DecompositionModelComparator.ecl] Unmatched elements from original architecture model
[epsilon.ecl - DecompositionModelComparator.ecl] 1. Module s.x
[epsilon.ecl - DecompositionModelComparator.ecl] 2. Module s.z
[epsilon.ecl - DecompositionModelComparator.ecl] 3. Module s
[epsilon.ecl - DecompositionModelComparator.ecl] Unmatched elements from reconstructed architecture model
[epsilon.ecl - DecompositionModelComparator.ecl] 1. Module s.x
[epsilon.ecl - DecompositionModelComparator.ecl] 2. Module s
[epsilon.ecl - DecompositionModelComparator.ecl] 3. Module s.x.k.k1
[epsilon.ecl - DecompositionModelComparator.ecl] 4. Module s.x.k
[epsilon.ecl - DecompositionModelComparator.ecl] 5. Module s.x.k.k2
BUILD SUCCESSFUL
Total time: 930 milliseconds

```

Figure 7-9: Sample Output from Execution

7.4.3 Eclipse Tool

Figure 7-10 shows the eclipse workbench for software architecture conformance analysis. The left part shows the package explorer in which we have partitioned our logic by each viewpoint. Each viewpoint consists of a metamodel file, executable ant build file, ECL file and two architecture view models in HUTN format. The section on the top right shows the editor support

from the Eclipse Epsilon from framework. The lower right section presents the execution result of the ant build file, which prints out architecture deviations.

Figure 7-10: Snapshot of the eclipse workbench showing the conformance analysis

7.5 CASE STUDY DESIGN

To validate our architecture conformance analysis approach, we have adopted the case study empirical evaluation protocol as discussed by Runeson and Höst [77]. The protocol consists of the following steps: (1) case study design (2) preparation for data collection (3) execution with data collection on the studied case (4) analysis of collected data (5) reporting. Table 7-2 reports the case study design steps for the selected case study.

The case study design approach is in the category of the applied research type and as such the primary purpose is to understand the impact of the adoption of architecture conformance analysis approach within the real industrial context. One research question has been defined which relates to how effective the adopted architecture conformance analysis is. The effectiveness of the proposed approach is calculated by the number of divergences and absences discovered on the implementation of the system with respect to present divergences and absences. To this end, we have applied the approach on the fault injected real code to detect the inconsistencies. We have reconstructed relevant architecture views from the implementation and applied conformance analysis using the described tool. We have applied exhaustive testing to trigger every flow path in Table 7-1 for each architecture view in industrial case study. As shown in Table 7-2 our first-degree information sources are software developers, meetings and interviews. We also analyzed technical reports, technical documents along with the source code as second-degree information source. At last, we analyzed official documents provided by the company to other stakeholders as third-degree information source. Table 7-2 also reports the data collection methods from the information sources. We have conducted semi-structured interviews and meeting with the software developers. Then, we independently analyzed

technical reports and official documents. Finally, the results of the previous steps have been reported.

Table 7-2: Case Study Design

Case Study Design Activity	Case Study
<i>Goal</i>	Assessing the effectiveness of the adapted architecture conformance analysis approach
<i>Research Questions</i>	RQ1: How effective is the adopted architecture conformance analysis approach and tool?
<i>Background and source</i>	Software Engineers (1 st degree) Meetings and interviews (1 st degree) Technical documents and reports (2 nd degree) Source Code (2 nd degree) Official documents (3 rd degree)
<i>Data Collection</i>	Direct data collection through semi-structured interviews and meetings Independent data collection based on document analysis (the papers and technical reports) Indirect data collection based on source code analysis.
<i>Data Analysis</i>	Quantitative Data Analysis using Tables

7.5.1 Industrial Case Study: E-commerce

In this section, we describe the industrial case in which the approach has been validated at. Due to confidentiality reasons, the name of the private company is not provided. We have conducted our approach on infrastructure of real industrial context for e-commerce software. There are five major e-commerce software types which are business to business (B2B), business to customer (B2C), customer to business (C2B), customer to customer (C2C) and public administration. In our case we are dealing with B2C e-commerce system. The system is being extensively used by customers for money transactions which is resulting in huge data being generated related to customers transactions. We present the architecture view models for each viewpoint which we have presented before. The following architecture models given in subsections are original architecture models of the system which is used for conformance analysis along with the deviated architecture models. Based on the architecture documentation that consists of view descriptions, the system has been implemented. For the development and maintenance of the system, many developers and testers have been assigned who are responsible for continuous maintenance and evolution of the system. For managing such a large

system, it is important that the corresponding code is consistent with the architecture. Testing is carried out for different quality concerns. One important concern is also the alignment with the architectural design decisions.

7.5.1.1 Shared Data View

Figure 7-11 presents the shared data view for original architecture. It has one repository with add, delete, update and query methods on single database called DB.

Figure 7-11: Shared data view of adopted project infrastructure

7.5.1.2 Decomposition View

Figure 7-12 presents the decomposition view for the original architecture. There are nested packages inside each other and total of twenty-two packages. As can be seen from the figure *samplecase* package is divided into controller, service, util, repository and infra. Each of these packages are decomposed into smaller packages which forms the decomposition relation between the packages in hierarchy which also can be seen from the figure in detail.

Figure 7-12: Decomposition view of adopted project infrastructure

7.5.1.3 Uses View

The uses view of the system is shown in Figure 7-13. Here the arrows represent a uses relation that defines the dependency on the correct function of module to the used module. As can be seen from the figure there are 18 modules and 24 uses relations between these modules. Mostly used module is *infra.maintenance.log* module and rest of the uses relations are distributed between the modules in a similar manner.

Figure 7-13: Uses view of adopted project infrastructure

7.5.1.4 Layered View

Figure 7-14 shows the layered view of original architecture. Controller is the highest layer in the architecture where it is allowed to use its subpackages, service and security packages from infrastructure. Moreover, security layer has its own layering within where *infra.security.dto* package is the lowest layer in this view. We can also notice that *infra.maintenance.log* is used by three higher layers in different ranking which are: *controller*, *service*, *repository*. Service layer is allowed to use *repository* and its subpackage *helper*. This layer then allowed to use *util*, *infra.maintenance.alert* and *infra.security.authorization*.

Figure 7-14: Layered view of adopted project infrastructure

7.5.1.5 Generalization View

In the system we have also defined the generalization view to show the generalization specialization relations among the modules in the system. Figure 7-15 presents generalization view of original architecture. In the figure blue squares denote interfaces, black squares denote classes, dotted arrows denote the implementation relation and solid arrows denote extension relation. In the figure, three types of generalizations are shown including 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. As can be seen from the figure there are 21 classes, 6 interfaces, 9 inheritance relations and 9 implementation relations. Implementation relations are between classes and interface whereas extension relations are between same type of entities (eg: class to class, interface to interface).

Figure 7-15: Generalization view of adopted project infrastructure

7.5.2 How effective is the adopted VOSACAM approach?

In this section we explain our exhaustive fault-based testing on our case to validate our adapted approach. Software testing is considered to be fault-based testing when the objective is to demonstrate absence of predefined faults [73][114]. Therefore, we created a mutant copy of our case per architecture viewpoint according criterion defined in Table 7-1. Mutant copy of a program is a program under test seeded with structural changes or bugs [23]. In our approach we are executing ECL scripts on the reconstructed architecture (mutant copy) and original architecture of the case to detect and kill all mutants. Killing mutant is detecting the injected bug on our reconstructed architecture which implies that our approach is effective at finding

real life defects. In our experiments we introduced absence and divergence of relations which corresponds to removal and altering of a relation. Each viewpoint includes its own absence and divergence relations depending on the structure of the architecture viewpoint. Different strategies for mutation testing can be identified based on either first-order or high-order mutants. Jia and Harman introduced higher-order mutation testing in which mutation operators are applied more than once in their study [115]. For our testing purposes this is inapplicable since our operators are conflicting with each other meaning that a relation cannot be removed and altered consecutively. Therefore, we applied first order mutation testing.

Firstly, we generated the reconstructed architecture from the code using our novel software architecture reconstruction method presented in chapter 6. Subsequently, we use our architecture conformance analysis described in this chapter. Rest of this section provides mutants we created and result of our approach per each viewpoint.

7.5.2.1 Shared Data View

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for shared data viewpoint:

- Add new database called DB2
- Add new repository called repository2
- Add insert and search methods to repository2
- Change add method type to data read
- Change query method type to data write

Figure 7-16 presents the execution result for shared data viewpoint conformance analysis. We can see that original architecture has two unmatched items which are query and add methods, whose types was altered in the reconstructed architecture. Also, we can see six unmatched items from the reconstructed architecture. Four of them are newly added items and remaining two of them were changed from the original architecture.

```
Unmatched elements from original architecture model
1. query reads data using samplecase.infra.repository.Repository from the repository db
2. add writes data using samplecase.infra.repository.Repository from the repository db
Unmatched elements from reconstructed architecture model
1. Data Accessor samplecase.infra.repository.Repository2
2. insert writes data using samplecase.infra.repository.Repository2 from the repository db2
3. Repository db2
4. query writes data using samplecase.infra.repository.Repository from the repository db
5. add reads data using samplecase.infra.repository.Repository from the repository db
6. search reads data using samplecase.infra.repository.Repository2 from the repository db2
```

Figure 7-16: Shared data conformance analysis result

7.5.2.2 Decomposition View

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for decomposition viewpoint:

- Add new service package under maintenance
- Remove dto package from authentication package

We expect to detect unmatched dto and authentication package from the original architecture. Furthermore, we also expect to detect newly added service package and diverged maintenance package at the reconstructed architecture. Figure 7-17 presents the output of conformance

analysis between original and reconstructed architecture. We can see that removing dto package under authentication package caused nesting packages to diverge with respect to original architecture. On the reconstructed architecture side, we can see similar effect as dto package is not present, it causes nesting packages to diverge from the original architecture. Also new service package is added under maintenance which triggered divergence both in original and reconstructed architectures. Maintenance in original architecture was not matched with reconstructed architecture. Whereas maintenance and service packages in reconstructed architecture was not matched to original architecture.

```
Unmatched elements from original architecture model
1. Module samplecase.infra.maintenance
2. Module samplecase
3. Module samplecase.infra.security.authentication
4. Module samplecase.infra.security.authentication.dto
5. Module samplecase.infra
6. Module samplecase.infra.security
Unmatched elements from reconstructed architecture model
1. Module samplecase.infra.maintenance
2. Module samplecase
3. Module samplecase.infra.security.authentication
4. Module samplecase.infra.maintenance.service
5. Module samplecase.infra
6. Module samplecase.infra.security
```

Figure 7-17: Decomposition viewpoint conformance analysis result

7.5.2.3 Uses View

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for uses viewpoint:

- Add new uses relation from infra.maintenance.monitor module to util module
- Remove relation between repository and infra.repository modules

Figure 7-18 shows the execution result of conformance analysis for the uses viewpoint. As can be seen from the figure, conformance analysis detected one divergence (newly added relation) and one absence (removed relation) between architecture models.

```
Unmatched elements from original architecture model
1. samplecase.repository uses samplecase.infra.repository
Unmatched elements from reconstructed architecture model
1. samplecase.infra.maintenance.monitor uses samplecase.util
```

Figure 7-18: Uses viewpoint conformance analysis result

7.5.2.4 Layered View

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for layered viewpoint:

- Remove uses relation between infra.configuration and infra.maintenance.log modules
- Add new uses relation from repository module to controller module
- Add new uses relation from util module to service.helper module
- Add new uses relation from controller module to infra.security.authorization.dto

Figure 7-19 presents results of the conformance analysis execution result. We can see that due to first three changes reconstructed architecture has seven absent allowed to use relation. Moreover, last change caused divergence in reconstructed architecture from original architecture. We can see that biggest impact was done by second change in which we introduced a cyclic relation in layered relation. This change caused all layered relations between controller, service and repository to be violated.

```

There are: 28 matched elements
There are: 7 unmatched elements in original architecture that are not present in reconstructed architecture
There are: 1 unmatched elements in reconstructed architecture that are not present in original architecture
Unmatched elements from original architecture model
1. Layer samplecase.infra.configuration
2. Layer samplecase.util
3. samplecase.controller is allowed to use below samplecase.service
4. samplecase.service.helper is allowed to use below samplecase.util
5. samplecase.service.helper is allowed to use below samplecase.repository
6. samplecase.infra.maintenance.log is allowed to use below samplecase.infra.configuration
7. samplecase.service is allowed to use below samplecase.repository
Unmatched elements from reconstructed architecture model
1. samplecase.controller is allowed to use below samplecase.infra.security.authorization.dto

```

Figure 7-19: Layered viewpoint conformance analysis result

7.5.2.5 Generalization View

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for generalization viewpoint: Change type Authentication from class to interface, add new class EnvironmentConfiguration, extend EnvironmentConfiguration from Configuration class, add new interface ComplexLogger extend ComplexLogger from Logger, add new class ConcreteLogger, implement Logger at ConcreteLogger, remove Entity interface. Figure 7-20 shows the execution result of the conformance analysis. Conformance analysis shows that there are three divergences from the original architecture due to removal of Entity interface and changing type Authentication from class to interface. Also, there are seven new elements that are existing in reconstructed architecture that does not exist in original architecture. First divergence relates to changing type of Authentication in which the relationship between LocationAuthentication changes from inheritance to implementation. Second and third divergences are due to introduction of ConcreteLogger and its implementation relation with Logger. Fourth and sixth divergences are due to introduction ComplexLogger and its inheritance relation with Logger. Fifth and seven divergences are due to introduction of EnvironmentConfiguration and its inheritance relation with Configuration.

```

Unmatched elements from original architecture model
1. samplecase.infra.security.authentication.dto.LocationAuthentication inherits samplecase.infra.security.authentication.dto.Authentication
2. Module samplecase.infra.repository.Entity
3. samplecase.repository.PaymentEntity implements samplecase.infra.repository.Entity
Unmatched elements from reconstructed architecture model
1. samplecase.infra.security.authentication.dto.LocationAuthentication implements samplecase.infra.security.authentication.dto.Authentication
2. Module samplecase.infra.maintenance.log.ConcreteLogger
3. samplecase.infra.maintenance.log.ConcreteLogger implements samplecase.infra.maintenance.log.Logger
4. Module samplecase.infra.maintenance.log.ComplexLogger
5. samplecase.infra.configuration.platform.EnvironmentConfiguration inherits samplecase.infra.configuration.Configuration
6. samplecase.infra.maintenance.log.ComplexLogger inherits samplecase.infra.maintenance.log.Logger
7. Module samplecase.infra.configuration.platform.EnvironmentConfiguration

```

Figure 7-20: Generalization viewpoint conformance analysis result

7.5.2.6 Summary

Table 7-3 reports the execution result of our approach with respect to each viewpoint. The table consists of three sections including divergences, absences and conformances. Divergences mean the alterations on the code from the intended architecture. The first column under divergences section shows the number of divergences that are present in code and were found by fault injection to the code. The second column under divergences section presents the number of divergences found out by our tool. Moreover, second section is the absences which presents elements that are present in intended architecture and not present in the code under analysis. This section again is divided into two parts, first part shows number of present absences in the code under analysis found out by fault injections. The second part shows the number of absences found out by our approach. The third part on the table reports the conformance relations found during our experiment and this section is again divided in similar manner as the previous ones. As can be inferred from the table, our approach has full coverage on both divergence and absence detection in architecture conformance analysis.

Table 7-3: Execution result of our approach

Viewpoint	Divergences		Absences		Conformances	
	# of Divergences Present	# of Divergences Found	# of Absences Present	# of Absences Found	# of Conformances Present	# of Conformances Found
Generalization Viewpoint	7	7	3	3	70	70
Decomposition Viewpoint	6	6	6	6	16	16
Layered Viewpoint	1	1	7	7	28	28
Uses Viewpoint	1	1	1	1	59	59
Shared Data Viewpoint	6	6	2	2	4	4

7.6 DISCUSSIONS

Our architecture conformance analysis approach has been evaluated with respect to effectiveness. We have conducted industrial case study to assess the effectiveness of our approach using case study protocol defined by Runeson and Höst [77]. We can state that the approach was effective in detecting deviations between the original and final architecture of the software system which are absence and divergence relations. It executes successfully on an industrial software system and presents plausible results.

Similar to any case study research, our study also has some validity threats. Internal validity refers to casual relation between treatment and outcome. While assessing the effectiveness of the suggested approach, we have applied formal fault-based testing and reported the results in

an isolated environment that no external variable can affect. One can argue that injected violations are not real-world system faults but faults our approach aimed to detect do not change in real-world systems. Nevertheless, our experiments are executed on real industrial case. External validity refers to concern of generalizing the results of a scientific study. In our case study, we have applied our suggested approach on B2C e-commerce system. However, architecture deviation problem is not related to specific domain but characteristic problem which might occur when software is developed. Also, our approach is executed under one case study but what really matters in our context is not the number of case studies but the size and complexity of the systems. In our approach, we have adopted five different viewpoints, defined the metamodels and domain specific languages for these, implemented the viewpoint-oriented software architecture reconstruction method and implemented viewpoint-oriented software architecture conformance analysis. The approach is generic enough and could be generalized for different viewpoints and different software system in any domain. As stated before, the system that is being developed can be a complex system, and hence, we could state that we have adopted a sufficiently representative system to support the external validity. Obviously, in practice, even larger systems could be identified, but the properties of architecture code consistency will be similar. In our future research, we will focus on further applications of the approach.

7.7 RELATED WORK

Several studies have been proposed for architecture conformance analysis. Architecture conformance analysis can both be driven in the code level or architecture level. Code level architecture conformance analysis is where software architecture models are transformed into code level tests which are then executed against code under test. Architecture level conformance analysis is where software architecture models are reconstructed and executed against architecture under test. In our previous work [111] we have conducted a domain-driven analysis of architecture reconstruction methods in order to create a feature model and generic business process model from 17 studies that we have selected. We have developed a novel reconstruction method based on the outcome of the synthesis of the identified primary studies. In [116], we have adopted a model-based testing approach for deriving test cases to be used for architecture drift analysis.

Various studies propose different approaches for architecture conformance analysis. In [110], the authors propose a tool called SCHOLIA that extracts runtime object graph (ROG) to derive ownership object graph (OOG), which represents the static hierarchical relations within software systems from the code. This study utilizes annotated code and executes conformance analysis between the generated model and existing model automatically. The tool has been evaluated using a case study. In study [117], the authors propose a conformance analysis tool called jRMTool utilizing reflexion modeling. The corresponding tool is evaluated for an internally used application in an organization. Reflexion model (RM) is created from the code and pre-defined architecture model. RM model is then analyzed for detecting divergence and absence relations for conformance analysis. Study [112], focuses on the architecture constraint analysis to detect whether original architecture and resulting architecture conforms to each other or there are deviations in this relation. Authors propose tool called dclcheck that executes architectural constraints defined by software architect against the code to detect any violation. Detected violations means that there is an erosion in the software architecture which leads to nonconformance between original architecture and the code. Study [118] proposes a tool called ArchRuby to detect architectural violations in the code similar to study [112]. Architectural

rules are defined and executed against source code to detect architectural erosions which leads to nonconformance between original architecture and the resulting code. RM model is created from the rule executions and analyzed for detecting divergence and absences relations. Study [114] presents a tool called LISA to check software systems against the reference architecture defined. At first tool reconstructs architecture models from the code in LISA model format and then apply conformance analysis. Study [119] proposes yet another tool called ConQAT in which graph-based model is reconstructed from code and then if same mappings are checked for existence between original architecture and reconstructed architecture model. Study [120] proposes a tool extends static analysis platform Magellan. Firstly, tool reconstructs software architecture from the code by static analysis of files. Then predefined constraints are executed on the reconstructed architecture to detect deviations in the architecture. Study [71] presents a tool called SAVE, which reconstructs architecture models from code in Data model format. Then reconstructed architecture and original system architecture are checked for absence and divergence relations. Study [115] proposes ExplorViz tool, which is a web-based architecture model visualization tool. Authors argue that manual conformance analysis can be performed using tool by checking architecture model and then analyzing code. Study [121] proposes a tool called SARTE, which takes input of software architecture specifications as form of finite state process model (FSP), test criteria. Tool generates test cases that are mapping architectural constraints to source code level. Study [122] presents a tool called CHARMY which transforms original architecture of software systems into code level tests. Tests are executed against the source code manually to detect the violations against the original architecture.

Table 7-4 reports the summary of the related work for architecture conformance analysis. The first row represents the method that we have proposed in the chapter, while the others represent the related approaches. We have focused on seven different features to characterize these studies. The first column shows whether the indicated approach uses architecture reconstruction or not. It appears that not all studies depend on architecture reconstruction; these methods map architectural specifications to code level testing. Conformance analysis is both done at the architecture level and code level. The table also reports that most of the studies checked absence and divergence relations together and only four studies checked for only one relation. We can also derive from the table that there is also no unified architecture model type for carrying out architectural conformance analysis. The most used architecture model types are RM and HUTN models. None of the studies explicitly focus the notion of viewpoints, but we still derived and interpreted which architecture viewpoint conformance analysis that have been used in the conformance analysis. We can see that uses, decomposition and layered viewpoints are the mostly used viewpoints. Furthermore, we can see most of the studies are using automated tools but there are some tools that are not fully automated and use manual checking. The studies that were checked do not evaluate the approach using a formal case study research protocol. Given this result, we can conclude that our approach is both complementary to the existing approaches and novel in the sense that it provides an integrated viewpoint-oriented approach that is validated within an industrial context.

Table 7-4: Overview of existing architecture conformance analysis and architecture reconstruction method

Tool / Conformance Analysis Feature	Architecture Reconstructio n Applied	Conformance Analysis Execution Level	Checked Relations	Architecture View	Architecture Model Type	Execution	Case Study Protocol Applied
VOSACAM	Yes	Architecture Level	Divergence Absence	Decomposition Shared Data Uses Generalization Layered	HUTN	Automated	Yes
MDABT	No	Code Level	Divergence	Decomposition Shared Data Uses Generalization Layered	HUTN	Automated	Yes
SCHOLIA	Yes	Architecture Level	Divergence Absence	Generalization Uses	OOG	Semi- automated	No
jRMTTool	Yes	Architecture Level	Divergence Absence	Uses	RM	Semi- automated	No
dclcheck	No	Code Level	Divergence Absence	Uses Layered Generalization	NA	Automated	No
ArchRuby	No	Code Level	Divergence Absence	Uses	RM	Automated	No
LISA	Yes	Architecture Level	Divergence Absence	Decomposition Layered Uses Generalization	LISA	Automated	Yes
ConQAT	Yes	Architecture Level	Absence	Decomposition Uses Layered	Graph	Automated	No
Magellan Ext	Yes	Architecture Level	Divergence Absence	Decomposition Uses	LogEn	Automated	No
SAVE	Yes	Architecture Level	Divergence Absence	Decomposition Uses	Data	Semi- Automated	No
ExplorViz	No	NA	Divergence Absence	NA	NA	Manual	No
SARTE	No	Code Level	Absence	Uses Decomposition	FSP	Semi- Automated	No
CHARMY	No	Code Level	Absence	Uses Decomposition	Charmy Specs	Manual	No

7.8 CONCLUSION

Software systems are rarely static and often need to be adapted to fix the bugs or implement new requirements. The architecture drift problem is a problem in many software projects and indicates the discrepancy between the architecture description and the resulting implementation [110]. Ensuring that the architecture drift problem is solved, and thus the code and the architecture are aligned, is important to guide and manage software projects.

In this chapter, we have presented an integrated and model-driven architecture conformance analysis approach that can be used to check the consistency between architecture views and the corresponding reconstructed architecture views from the code. We have adopted five different viewpoints for which we defined the metamodels and domain-specific languages. For each of the viewpoints, we have implemented the software architecture reconstruction method and the viewpoint-oriented software architecture conformance analysis, together with the toolset. We have applied the method and the tool for a case study of a commercial company.

We can conclude that our approach is both complementary to the existing methods and novel in the sense that it provides an integrated viewpoint-oriented approach that is validated within an industrial context. Both the method and the tool are now being used in the company of the case study. In our future work, we plan to apply the approach and the toolset for various case studies.

8

GENERAL DISCUSSION

8.1 INTRODUCTION

Our main objective in this thesis is to define model-driven conformance analysis methodologies to detect the absence and divergence relations of software systems. To achieve our objective, we first conducted a systematic literature review on model-driven architecture-based testing. We derived a generic process model for architecture-based testing and derived our architecture conformance analysis tools based on this process model. Afterwards, we applied design science research for big data e-government systems and derived reference architecture scoping in both big data and e-government systems. We then started implementing code level architecture conformance analysis tool and apply a case study research on big data e-government system. Later, we conducted a domain-driven analysis on architecture reconstruction methods in which we derived a feature model of architecture reconstruction. Furthermore, we implemented view driven architecture reconstruction tool and architecture level architecture conformance analysis tool. Finally, we evaluated our conformance analysis tool on an e-commerce system. The next sections will provide details for addressing research questions and stating corresponding contributions.

8.2 RESEARCH QUESTIONS

The following four research questions have been defined to thesis objective.

RQ1: What are current solutions for model-driven architecture-based testing?

RQ2: What are the limitations and benefits of code level model-based architecture conformance analysis using architecture viewpoints?

RQ3: What are the characteristic features of architecture reconstruction methods?

RQ4: What are the limitations and benefits of architecture level model-based architecture conformance analysis using architecture viewpoints?

We conducted five different research methodologies namely systematic literature review, case study research, survey study, domain-driven analysis and design science research for answering stated research questions.

8.2.1 *What are current solutions for model-driven architecture-based testing?*

In order to develop model-based architecture conformance analysis solutions, we conducted a systematic literature review using Kitchenham et al. guidelines[13]. We thoroughly reviewed 31 primary studies in the field, which are filtered out of 739 studies according to defined study selection criteria. In this work, we answered three research questions for the MDABT domain which are:

- What are the addressed concerns for applying MDABT?
- What are the proposed solutions in MDABT?
- What are the existing research directions within MDABT?

Firstly, we have been able to categorize concerns for applying MDABT into two categories which are: code to architecture conformance checking and internal architecture consistency checking. In our selected list of studies, 27 out of 31 studies addresses the concern for architecture conformance checking in their work. The rest of the studies focuses on internal architecture consistency concerns. Secondly, we derived a generic process model for MDABT

based on selected studies. Figure 2-5 presents the generic process model and Table 2-7 presents the mapping of these studies to our process model. We have explained each primary study in detail at the corresponding sections of this thesis. Lastly, we identified current obstacles and existing research directions based on selected primary studies. We identified seven obstacles for MDABT which are: the need for executable models for representing architecture, coping with the abstraction difference between architecture and code, the combinatorial state explosion problem, the lack of architecture views to consider multiple different concerns, the need for automating the process by tools, the cost-benefit analysis, and the need for applying MDABT on complex real systems to provide evidence of the approaches with respect to conventional approaches. Moving forward, given the derived generic process model for MDABT, one can derive different MDABT approaches to optimize the benefits for future research. Also, MDABT can be utilized for different quality concerns such as safety and security. We based our implementations for architecture conformance checking on this study, in which we utilized architecture viewpoint definitions and industrial case studies to verify our approaches.

Moreover, there are various threats to the validity of our systematic literature review that needs to be addressed. One of the primary dangers to the legitimacy of SLR is publication bias. We overcome the publication bias by performing a research protocol containing research questions. Moreover, incompleteness is another threat to the study. We mitigated away from incompleteness by defining search scope, search method and constructing search strings for different publication platforms. However, search keywords are another danger in incompleteness context. To mitigate this risk, we constructed a keyword list in a repeated manner by pre-performing sample searches in the domain. Publication platforms are missing some study types such as technical reports, theses and company journals which can also contribute to incompleteness of the study. Therefore, such studies have been neglected which can be important contributor in terms of completeness and validation by strong case study. Also, we defined inclusion and exclusion criteria to select studies to address limited functional operations on publication platforms such as length of search queries.

8.2.2 What are the limitations and benefits of code level model-based architecture conformance analysis using architecture viewpoints?

We developed code level architecture conformance analysis tool based on the generic process model derived in our SLR study. We conducted a case study using Runeson et al. guidelines [77] on an e-government system to assess the effectiveness of our developed tool. Also, we conducted a survey study based on Kitchenham et al. guidelines [66][67][68][69][70] for assessing the practicality of our approach. In our approach, we focused on architecture conformance checking concern. Our approach takes architecture view models and translates them into unit tests that can be executed on the code to verify the absence relations. Since we translate architecture models, in this approach we accept architecture models of a system as a source of truth and implemented code should conform to these architecture models. Therefore, we focused on architectural elements that are present in architecture models but missing on implementation of a system. We evaluated our approach and reported our results based on an e-government system both with original and mutant copies of the system. Moreover, we conducted surveys to developers both before and after the execution of our approach. We concluded that our approach was practical and useful but also has a flaw back with the architecture model specification language we utilized. We used EMF based models for

representing architecture and it would have been better if UML was used instead. Lastly, based on test execution results we concluded that our approach is effective for checking architecture conformance relations.

Our case study research has threats to validity like any other case study research. We identified three types of threats to validity in our work. Construct validity refers to misalignment between what researchers measure and what the study suggests. We consider the intended architecture of a software system as a source of the truth for checking divergence relations in our code-level architecture conformance analysis approach. Therefore, we are not checking if the implemented architecture has extra elements with respect to the intended architecture. We also applied fault-based testing techniques to evaluate the effectiveness of our proposed approach on our industrial case. Moreover, we addressed construct validity threats in our survey study. We defined exclusion criteria for sample population and applied probabilistic sampling and convenience sampling to mitigate the risk of selecting the wrong persons for interviews. We provided both open and closed questions to mitigate the risk for misinterpretation of questions by interviewees. We analyzed the total population according to our survey study guideline [67] to mitigate the risk of participants. Internal validity refers to the casual relation between treatment and outcome. We executed our fault-based testing for assessing effectiveness of our approach in an isolated environment where no external variable can have an effect on it. Furthermore, we provided both open and closed questions in our survey study to assess the practicality of our approach for revealing participants explicitly-held and implicitly-held perceptions. Another issue that we find out is the possibility of experimenter effects/the Hawthorne effect. This implies that the participants could have behaved differently due to the researcher's presence during different stages of this study including architecture modelling, testing process and answering survey questions. Further studies should be carried out to reduce the Hawthorne effect. Finally, external validity refers to the concern of generalizing the results of a scientific study. Our approach is validated on a single case study and this implies a threat to the external validity of our study. However, we consider the case study as representative for the challenges that many different systems have to deal with when it comes to coping with architecture consistency. What matters in our context is not the number of case studies but the size and complexity of the systems. Obviously, in practice even larger systems could be identified, but the properties of architecture code consistency will be similar. In addition, to enhance the external validity, although we have one case study, we carried out two different experiments including the one with testing the real code, and the other using an exhaustive fault-based testing approach.

8.2.3 *What are the characteristic features of architecture reconstruction methods?*

We conducted domain-driven analysis to derive a feature model for software architecture reconstruction methods. The derived feature model, which can be seen in Figure 5-3, contains common and variant features for SAR methods. To this end, we selected and analysed a set of papers for SAR methods. We discovered a set of repeating common and variant features for SAR methods. Furthermore, we based our SAR method implementation on the derived feature model for our second architecture conformance analysis approach. During our research, we discovered the need for a well-defined SLR study and this can be conducted as a future research.

Furthermore, we need to address concerns for threats to the validity of our approach. First of all, we did not choose a systematic literature review for deriving our domain feature model. Although, we applied a domain analysis process in which we selected a sufficient set of studies that we believed to cover the adopted features. We would have selected more studies if we had chosen a systematic literature review, but we can confidently state that the studies that we have selected for our review appeared to be repeating the similar set of common and variant features. This proved the convergence of the domain model and increased our confidence in the validity of the derived feature model. The family feature model for the SAR methods defines the common and variant features that can be used to describe individual SAR methods. With the feature model, we indeed derive many different alternative SAR methods, describe existing SAR methods, and support the identification of novel SAR methods. Although possible, in this study our goal was not to provide a survey of existing SAR methods. We aim to provide this in our future work.

8.2.4 What are the limitations and benefits of architecture level model-based architecture conformance analysis using architecture viewpoints?

We conducted our second case study for architecture level architecture conformance analysis approach in chapter 7 of this thesis. Similar to our previous case study, we based our case study on Runeson et al. guidelines [77] for an industrial e-commerce system. Both of our implemented approaches are checking for architecture conformance using model-based testing which are on different levels. In this approach, we both check for absence and divergence relations for conformance analysis. We reconstructed the software architecture from the system's code which results in the implemented architecture of a system. Then, we check it against the intended architecture which is the documented architecture of the system. We find and report both architectural elements that are missing or additional with respect to intended architecture. We can conclude that our approach is useful and effective in detecting both absence and divergence relations for architecture conformance analysis given our case study results. However, since we use EMF based models which are not industry standard for documenting software architecture, there is additional work needs to be done to translate documented architecture models to our architecture view models.

Like any other case study, our case study also has threats to its validity. While assessing the effectiveness of the suggested approach, we have applied formal fault-based testing and reported the results in an isolated environment that no external variable can affect. One can argue as a threat to internal validity that injected violations are not real-world system faults since we applied exhaustive fault-based testing. However, faults of our approach aimed to detect the change in real-world systems. Nevertheless, our experiments are executed on the real industrial case. Moreover, we executed case study research on a single B2C software system and this suggests threats to external validity. However, the architecture deviation problem is not related to a specific domain but a characteristic problem which might occur when the software is developed. Although our approach is evaluated on a single software system, what matters is the size and complexity of the software system. In our future works, we will apply our architecture-level architecture conformance analysis to different software systems.

APPENDICES

Appendix-A Search Strings

Appendix-B List of Primary Studies

Appendix-C Study Quality Assessment

Appendix-D Data Extraction Form

Appendix-E Primary Studies for Software Architecture Models

APPENDIX-A SEARCH STRINGS

Electronic Database	Search String
IEEE Xplore	<p>(("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</p> <p>(("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")</p>
ACM Digital Library	<p>acmdlTitle:(+architecture +software +test "model" "based" "driven")</p> <p>OR</p> <p>recordAbstract:(+architecture +test +software +model +("based" "driven"))</p>
Wiley Interscience	(model test software architecture in Abstract) OR (model test software architecture in Publication Titles)
Science Direct	<p>((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</p>

	<p>Title(model driven test) OR Title(model driven software test) OR</p> <p>Title(model-driven test) OR Title(model-driven software test)) AND</p> <p>(Title(architecture)))</p> <p>OR (</p> <p>Title(architecture based testing) OR Title(architecture based software testing) OR</p> <p>Title(architecture-based testing) OR Title(architecture-based software testing) OR</p> <p>Title(architecture driven testing) OR Title(architecture driven software testing) OR</p> <p>Title(architecture-driven testing) OR Title(architecture-driven software testing) OR</p> <p>Title(architecture based test) OR Title(architecture based software test) OR</p> <p>Title(architecture-based test) OR Title(architecture-based software test) OR</p> <p>Title(architecture driven test) OR Title(architecture driven software test) OR</p> <p>Title(architecture-driven test) OR Title(architecture-driven software test)))</p> <p>OR</p> <p>((Abstract(model based testing) OR Abstract(model based software testing) OR</p> <p>Abstract(model-based testing) OR Abstract(model-based software testing) OR</p> <p>Abstract(model driven testing) OR Abstract(model driven software testing) OR</p> <p>Abstract(model-driven testing) OR Abstract(model-driven software testing) OR</p> <p>Abstract(model based test) OR Abstract(model based software test) OR</p> <p>Abstract(model driven test) OR Abstract(model driven software test) OR</p> <p>Abstract(model-driven test) OR Abstract(model-driven software test)) AND</p> <p>(Abstract(architecture)))</p> <p>OR (</p> <p>Abstract(architecture based testing) OR Abstract(architecture based software testing) OR</p> <p>Abstract(architecture-based testing) OR Abstract(architecture-based software testing) OR</p> <p>Abstract(architecture driven testing) OR Abstract(architecture driven software testing) OR</p> <p>Abstract(architecture-driven testing) OR Abstract(architecture-driven software testing) OR</p> <p>Abstract(architecture based test) OR Abstract(architecture based software test) OR</p> <p>Abstract(architecture-based test) OR Abstract(architecture-based software test) OR</p> <p>Abstract(architecture driven test) OR Abstract(architecture driven software test) OR</p> <p>Abstract(architecture-driven test) OR Abstract(architecture-driven software test)))</p>
Springer	'software AND architecture AND test AND model AND "based testing" AND (based OR driven) AND NOT (hardware AND circuit)'
ISI Web of Knowledge	<p>((TS="model based testing" OR TS="model based software testing" OR</p> <p>TS="model-based testing" OR TS="model-based software testing" OR</p> <p>TS="model driven testing" OR TS="model driven software testing" OR</p> <p>TS="model-driven testing" OR TS="model-driven software testing" OR</p> <p>TS="model based test" OR TS="model based software test" OR</p> <p>TS="model driven test" OR TS="model driven software test" OR</p> <p>TS="model-driven test" OR TS="model-driven software test") AND</p> <p>(TS="architecture"))</p> <p>OR (</p> <p>TS="architecture based testing" OR TS="architecture based software testing" OR</p> <p>TS="architecture-based testing" OR TS="architecture-based software testing" OR</p> <p>TS="architecture driven testing" OR TS="architecture driven software testing" OR</p> <p>TS="architecture-driven testing" OR TS="architecture-driven software testing" OR</p> <p>TS="architecture based test" OR TS="architecture based software test" OR</p> <p>TS="architecture-based test" OR TS="architecture-based software test" OR</p>

	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. Bertolino, A.; Corradini, F.; Inverardi, P.; Muccini, H. (2000). Deriving Test Plans from Architectural Descriptions. pp. 220--229.
- B. Jin, Z.; Offutt, J. (2001). Deriving Tests From Software Architectures. pp. 308--313.
- C. Muccini, H.; Bertolino, A.; Inverardi, P. (2004). Using Software Architecture for Code Testing. pp. 160--171.
- D. Muccini, H.; Dias, M.; Richardson, D. J. (2004). Systematic Testing of Software Architectures in the C2 Style. pp. 295--309.
- E. A. Bucchiarone, H. Muccini, P. Pelliccione, P. Pierini. (2004). Model-Checking plus Testing from Software Architecture Analysis to Code Testing. pp. 351--365.
- F. Scollo, G.; Zecchini, S. (2005). Architectural Unit Testing. pp. 27--52.
- G. Winbladh, K.; Alspaugh, A. T.; Ziv, H.; Richardson, D. (2006). Architecture Based Testing Using Goals and Plans. pp. 64--68.
- H. Muccini, H.; Dias, M.; Richardson, J. D. (2006). Software Architecture-Based Regression Testing. pp. 1--18.
- I. Johnsen, A.; Pettersson, P.; Lundqvist, K. (2009). An Architecture-Based Verification Technique for AADL Specifications. pp. 105--113.
- J. Reza, H.; Lande, S. (2010). Model Based Testing Using Software Architecture. pp. 188--193.
- K. Keum, C.; Kang, S.; Kim, M. (2013). Architecture-Based Testing of Service-Oriented Applications In Distributed Systems. pp. 1212--1223.
- L. Lochau, M.; Lity, S.; Lachmann, R.; Schaefer, I.; Goltz, U. (2013). Delta-oriented Model-Based Integration Testing of Large-Scale Systems. pp. 63--84.
- M. Li, N.; Escolana, Anthony.; Kamal, T. (2016). Skyfire: Model-Based Testing With Cucumber. pp. 393--400.
- N. Elallaoui, M.; Nafil, K.; Touahni, R.; Messoussi, R. (2016). Automated Model Driven Testing Using AndroMDA and UML2 Testing Profile in Scrum Process. pp. 221--228.
- O. Bertolino, A., Inverardi, P., & Muccini, H. (2001, July). An explorative journey from architectural tests definition down to code tests execution. In Proceedings of the 23rd International Conference on Software Engineering (pp. 211-220). IEEE Computer Society.
- P. Javed, A. Z., Strooper, P. A., & Watson, G. N. (2007, May). Automated generation of test cases using model-driven architecture. In Automation of Software Test, 2007. AST'07. Second International Workshop on (pp. 3-3). IEEE.
- Q. Lalchandani, J. T., & Mall, R. (2008, February). Regression testing based-on slicing of component-based software architectures. In Proceedings of the 1st India software engineering conference (pp. 67-76). ACM.
- R. Lima, B., & Faria, J. P. (2015, July). An approach for automated scenario-based testing of distributed and heterogeneous systems. In Software Technologies (ICSOFTE), 2015 10th International Joint Conference on (Vol. 1, pp. 1-10). IEEE.
- S. Iyengar, P., Westerkamp, C., Wuebbelmann, J., & Pulvermueller, E. (2010, September). An architecture for deploying model based testing in embedded systems. In Specification & Design Languages (FDL 2010), 2010 Forum on (pp. 1-6). IET.
- T. Dan, H., & Hierons, R. M. (2011, March). Conformance testing from message sequence charts. In Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on (pp. 279-288). IEEE.
- U. Havlice, Z., Szabóová, V., & Vízi, J. (2013, January). Critical knowledge representation for model-based testing of embedded systems. In Applied Machine Intelligence and Informatics (SAMI), 2013 IEEE 11th International Symposium on (pp. 169-174). IEEE.

- V. Caliebe, P., Herpel, T., & German, R. (2012, April). Dependency-based test case selection and prioritization in embedded systems. In *Software Testing, Verification and Validation (ICST)*, 2012 IEEE Fifth International Conference on (pp. 731-735). IEEE.
- W. Carrozza, G., Faella, M., Fucci, F., Pietrantuono, R., & Russo, S. (2012, November). Integrating mdt in an industrial process in the air traffic control domain. In *Software Reliability Engineering Workshops (ISSREW)*, 2012 IEEE 23rd International Symposium on (pp. 225-230). IEEE.
- X. Wu, C. S., & Huang, C. H. (2013, May). The web services composition testing based on extended finite state machine and UML model. In *Service Science and Innovation (ICSSI)*, 2013 Fifth International Conference on (pp. 215-222). IEEE.
- Y. Schulz, S., Honkola, J., & Huima, A. (2007, March). Towards model-based testing with architecture models. In *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the* (pp. 495-502). IEEE.
- Z. Buchs, D., Pedro, L., & Lúcio, L. (2006). Formal test generation from UML models. *Research Results of the DICS Program*, 4028, 145-171.
- AA. Lamancha, B. P., Reales, P., Polo, M., & Caivano, D. (2011, June). Model-Driven Test Code Generation. In *International Conference on Evaluation of Novel Approaches to Software Engineering* (pp. 155-168). Springer, Berlin, Heidelberg.
- BB.Lima, B. M. C., & Faria, J. C. P. (2017, March). Towards Decentralized Conformance Checking in Model-Based Testing of Distributed Systems. In *Software Testing, Verification and Validation Workshops (ICSTW)*, 2017 IEEE International Conference on (pp. 356-365). IEEE.
- CC.Wu, C. S., & Lee, Y. T. (2012, December). Automatic SaaS test cases generation based on SOA in the cloud service. In *Cloud Computing Technology and Science (CloudCom)*, 2012 IEEE 4th International Conference on (pp. 349-354). IEEE.
- DD. Xu, D. (2011). A tool for automated test code generation from high-level Petri nets. *Applications and Theory of Petri Nets*, 308-317.
- EE.Wehrmeister, M. A., & Berkenbrock, G. R. (2013, June). Automatic Execution of Test Cases on UML Models of Embedded Systems. In *International Embedded Systems Symposium* (pp. 39-48). Springer, Berlin, Heidelberg.

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
I	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
M	1	1	1	1	0,5	1	0	1	1	1	8,5
N	1	1	1	1	0,5	1	0	1	1	0,5	8
O	1	1	1	1	1	1	0	1	1	0,5	8,5
P	1	1	1	1	0,5	1	0	1	1	0,5	8
Q	1	1	1	1	1	1	0	1	1	1	9
R	1	1	1	1	0,5	1	0	1	1	1	8,5
S	1	1	1	1	1	1	0	1	0,5	1	8,5
T	1	1	1	1	0,5	1	0	1	1	1	8,5
U	1	1	1	1	1	1	0	1	0,5	1	8,5
V	1	1	1	1	1	1	0	1	1	1	9
W	1	1	1	1	1	1	0	1	1	1	9
X	1	1	1	1	0,5	1	0	0,5	1	0,5	7,5

Y	1	1	1	1	0,5	0,5	0	0,5	1	0,5	7
Z	1	1	1	1	1	1	0	1	1	1	9
AA	1	1	1	1	1	1	0	1	1	1	9
BB	1	1	1	0,5	1	1	0	1	1	0,5	8
CC	1	1	1	1	1	1	0	1	0,5	0,5	8
DD	1	1	1	1	1	1	0	1	1	0,5	8,5
EE	1	1	1	0,5	1	1	0	1	1	0,5	8

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		
10	Addressed Concern	☐ Code to Architecture Conformity ☐ Internal Architectural Consistency Checking
11	Test Criteria	☐ Coverage Criteria ☐ Test Purpose Matching
12	Software Architecture Description Language	CHAM, Wright ADL, FSP Model, UML Diagrams, GoalML, AADL, Acme ADL, BPEL
14	Test Model	LTS , BG, Promela, LTL Formulae, Buchi Automata, IOLTS, Uppaal, HPrTNS, ECFG, eDeltaModels, Test Graph, U2TP Diagram
15	Test Case Execution	☐ Automatic ☐ Manual
16	Test Case Generation	☐ Automatic ☐ Manual
17	Test Oracle	☐ Automatic ☐ Manual
18	Assessment Approach	☐ Case Study ☐ Experiment ☐ Small Example
19	Findings	
20	Constraints / Limitations	
Evaluation		
21	Personal note	The opinions of the reviewer about the study
22	Additional note	Publication details
23	Quality Assessment	Detailed quality scores

APPENDIX-E Primary Studies for Software Architecture Reconstruction Methods

Primary Study	Reference
Anquetil, N., & Lethbridge, T. C. (1999). Recovering software architecture from the names of source files. <i>Journal of Software Maintenance: Research and Practice</i> , 11(3), 201-221.	[82]
El Boussaidi, G., Belle, A. B., Vaucher, S., & Mili, H. (2012, October). Reconstructing architectural views from legacy systems. In <i>2012 19th Working Conference on Reverse Engineering</i> (pp. 345-354). IEEE.	[83]
Van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., & Riva, C. (2004, June). Symphony: View-driven software architecture reconstruction. In <i>Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on</i> (pp. 122-132). IEEE.	[85]
von Detten, M. (2012, October). Archimetrix: A tool for deficiency-aware software architecture reconstruction. In <i>Reverse Engineering (WCRE), 2012 19th Working Conference on</i> (pp. 503-504). IEEE.	[87]
Favre, J. M. (2004, November). Cacophony: Metamodel-driven software architecture reconstruction. In <i>Reverse Engineering, 2004. Proceedings. 11th Working Conference on</i> (pp. 204-213). IEEE.	[89]
Fiutem, R., Antoniol, G., Tonella, P., & Merlo, E. (1999). Art: an architectural reverse engineering environment. <i>Journal of Software Maintenance: Research and Practice</i> , 11(5), 339-364.	[90]
Fontana, F. A., & Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. <i>Information sciences</i> , 181(7), 1306-1324.	[91]
Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., & Di Salle, A. (2017, April). Microart: A software architecture recovery tool for maintaining microservice-based systems. In <i>IEEE International Conference on Software Architecture (ICSA)</i> .	[95]
Guo, G. Y., Atlee, J. M., & Kazman, R. (1999). A software architecture reconstruction method. In <i>Software Architecture</i> (pp. 15-33). Springer, Boston, MA.	[96]

Maqbool, O., & Babri, H. A. (2007, April). Bayesian learning for software architecture recovery. In <i>Electrical Engineering, 2007. ICEE'07. International Conference on</i> (pp. 1-6). IEEE.	[98]
Risi, M., Scanniello, G., & Tortora, G. (2012). Using fold-in and fold-out in the architecture recovery of software systems. <i>Formal Aspects of Computing</i> , 24(3), 307-330.	[99]
Riva, C. (2002). Architecture reconstruction in practice. In <i>Software Architecture</i> (pp. 159-173). Springer, Boston, MA.	[100]
Roy, B., & Graham, T. N. (2008, September). An iterative framework for software architecture recovery: An experience report. In <i>European Conference on Software Architecture</i> (pp. 210-224). Springer, Berlin, Heidelberg.	[101]
Sartipi, K., & Kontogiannis, K. (2003, September). On modeling software architecture recovery as graph matching. In <i>International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.</i> (pp. 224-234). IEEE. Maintenance, 2003. ICSM 2003. Proceedings. International Conference on (pp. 224-234). IEEE.	[102]
Şora, I., Glodean, G., & Gligor, M. (2010, May). Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In <i>Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on</i> (pp. 259-264). IEEE.	[103]
Stoermer, C., Rowe, A., O'Brien, L., & Verhoef, C. (2006). Model-centric software architecture reconstruction. <i>Software: Practice and Experience</i> , 36(4), 333-363.	[105]
Stringfellow, C., Amory, C. D., Potnuri, D., Andrews, A., & Georg, M. (2006). Comparison of software architecture reverse engineering methods. <i>Information and Software Technology</i> , 48(7), 484-497.	[106]

REFERENCES

- [1]. [ISO/IEC/IEEE 42010:2011] Systems and Software Engineering – Architecture Description ([ISO/IEC/IEEE 42010), Oct 2020.
- [2]. [ANSI/IEEE 1059:1993] Software Verification and Validation Plan (ANSI/IEEE 1059), May 2014.
- [3]. Baier, C., Katoen, J.P., Larsen, K.G. Principles of Model Checking, MIT Press, 2008
- [4]. Clarke, E., Grumberg, O., and Peled. D. Model Checking. MIT Press, 2000.
- [5]. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. Documenting Software Architectures: Views and Beyond. Second Edition. Addison-Wesley, 2010.
- [6]. Demirli, E., Tekinerdogan, B. Software Language Engineering of Architectural Viewpoints, in Proc. of the 5th European Conference on Software Architecture (ECSA 2011), LNCS 6903, pp. 336–343, 2011.
- [7]. Felderer, M., Zech, P., Breu, R., Büchler, M., Pretschne, A. Model-based security testing: a taxonomy and systematic classification, Journal of Software: Testing, Verification and Reliability, DOI: 10.1002/stvr.1580, 2015.
- [8]. Fowler, M. Domain-specific Languages. Addison-Wesley Professional, 2011.
- [9]. Garousi, V., Mäntylä, Mika V. When and what to automate in software testing? A multi-vocal literature review. Information and Software Technology. 76: 92–117. doi:10.1016/j.infsof.2016.04.015, 2016.
- [10]. Gurbuz, H.G., Tekinerdogan, B. Model-Based Testing for Software Safety: A Systematic Literature Review, Software Quality Journal, DOI 10.1007/s11219-017-9386-2, 2017.
- [11]. Hartman, A., Katara, M., & Olvovsky, S. Choosing a Test Modeling Language: A Survey. doi:10.1007/978-3-540-70889-6_16, 2006
- [12]. ISO/IEC 42010:2007] Recommended practice for architectural description of software-intensive systems (ISO/IEC 42010), 2011.
- [13]. Kitchenham, B., Charters, S. 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, 2007.
- [14]. Kitchenham, B., Budgen, D., Brereton, P. Evidence-Based Software Engineering and Systematic Reviews, CRC Press Taylor & Francis Group, 2016.
- [15]. Mäkinen, M.A., Model Based Approach to Software Testing. Thesis, Helsinki University of Technology Department of Computer Science and Engineering Software Business and Engineering Institute, 2007.
- [16]. Myers, G.J., Sandler, C., Badgett, T. The Art of Software Testing, John Wiley & Sons, Inc, Hoboken, New Jersey, 2012.
- [17]. Muccini, H., Bertolino, A., and Inverardi, P. Using Software Architecture for Code Testing, IEEE Trans. Software Eng., vol. 29, no. 3, pp. 160-171, Mar. 2003.
- [18]. Mellor, Stephen J., et al. MDA distilled: principles of model-driven architecture. Addison-Wesley Professional, 2004.
- [19]. Mistrik, Ivan, et al., eds. Software quality assurance: in large scale and complex software-intensive systems. Morgan Kaufmann, 2015.
- [20]. Murphy, Gail C., David Notkin, and Kevin J. Sullivan. "Software reflexion models: Bridging the gap between design and implementation." IEEE Transactions on Software Engineering 27.4 (2001): 364-380.

- [21]. Neto A. C. D., Subramanyan R., Vieira M., Travassos G. H. A Survey on Model-based Testing Approaches: A Systematic Review. Proceedings of the 2007 ACM symposium on Applied Computing, 2007.
- [22]. Nguyen, Cu.D., Marchetto, A., and Tonella, P. Combining model-based and combinatorial testing for effective test case generation. In Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012). ACM, New York, USA, 100-110. DOI=<http://dx.doi.org/10.1145/2338965.2336765>, 2012
- [23]. Rafi, D.M.; Moses, K.R.K.; Petersen, K.; Mantyla, M.V., Benefits and limitations of automated software testing: Systematic literature review and practitioner survey, Automation of Software Test (AST), 2012 7th International Workshop on, vol., no., pp.36, 42, 2 -3, June 2012.
- [24]. Souza, N.M., Dias, D., Ruas de Oliveira, L.B., Aparecida Lana, C., Yumi Nakagawa, E., Maldonado, J.C. Exploring together software architecture and software testing: A systematic mapping, International Conference of the Chilean Computer Science Society - SCCC2016, At Valparaíso, Chile, Volume: 35, October 2016
- [25]. Tekinerdogan, B., Ali, N., Grundy, J., Mistrik, I., Soley, R. Quality Concerns in Large Scale and Complex Software-intensive Systems. In: I. Mistrik, R. Soley, N. Ali, J. Grundy, B. Tekinerdogan. Software Quality Assurance in Large Scale and Complex Software-Intensive Systems Elsevier, -p.1-17, 2015.
- [26]. Tekinerdogan, B. Software Architecture, chapter in: T. Gonzalez and J.L. Díaz-Herrera, Computer Science Handbook, Second Edition, Volume I: Computer Science and Software Engineering, Taylor and Francis, 2014.
- [27]. Tekinerdogan, B. Architectural Drift Analysis using Design Structure Reflexion Matrices. In: I. Mistrik, R. Soley, N. Ali, J. Grundy, B. Tekinerdogan. Software Quality Assurance in Large Scale and Complex Software-Intensive Systems Elsevier, - p. 221 - 236.
- [28]. Utting, M., Pretschner, A., Legeard, B. A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability, 2011.
- [29]. Utting, M., Legeard, B. Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann, 2007.
- [30]. Zhang, P., Muccini, H., and Li, B. A Classification and Comparison of Model Checking Software Architecture Techniques, Journal of Systems and Software, vol. 83, no. 5, pp. 723-744, 2010.
- [31]. Almarabeh, Tamara, and Amer AbuAli. "A general framework for e-government: definition maturity challenges, opportunities, and success." European Journal of Scientific Research 39.1 (2010): 29-42.
- [32]. Angelov, Samuil, Paul Grefen, and Danny Greefhorst. "A classification of software reference architectures: Analyzing their success and effectiveness." 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture. IEEE, 2009.
- [33]. Ballard, Chuck, et al. Information governance principles and practices for a big data landscape. IBM Redbooks, 2014.
- [34]. Begoli, Edmon, and James Horey. "Design principles for effective knowledge discovery from big data." 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture. IEEE, 2012.
- [35]. Chapelle , D. 2013. Big Data & Analytics Reference Architecture, An Oracle White Chapter.
- [36]. Clements, P. Northrop, L. Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., 2001.

-
- [37]. Debnath, Narayan, et al. "A feature model of E-government systems integrated with formal specifications." Fifth International Conference on Information Technology: New Generations (itng 2008). IEEE, 2008.
 - [38]. Galster, Matthias, and Paris Avgeriou. "Empirically-grounded reference architectures: a proposal." Proceedings of the joint ACM SIGSOFT conference--QoSA and ACM SIGSOFT symposium--ISARCS on Quality of software architectures--QoSA and architecting critical systems--ISARCS. 2011.
 - [39]. Geerdink, Bas. "A reference architecture for big data solutions introducing a model to perform predictive analytics using big data technology." 8th international conference for internet technology and secured transactions (ICITST-2013). IEEE, 2013.
 - [40]. Gorton, Ian, and John Klein. "Distribution, data, deployment: Software architecture convergence in big data systems." IEEE Software 32.3 (2014): 78-85.
 - [41]. Kang, Kyo C., et al. Feature-oriented domain analysis (FODA) feasibility study. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.
 - [42]. Kassahun, Ayalew, Rob JM Hartog, and Bedir Tekinerdogan. "Realizing chain-wide transparency in meat supply chains based on global standards and a reference architecture." Computers and Electronics in Agriculture 123 (2016): 275-291.
 - [43]. Data, Big. "Principles and Best Practices of Scalable Realtime Data Systems, by Nathan Marz." James Warren 328 (2015).
 - [44]. Medjahed, Brahim, et al. "Infrastructure for e-government web services." IEEE Internet Computing 7.1 (2003): 58-65.
 - [45]. Oracle, 2013. Information Management and Big Data A Reference Architecture, An Oracle White Chapter.
 - [46]. Öztürk, Karahan, and Bedir Tekinerdogan. "Feature modeling of software as a service domain to support application architecture design." International Conference on Software Engineering Advances. IARIA. 2011.
 - [47]. Pääkkönen, Pekka, and Daniel Pakkala. "Reference architecture and classification of technologies, products and services for big data systems." Big data research 2.4 (2015): 166-186.
 - [48]. Salma, Cigdem Avci, Bedir Tekinerdogan, and Ioannis N. Athanasiadis. "Feature Driven Survey of Big Data Systems." IoTBD. 2016.
 - [49]. Sang M. Lee, X. Tan, and S. Trimi. Current practices of leading e-government countries. Communications. ACM 48, 10 (October 2005), 99-104. DOI=<http://dx.doi.org/10.1145/1089107.1089112>, 2005.
 - [50]. Tekinerdoğan, Bedir, Sevcen Bilir, and Cem Abatlevi. "Integrating platform selection rules in the model driven architecture approach." Model driven architecture. Springer, Berlin, Heidelberg, 2004. 159-173.
 - [51]. Mahmood, Zaigham. Software engineering frameworks for the cloud computing paradigm. Ed. Saqib Saeed. London: Springer, 2013.
 - [52]. Tekinerdogan, Bedir, Hasan Sozer, and Mehmet Aksit. "Feature-based rationale management system for supporting software architecture adaptation." International Journal of Software Engineering and Knowledge Engineering 22.07 (2012): 945-964.
 - [53]. Tekinerdoğan, Bedir, and Mehmet Akşit. "Classifying and evaluating architecture design methods." Software Architectures and Component Technology. Springer, Boston, MA, 2002. 3-27.
 - [54]. Tüzün, Eray, et al. "Empirical evaluation of a decision support model for adopting software product line engineering." Information and Software Technology 60 (2015): 77-101.

- [55]. Yin, R. K. "Case Study Research, design and methods, Sage Publications Inc. Thousand Oaks, California." (2003).
- [56]. Ali, N., Baker, S., O’Crowley, R., Herold, S., & Buckley, J. (2018). Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, 23(1), 224-258.
- [57]. Antoniol, G., Canfora, G., Casazza, G., & De Lucia, A. (2001). Maintaining traceability links during object-oriented software evolution. *Software: Practice and Experience*, 31(4), 331-355.
- [58]. Barbosa, R., Silva, N., Duraes, J., & Madeira, H. (2007, February). Verification and validation of (real time) COTS products using fault injection techniques. In *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2007. ICCBSS'07. Sixth International IEEE Conference on* (pp. 233-242). IEEE.
- [59]. Bertolino, A., Corradini, F., Inverardi, P., & Muccini, H. (2000, June). Deriving test plans from architectural descriptions. In *Proceedings of the 22nd international conference on Software engineering* (pp. 220-229). ACM.
- [60]. Díaz-Pace, J. A., Soria, Á., Rodríguez, G., & Campo, M. R. (2012). Assisting conformance checks between architectural scenarios and implementation. *Information and Software Technology*, 54(5), 448-466.
- [61]. Epsilon – model driven development IDE, <http://eclipse.org/epsilon>, last accessed on April, 2015.
- [62]. Javed, A. Z., Strooper, P. A., & Watson, G. N. (2007, May). Automated generation of test cases using model-driven architecture. In *Automation of Software Test, 2007. AST'07. Second International Workshop on* (pp. 3-3). IEEE.
- [63]. Jia, Y., & Harman, M. (2009). Higher order mutation testing. *Information and Software Technology*, 51(10), 1379-1393.
- [64]. Jin, Z., & Offutt, J. (2001, November). Deriving tests from software architectures. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on* (pp. 308-313). IEEE.
- [65]. Johnsen, A., Pettersson, P., & Lundqvist, K. (2011). An architecture-based verification technique for AADL specifications. *Software Architecture*, 105-113.
- [66]. Kitchenham, B. A. (2002). Principles of Survey Research Part 2: Designing a Survey Barbara A. Kitchenham Department of Computer Science Keele University, Staffordshire, UK. *ACM SIGSOFT Software Engineering Notes*, 27(1), 18..
- [67]. Kitchenham, B. A., Pfleeger, S. L.(2002). Principles of survey research part 3: constructing a survey instrument. *Software Engineering Notes*, 27(2), 20-24.
- [68]. Kitchenham, B., & Pfleeger, S. L. (2002). Principles of survey research part 4: questionnaire evaluation. *SIGSOFT Softw. Eng. Notes*, 27(3), 20-23.
- [69]. Kitchenham, B., & Pfleeger, S. L. (2002). Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Software Engineering Notes*, 27(5), 17-20.
- [70]. Kitchenham, B., & Pfleeger, S. L. (2003). Principles of survey research part 6: data analysis. *ACM SIGSOFT Software Engineering Notes*, 28(2), 24-27.
- [71]. Knodel, J., & Popescu, D. (2007, January). A comparison of static architecture compliance checking approaches. In *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on* (pp. 12-12). IEEE.
- [72]. Koschke, R., & Simon, D. (2003, November). Hierarchical Reflexion Models. In *WCRE (Vol. 3, pp. 186-208)*.
- [73]. Morell, L. J. (1990). A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8), 844-857.

-
- [74]. Michal, Y. (2008). *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons.
- [75]. Reza, H., & Lande, S. (2010, April). Model based testing using software architecture. In *Information Technology: New Generations (ITNG)*, 2010 Seventh International Conference on (pp. 188-193). IEEE.
- [76]. Rosik, J., Le Gear, A., Buckley, J., Babar, M. A., & Connolly, D. (2011). Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience*, 41(1), 63-86.
- [77]. Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2), 131.
- [78]. Tekinerdogan, B., & Demirli, E. (2013, June). Evaluation framework for software architecture viewpoint languages. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures* (pp. 89-98). ACM.
- [79]. Terra, R., Valente, M. T., Czarnecki, K., & Bigonha, R. S. (2015). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3), 315-342.
- [80]. Uzun, B., & Tekinerdogan, B. (2018). Model-Driven Architecture Based Testing: A Systematic Literature Review. *Information and Software Technology*.
- [81]. Voas, J. M., & McGraw, G. (1997). *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc.
- [82]. Anquetil, N., & Lethbridge, T. C. (1999). Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11(3), 201-221.
- [83]. El Boussaidi, G., Belle, A. B., Vaucher, S., & Mili, H. (2012, October). Reconstructing architectural views from legacy systems. In *2012 19th Working Conference on Reverse Engineering* (pp. 345-354). IEEE.
- [84]. Czarnecki, K., Hwan, C., Kim, P., & Kalleberg, K. T., 2006. Feature models are views on ontologies. In *Software Product Line Conference, 2006 10th International* (pp. 41-51). IEEE.
- [85]. Van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., & Riva, C. (2004, June). Symphony: View-driven software architecture reconstruction. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on* (pp. 122-132). IEEE.
- [86]. van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C.: Viewpoints in Software Architecture Reconstruction. In: *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 122–132 (2004)
- [87]. von Detten, M. (2012, October). Archimetrix: A tool for deficiency-aware software architecture reconstruction. In *Reverse Engineering (WCRE), 2012 19th Working Conference on* (pp. 503-504). IEEE.
- [88]. Duenas, J. C., de Oliveira, W. L., & de la Puente, J. A. (1998, March). Architecture recovery for software evolution. In *Software Maintenance and Reengineering, 1998. Proceedings of the Second Euromicro Conference on* (pp. 113-119). IEEE.
- [89]. Favre, J. M. (2004, November). Cacophony: Metamodel-driven software architecture reconstruction. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on* (pp. 204-213). IEEE.
- [90]. Fiutem, R., Antoniol, G., Tonella, P., & Merlo, E. (1999). Art: an architectural reverse engineering environment. *Journal of Software Maintenance: Research and Practice*, 11(5), 339-364.

- [91]. Fontana, F. A., & Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Information sciences*, 181(7), 1306-1324.
- [92]. Garcia, J., Ivkovic, I., & Medvidovic, N. (2013, November). A comparative analysis of software architecture recovery techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (pp. 486-496). IEEE Press.
- [93]. Gorton, I., & Zhu, L. (2005, May). Tool support for just-in-time architecture reconstruction and evaluation: an experience report. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on* (pp. 514-523). IEEE.
- [94]. Guamán, D., Pérez, J., & Díaz, J. (2018, September). Towards a (semi)-automatic reference process to support the reverse engineering and reconstruction of software architectures. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings* (p. 8). ACM.
- [95]. Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., & Di Salle, A. (2017, April). Microart: A software architecture recovery tool for maintaining microservice-based systems. In *IEEE International Conference on Software Architecture (ICSA)*.
- [96]. Guo, G. Y., Atlee, J. M., & Kazman, R. (1999). A software architecture reconstruction method. In *Software Architecture* (pp. 15-33). Springer, Boston, MA.
- [97]. Koschke, R. (2006). Architecture reconstruction. In *Software Engineering* (pp. 140-173). Springer, Berlin, Heidelberg.
- [98]. Maqbool, O., & Babri, H. A. (2007, April). Bayesian learning for software architecture recovery. In *Electrical Engineering, 2007. ICEE'07. International Conference on* (pp. 1-6). IEEE.
- [99]. Risi, M., Scanniello, G., & Tortora, G. (2012). Using fold-in and fold-out in the architecture recovery of software systems. *Formal Aspects of Computing*, 24(3), 307-330.
- [100]. Riva, C. (2002). Architecture reconstruction in practice. In *Software Architecture* (pp. 159-173). Springer, Boston, MA.
- [101]. Roy, B., & Graham, T. N. (2008, September). An iterative framework for software architecture recovery: An experience report. In *European Conference on Software Architecture* (pp. 210-224). Springer, Berlin, Heidelberg.
- [102]. Sartipi, K., & Kontogiannis, K. (2003, September). On modeling software architecture recovery as graph matching. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on* (pp. 224-234). IEEE.
- [103]. Şora, I., Glodean, G., & Gligor, M. (2010, May). Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on* (pp. 259-264). IEEE.
- [104]. Sozer, H., & Tekinerdogan, B. Introducing recovery style for modeling and analyzing system recovery, in *Proc. of Seventh Working IEEE/IFIP Conference on Software Architecture*, pp 167-176, 2008.
- [105]. Stoermer, C., Rowe, A., O'Brien, L., & Verhoef, C. (2006). Model-centric software architecture reconstruction. *Software: Practice and Experience*, 36(4), 333-363.
- [106]. Stringfellow, C., Amory, C. D., Potnuri, D., Andrews, A., & Georg, M. (2006). Comparison of software architecture reverse engineering methods. *Information and Software Technology*, 48(7), 484-497.

-
- [107]. Zahid, M., Mehmmod, Z., & Inayat, I. (2017, December). Evolution in software architecture recovery techniques—A survey. In *Emerging Technologies (ICET), 2017 13th International Conference on* (pp. 1-6). IEEE.
- [108]. Pfleeger, S. L., & Kitchenham, B. A. (2001). Principles of survey research: part 1: turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes*, 26(6), 16-18.
- [109]. Hevner, A. R. (2007). A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2), 4.
- [110]. Abi-Antoun, Marwan, and Jonathan Aldrich. "Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations." *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 2009.
- [111]. Uzun, Burak, and Bedir Tekinerdogan. "Domain-driven analysis of architecture reconstruction methods." *Model Management and Analytics for Large Scale Systems*. Academic Press, 2020. 67-84.
- [112]. Terra, Ricardo, and Marco Tulio Valente. "A dependency constraint language to manage object-oriented software architectures." *Software: Practice and Experience* 39.12 (2009): 1073-1094.
- [113]. Java runtime metadata analysis, <https://github.com/ronmamo/reflections>, last accessed on May, 2020.
- [114]. Weinreich, R., & Buchgeher, G. (2014, April). Automatic reference architecture conformance checking for soa-based software systems. In *2014 IEEE/IFIP Conference on Software Architecture* (pp. 95-104). IEEE.
- [115]. Fittkau, F., Stelzer, P., & Hasselbring, W. (2014, August). Live visualization of large software landscapes for ensuring architecture conformance. In *Proceedings of the 2014 European Conference on Software Architecture Workshops* (pp. 1-4).
- [116]. Uzun, Burak, and Bedir Tekinerdogan. "Architecture conformance analysis using model-based testing: A case study approach." *Software: Practice and Experience* 49.3 (2019): 423-448.
- [117]. Rosik, Jacek, et al. "An industrial case study of architecture conformance." *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 2008.
- [118]. Miranda, Sergio, et al. "Architecture Conformance Checking in Dynamically Typed Languages." *J. Object Technol.* 15.3 (2016): 1-1.
- [119]. Deissenboeck, F., Heinemann, L., Hummel, B., & Juergens, E. (2010, May). Flexible architecture conformance assessment with ConQAT. In *2010 ACM/IEEE 32nd International Conference on Software Engineering* (Vol. 2, pp. 247-250). IEEE.
- [120]. Eichberg, M., Kloppenburg, S., Klose, K., & Mezini, M. (2008, May). Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering* (pp. 391-400).
- [121]. Muccini, Henry, Marcio Dias, and Debra J. Richardson. "Systematic Testing of Software Architectures in the C2 style." *International Conference on Fundamental Approaches to Software Engineering*. Springer, Berlin, Heidelberg, 2004.
- [122]. Bucchiarone, Antonio, et al. "Model-checking plus testing: from software architecture analysis to code testing." *International Conference on Formal Techniques for Networked and Distributed Systems*. Springer, Berlin, Heidelberg, 2004.

SUMMARY

Every software system has an architecture that defines the gross-level structure of the system and consists of architectural components, relations and their interactions. A common practice to model software architectures is based on the notion of so-called architecture viewpoints, which define the templates and guidelines for the architecture from the perspective of one or more stakeholder concerns. Based on architecture viewpoints, architecture views can be derived for a particular system. Usually, the architecture of a system consists of multiple architecture views. Since software systems constantly evolve, the software code can start to diverge from the documented software architecture. Architecture conformance analysis methods have been proposed that compare and check the differences between the code and the software architecture. Elements in the code might conform to the elements in the architecture, elements might be absent, or the code might diverge from the architecture and introduce new elements. Conformance analysis is performed on two levels which are code and architecture levels. Code-level conformance analysis employs intended architecture to generate runnable specifications that are executed on the code to reveal divergence or absence relations. Architecture level conformance analysis employs reconstructed architecture from implementation and compares intended architecture with reconstructed architecture. Manually checking for divergence and absence relations in large-scale software systems is time-consuming and cumbersome; hence, most conformance analysis methods in the industry are either semi-automated or fully automated. Software testing can be used to detect the divergence and absence relations. Automation in software testing can be achieved by model-based testing techniques. Model-based testing is a software testing technique in which certain aspects of software system behaviours are defined in a model that is a basis for abstract tests. Concrete tests are derived from these abstract tests and executed against the software system under test.

The main objective in this thesis is to define model-driven conformance analysis methodologies to detect the absence and divergence relations of software systems. To this end we answered the following research questions:

RQ1: What are current solutions for model-driven architecture-based testing?

RQ2: What are the limitations and benefits of code level model-based architecture conformance analysis using architecture viewpoints?

RQ3: What are the characteristic features of architecture reconstruction methods?

RQ4: What are the limitations and benefits of architecture level model-based architecture conformance analysis using architecture viewpoints?

To answer these research questions, we used five different research methodologies which are: Systematic Literature Review, Case Study Research, Survey Study, Domain Analysis, Design Science Research.

In chapter 2, we applied a systematic literature review for model-driven architecture based testing. We defined three research questions which are related to identifying addressed concerns, proposed solutions and existing research directions. We thoroughly analyzed 31 primary studies out of 739 studies. Firstly, we identified two concerns for applying MDABT which are: code to architecture conformance checking and internal architecture consistency checking. Furthermore, we derived a generic process model for model-driven architecture-

based testing. Lastly, we identified current obstacles and existing research directions based on selected primary studies. We identified seven obstacles for MDABT which are: the need for executable models for representing architecture, coping with the abstraction difference between architecture and code, the combinatorial state explosion problem, the lack of architecture views to consider multiple different concerns, the need for automating the process by tools, the cost-benefit analysis, and the need for applying MDABT on complex real systems to provide evidence of the approaches with respect to conventional approaches.

In chapter 3, we applied design science research to familiarize ourselves with the big data e-government system domain. We synthesized a generic process model for deriving big data e-government architectures. Our process model takes variability models for big data and e-government systems along with e-government reference business process model and reference architecture to create a concrete big data e-government system architecture.

In chapter 4, we presented our code level architecture conformance analysis method alongside case and survey studies. Our method takes architecture view models as input and translates them into unit tests that can be executed on the code to verify the absence relations. We evaluated our results on both original and mutant copy of big data e-government system. Moreover, we conducted surveys to developers both before and after the execution of our method. We concluded that our method was practical and useful for architecture conformance analysis.

In chapter 5, we presented our domain analysis on software architecture reconstruction methods. We derived a feature model that contains common and variant features for software architecture reconstruction methods. To this end, we selected and analysed a set of papers for software architecture reconstruction methods. We discovered a set of repeating common and variant features for software architecture reconstruction methods. During our research, we discovered the need for a well-defined SLR study and this can be conducted as future research.

In chapter 6, we presented our view-driven architecture reconstruction approach. We implemented our approach so that it can be executed while the software system is running. We extract view models for selected viewpoints from the source code. We defined selected viewpoints for this study along side criteria for reconstructing each architecture viewpoint.

In chapter 7, we presented our architecture level conformance analysis approach alongside a case study. We used our previously presented architecture reconstruction method to derive architecture view models from the source code. Our approach takes in reconstructed architecture view models as parameters for detecting absence and divergence relations between implemented and intended architecture. We evaluated our approach on an e-commerce software system. We concluded that our approach was useful for detecting absence and divergence relations for architecture conformance analysis.

Chapter 8, concludes this thesis by summarizing and discussing the contributions.

SAMENVATTING

Elk software systeem heeft een architectuur die de fundamentele structuur van het systeem definieert en bestaat uit architecturale componenten, relaties en hun interacties. Een gangbare praktijk voor het modelleren van software architecturen is gebaseerd op de notie van zogenaamde architecture viewpoints, die de sjablonen en richtlijnen voor de architectuur definiëren vanuit het perspectief van de verschillende belanghebbenden. Op basis van architecture viewpoints kunnen architecture views worden afgeleid voor een bepaald systeem. Gewoonlijk bestaat de architectuur van een systeem uit meerdere architecture views. Omdat softwaresystemen voortdurend evolueren, kan de code gaan afwijken van de gedocumenteerde software architectuur. Er zijn architectuur conformiteitsanalyse methoden voorgesteld die de verschillen tussen de code en de software architectuur vergelijken en controleren. Elementen in de code kunnen overeenkomen met de elementen in de architectuur, elementen kunnen ontbreken, of de code kan afwijken van de architectuur en nieuwe elementen introduceren. Conformiteitsanalyse wordt uitgevoerd op twee niveaus: code- en architectuurniveau. Conformiteitsanalyse op codeniveau maakt gebruik van de beoogde architectuur om uitvoerbare specificaties te genereren die op de code worden uitgevoerd om divergentie- of afwezigheidsrelaties aan het licht te brengen. Conformiteitsanalyse op architectuurniveau maakt gebruik van gereconstrueerde architectuur vanaf implementatie en vergelijkt beoogde architectuur met gereconstrueerde architectuur. Het handmatig controleren op divergentie- en afwezigheidsrelaties in grootschalige softwaresystemen is tijdrovend en omslachtig; vandaar dat de meeste conformiteitsanalysemethoden in de industrie ofwel semi-geautomatiseerd of volledig geautomatiseerd zijn. Met softwaretests kunnen de divergentie- en afwezigheidsrelaties worden opgespoord. Automatisering bij het testen van software kan worden bereikt door op modellen gebaseerde testtechnieken. Modelgebaseerd testen is een softwaretesttechniek waarbij bepaalde aspecten van het softwaresysteemgedrag worden gedefinieerd in een model dat de basis vormt voor abstracte tests. Concrete tests worden afgeleid uit deze abstracte tests en uitgevoerd tegen het te testen softwaresysteem.

Ons hoofddoel in dit proefschrift is het definiëren van modelgestuurde conformiteitsanalyse methoden om de afwezigheid en divergentierelaties van softwaresystemen te detecteren. Daartoe hebben we de volgende onderzoeksvragen beantwoord:

1. Wat zijn de huidige oplossingen voor modelgestuurde, op architectuur gebaseerde testen?
2. Wat zijn de beperkingen en voordelen van modelgebaseerde architectuur conformiteitsanalyse op codeniveau met behulp van architectuurstandpunten?
3. Wat zijn de karakteristieke kenmerken van architectuur reconstructiemethoden?
4. Wat zijn de beperkingen en voordelen van modelgebaseerde architectuur conformiteitsanalyse op architectuurniveau met behulp van architectuurstandpunten?

Om deze onderzoeksvragen te beantwoorden, hebben we vijf verschillende onderzoeksmethodologieën gebruikt, namelijk: systematische literatuurstudie, casestudyonderzoek, enquêteonderzoek, domeinanalyse, ontwerpwetenschappelijk onderzoek.

In hoofdstuk 2 hebben we een systematisch literatuuronderzoek toegepast voor modelgestuurde, op architectuur gebaseerde testen. We hebben drie onderzoeksvragen gedefinieerd die betrekking hebben op het identificeren van geadresseerde zorgen, voorgestelde oplossingen en bestaande onderzoeksrichtingen. We hebben 31 primaire onderzoeken grondig geanalyseerd uit 739 onderzoeken. Ten eerste hebben we twee problemen geïdentificeerd voor het toepassen van MDABT, namelijk: controle van de conformiteit van code naar architectuur en controle van de consistentie van de interne architectuur. Bovendien hebben we een generiek procesmodel afgeleid voor modelgestuurd, op architectuur gebaseerd testen. Ten slotte hebben we op basis van geselecteerde primaire onderzoeken actuele obstakels en bestaande onderzoeksrichtingen geïdentificeerd. We identificeerden zeven obstakels voor MDABT, te weten: de behoefte aan uitvoerbare modellen om architectuur weer te geven, het omgaan met het abstractieverschil tussen architectuur en code, het explosieprobleem in de combinatorische toestand, het gebrek aan architectuuropvattingen om rekening te houden met meerdere verschillende problemen, de noodzaak om de proces door middel van tools, de kosten-batenanalyse en de noodzaak om MDABT toe te passen op complexe reële systemen om de benaderingen met betrekking tot conventionele benaderingen te bewijzen.

In hoofdstuk 3 hebben we design science research toegepast om ons vertrouwd te maken met het domein van big data gebaseerde e-government systemen. We hebben een generiek procesmodel samengesteld voor het afleiden van big data-e-government architecturen. Ons procesmodel maakt gebruik van variabiliteitsmodellen voor big data en e-overheidssystemen, samen met het referentiebedrijfsprocesmodel voor e-overheid en referentiearchitectuur om een concrete big data e-government systeemarchitectuur te creëren.

In hoofdstuk 4 hebben we onze analysemethode voor conformiteitsanalyse op codeniveau gepresenteerd, naast casus- en enquêtestudies. Onze methode neemt architectuurview-modellen als invoer en vertaalt deze naar unit-tests die op de code kunnen worden uitgevoerd om de afwezigheidsrelaties te verifiëren. We hebben onze resultaten geëvalueerd op zowel de originele als de gemuteerde kopie van het e-overheidssysteem met big data. Bovendien hebben we zowel voor als na de uitvoering van onze methode enquêtes gehouden onder ontwikkelaars. We concludeerden dat onze methode praktisch en bruikbaar was voor analyse van architectuurconformiteit.

In hoofdstuk 5 hebben we onze domeinanalyse over reconstructiemethoden voor software-architectuur gepresenteerd. We hebben een kenmerkmodel afgeleid dat algemene en afwijkende kenmerken bevat voor reconstructiemethoden voor software-architectuur. Daartoe hebben we een reeks papers geselecteerd en geanalyseerd voor reconstructiemethoden voor software-architectuur. We ontdekten een reeks herhalende gemeenschappelijke en afwijkende functies voor reconstructiemethoden voor software-

architectuur. Tijdens ons onderzoek ontdekten we de behoefte aan een goed gedefinieerde SLR-studie en dit kan als toekomstig onderzoek worden uitgevoerd.

In hoofdstuk 6 hebben we onze viewpoint benadering van architectuurreconstructie gepresenteerd. We hebben onze aanpak geïmplementeerd zodat deze kan worden uitgevoerd terwijl het softwaresysteem draait. We extraheren weergavemodellen voor geselecteerde gezichtspunten uit de broncode. We definieerden geselecteerde gezichtspunten voor deze studie naast aanvullende criteria voor het reconstrueren van elk architectuurgezichtspunt.

In hoofdstuk 7 hebben we onze benadering van conformiteitsanalyse op architectuurniveau gepresenteerd naast een casestudy. We hebben onze eerder gepresenteerde architectuurreconstructiemethode gebruikt om architectuurweergavemodellen af te leiden uit de broncode. Onze aanpak neemt gereconstrueerde architectuurview-modellen als parameters voor het detecteren van afwezigheids- en divergentierelaties tussen geïmplementeerde en beoogde architectuur. We hebben onze aanpak geëvalueerd op een e-commerce softwaresysteem. We concludeerden dat onze aanpak nuttig was voor het detecteren van afwezigheids- en divergentierelaties voor analyse van architectuurconformiteit.

Hoofdstuk 8 sluit dit proefschrift af met een samenvatting en bespreking van de bijdragen.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere thanks and gratitude to my supervisor Professor Bedir Tekinerdogan, who guided me throughout my MSc and PhD degrees since 2012, and helped me to reach my academic and business endeavours, with endless patience and support.

I would like to thank Wageningen University for giving me the chance for PhD program, and helpful members of information technology group and the secretaries who made this journey possible for me.

Finally, I would like to thank my wife and family for supporting and encouraging me during the compilation of this dissertation.

ABOUT THE AUTHOR

Burak Uzun was born on the 21st of September 1991 in Corlu, Turkey.

He is graduated from Bilkent University where he received his MSc degree (2015) in Computer Engineering. He worked for TUBITAK Software Technologies Research Institute as senior software engineer for six years, 2012-2018. As full stack engineer, he mainly focused on designing and implementing e-government software systems. Hereafter, he directed his career path towards finance and e-commerce industry. He worked for Klarna Bank for two years between 2018-2021 as senior software engineer on e-commerce banking systems. He is currently working for NASDAQ since April 2021 as senior software engineer, where he uses his hands-on engineering expertise on daily stock market products.

His main research topics include distributed systems, software architecture design, system design, model-based testing, and model-driven development.

In 2015 Burak Uzun started his PhD at Wageningen University. In his PhD research he studied Software Architecture Conformance Analysis and Model-Based Testing. The graduation ceremony of his PhD was in June 2021.

His research focuses on software architecture, conformance analysis and model-based testing.

His purpose is to integrate feasible software architecture conformance analysis solutions to continuous integration and continuous delivery pipelines.