

ULRR

Consistency checking of runtime feature dependencies in software product lines

Item Type	Thesis
Authors	Abid, Saad bin
Download date	2026-05-16 07:57:29
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/4012



University of Limerick
OLLSCOIL LUIMNIGH

**Consistency Checking of Runtime Feature Dependencies in
Software Product Lines**

**Submitted by Saad Bin Abid
For the award of Doctor of Philosophy**

**Supervised by
Dr. Norah Power
Dr. Goetz Botterweck
Submitted to the University of Limerick
June 2014**

To My Family

Abstract

Feature dependency is a kind of feature interaction which can be observed in the features that realize the functionality of a software product, and particularly in the features of Software Product Lines (SPLs). In feature modelling for SPLs, common and variable features are organized both in terms of their structural dependencies (e.g., aggregation and specialization) and in terms of their configuration dependencies (e.g., excluded and required). These dependencies are essential but are not sufficient for developing reusable and adaptable SPL components. There are various other types of feature dependencies (e.g., runtime feature dependencies) that are responsible for implementing the behaviour of an end product. The aim of this research is to facilitate a software product line engineer to detect whether the latter types of feature dependencies have been implemented as specified and as intended.

This thesis proposes and evaluates a four-step technique for consistency checking of runtime feature dependencies in SPL artefacts. The four-step process uses a domain-specific language developed for specifying the product line assets (i.e., features, interdependencies among the features and the implementation source code). The proposed technique uses existing reverse engineering techniques to perform a code-to-model transformation. Aspect-oriented (AO) pattern detection algorithms were developed that implement the runtime feature dependencies. The simple detection of AO-patterns does not guarantee that the behaviour is implemented as intended. Model validation constraints are then applied to provide feature dependency related feedback to a product line engineer.

The key contributions of this thesis are: (i) the extension of previous work on the modularization of runtime feature dependencies, by providing a technique for consistency checking of runtime feature dependencies at various abstraction levels, (ii) the theoretical contribution in the form of a model-driven technique based on round-trip engineering (RTE), (iii) a plug-in based prototype realizing the proposed model-driven technique and (iv) a validation mechanism inspired by a technique called mutation testing which is applied using an existing product line case study.

Declaration

The work presented in this thesis is entirely my own work. It has not been submitted previously to this or any other institute for this or any other academic award. Where use has been made of the work of other people, it has been acknowledged and referenced.

Signed: _____

Date: _____

Saad Bin Abid

Acknowledgements

First of all I would like to thank my lord ALLAH (SWT) for enabling me to accomplish my PhD research in the exciting field of software engineering. Without HIS (SWT) will it wouldn't have been possible.

I would like to thank my supervisors Dr. Norah Power and Dr. Goetz Botterweck for their kind support throughout the PhD process. I would especially like to thank Dr. Norah Power for her efforts towards the completion of my thesis. She always motivated and inspired me for my research and provided me an effective feedback on my research topic. I cannot thank her enough just by writing a few words in this acknowledgments section. I would also like to mention Dr. Goetz Botterweck's technical support that he provided me during my research. I would also like to give special thanks to Dr. Brian Lee for his moral and financial support.

I would like to dedicate my work to my parents (Naseem Abid and Chaudhry Abid). Without their prayers, love and compassion I wouldn't have been able to motivate morally and complete this work. Mother I know you suffered a lot because of my studies as I wasn't able to meet you for years due to my PhD studies. I would also like to dedicate my work to my late grandmother who passed away while I was completing my PhD and I couldn't see her before her last journey towards the heavens.

I would also like to thank a stranger who helped me out financially while I was out of funding and his help supported me during my PhD studies.

I would also like to thank the following people for their help and support in providing valuable feedback on my research work for evaluation purpose

- Dr. Dimitris Kolovos, Project Leader (Epsilon), University of York, England
- Dr. Deepak Dhungana, Senior Researcher, Siemens Research, Austria
- Dr. Rick Rabiser, Senior Researcher, Johannes Kepler Universität (JKU), Linz, Austria
- Dr. Yuansong Qiao, Principal Researcher, Software Research Institute (SRI), Athlone IT, Ireland

- Dr. Jad El-Khoury, Senior Researcher in Machine Design Lab, Royal Institute of Technology (Kth), Sweden
- Dr. Abdullah Sandhu, Senior Software Architect/Designer, Ericsson, Athlone, Ireland
- Shuaijun Zhang, Senior Software Engineer, Software Research Institute (SRI), Athlone IT, Ireland
- Martin O'Hara, Senior Software Engineer and Researcher, Software Research Institute (SRI), Athlone IT, Ireland

Last but not the least I would like to thank my family here in Ireland specially my wife who helped me during this daunting PhD process. She always held her head high, made me feel comfortable and gave me moral motivation during my PhD studies. My daughter who always became my strength and her smiling face always gave me strength to work hard and complete my PhD thesis.

Table of Contents

DECLARATION	III
ACKNOWLEDGEMENTS	IV
LIST OF FIGURES	X
LIST OF TABLES	XII
TABLE OF ABBREVIATIONS	XIII
CHAPTER ONE: INTRODUCTION AND MOTIVATION	1
1.1 OVERVIEW	1
1.2 INTRODUCTION	1
1.3 FEATURE MODELLING.....	3
1.4 RUNTIME FEATURE DEPENDENCIES	4
1.5 CONTEXT	5
1.6 MOTIVATION	7
1.7 RESEARCH CONTRIBUTIONS	12
1.8 OUTSIDE THE SCOPE OF THE WORK.....	12
1.9 STRUCTURAL OVERVIEW OF THE DISSERTATION	13
CHAPTER TWO: LITERATURE REVIEW	15
2.1 OBJECTIVE.....	15
2.2 RESEARCH QUESTION FOR LITERATURE REVIEW	15
2.3 IDENTIFIED APPROACHES.....	15
2.3.1 (<i>Ferber et al., 2002</i>)	15
2.3.2 (<i>Lee and Kang, 2004</i>)	16
2.3.3 (<i>Ye and Liu, 2005</i>)	17
2.3.4 (<i>Zhang et al., 2006</i>).....	18
2.3.5 (<i>Lee et al., 2006</i>)	20
2.3.6 (<i>Savolainen et al., 2007</i>)	21
2.3.7 (<i>Silveira et al., 2007</i>)	23
2.3.8 (<i>Kim et al., 2008</i>).....	24
2.3.9 (<i>Parra et al., 2010</i>).....	26
2.3.10 (<i>Apel et al., 2011</i>).....	28
2.3.11 (<i>Batory et al., 2011</i>)	28
2.3.12 (<i>Mosser et al., 2012</i>).....	29
2.3.13 <i>Feature Dependency Analysis in Other Software Engineering Domains</i>	29
2.4 DISCUSSION OF LITERATURE COMPARISON CRITERIA.....	31
2.5 COMPARISON OF THE IDENTIFIED LITERATURE	32
2.5.1 <i>Extending Feature Models</i>	32
2.5.2 <i>Well Defined Analysis Process for Feature Dependencies</i>	33
2.5.3 <i>Analysis of Feature Dependencies in Product Line Assets</i>	34
2.5.4 <i>Tool Support</i>	35

2.5.5 <i>Specific Product Line Stakeholders</i>	36
2.5.6 <i>Runtime Feature Dependencies</i>	36
2.5.7 <i>Evaluation Criteria</i>	37
2.6 IDENTIFIED RESEARCH GAPS.....	37
2.6.1 <i>Lack of focus on runtime feature dependencies analysis</i>	37
2.6.2 <i>Lack of guidelines to analyse modularize runtime feature interactions</i>	38
2.6.3 <i>Lack of Focus on Analysis of Model to Code Feature Dependencies</i>	38
2.6.4 <i>Lack of Focus on Product Line Stakeholders</i>	39
2.6.5 <i>Lack of Tool Supported Approaches</i>	39
2.7 SUMMARY.....	39
CHAPTER THREE: CONSISTENCY CHECKING OF RUNTIME FEATURE DEPENDENCIES IN	
PRODUCT LINE ASSETS	41
3.1 OBJECTIVE.....	41
3.2 PROPOSED TECHNIQUE ASSUMPTIONS AND INVOLVED STAKEHOLDERS.....	41
3.2.1 <i>Assumptions</i>	41
3.2.2 <i>Involved Stakeholders</i>	42
3.3 PROBLEM OVERVIEW OF RUNTIME FEATURE DEPENDENCIES IN A PRODUCT LINE CONTEXT.....	42
3.4 ROUND-TRIP ENGINEERING (RTE)	43
3.5 INCONSISTENCY SCENARIOS RELATED TO RUNTIME FEATURE DEPENDENCIES IMPLEMENTATIONS	44
3.5.1 <i>No Runtime Feature Dependencies Implementation</i>	45
3.5.2 <i>Incompatibility between the Specified Type and its Respective Implementation</i>	45
3.5.3 <i>Incompatible Specified Features Pair and its Respective Implementation Pair</i>	46
3.5.4 <i>Incomplete Implementation of the Runtime Feature Dependencies</i>	46
3.5.5 <i>Implementation of Runtime Feature Dependency Not Preserving the Specified</i>	
<i>Direction</i>	46
3.5.6 <i>Semantic Inconsistency Related to Runtime Feature Dependencies</i>	47
3.5.7 <i>An overview of Inconsistency Scenarios and Classification</i>	48
3.6 DOMAIN MODELLING OF THE PRODUCT LINE ASSETS.....	49
3.6.1 <i>Dependency-Oriented Feature Model (DOFM)</i>	49
3.6.2 <i>Implementation Model (IM)</i>	51
3.6.3 <i>Pattern Model (PM)</i>	52
3.7 PROCESS MODEL OF THE PROPOSED TECHNIQUE	59
3.7.1 <i>Extracting Source Code Concepts/Code to Model Transformation</i>	60
3.7.2 <i>Mapping Dependency-Oriented Feature Model to Implementation Model</i>	60
3.7.3 <i>Aspect-Oriented (AO) Pattern-based Implementation Detection</i>	61
3.7.4 <i>Applying Constraints for Consistency Checking in Product Line Assets</i>	62
3.8 PROPOSED TECHNIQUE PREREQUISITES.....	63
3.9 LIMITATIONS.....	64
3.10 DISCUSSION.....	65
3.11 SUMMARY.....	67
CHAPTER FOUR: IMPLEMENTATION CONCEPTS AND PROTOTYPE SUPPORT	69
4.1 OVERVIEW	69
4.2 CONSISTENCY CHECKING TECHNIQUE - A CONCEPTUAL REALIZATION	69
4.2.1 <i>Code to Model Transformation and AO-Pattern Detection Plug-in</i>	69

4.2.2 Mapping the DOFM and the IM.....	70
4.2.3 Applying the EVL Constraints for the Consistency Checking	72
4.3 ASPECT-ORIENTED PATTERN DETECTION PROCESS – AN IMPLEMENTATION OVERVIEW	73
4.4 ASPECT-ORIENTED PATTERN DETECTION ALGORITHMS (PSEUDO-CODE)	75
4.4.1 Runtime Modification Dependency Pattern Detection Algorithm	76
4.4.2 Runtime Required Activation Pattern Detection Algorithm	76
4.4.3 Runtime Excluded Activation Pattern Detection Algorithm	78
4.4.4 Runtime Sequential Activation Pattern Detection Algorithm	80
4.4.5 Runtime Concurrent Activation Detection Algorithm	82
4.5 TOOL SUPPORT	84
4.5.1 Code to Model Transformation and AO-Pattern Detection Plug-in	84
4.5.2 Mapping Specified Concepts in DOFM to Respective Implementation in IM	86
4.5.3 Applying EVL Constraints	87
4.6 DISCUSSION.....	89
4.7 SUMMARY.....	90
CHAPTER FIVE: VALIDATION AND EVALUATION	93
5.1 OVERVIEW	93
5.2 VALIDATION.....	93
5.2.1 Validation Strategy and Hypothesis.....	93
5.2.2 Mutation Testing-Inspired Approach for Validation	94
5.2.3 Experimentation- Scientific Calculator Product Line Case Study.....	95
5.2.4 Validation Example from Scientific Calculator Implementation of Runtime Required Activation Dependency (RTRAD)-An Example Case	96
5.3 EVALUATION BASED ON FORMAL INTERVIEWS	106
5.3.1 Candidate Selection	106
5.3.2 Evaluation Package Formulation	106
5.3.3 Questionnaire Formulation.....	107
5.3.4 Formal Evaluation Process.....	107
5.3.5 Key Findings of Evaluation Results Discussion	108
5.4 DISCUSSION OF VALIDATION RESULTS AND LESSONS LEARNED.....	109
5.5 COMPARATIVE EVALUATION OF THE PROPOSED APPROACH WITH STATE OF THE ART	112
5.6 QUALITATIVE ANALYSIS OF THE PROPOSED SOLUTION	114
5.6.1 Extensibility.....	114
5.6.2 Scalability.....	115
5.6.3 Usability	116
5.6.4 Variability.....	117
5.7 THREATS TO VALIDITY	117
5.8 SUMMARY.....	118
CHAPTER SIX: CONCLUSIONS AND FUTURE DIRECTIONS	119
6.1 OVERVIEW	119
6.2 SUMMARY OF THE DISSERTATION.....	119
6.3 CONTRIBUTIONS.....	122
6.4 LIMITATIONS.....	123
6.5 FUTURE WORK.....	124

6.5.1 Further Case studies.....	124
6.5.2 Improving Pattern Detection Algorithms.....	124
6.5.3 Dynamic Analysis of RTFDs.....	124
6.5.4 Scalability of the Approach.....	124
6.5.5 Providing Graphical User Interfaces.....	125
6.5.6 Model to Code Inconsistency Traceability and Vice Versa.....	125
6.5.7 Comparison with new Approaches.....	125
6.5.8 Traceability and Reverse Engineering.....	125
6.5.9 Extending the Domain.....	125
REFERENCES.....	127
APPENDIX A: ABSTRACT CONCEPTS OF EPSILON VALIDATION LANGUAGE.....	133
APPENDIX B: MUTATION TESTING-INSPIRED VALIDATION.....	137
APPENDIX C: PATTERN DETECTION ALGORITHMS IMPLEMENTATION FOR PARSING OF ASPECTJ IMPLEMENTATION USING JAVA (SOURCE CODE).....	163
APPENDIX D: QUESTIONNAIRE FOR THE FORMAL INTERVIEWS.....	187
APPENDIX E: EVL CONSTRAINTS.....	189
APPENDIX F: LIST OF PUBLICATIONS.....	193
APPENDIX G: INTERVIEW TRANSCRIPTS.....	195

List of Figures

<i>Figure 1-1 Domain and Application Engineering Processes (Pohl et al., 2005)</i>	2
<i>Figure 1-2 Feature Model of a Scientific Calculator Product Line (An Excerpt) (Lee et al., 2009a)</i>	4
<i>Figure 1-3 Runtime Feature Dependency Classification Diagram (Lee and Kang, 2004)</i>	5
<i>Figure 1-4 Scientific Calculator RTFD Motivational Example</i>	9
<i>Figure 1-5 Motivational Scenario</i>	11
<i>Figure 1-6 Thesis Chapters Overview and Research Approach Adopted</i>	14
<i>Figure 2-1 Feature Interaction and Dependency Representation Model (Ferber et al., 2002)</i> ...	16
<i>Figure 2-2 Part of Activation Dependencies in ECB Product Line (Lee and Kang, 2004)</i>	17
<i>Figure 2-3 Feature Relationships and Notations (Ye and Liu, 2005)</i>	18
<i>Figure 2-4 Detecting Feature Dependencies Anomalies (Zhang et al., 2006)</i>	19
<i>Figure 2-5 Tables for Eigenvalues of Dependencies (Upper) and Feature Dependencies (Lower)</i> <i>(Lee et al., 2006)</i>	21
<i>Figure 2-6 A Dependency Model of Publish/Subscribe Main variation points and concerns</i> <i>(Silveira et al., 2007)</i>	23
<i>Figure 2-7 Summary of contextual annotations used in YANCEES (Silveira et al., 2007)</i>	24
<i>Figure 2-8 Feature Dependency Graph Construction (Kim et al., 2008)</i>	25
<i>Figure 2-9 Different Incremental wise development of the Buffer Program in CIDE (Kim et al.,</i> <i>2008)</i>	26
<i>Figure 2-10 Feature Aspect meta-model DSL (Parra et al., 2010)</i>	27
<i>Figure 2-11 The Approach (Parra et al., 2010)</i>	27
<i>Figure 3-1 An Example to Demonstrate Inconsistency Scenarios in the Implementation of RTFDs</i>	44
<i>Figure 3-2 No Implementation of Specified Runtime Feature Dependency</i>	45
<i>Figure 3-3 Specified Runtime Feature Dependency Order not Preserved in Respective</i> <i>Implementation</i>	47
<i>Figure 3-4 Dependency-Oriented Feature Model (DOFM)</i>	51
<i>Figure 3-5 Implementation Model (IM) Meta-Model</i>	52
<i>Figure 3-6 Runtime Modification Pattern Meta-Model</i>	54
<i>Figure 3-7 Runtime Required Activation Pattern Meta-Model</i>	55
<i>Figure 3-8 Runtime Excluded Activation Pattern Meta-Model</i>	56
<i>Figure 3-9 Runtime Concurrent Activation Pattern Meta-Model</i>	57
<i>Figure 3-10 Runtime Sequential Activation Pattern Meta-Model</i>	59
<i>Figure 3-11 Consistency Checking Process Model</i>	60
<i>Figure 3-12 Consistency Checking Process Model</i>	66
<i>Figure 4-1 Code to IM Transformation and AO Pattern Detection</i>	70
<i>Figure 4-2 Specification of RTFDs and Integration of the DOFM and the IM</i>	71
<i>Figure 4-3 Applying the Developed EVL Constraints</i>	73
<i>Figure 4-4 Code to IM Generation and AO Pattern Detection Process</i>	74
<i>Figure 4-5 Class Diagram of the Code2Model Plug-in</i>	75
<i>Figure 4-6 Code2IM and AOP-Pattern Detection Transformation Plug-in</i>	85
<i>Figure 4-7 Generated IM and PM Meta-Model Instances</i>	86
<i>Figure 4-8 Manually Mapping the DOFM and the IM</i>	87

<i>Figure 4-9 Applying Developed EVL Constraints</i>	88
<i>Figure 4-10 Example of the Generated Error Markers</i>	88
<i>Figure 5-1 Scientific Calculator Product Line Feature Model (an excerpt) (Lee et al., 2009a)</i>	96
<i>Figure 5-2 Specification of Runtime Required Activation between features “And” and “HexPanel”</i>	97
<i>Figure 5-3 XMI-based representation of Feature Model (DOFM) with Runtime Required Activation between features “And” and “HexPanel”</i>	98
<i>Figure 5-4 Actual Implementation of Runtime Required Activation Dependency between “HexPanel” and “And” Implementations</i>	99
<i>Figure 5-5 Detected Runtime Required Activation Pattern Model Instance</i>	99
<i>Figure 5-6 Extendibility of the Domain</i>	115
<i>Figure A-1 Abstract Syntax of EVL (Kolovos et al., 2012b)</i>	133
<i>Figure A-2 Concrete Syntax of an EVL Context (Kolovos et al., 2012b)</i>	134
<i>Figure A-3 Concrete Syntax of an EVL Invariant (Kolovos et al., 2012b)</i>	134
<i>Figure A-4 Concrete Syntax of an EVL Fix (Kolovos et al., 2012b)</i>	135
<i>Figure B-1 Identified Runtime Ex-Act Dependency between “DelButton” and “ModeFM” Features</i>	137
<i>Figure B-2 XMI-Based Feature Model Instance for Specifying “DelButton” and “ModeFM” Features and their Excluded Activation Dependency</i>	138
<i>Figure B-3 Actual Runtime Excluded Activation Pattern-Based Implementation between “DelButton.java” and “ModeFM.java”</i>	139
<i>Figure B-4 Identified Runtime Sequential Activation Dependency between “EqualsButton” and “BaseFM” Features</i>	144
<i>Figure B-5 XMI-Based Feature Model Instance for Specifying “EqualsButton” and “BaseFM” Features and their Sequential Activation Dependency</i>	145
<i>Figure B-6 Actual Runtime Sequential Activation Pattern-Based Implementation between “EqualsButton.java” and “BaseFM.java”</i>	146
<i>Figure B-7 Identified Runtime Concurrent Activation Dependency between “M5” and “M1” Features</i>	151
<i>Figure B-8 XMI-Based Feature Model Instance for Specifying “M5” and “M1” Features and their Concurrent Activation Dependency</i>	152
<i>Figure B-9 Actual Runtime Concurrent Activation Pattern-Based Implementation between “M5.java” and “M1.java”</i>	152
<i>Figure B-10 Identified Runtime Modification Dependency between “OffButton” and “ShiftFM” Features</i>	157
<i>Figure B-11 XMI-Based Feature Model Instance for Specifying “ShiftKey” and “OffButton” Features and their Runtime Modification Dependency</i>	158
<i>Figure B-12 Actual Runtime Modification Pattern-Based Implementation between “OffButton.java” and “ShiftFM.java”</i>	158

List of Tables

<i>Table 2-1 Table of Approaches and Respective Evaluation Examples</i>	37
<i>Table 5-1 Validation of Implementation Mutants for RTRAD Relationship Example</i>	104
<i>Table 5-2 Specification Mutants Validation for RTRAD Relationship Example</i>	105
<i>Table 5-3 Validation Results for “Scientific Calculator” Case Study</i>	112
<i>Table 5-4 Existing SPLE Approaches/Tools Supporting Identification and Resolution of Runtime Feature Dependencies Related Inconsistencies (-- Not Supported, - Some Supported, + Supported, ++ Very Well Supported)</i>	113
<i>Table B-1 Validation of the Example Runtime Excluded Activation Example between DelButton and ModeFM</i>	143
<i>Table B-2 Validation of the Example Runtime Sequential Activation Example between EqualsButton and BaseFM</i>	150
<i>Table B-3 Validation of the Example Runtime Concurrent Activation Dependency between “M5” and “M1” Features</i>	156
<i>Table B-4 Validation of the Example Runtime Modification Dependency between "Shiftkey" and "OffButton"</i>	161

Table of Abbreviations

SPLs	Software Product Lines
MDD	Model-Driven Development
DE	Domain Engineering
AE	Application Engineering
FODA	Feature-Oriented Domain Analysis
RTFDs	Runtime Feature Dependencies
O&AFDs	Operational and Activation Feature Dependencies
DOFM	Dependency-Oriented Feature Model
PM	Pattern Model
IM	Implementation Model
DSL	Domain-Specific Language
AO-Patterns	Aspect-Oriented Patterns
AST	Abstract Syntax Tree
AOP	Aspect-Oriented Programming
SDK	Software Development Kit
PDE	Plug-in Development Environment
JDT	Java Development Tooling
IDE	Integrated Development Environment
EVL	Epsilon Validation Language
OCL	Object-Constraint Language
MT	Mutation Testing
EOL	Epsilon Object Language
RTE	Round-trip Engineering
ATL	Atlas Transformation Language
DFS	Depth First Search

Chapter One: Introduction and Motivation

1.1 Overview

This chapter introduces the thesis domain, provides the motivation and sketches the research question under investigation. Subsequently, the chapter summarizes the main contributions of the thesis and provides an overview of the remaining chapters.

1.2 Introduction

The reuse of software has been a driving force of software engineering methodologies for a long time. “*Systematic software reuse is a promising approach to reduce cost and development cycle time, improve software quality and productivity*” (Clements and Northrop, 2002). In this context, the notion of a software product line (SPL) has gained importance for large-scale systematic software reuse. A SPL is a set of software-intensive systems sharing a common managed set of features that specify the needs of a particular market segment and that are developed from a common set of core assets (Clements and Northrop, 2002).

In a feature-oriented software product line, functionalities are observed and built as a set of common and variable features (Kang et al., 2002). A feature is defined as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" (Kang et al., 1990). Product line features realizing the functionality of an established product line often depend on one another. This phenomenon is called feature dependency, and it can be observed at different levels of abstraction in a software product line (Silveira et al., 2007, Lee et al., 2006).

Product line features are realized as a set of architectural components in the product line architecture (Savolainen et al., 2007). As mentioned earlier, features depend on each other to provide the functionality of a product leading towards component dependencies in product line architecture. In general, software engineering, a *software component* can take the shape of a software package, a web service, a web resource, a software module or even a class object or collection of objects. In the component and connector (C&C) view of Architecture Description Language (i.e., XADL 2.0) (Dashofy et al., 2002), a feature can be implemented as a component or a set of components whereas feature dependency can be implemented as connectors and interfaces among architectural components.

SPL engineering (SPLE) consists of two interrelated processes, namely domain engineering (DE) and application engineering (AE) (Figure 1-1).

“Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. Reusable work products), as well as providing an adequate means for reusing these assets (i.e. Retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems.” (Czarnecki, 1998)

During application engineering (AE), concrete products are derived from the product line assets by selecting, customizing, integrating and deploying the assets that have been developed during the domain engineering process. The process of deriving concrete products from the product line is often referred as product derivation and is one of the activities in AE process.

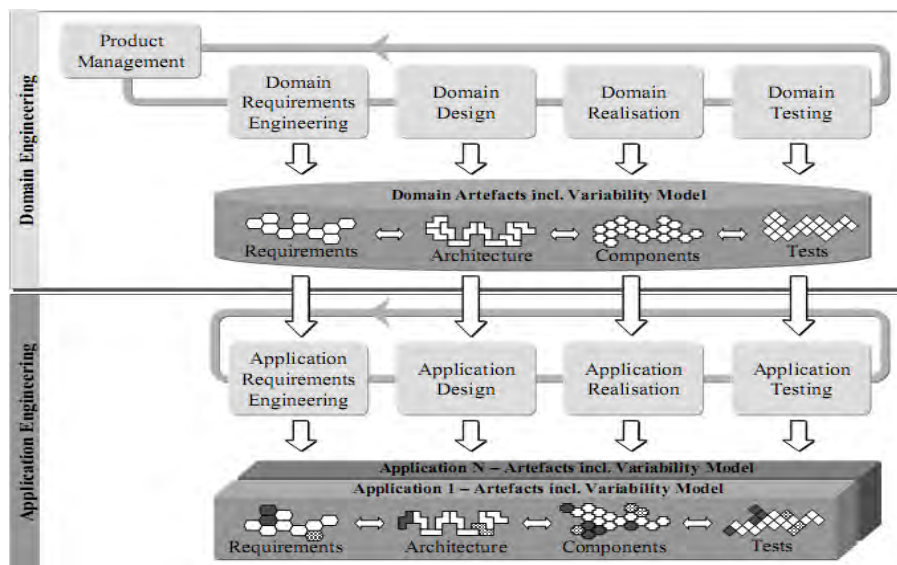


Figure 1-1 Domain and Application Engineering Processes (Pohl et al., 2005)

Although understanding feature dependencies among various features is a critical task in product line asset development, such dependencies have not been analysed and documented explicitly (Silveira et al., 2007). A feature dependency, if not analysed properly, may affect a large part of a product line assets’ maintenance and deployment. Inability to manage the feature dependencies between different levels of abstraction, (i.e., between feature model and implementation levels) can lead to erroneous product configuration and derivation.

1.3 Feature Modelling

Software Product Line can be seen as a managed set of features. In SPL engineering feature modelling has emerged as one of the most popular domain analysis technique (Kang et al., 2002). Feature modelling allows domain experts to analyse commonality and variability in a particular domain in order to develop highly reusable core assets of a SPL. Feature modelling performs domain analysis by taking into account the domain knowledge. The outcome of a feature modelling process is a feature model. A feature model represents a compact representation of all the possible products of the SPL. Feature models were first introduced in the feature-oriented domain analysis (FODA) method (Kang et al., 1990). Feature models are visually represented by feature diagrams. The purpose of a feature model defines features and inter-dependencies among them, typically in a form of a feature diagram. Feature diagram enables one to extract the structure, visualize the commonality and the variability of a particular domain or a set of products. Common features are also referred to as mandatory features whereas variable features represent a choice that can be present in one product and while absent in other.

A relatively new approach called orthogonal variability modelling (OVM) or simply variability modelling is utilized to perform modelling of the variability in design and realization documents (Bencomo, 2008).

Unlike feature model OVM is utilized to model explicitly define and manage variability of a product line without taking into account the common features. An example of a scientific calculator (SciCalc) feature model (Botterweck and Lee, 2009) is shown in Figure 1-2 with a set of common and variable features along with various types of dependencies (e.g., Excluded-ActivationDep, Required-ActivationDep) represented between them. The types of dependencies represented in the SciCalc feature model are defined in Section 1.4. SciCalc product line is composed of various features (e.g., Off, History, Buttons). Hollow circles show the optional features whereas filled circles represent a mandatory feature that should be configured for all the variants of SciCalc product line.

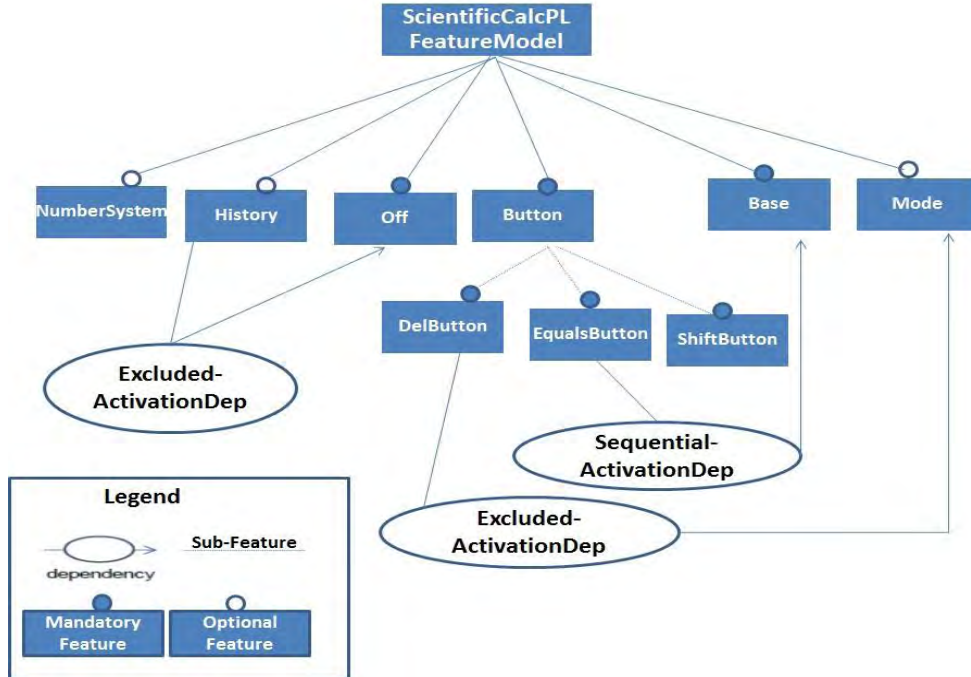


Figure 1-2 Feature Model of a Scientific Calculator Product Line (An Excerpt) (Lee et al., 2009a)

1.4 Runtime Feature Dependencies

Work by Lee and Kang (2004) classified operational and activation dependencies. Operational dependencies are defined as “*directly or indirectly create relationships between features during the operation of the system in such a way that the operation of one feature is dependent on one of those of other features*”. Figure 1-3 represents a classification diagram as a UML class diagram.

Types of operational dependencies are as follows,

- 1) *Usage dependency*: “A usage dependency between two features means one feature (a usable client) depends on another feature (a usage supplier) for its correct functioning”,
- 2) *Modification dependency*: “A feature (a modifier) may modify the behavior of other feature (a modifyee) during its activation”, and
- 3) *Activation dependency*: “if an activation of one feature depends on that of other feature”.

Activation dependencies are type of operational dependencies. Work by Lee et al. (2009a) further sub-classified activation dependencies into four categories namely,

3.1) Exclusive-activation dependencies: features should be mutually exclusive with each other during activation,

3.2) Subordinate activation dependency: during activation of features there may exist a feature (a subordinate) which can only be active while other feature (a superior) is active,

3.3) Concurrent activation dependency: there may exist a scenario during activation where some subordinates of a superior are active at the same time while the superior is active and

3.4) Sequential activation dependency: where subordinates of a superior may have to be active sequentially while the superior is active.

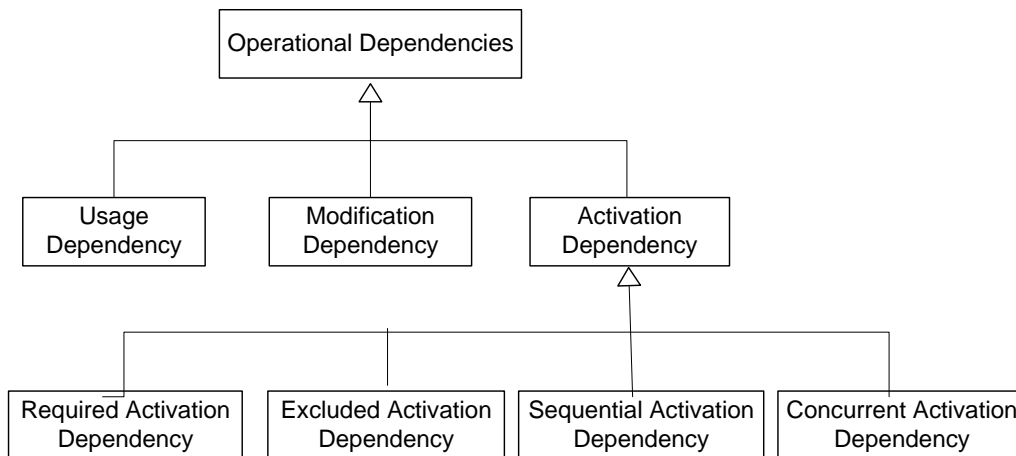


Figure 1-3 Runtime Feature Dependency Classification Diagram (Lee and Kang, 2004)

1.5 Context

In literature, the feature modelling technique has been used to analyse feature dependencies among the features within the context of a product line by different authors (Kang et al., 1990, Ferber et al., 2002, Ye and Liu, 2005, Lee et al., 2006). They extended the feature model with various types of feature dependencies, for instance, configuration dependencies (e.g., requires,

excludes), structural dependencies (e.g., aggregation, generalization) and runtime dependencies (e.g., operational and activation dependencies).

Structural and configuration dependencies are of an important nature but not sufficient for developing the reusable and adaptable product line components (Lee and Kang, 2004). *Structural dependencies* are kind of dependencies where common and variable features of a product line are organized as specialization and aggregation. *Configuration dependencies* are constraints (i.e., requires and excludes) between common and variable features of a product line, that represent the configuration options when a particular product variant is configured. However, there exists other types of feature dependencies (i.e., runtime activation and operational dependencies) that are responsible for implementing the product behavior (Lee and Kang, 2004).

Feature dependency have to implemented as a separate reusable asset (Lee and Kang, 2004, Lee et al., 2009b, Botterweck and Lee, 2009, Cho et al., 2008) in order to modularize them. Implementations of feature dependencies are hard to identify, because they are embedded within the product line implementations. Unable to understand and manage the feature dependencies embedded in the source code might lead towards tangled code issue and error-prone product derivation during AE process.

Various authors have addressed the issue of managing feature dependencies by applying different methodologies in literature (Luo and Diao, 2009, Ferber et al., 2002, Cho et al., 2008, Lee and Kang, 2004, Satyananda et al., 2007, Silveira et al., 2007, Jayaraman et al., 2007, Lienhard et al., 2007, Zhang et al., 2005, Lee et al., 2009b, Botterweck and Lee, 2009). Feature dependencies and interactions need to be implemented and managed as explicit components in the source code (Botterweck and Lee, 2009, Lee et al., 2009a, Lee et al., 2009b, Lee and Kang, 2004). Inability to manage feature dependencies at different levels of abstraction (i.e., between feature model and implementation levels) can lead to erroneous product configuration and derivation.

Crosscutting features are those features that have functionalities scattered across in implementations of various features. Similarly the dependencies exist between features interacting with each other are embedded within their respective modular implementations. Work done by Lee et al., (Botterweck and Lee, 2009, Lee et al., 2009a, Lee et al., 2009c, Lee and Kang, 2004, Lee et al., 2006) provided modular implementations by using a particular type of design pattern called aspect-oriented design patterns using the AspectJ technology (Colyer et al., 2004) to design, develop and reuse the RTFDs (Section 1.4) in

product line implementations. An aspect is a separate modular unit encapsulating any crosscutting concern, which would otherwise be scattered across multiple components. AspectJ technology provides support for encapsulation of crosscutting features into new modular units called the aspects.

However, the proposed AO-design patterns only enable the product line engineer to implement the RTFDs as a separate component in the source code.

An example problematic scenario related to RTFDs can arise when two features want to access a shared resource sequentially but during the product execution, they fail to access the resource in a sequential manner. The challenge for a product line engineer in this situation is to check the consistency between modular implementation of sequentially accessing the shared resource in the source code versus its specification in order to develop a functionally correct product.

Following research gaps were identified in the work by Lee et al. (Lee and Kang, 2004, Lee et al., 2009b, Botterweck and Lee, 2009, Cho et al., 2008),

- Checking consistency mechanism between the prescribed AO-design patterns and their respective specifications was missing
- No specification was provided for the prescribed AO-design patterns

This research work extends the work started by Lee et al., by introducing the specification language (i.e., DOFM) and implementation model layers on top of the source code containing the AO pattern-based implementations.

This work represents a verification challenge of

- RTFDs implementations with respect to their specifications in Domain Engineering (DE) of a product line
- Where the runtime behaviour of features depending on each other in a product line is designed and implemented as a set of RTFDs using the AO-design patterns prescribed in the work by Lee et al, (Lee and Kang, 2004, Lee et al., 2009b, Botterweck and Lee, 2009, Cho et al., 2008).

1.6 Motivation

After the emergence of FODA (Kang et al., 1990), researchers soon realized the need for other types of feature dependencies to be represented in the models developed after performing FODA. They proposed new extensions to the first generation modelling languages. The extensions resulted in different types of feature dependencies for instance, the introduction of “intentional”, “environmental”, and “usage” dependencies (Ferber et al., 2002). Even though

these approaches tried to resolve the feature dependency management issues in software systems, but there still exists the research gap of consistency checking between the feature dependencies on model level and their respective implementations in the source code.

Runtime feature dependencies need to be analysed in software product line assets to detect any undesired end product behaviour during execution. Operational and Activation feature dependencies (O&AFDs) are in the focus of this research. Unable to perform consistency checking of O&AFDs among product line artefacts can lead to various types of operational and activation related inconsistencies in the end product.

In the thesis, definition of inconsistency related to RTFDs is as following,

“A situation where a modular implementation of a runtime feature dependency in the source code doesn’t comply with its respective specification(s).”

The approaches like (Apel et al., 2011, Kaestner et al., 2007, Leich et al., 2005) deal with the feature interactions by encoding them and applying the off-the-shelf model checking technology. However, they are not focusing on runtime feature dependencies established by the work (Lee et al., 2009c, Lee et al., 2009a, Botterweck and Lee, 2009, Cho et al., 2008, Lee and Kang, 2004).

During the DE process, the software assets are developed and maintained to address not only the software system specification but also the market needs. Once the DE assets are developed, they tend to evolve over time leading to continuous maintenance and verification of the product line assets. It is often seen that the product line assets are developed in a distributed and collaborative environments (Mens and Straeten, 2007). Evolution of product line assets requires adding, deleting, updating the product line assets over time. In the context of this research, various stakeholders are involved in contributing towards evolving the feature dependencies in product line assets. Different stakeholders are involved during the development of product line assets.

Figure 1-5 motivates the need of developing the consistency checking approach for runtime feature dependencies in product line assets developed in distributed environments. For instance during the DE process, a requirements engineer using a particular specification language (e.g., feature modelling) to specify features and feature dependencies (e.g., structural, configuration and runtime dependencies) and a developer implementing specified product line features and feature dependencies in the source code.

For instance, consider a scenario (i.e., example scenario considering Figure 1-2) where the domain expert is performing the domain analysis and specifying the runtime behaviour of features History and Off (i.e., Runtime Excluded Activation). The specified behaviour means the following,

If Feature Off is activated the Feature History should turn off/deactivated during the execution of the product containing these features.

On the other hand the developer is implementing the specified Runtime excluded activation behaviour using the AO-design patterns in the source code.

The product line engineer wants to verify, if the implemented behaviour as runtime excluded activation is implemented as intended (i.e., using the AO-design pattern for runtime excluded activation) and not something else (i.e., using the AO-design pattern for runtime included activation).

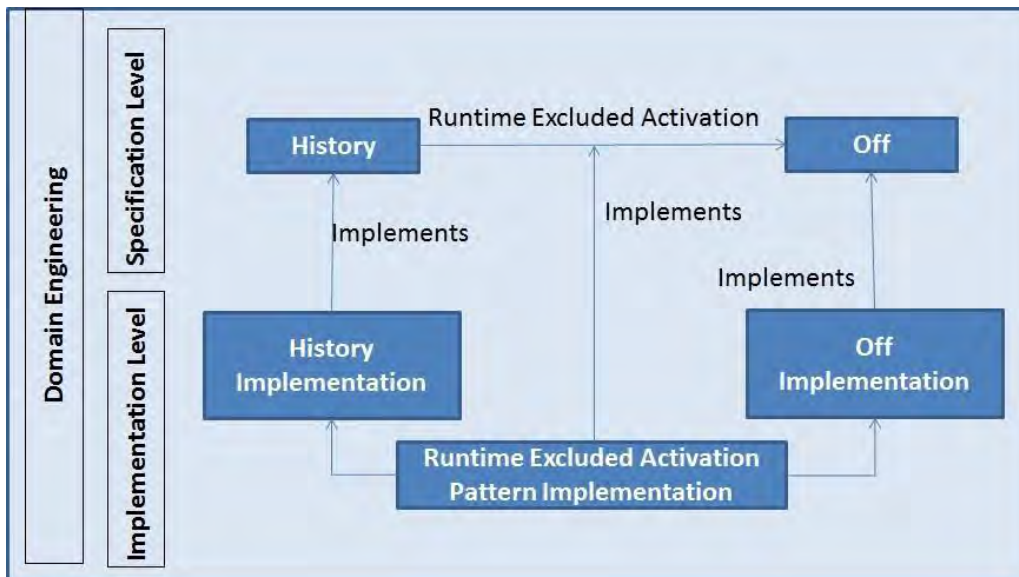


Figure 1-4 Scientific Calculator RTFD Motivational Example

This situation requires a consistency checking support for the product line engineer to facilitate him to perform the verification challenge. In the absence of consistency checking support the product line engineer have to manually identify the runtime excluded pattern in the source code and then verify it with respect to its specification. In a large product line, where there are thousands of features and many more dependencies among them, it requires an extensive manual work which might lead towards inconsistent product behaviour been implemented in the final product.

Research Question/Challenge¹: *How can we automate or semi-automate the consistency checking of runtime feature dependencies in product line assets to facilitate the product line engineer to identify runtime operational and activation inconsistencies in the end product during the domain engineering process?*

¹ Research question was inspired by a scenario taken from work conducted in the SFI funded project called model-based product derivation (Lero-SPL3) by Botterweck et al.

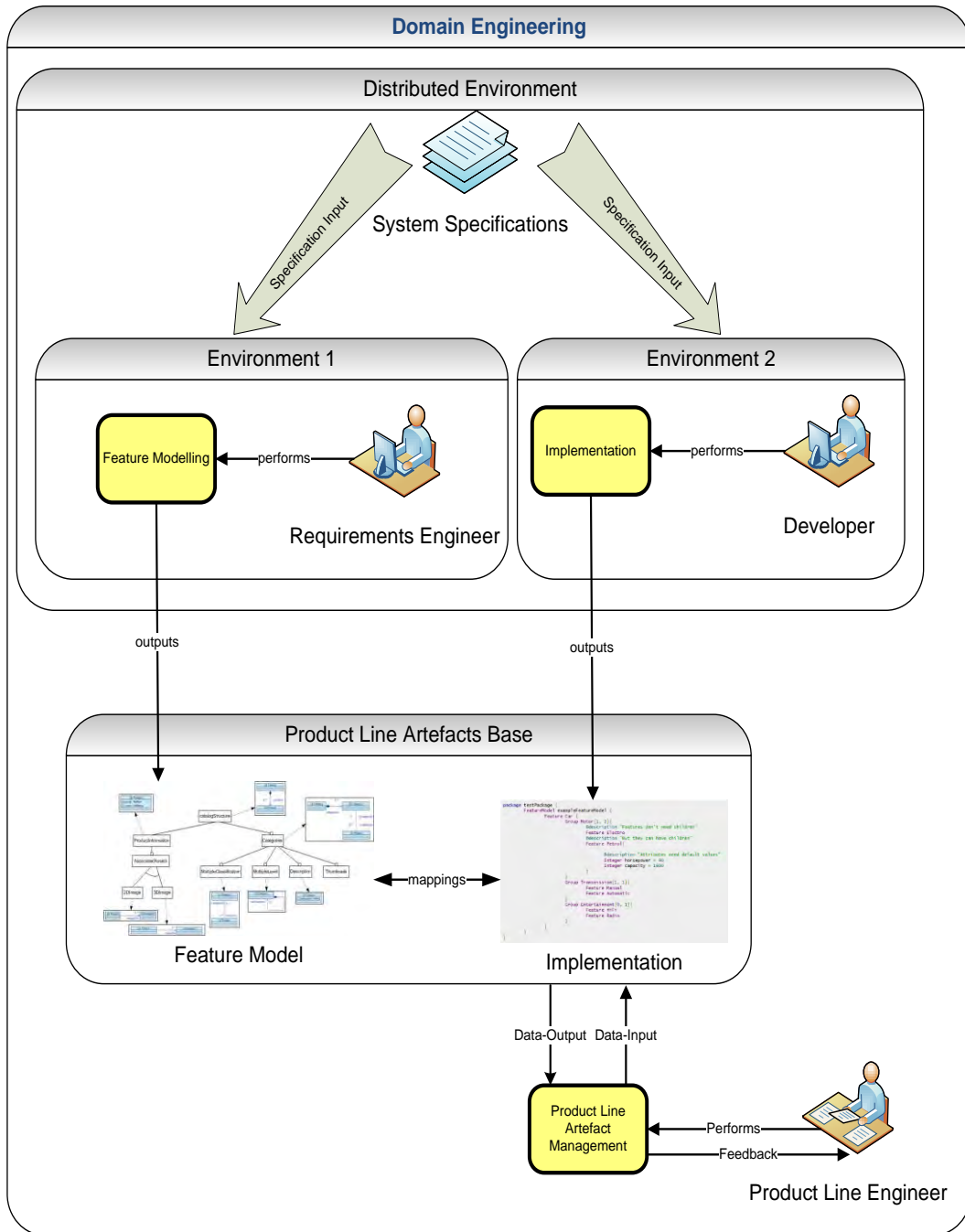


Figure 1-5 Motivational Scenario

This thesis work extends the work started by Lee et al., by introducing a specification language (i.e., DOFM²) and an implementation model on top of the source code containing the AO pattern-based implementations. This research work is in the context of the product line maintenance and assets verification, where a product line engineer wants to perform a consistency checking between specified RTFDs and its respective modular implementation (i.e., using a particular design pattern) in the source code.

1.7 Research Contributions

This section provides a list of contributions this research has made

- An extension to the existing work started by Lee et al. (Lee and Kang, 2004, Lee et al., 2009b, Botterweck and Lee, 2009, Cho et al., 2008).
- A model-based consistency checking technique (Section 3.7)
- A set of inconsistency scenarios related to RTFDs (Section 3.5)
- A Domain Specific Language (DSL) realizing the product line assets (i.e., DOFM, IM and PM) (Section 0)
- A set of AspectJ-based patterns detection algorithms (Section 4.4)
- An prototype realizing the AspectJ-based patterns detection algorithms (Section 4.5)
- A mutation testing inspired technique to validate the proposed consistency checking technique (Section 5.2.2)
- A Literature review focused on feature interactions and feature dependency management techniques in product lines (Section 2.3)

1.8 Outside the Scope of the work

The following aspects are outside the scope of this work

- Structural and configuration features' dependencies (Lee and Kang, 2004)
- AE activities like the product configuration, the assembly and the deployment (Jézéquel, 2012)

² Dependency-Oriented Feature Model (Section 3.6.1)

- Visualization techniques for model-driven product lines (Cawley et al., 2009)

1.9 Structural Overview of the Dissertation

Figure 1-6 provides a high level overview of the thesis along with the research question and the research approach adopted to resolve the formulated research questions for each of the chapters. The dissertation is organized as follows,

Chapter Two: Literature Review. The chapter provides an overview of the research challenge this thesis is attempting to resolve. The chapter formulates a research question to perform a state-of-art literature review of approaches that are focusing on feature dependencies consistency checking and management in product line assets. The chapter discusses the research gap identified and concludes with the summary of the chapter.

Chapter Three: Consistency Checking of Runtime Feature Dependencies in Product Line Assets. This chapter presents the conceptual framework contribution in the form of a model-driven process model that allows a product line engineer to perform semi-automatic consistency checking of RTFDs in product line assets. The chapter also discusses the inconsistency scenarios that are taken into consideration by the consistency checking approach, the prerequisites and assumptions and the stakeholders involved in the processes of the proposed technique.

Chapter Four: Implementation Concepts and Prototype Support. This chapter provides a detailed discussion on the working of the proposed technique along with the tool support for the proposed model-driven technique discussed in chapter three, which allows a product line engineer to semi automatically detect inconsistencies between the runtime feature dependencies specified and their respective modular implementations. The implementation choices for the tool support are discussed with the domain specific language development, algorithms for pattern detection and the constraint language for validation purposes.

Chapter Five: Validation and Evaluation. The aims of this chapter are to describe the validation of the solution presented in previous chapters and to demonstrate that the proposed technique works in practice. Evaluation of the proposed approach is performed based on the formal interviews.

Chapter Six: Conclusions and Future Directions. This chapter concludes the dissertation and provides an overview to the future directions. Subsequently the chapter summarizes the thesis, presents the key contributions of this thesis.

Summary of Appendices: **Appendix A**, provides overview on the EVL constraints. **Appendix B**, provides remaining validation discussed in Chapter 5 based on mutation testing inspired technique. **Appendix C**, provides Java-based implementation of the pattern detection algorithms discussed in Chapter 4. **Appendix D**, provides a questionnaire developed for formal evaluation of the proposed technique. **Appendix E** provides implementation of the EVL constraints. **Appendix F** provides list of accepted publications. **Appendix G** provides transcript of the interviews conducted for the formal evaluation of the proposed techniques.

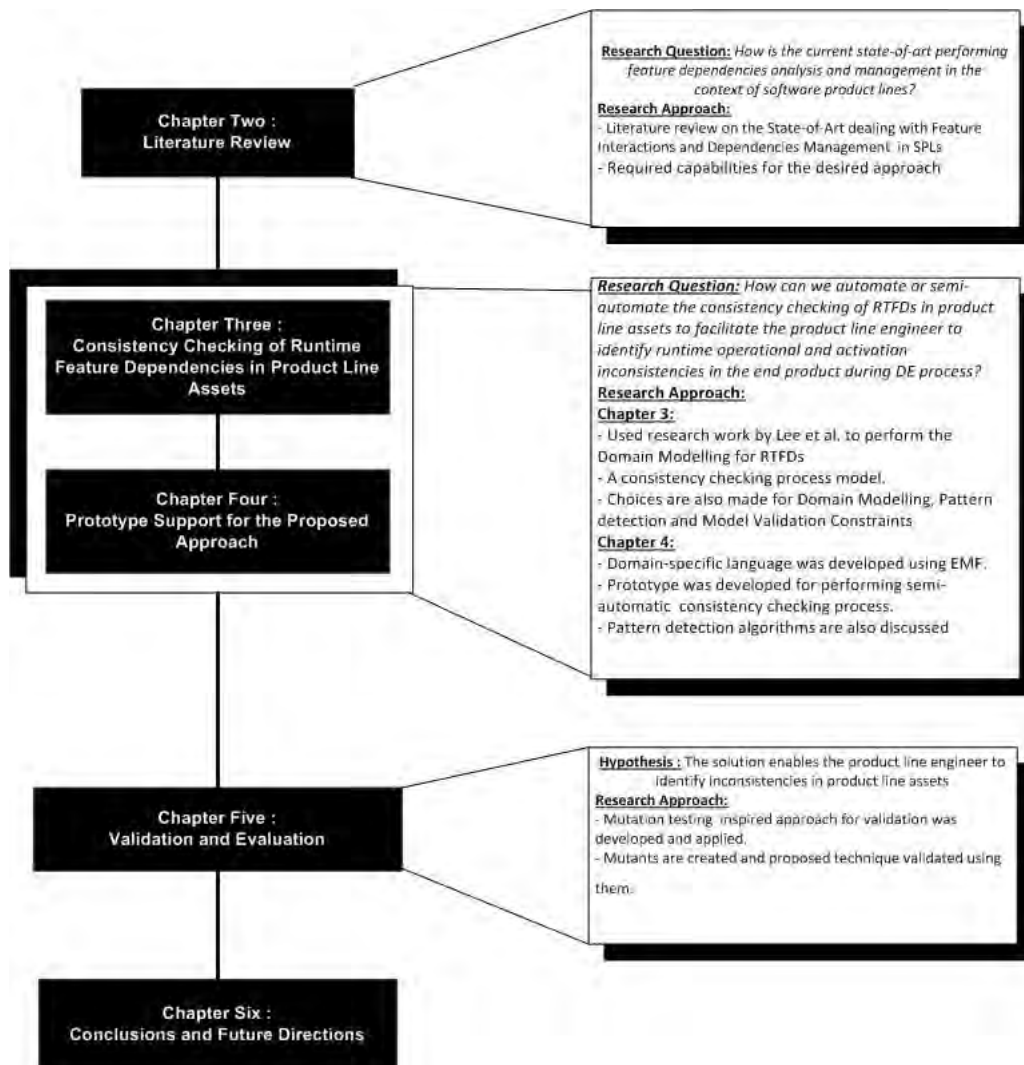


Figure 1-6 Thesis Chapters Overview and Research Approach Adopted

Chapter Two: Literature Review

2.1 Objective

The objective of this literature review is to identify approaches that provide consistency checking and management of feature dependencies in the context of software product lines and to identify the research gap that motivates the need for the technique presented in Chapters 3 and 4.

2.2 Research Question for Literature Review

The formulated research question for the literature review is as follows,

How is the current state-of-art performing feature dependencies analysis and management in the context of software product lines?

2.3 Identified Approaches

This section discusses identified approaches based on the research question. I have used various data sources (e.g., Conferences, Workshops, Journals, ACM Digital Library, IEEE Computer Society Digital Library, Google and SpringerLink) for identifying the related literature. The following sections summarize each of the twelve identified approaches along with the identified approaches performing feature analysis in other software engineering domain (i.e., Section 2.3.13).

2.3.1 (Ferber et al., 2002)

The approach focuses on investigating feature dependencies and interactions that restrict the extraction of the variants from legacy assets of the product line. The authors provide two complementary views of a feature model in order to specify the features and the dependencies among them of a legacy product line. One view is providing tree-like structure to specify features whereas the second view provides dependency information between the specified features already specified in a tree-like structure. In feature dependencies and interaction view authors specify various types of feature dependencies namely, *Intentional*, *Resource Usage*, *Environment Induced*, *Usage*, *Excluded Relation* (Figure 2-1). The authors have provided a detailed classification of the specified feature dependencies. Only views were provided but there are no guidelines as to how to implement or design the feature interactions and dependencies. The approach uses a XML-based feature model to specify features and dependencies in the

proposed two views. The approach adopted in two industrial case studies, 1) Engine Position Management, 2) Lambda Feedback Control for the evaluation purposes.

The approach provides a better solution to manage large number of feature dependencies of legacy systems by specifying them in the feature dependency view. It also provides a better understanding of the overall legacy system by using the proposed views.

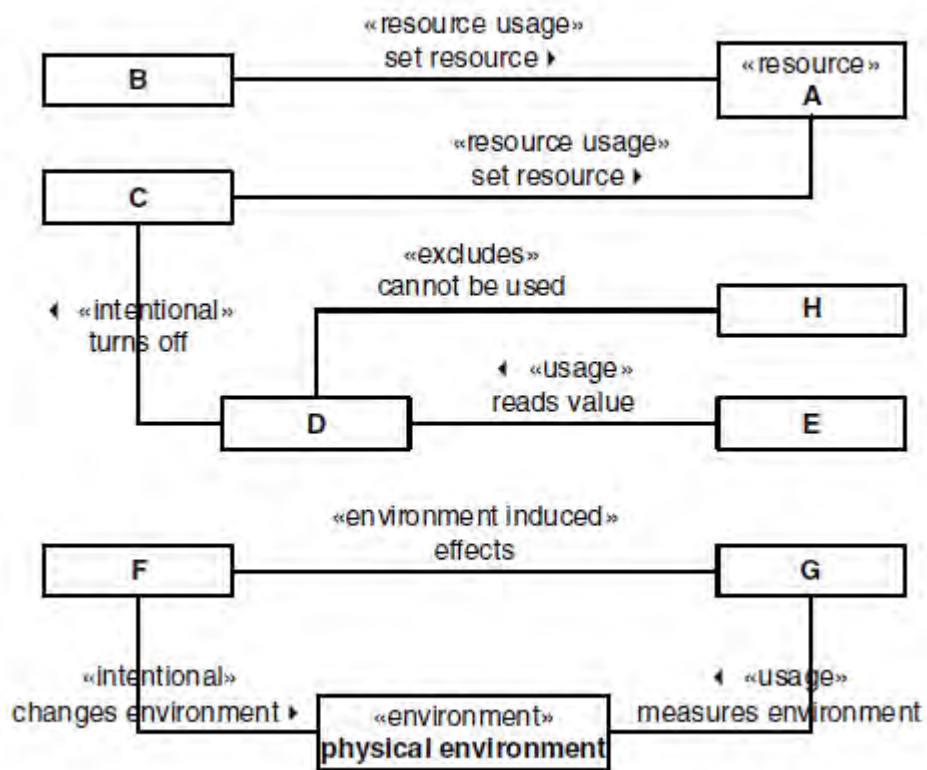


Figure 2-1 Feature Interaction and Dependency Representation Model
(Ferber et al., 2002)

2.3.2 (Lee and Kang, 2004)

The approach focuses on extending the feature model to analyze feature dependencies that are useful in designing reusable and adaptable product line components. The authors argue that structural relationships and configuration dependencies are not sufficient to develop reusable and adaptable product line assets. Other types of dependencies among features called operational

dependencies also have significant influences on the design of product line assets. The authors conducted feature dependency analysis to classify operational dependencies namely, 1) *Modification dependency*, 2) *Usage dependency*, 3) *Exclusive-activation dependency*, 4) *subordinate-activation dependency*, 5) *concurrent-activation dependency*, 6) *sequential activation dependency*. The authors then provided guidelines (i.e., *separating commonality from variability* and *hiding variable features from their usage client features*) about how to design reusable and adaptable components based on the results obtained by the feature-oriented analysis of the product line. The approach provides a solution to separate feature dependencies from feature implementations and designs them explicitly as components in the product line implementation. Elevator control software (ECS) case study was used to evaluate the approach.

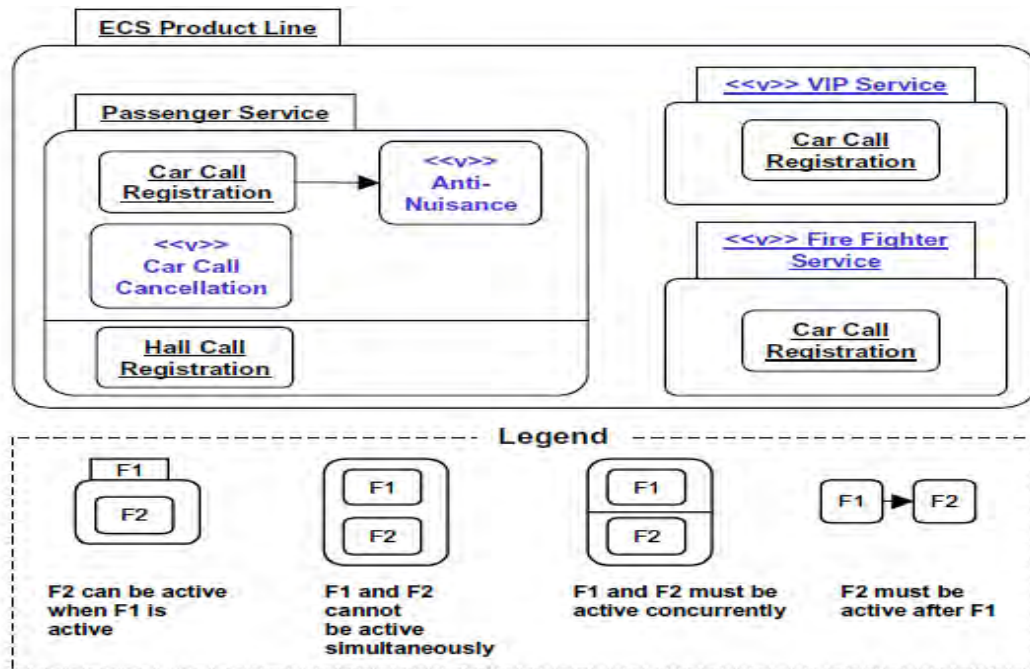


Figure 2-2 Part of Activation Dependencies in ECB Product Line (Lee and Kang, 2004)

2.3.3 (Ye and Liu, 2005)

The approach extends the feature model and provided two views namely 1) a tree view and 2) a dependency view. The approach focuses on the dependency view to specify the feature dependencies namely, 1) *Composition*, 2) *Generalization/Specialization*, 3) *Excludes*, 4) *Requires*, 5) *Variation Point*, 6) *Impacts* (Figure 2-3 for more details). Very generic classification detail was

provided of the feature dependencies by the approach of the authors. Feature relationships represented as a dependency matrix and a set of individual dependency diagrams. The dependent relationships *requires*, *excludes* and *impacts* are represented by standard UML notations and stereotypes are used to differentiate them from each other. Validation of feature dependencies is based on *requires* and *excludes* information contained in the dependency matrix. For feature dependency, analysis purposes the dependency view that consists of the dependency matrix along with the set of dependency diagrams is used. The dependency diagrams were generated by using an algorithm using the feature dependency information stored in a dependency matrix.

Rather than working with single dependency model to specify feature dependencies, the approach provides a scalable solution that consists of the feature dependency matrix to store the feature dependencies information and generate the feature dependency diagrams using the algorithm. For the evaluation purpose, the authors have applied their approach in the academic case study of Credit Card system.




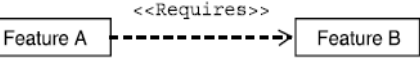
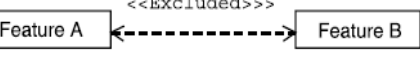
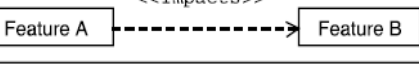
Relationships	Notation	Semantics
Composition		Feature A consists of Feature B
Generalisation/ specialisation		Feature A is a generalised feature of Feature B
Variation point		Feature A has one direct child Feature B who is a variable feature. Feature A represents a variation point
Requires		Feature A requires Feature B
Excludes		Feature A excludes Feature B and Feature B excludes Feature A (bi-directional)
Impacts		Feature A impacts on Feature B


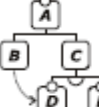


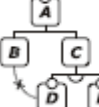
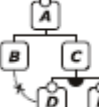
Figure 2-3 Feature Relationships and Notations (Ye and Liu, 2005)

2.3.4 (Zhang et al., 2006)

The approach focuses on feature interactions that influence the product requirements. For that purpose, the authors of the approach extend the feature model with new types of dependencies by partly analyzing the

operationalization and assignment of responsibilities between features. The introduced feature dependencies are classified in detail, which are namely 1) *refinement*, 2) *constraints*, 3) *influence* and 4) *interaction*. The authors also discussed three kinds of interconnections between the introduced dependencies, which are connections 1) between refinement and constraints, 2) between constraints and interaction and 3) between influence and interaction dependencies. The authors argue that interactions between the requirements need to be analyzed since they contribute to the development of the software.

For requirement dependency analysis, the approach uses identified dependencies along with their interconnections (Please consider Figure 2-4). Based on the results of feature dependency analysis, the approach provides an approach to design high-level software architecture. The approach provides two applications, which are 1) Simple document editor and 2) email client for the purposes of evaluation.

ID	Scenario	Constraints	Binding-State	Verification Result
1		<i>all-group</i> (A, B), C <i>requires</i> A, B <i>requires</i> C.	<i>Undecided</i> = {A, B, C} <i>Bound</i> = {} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Satisfied. <i>Crit3</i> : Satisfied.
			<i>Undecided</i> = {C} <i>Bound</i> = {A, B} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Satisfied. <i>Crit3</i> : Violated by C.
2		<i>all-group</i> (A, B, C), <i>multi-bound</i> (D, E) m- <i>requires</i> C, D <i>excludes</i> E, B <i>requires</i> D.	<i>Undecided</i> = {A, B, C, D, E} <i>Bound</i> = {} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied <i>Crit2</i> : Violated by E. <i>Crit3</i> : Satisfied.
			<i>Undecided</i> = {D, E} <i>Bound</i> = {A, B, C} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied <i>Crit2</i> : Violated by E. <i>Crit3</i> : Violated by D.
3		<i>all-group</i> (A, B, C), <i>multi-bound</i> (D, E) m- <i>requires</i> C, B <i>requires</i> D.	<i>Undecided</i> = {A, B, C, D, E} <i>Bound</i> = {} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Satisfied. <i>Crit3</i> : Satisfied.
			<i>Undecided</i> = {D, E} <i>Bound</i> = {A, B, C} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Satisfied. <i>Crit3</i> : Violated by D.
4		<i>all-group</i> (A, B), C <i>requires</i> A, B <i>excludes</i> C.	<i>Undecided</i> = {A, B, C} <i>Bound</i> = {} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Violated by C. <i>Crit3</i> : Satisfied.
			<i>Undecided</i> = {C} <i>Bound</i> = {A, B} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Violated by C. <i>Crit3</i> : Satisfied.
5		<i>all-group</i> (A, B, C), <i>multi-bound</i> (D, E) m- <i>requires</i> C, D <i>excludes</i> E, B <i>excludes</i> D.	<i>Undecided</i> = {A, B, C, D, E} <i>Bound</i> = {} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Violated by D. <i>Crit3</i> : Satisfied.
			<i>Undecided</i> = {D, E} <i>Bound</i> = {A, B, C} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Violated by D. <i>Crit3</i> : Violated by E.
6		<i>all-group</i> (A, B, C), <i>multi-bound</i> (D, E) m- <i>requires</i> C, B <i>excludes</i> D.	<i>Undecided</i> = {A, B, C, D, E} <i>Bound</i> = {} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Violated by D <i>Crit3</i> : Satisfied.
			<i>Undecided</i> = {D, E} <i>Bound</i> = {A, B, C} <i>Removed</i> = {}	<i>Crit1</i> : Satisfied. <i>Crit2</i> : Violated by D <i>Crit3</i> : Violated by E.

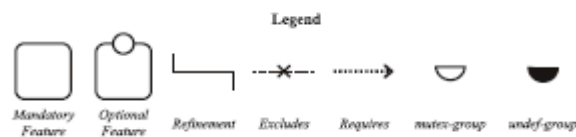


Figure 2-4 Detecting Feature Dependencies Anomalies (Zhang et al., 2006)

2.3.5 (Lee et al., 2006)

The approach provides a feature-oriented approach to manage domain requirement dependencies. It uses a graph-based technique to facilitate product releasing. The approach classifies feature dependencies in two groups, which are 1) static dependencies and 2) dynamic dependencies. The authors of the approach further classified static and dynamic dependencies as following;

- 1) Static dependencies are *decomposition*, *generalization* and *static constraints*
- 2) Dynamic dependencies are *serial*, *collateral*, *synergetic* and *change*. *Change* dependency is further classified as 1) *state change*, 2) *behaviour change*, 3) *data change* and 4) *code change* dependencies.

The authors used contiguous lines to represent static dependencies and broken lines for dynamic dependencies. Figure 2-5 represents the tables for assignment of eigenvalues to each type of dependencies and feature dependency table. Stereotypes are used in the feature dependency model to distinguish them from each other. Feature dependencies are managed in a feature dependency table representing the directed dependencies among the features. The feature dependency table is then transformed into a feature dependency forest. Each of the features has a dependency tree attached to it whose root is a feature. Maximal connective feature dependency graphs are generated using the algorithm called *MaximalConnectiveDependencyGraphGenerator*. The generated graphs are further utilized to incrementally derive products by identifying which features are to be included in the release. The authors utilized spots and futures transaction product line as a case study to validate their approach.

Eigenvalue of dependencies

Type of dependency	Assigned eigenvalue(binary)
decomposition	01000000 (0x 40)
generalization	00100000 (0x 20)
required	00010000 (0x 10)
excluded	10000000 (0x 80)
serial	00001000 (0x 08)
collateral	00000100 (0x 04)
synergetic	00000010 (0x 02)
change	00000001 (0x 01)

Feature dependencies table

Feature(source)	Dependency	Feature(destination)
Feature1	decomposition (0x40)	Feature7
Feature2	Synergetic(0x02)	Feature1
Feature1	Synergetic(0x02)	Feature2
Feature3	required (0x10)	Feature4
Feature3	serial (0x08)	Feature4
Feature3	change (0x01)	Feature5
Feature5	Collateral(0x04)	Feature2
Feature2	Collateral(0x04)	Feature5
Feature6	excluded (0x80)	Feature1
Feature1	excluded (0x80)	Feature6

Figure 2-5 Tables for Eigenvalues of Dependencies (Upper) and Feature Dependencies (Lower) (Lee et al., 2006)

2.3.6 (Savolainen et al., 2007)

The approach focuses on analyzing the product line feature dependencies to identify unnecessary features. The approach also provides guidelines to resolve the situation when unnecessary features are identified during feature dependency analysis process. The approach uses dependency model to represent feature dependencies. No particular classification is but stereotypes like <<functionality>> is used to represent dependencies among features in the feature model.

The approach provides six separate sequential steps for feature management method. The authors manually perform the steps. The brief description of six sequential steps is as following;

Step 1: *Artifact consolidations*; the method assumes that product features specification is at hand along with the feature decomposition model.

Step 2: *Feature Dependency Analysis*; Manual analysis was performed based on the existing artifacts discussed in step 1. Feature dependency was traced forward and excess features (not specified in the product features specification) are identified and recorded.

Step 3: *Feature Dependency Restructuring*; Based on the identified excess the authors proposed two guidelines to resolve the excess feature situation. The authors suggest that either the product programs that are used to specify the product specification may not understand the decomposition model and dependency and made the wrong selections or the feature decomposition is performed wrongly leading to addition of unnecessary feature dependency to less need features. In first situation, the authors proposed that the excess feature needs to be specified in the original product specification. The second situation can be cured by restructuring the decomposition and dependencies in a manner that removes the unnecessary dependencies.

Step 4: *Artifact Consolidation*; Since Step 1 assumes that product features specification and decomposition model is already present, based on these pre-existing artefacts *realization dependency model* and *component dependency model* are obtained. The feature dependencies among features are realized by component dependencies called functional dependencies in the component dependency model. <<functionality>> stereotype is used to represent the functional dependencies between the components in the component dependency model. Components may need other services from other components to implement their responsibilities. In such situation <<implementation>> stereotype is used to mark such dependencies in the component dependency model.

Step 5: *Component Dependency Analysis*; During this step for each product specification a component configuration is derived based on the trace realization and implementation dependencies forward. For the derived component configuration record, the features it realized, by trace realization backwards. If the features realized is not found in the product specification then record the feature along with its component as excess features.

Step 6: *Component Dependency Analysis*; Based on the identified excess feature and realized component(s). The authors suggest two solutions, that are 1) change the realization mappings between father and realized component in a manner that dependent features are realized by same architectural elements and

2) restructuring component dependencies by changing component decompositions.

The authors of the approach argue that they have applied their approach in an industrial case study of mobile phone product line for evaluation purposes.

2.3.7 (Silveira et al., 2007)

The approach focuses on avoiding interferences during extension of the product line by presenting strategies related to product line assets documentation and management of feature dependencies. The authors of the approach argue that when software engineers want to extend product line functionalities or wants to configure a product then feature interference cause major problems. They point out inadequate documentation and management of feature dependencies are among the major reasons that restricts product configuration and product line functionalities extension.

During the development of YANCEES product line (Filho and Redmiles, 2005), the authors of the approach observed and classifies feature dependencies namely 1) *fundamental dependencies*, 2) *configuration-specific*, 3) *incidental dependencies* and 4) *dependencies on emerging system properties*.

Managing of feature interactions in YANCEES product line is by two related activities. Firstly feature dependencies are specified in the dependency model (Please consider Figure 2-6). The dependency model uses UML class like diagram along with stereotypes are used for both feature dependencies and features.

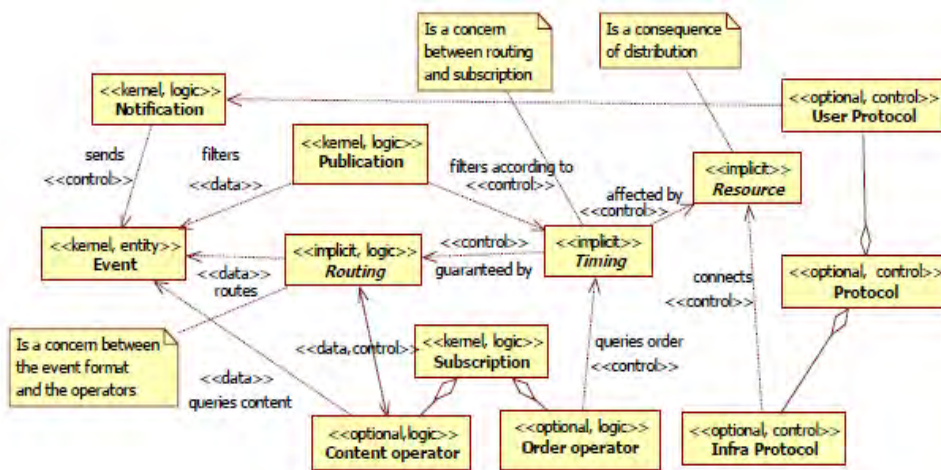


Figure 2-6 A Dependency Model of Publish/Subscribe Main variation points and concerns (Silveira et al., 2007)

Secondly, once the dependencies are identified and modelled they are formally incorporated in the implementation of the infrastructure. In YANCEES product line assets approach, it was achieved by the use of source code annotation using JAVA annotation API in both the common code and in the feature implementations (Please consider Figure 2-7).

The approach is considering software engineers as key stakeholders who are responsible for extending the product line functionalities and configuring products.

DEPEND.	ANNOTATIONS	DESCRIPTION
Fundamental	@DependsOnVP @DependsOnProperty	Expresses a general dependency existing between variation points and between properties.
Configuration-specific	@RequireFeature @CompatibleWithFeature @CompatiblWWithProperty	Express a dependency on a specific feature on a variation point. Expresses compatibility with existing features and emerging properties
Traceability	@ImplementsFeature @ImplementsVariationPoint	Marks classes that implement variation points and features in the code.
Incidental	@ProvidedGuarantees @RequiredGuarantees	Specifies the provided and required guarantees of the extension

Figure 2-7 Summary of contextual annotations used in YANCEES (Silveira et al., 2007)

2.3.8 (Kim et al., 2008)

The approach focuses and argues on the following;

- The features modules can be quantized as compositions of smaller modules called derivatives based on the new model of feature interactions as a result of work performed in virtual separation of concerns.
- A given program can be reconstructed by composing features in any order and the contents of a feature module that are expressed as a composition of derivatives is determined automatically by a feature order.
- Different feature composition ordering allows one to manage the contents of a feature module to isolate and study the impact of feature interactions.
- Proposes a Feature Dependency Graph (FDG) to capture derivative in the tree model interaction (Please consider Figure 2-8).

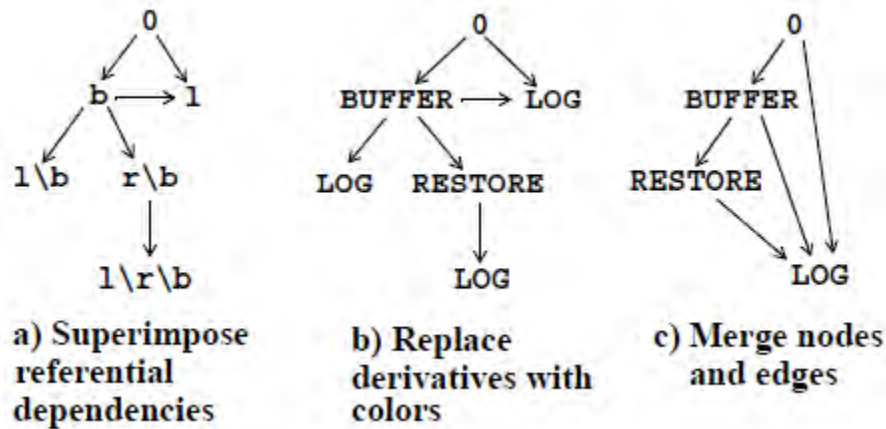


Figure 2-8 Feature Dependency Graph Construction (Kim et al., 2008)

The approach provides tree model interaction to represent feature interactions. The approach uses a tool support called CIDE (Colored Integrated Development Environment) to indicate feature interactions by nesting of colors. CIDE represents an SPL as a single program rather than as separate composable feature modules. A painted program using CIDE can be represented by a tree of interactions called the *derivative tree*. The approach uses incremental compose of feature from a Tree Model Interaction (Please consider Figure 2-9).

The approach uses low-level program details to determine feature interactions. For the analysis of SPLs, the approach uses the *Safe Composition* approach that proves that every legal composition of feature modules provides a type-safe program. The approach provides an algorithm called FCMA that traverse the derivative tree reassembles a program given a particular feature order, also produces contents of feature modules for that order. The authors evaluated their approach by converting AHEAD feature-composed programs into CIDE-form to analyze derivative trees, their safe composition and natural orders.

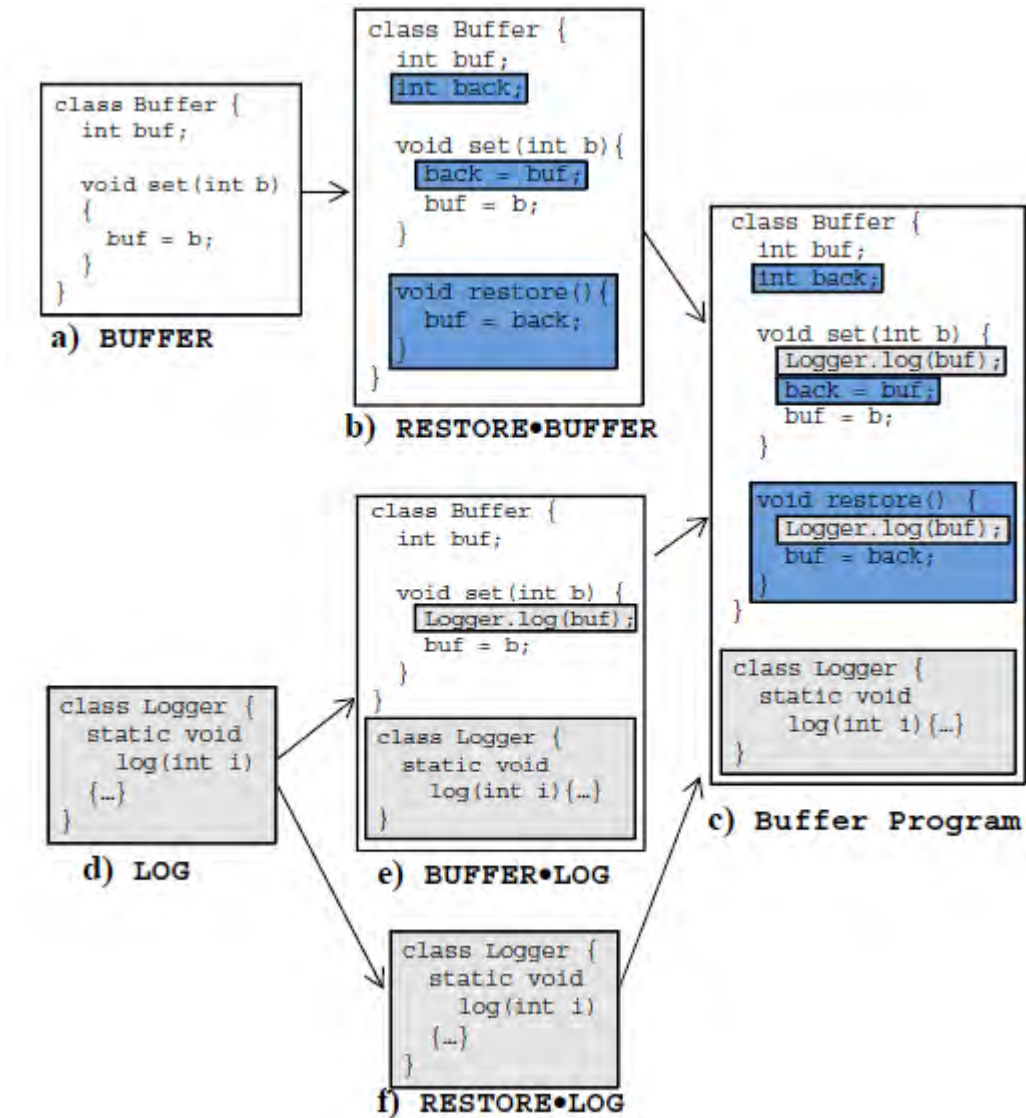


Figure 2-9 Different Incremental wise development of the Buffer Program in CIDE (Kim et al., 2008)

2.3.9 (Parra et al., 2010)

The approach focuses on feature-based composition approach to automatically derive a product architecture from a given feature configuration. The approach was developed in the context of application engineering process of a software product line. In order to automate the product architecture derivation process, the approach follows following steps;

Step 1: Implemented a Domain Specific Language (DSL) to represent each feature as a high-level aspect model (Please consider Figure 2-10).

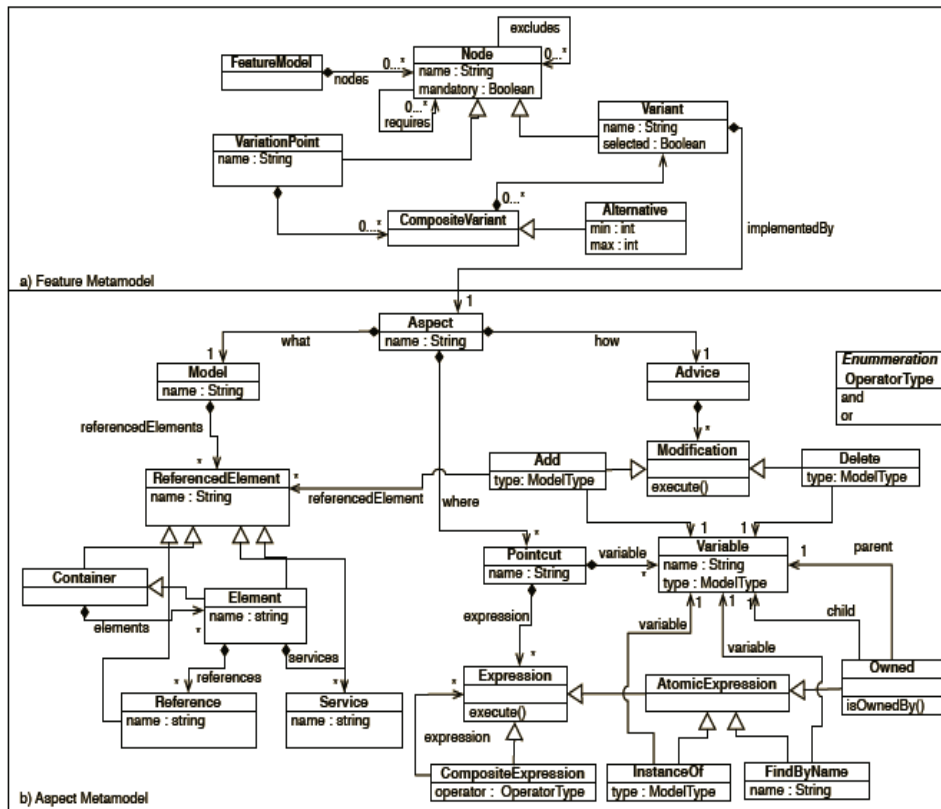


Figure 2-10 Feature Aspect meta-model DSL (Parra et al., 2010)

Step 2: For a particular given product configuration set of corresponding set of aspect models are weaved according to weaving strategy that is derived from in depth cross analysis of both feature interactions and aspect model dependencies.

The approach takes into account the structural dependencies (i.e., Requires and Excludes) with the given product configuration. Figure 2-11 represents the approach.

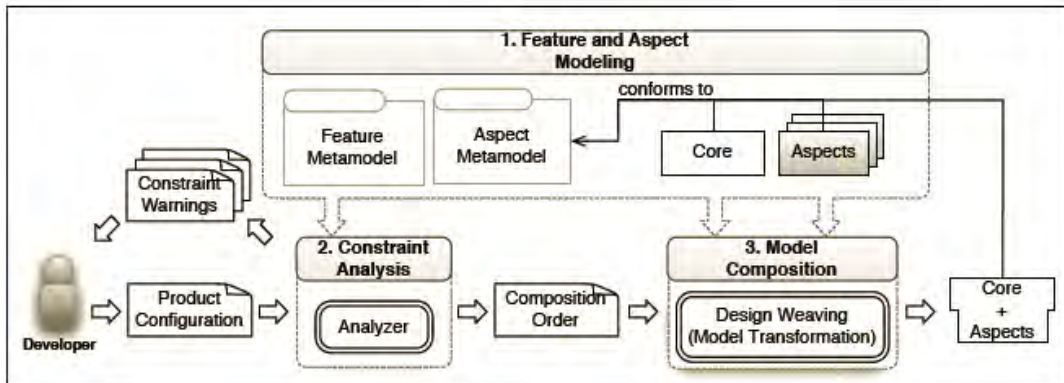


Figure 2-11 The Approach (Parra et al., 2010)

Part of a process for automatically deriving architecture based on given product configuration is analyzing structural feature dependencies. Feature interactions were analyzed using the constraint analysis activity and developed algorithm in the process. The approach is facilitating developer to automatically derive the product architecture. The approach was evaluated by using self-developed case study.

2.3.10 (Apel et al., 2011)

The approach focuses on automatic detection of feature interactions in software product lines. It uses product line verification techniques, variability encoding and off-the-shelf model checking technology. The approach is called *feature-aware verification*. According to the approach the features need to be specified using a developed automaton-based specification language as separate and composable units. The behavior of the feature is specified along with it using the developed automaton-based language. The specification of features also contains the constraints that can violate the overall system behaviour. The process of detecting feature interactions is based on local-specification of the feature. The approach provides two ways to detect features, which are 1) detection in the products and 2) detection in the product line. The approach is supported by a tool chain called *SPLverifier*. Two variations of *SPLverifier* are developed which are 1) for C language-based product line and 2) Java language-based product lines. The C version of *SPLverifier* uses the tools *FeatureHouse* for feature composition, *AutoFeature* for feature specification and *CBMC* or *CPAchecker* for model checking. The Java version of *SPLverifier* uses *FeatureHouse* for feature composition, *AspectJ* for feature composition and *JavaPathfinder* or its extension *jpf-bdd* for model checking purposes. The approach was evaluated in the context of AT&T email client.

2.3.11 (Batory et al., 2011)

The approach mainly focuses on understanding the relationship between features and feature interactions. The authors of the approach previously worked with sequential composition of features using the dot “.” operator. In this piece of research they introduced two new operators called the cross-product and the interaction composition. The authors gave a formal meaning to newly introduced operators. The formal model called coloring algebra is inspired by Colored Integrated Development Environment (CIDE) (Kim et al., 2008). The coloring algebra gives formal meaning to the sequential, the cross product and the interaction operation. The coloring algebra suggested a more general way to support coloring done in CIDE, so that documents of a product

line that were synthesized could be modified, and that these modifications could subsequently be back-propagated to their original feature-based product line representations. A general framework was defined to analyze feature interactions and compositions. Authors used XML-based MS Word documents to demonstrate the feasibility of their approach and understand relationship between feature modules and feature interactions. The approach has tool support called Paan. The tool support provides the following functionality 1) implement coloring algebra, 2) demonstrate the feasibility of back propagation and 3) working with MSWord documents.

2.3.12 (Mosser et al., 2012)

The approach focuses on stepwise process to deal with feature interactions at the domain level. The approach facilitates the developers to identify conflicts between features and create resolving strategies based on developer's intention to resolve the identified conflicts according to the developed algorithm. The approach directly interacts with the behavior of the end product. The feature interactions are implicit within the given product configuration which is an input to the algorithm that incrementally identify and resolves the feature interactions in a given product configuration. The approach was evaluated within the service-oriented architecture (SOA) domain. The approach works with given feature configuration and related business processes. For the tooling purposes the approach uses existing framework called ADORE and composition approach based on aspect weaving for building processes for given configuration. The approach was evaluated using an academic case study of JSEDUITE product line.

2.3.13 Feature Dependency Analysis in Other Software Engineering Domains

The main aim in identifying the state of the art was to detect the approaches that analyze and deal with feature dependencies in the product line domain, where feature dependencies are classified as first class citizens. However, during process, I recognized that other software engineering domains (e.g., software architecture and Model-Driven Development (MDD)) deal with similar feature dependency analysis challenges.

Software architecture provides a well defined, well documented view of a software system (Buckley et al., 2013). In the component and connector (C&C) architectural view of a software system, provided by an architecture description Language (i.e., XADL 2.0 (Dashofy et al., 2002)) , a feature can be implemented as a component or a set of components, and feature dependency

can be implemented as connectors and interfaces among architectural components. There is a strong need to perform consistency checking for identifying architectural drift (Buckley et al., 2013). Architectural drift is a situation, where implementation(s) of a software system's architecture do not remain consistent with its respective architectural view. Such a situation not only requires checking of the architectural components being implemented consistently but also the implementations of dependencies between the architectural components should also conform to their respective architectural component dependencies in the software system's architecture. Current works (Knodel et al., 2006, Buckley et al., 2013, Passos et al., 2010, Murphy et al., 1995) provide a reflection modelling-based solution to detect architectural drift as a result of lack of knowledge of the developers implementing the developed architecture over time. Further work (Abgaz et al., 2012) provided an analysis technique for dependencies for ontology-driven content-based systems of a software system.

Model-driven development (MDD) is a paradigm that considers models as a first class citizens that are utilized for development and maintenance of software systems. As discussed earlier, feature modelling(FM) is one of the techniques to specify the features (e.g., functional and non-functional) and dependencies among features of software systems. Current works use feature modelling as a technique to analyze feature dependencies (Daizhong Luo, 2009, Gibson, 1997, Kang et al., 1990, Kang et al., 2002). The mentioned approaches are different then the proposed technique in a way that the proposed approach the following,

- performs semi-automatic analysis of a particular types of feature dependencies (i.e., operational and activation dependencies),
- focuses on various stakeholders (e.g., developers and product line engineer),and
- performs static analysis of the source code to detect implemented design patterns for activation dependencies.

However, the mentioned approaches either extended the feature models to represent feature dependencies, or applied design patterns in the source code to manage feature dependencies. Also, the mentioned approaches provided manually intensive solutions to manage feature dependencies.

2.4 Discussion of Literature Comparison Criteria

Section 2.3 provided an overview of each of the identified approaches. This section provides the comparison criteria for the twelve identified approaches. The comparison criterion was developed while discussing the literature review. The approaches identified had various things in common that contributed towards identification of the comparison criteria. For instance, a few of approaches were focusing on extension of feature models to manage the feature dependencies in software product lines. This contributed towards a heading/category of “*Extending Feature Models*”. It is a bottom-up approach because it was already in the papers. The main purpose of the comparison criteria was to allow the comparison and contrast of the different solutions. The tabular comparison is later on visited in Chapter 5 (i.e., Section 5.5).

I have compared the identified literature/approaches under the following categories: 1) Extending Feature Models, 2) Well defined analysis process for feature dependencies, 3) Feature interaction and dependency analysis, 4) Tool supported approaches, 5) Focus on a specific stakeholder, 6) Focus on runtime feature dependencies and 7) Evaluation criteria.

The comparison criteria were also inspired by unpublished work by (Muñoz, 2009) on a systematic review of the feature interactions in software product line based on guidelines by the work (Kitchenham et al., 2009). Based on the research questions, the systematic review used the following comparison criteria: *representation of the feature interactions, roles of stakeholders described, defined activities to identify feature dependencies, was the approach evaluated in academic or industrial environment and automated solution for feature interaction* (Muñoz, 2009).

It must be noted that the comparison criteria factors generally change, based on the aim of the literature review. In the comparison criteria developed in this dissertation, a few of the important factors developed by the work (Muñoz, 2009).were reused in the context of detection and analysis of feature dependencies, in particular, *evaluation criteria, well defined processes for analysis, focus on a specific stakeholder*.

The comparison criteria can be applied to the model-driven domain (MDD) and software architecture, which have similar concerns, like behaviour conformance between architectural layers. For instance, these comparison criteria can be applied to the work by Qamar (2013), focusing on dependency management of feature dependencies in mechatronics domain. A brief discussion of each of the headings is as follows,

Extending Feature Models, under this heading, I have discussed all the approaches that established a new categorization of feature dependencies and used/extended feature modelling techniques to specify and analyze the newly introduced feature dependencies.

Well Defined Analysis Process for Feature Dependencies, under this heading, I have discussed all the approaches that have a defined process or steps in order to perform consistency checking/ analysis of feature dependencies.

Analysis of Feature Dependencies in Product Line Assets, I have provided discussion of approaches that are providing analysis of feature dependencies in product line assets at a different level of abstractions (e.g., Specification, design, implementation and source code).

Tool Supported Approaches, the discussion is made for all the approaches that are providing a commercial or a research prototype for their proof of concept.

Focus on a Specific Stakeholder, approaches focusing on a particular product line stakeholder are discussed under this heading.

Focus on Runtime feature dependencies, all the approaches that are providing analysis or management of feature dependencies that are related to the behavior of an end product are discussed under this heading.

Evaluation Criteria, approaches that have performed the evaluation of some kind using which type of case study (e.g., industrial or academia case study) for a proof of concept have been discussed under this heading.

2.5 Comparison of the Identified Literature

The twelve approaches are compared under the following criteria discussed above.

2.5.1 Extending Feature Models

Initially developed approaches focused more on providing the classification of feature dependencies for analysis and management of the product line assets. Six approaches were identified that introduced new types of feature dependencies and extend the feature model to specify the feature dependencies.

(Ferber et al., 2002) focused on investigating feature dependencies and interactions that restrict the extraction of the variants from legacy assets of a product line and provide two complementary views of a feature model in order to specify the features and the dependencies between them in a legacy software

product line. (Lee and Kang, 2004) focused on extending feature models with O&AFD relationships to analyze feature dependencies that are useful in designing reusable and adaptable product line components.

(Ye and Liu, 2005) extended the feature model and provided two views 1) a *tree view* and 2) a *dependency view*. The approach focuses on dependency view to specify the feature dependencies, 1) *composition*, 2) *generalization/specialization*, 3) *excludes*, 4) *requires*, 5) *variation point* and 6) *impacts*. (Zhang et al., 2006) focused on feature interactions that affect product requirements. For that purpose, the authors extend the feature model with new types of dependencies namely 1) *refinement*, 2) *constraints*, 3) *influence* and 4) *interaction* by partially analyzing the operationalization and assignment of responsibilities between features.

Work by Lee et al., (Lee et al., 2006) provided a feature-oriented approach to manage domain requirement dependencies and classify feature dependencies in static and dynamic dependencies. It uses a graph-based technique to ease the product releasing. Work by Silveira et al., (Silveira et al., 2007) focused on avoiding interferences during extension of the product line by presenting strategies related to product line assets documentation and management of feature dependencies. The authors classified feature dependencies namely 1) *fundamental dependencies*, 2) *configuration-specific*, 3) *incidental dependencies* and 4) *dependencies on emerging system properties*.

2.5.2 Well Defined Analysis Process for Feature Dependencies

A feature dependency analysis and management approach should also have a well-defined process or set of activities to follow. Five approaches provided a systematic process for feature dependencies analysis.

Work by Lee et al., (Lee et al., 2006) provided a feature-oriented approach to manage domain requirement dependencies comprising of three activities for managing domain requirement dependencies. Three activities are; 1) giving an eigenvalue to each of the requirement dependency types, 2) develop a dependency table representing type and dependency pair between features and 3) applying an algorithm *DependencyForsetGenerator* generates the dependency forest based on the table maintained in activity 2. Work by (Savolainen et al., 2007) provided six sequential activities for managing unwanted feature dependencies in a product line.

Work by Parra et al., (Parra et al., 2010) defined a semi-automatic process that applies constraints and algorithms for the analysis of configuration feature

dependencies (i.e., requires and excludes). Initially a DSL was developed to specify each feature as a high-level aspect model. Secondly a given product features configuration and corresponding set of aspect models are weaved according to weaving strategy and that is derived from in depth cross analysis of feature interactions and aspect model dependencies.

Work by Apel et al., (Apel et al., 2011) developed a *feature-aware verification* approach to automatically detect feature interactions in the context of software product lines based on local specifications of a feature. It uses verification techniques for a product line, variability encoding and off-the-shelf model checking technology. The automaton-based language was used to specify the feature behavior as separate and composable units along with the constraints that can violate the system behaviour.

Identified research (Mosser et al., 2012) provided an algorithm to incrementally identify and resolve the feature interactions. The approach directly interacts with the behavior of the end product. The feature interactions are implicit within the given product configuration which serves an input to the algorithm that incrementally identifies and resolves the feature interactions in a provided product configuration.

2.5.3 Analysis of Feature Dependencies in Product Line Assets

Five out of the 13 identified approaches provided analysis of feature dependencies in some form (e.g., feature interactions' detection, reasoning and inconsistency resolution) instead of introducing new classifications of feature dependencies in literature.

(Silveira et al., 2007) explicitly represented the feature dependencies in the dependency model and then manages them in the source code by using annotations for each type of dependencies. (Kim et al., 2008) resembles the work by (Silveira et al., 2007) as both managed the feature interactions between an abstract view of feature interactions and their respective source code implementation. However, work by (Kim et al., 2008) provided a mechanism to colour specific parts of the source code representing the feature interactions and respective tree model. The approach uses low-level program details to determine feature interactions. The approach provided an algorithm called FCMA that traverses the derivative tree reassembles a program given a particular feature order, also produces the contents of feature modules for that order.

(Savolainen et al., 2007) managed feature dependencies in a product line to identify unnecessary features by performing an analysis between two views called a feature dependency view and a component dependency view. However work by (Parra et al., 2010) provide a mechanism to manage both the structural and configuration dependencies during the application engineering process of the software product line. It provides a feature model and an aspect model views to perform model to code transformation. In order to provide consistency between feature and aspect model view constraints are applied to analyze structural constraint of a particular given product configuration.

(Mosser et al., 2012) provided a stepwise process to deal with feature interactions at the domain level. The approach analyzes feature dependencies between the configured feature diagram and business processes associated with each feature in the configured feature diagram using the developed algorithm.

2.5.4 Tool Support

Tool support has always been in question for the software engineering community. Tool support can vary from a simple prototype for proof of concept (POC) to a commercial tool suite. Most of the identified approaches rely on prototypes using off-the-shelf techniques for their proof of concept. Six approaches provided tool support in some form for analysis of feature dependencies in a product line.

(Parra et al., 2010) provided the approach supported by existing tools to analyze the configuration feature dependencies (i.e., include and exclude) between feature and aspect models. Work by (Apel et al., 2011) also used off-the-shelf model-checking technology for feature interaction detections in product line assets. (Apel et al., 2011) provided a tool supported approach called SPLverifier. SPLverifier is implemented in two tool chains, 1) one for the verification of the C product lines and 2) one for the verification of the Java product lines. The C version uses the tools FeatureHouse for feature composition, AutoFeature for feature specification, and CBMC or CPAchecker for model checking. The Java version uses FeatureHouse, AspectJ, and JavaPathfinder or its extension jpf-bdd for model checking.

(Kim et al., 2008) provided an idea of tool support but not automatic using a tool called CIDE. (Hasselbring, 2011) provided a tool support limited to only source code level called Keiker. (Batory et al., 2011) provided a tool support called Paan, that (1) implemented coloring algebra, (2) demonstrated the feasibility of back-propagation and (3) worked with MSWord documents.

(Mosser et al., 2012) also provided a tool support for their proposed approach that uses an existing framework called ADORE and composition approach based on aspect weaving for building business processes for given configuration.

2.5.5 Specific Product Line Stakeholders

Approaches developed by (Silveira et al., 2007) and (Hasselbring, 2011) took software engineers as target stakeholders to manage feature dependencies in the context of software product lines. Whereas approaches developed by (Parra et al., 2010) and (Mosser et al., 2012) focused on developers for managing the feature dependencies between product line artefacts.

2.5.6 Runtime Feature Dependencies

Work by Apel et al., (Apel et al., 2011) attempted to check the feature interactions in software product lines between the specified features and their behavior in separate composable units. The approach tried to make sure that the features of the product work properly together. (Batory et al., 2011) provided a formal model called coloring algebra is inspired by existing work called Colored Integrated Development Environment (CIDE) (Kim et al., 2008). The work focused on structural feature interactions. The coloring algebra gives formal meaning to the sequential, the cross product and the interaction operation. Each colored part of the code represents a feature in the code and coloring code is used for representing feature interactions in the feature implementation. CIDE approach lacks modularity as it represents an SPL as a single, general program rather than separate composable feature modules. The interactions colored in the implementation are represented as a tree if interactions called a *derivative tree*. A general framework is defined to analyze feature interactions and compositions.

Both of the works by (Apel et al., 2011) and (Batory et al., 2011) focused on detecting feature interactions in the source code. (Lee and Kang, 2004) classified the feature dependencies into runtime feature dependencies. The work by (Lee and Kang, 2004) differs from Apel et al. (2011) and Batory et al. (2011) as it provided classification and applied modular pattern-oriented solution in source code to manage feature dependencies in a product line. Both of the approaches resembled to the proposed approach in this thesis work to some extent but differs as it is not only focusing on feature dependencies that are modularized using particular design patterns but also to analyze them for behavioural consistency between their specifications and respective implementations.

2.5.7 Evaluation Criteria

Most of the identified approaches are using academic case studies for the evaluation. Only one of the identified approaches (i.e., (Ferber et al., 2002)) used an industrial case study for evaluation purpose. Rest of the approaches used academia case studies. Table 2-1 shows the approach and the respective evaluation methodology (i.e., industrial or academia case study).

Approach	Evaluation Methodology
(Ferber et al., 2002)	Industrial case studies: 1) Engine Position
(Lee and Kang, 2004)	Academia case study: Elevator control
(Ye and Liu, 2005)	Academia case study: Credit Card case
(Zhang et al., 2006)	Academia case studies: 1) simple document
(Lee et al., 2006)	Academia case study: spot and future
(Silveira et al., 2007)	Academia case study: YANCEES
(Kim et al., 2008)	Academia case study: Examples from
(Savolainen et al., 2007)	Academia case study: Mobile phone
(Parra et al., 2010)	Academia case study: Self developed case
(Apel et al., 2011)	Academia case study: AT&T email client
(Batory et al., 2011)	Academia case study: MS word documents
(Mosser et al., 2012)	Academia case study: JSEDUIITE product

Table 2-1 Table of Approaches and Respective Evaluation Examples

2.6 Identified Research Gaps

The previous section compared the identified approaches dealing with management and analysis of feature interactions and dependencies within the context of software product lines. The following sections discuss the research gaps after performing the comparison of the twelve identified approaches. The research gaps identified serve as a set of required capabilities for the required consistency checking approach.

2.6.1 Lack of focus on runtime feature dependencies analysis

Most of the identified approaches focused on the analysis of structural and configuration feature dependencies (i.e., exclude and include feature dependencies). However, the approaches like (Lee and Kang, 2004, Lee et al., 2009a, Lee et al., 2009c) and (Apel et al., 2011) focused on modularization and detection of runtime feature dependencies and interactions respectively. Kang et al. (2009a) provided a pattern-oriented solution to modularize the O&AFDs that facilitates the feature dependencies modularization and reuse in source code. Apel et al. (2011) focused on specifying behaviour of features in the feature implementation and check it using model checking techniques. However, the

proposed approach in this dissertation is focusing on the O&AFDs analysis in SPLs assets (i.e., specification and respective implementations).

Current approaches (Classen, 2007, Kim et al., 2008, Parra et al., 2010, Xue et al., 2010, Apel et al., 2011) provided feature interaction detection strategies and didn't focus on the analysis strategies. However, this research is focused on the modularized runtime feature dependencies and provides a conceptual framework as a process model that provides; 1) model-driven approach for specifying the runtime feature dependencies, 2) detection and 3) consistency checking of them to identify the inconsistencies in product line assets.

2.6.2 Lack of guidelines to analyse modularize runtime feature interactions

Current approaches (Classen, 2007, Kim et al., 2008, Parra et al., 2010, Xue et al., 2010, Apel et al., 2011) focused on the feature interactions and dependencies that are implicitly implemented within the source code. They don't provide processes to be followed in order to perform consistency checking of modularize runtime feature dependencies.

Current works by (Lee and Kang, 2004, Lee et al., 2009a, Lee et al., 2009c) provided guidelines to implement runtime feature dependencies as separate modules. The analysis process is not defined and only designing the components implementing runtime feature dependencies is provided. A few other approaches like (Savolainen et al., 2007, Apel et al., 2011, Kim et al., 2008) provided some guidelines to analyse OFDs. The approach in this dissertation is focusing on runtime feature dependencies that are modularized using the guidelines provided by the research work (Lee and Kang, 2004, Lee et al., 2009a, Lee et al., 2009c).

2.6.3 Lack of Focus on Analysis of Model to Code Feature Dependencies

Current approaches (Ferber et al., 2002, Lee and Kang, 2004, Lee et al., 2006, Zhang et al., 2006, Batory et al., 2011, Kim et al., 2008) focused on feature interaction management in the SPLs implementation assets (source code) by extending and specifying the feature model with the newly introduced and classified feature dependencies. In principle, the authors classified newly introduced feature dependencies and didn't provide guidelines that can facilitate the model to code analysis of feature dependencies. The proposed solution, provides a model view of the code as an implementation model and perform model validation to analyse the source code at a model level.

2.6.4 Lack of Focus on Product Line Stakeholders

Most of the approaches are not taking into account the various stakeholders of the product line contributing towards product line assets development. Only works by (Silveira et al., 2007, Parra et al., 2010, Apel et al., 2011, Mosser et al., 2012) focused on various stakeholders dealing with the feature dependencies in the product line domain. This lack of focus on product line stakeholders should be taken into account because the stakeholders are the end users that are facilitated by the approach and solve a particular problem. The proposed approach focuses on product line stakeholders (i.e., requirements engineer, developer and product line engineer).

2.6.5 Lack of Tool Supported Approaches

Current approaches working with the feature interactions and dependencies detection lacked an automated tool support for the analysis purposes. However, the tool supported approaches like (Kim et al., 2008, Apel et al., 2011, Batory et al., 2011, Mosser et al., 2012) provided solutions to detect feature interactions but are limited to either configuration dependencies or implicitly implemented runtime feature dependencies. The approach in this dissertation provides focuses on consistency checking of the modularize runtime feature dependencies by detection and analyzing them in product line assets.

2.7 Summary

This chapter provided an overview of the state-of-art dealing with the detection and management of feature interactions and dependencies in SPL assets. The chapter provided a formulated research question to perform the literature review. The chapter summarized 12 identified approaches. Discussion of the identified state-of-art provided under the established headings. A comparison criterion for the identified approaches were identified and discussed. The identified research gap helped to identify the required capabilities of the approach that is required to perform the consistency checking of the runtime feature dependencies since most of the approaches focused on structural and configuration types of feature dependencies. Such a consistency checking approach should also be able to provide both guidelines and tool support that enables the product line stakeholders (e.g., developers, software engineers) to perform consistency checking of runtime feature dependencies in SPLs assets (i.e., specifications and respective implementations).

Chapter Three: Consistency Checking of Runtime Feature Dependencies in Product Line Assets

3.1 Objective

Chapter Two advocated that there is still a need of an approach for consistency checking of runtime feature dependencies in product line assets. This chapter presents a model-driven technique as a conceptual framework that allows a product line engineer to semi-automatically detects inconsistencies between the specified runtime feature dependencies and their respective implementations in order to achieve a functionally correct product. The approach follows the round-trip engineering (RTE) software-development tools process. This chapter also discusses the prerequisites along with assumptions, and the stakeholders involved in the processes of the proposed technique. The chapter provides the problematic scenarios of the inconsistencies in product line assets (i.e., between the specified runtime feature dependencies and their respective implementations). Detail discussion is also provided for the inputs and the expected outputs of each process of the proposed technique.

3.2 Proposed Technique Assumptions and Involved Stakeholders

The following sections discuss assumptions of the proposed technique and the involved stakeholders during the deployment and execution of the proposed technique.

3.2.1 Assumptions

The proposed technique has the following assumptions for its development and deployment.

- The implementations of the product line features in the Java programming language and the runtime feature dependencies are implemented using aspect-oriented patterns (Lee et al., 2009a).
- The developer has enough information regarding how to implement runtime feature dependencies using prescribed aspect-oriented patterns developed and discussed in the work by (Lee et al., 2009a).

- The mapping of the features to the feature implementations and the runtime feature dependencies to their respective implementations in the implementation model is correct and at hand for a product line engineer.

3.2.2 Involved Stakeholders

A graphical overview of the stakeholders performing the specific tasks provided in Figure 1-5. The following sections will discuss the role of the stakeholders in the context of the proposed technique.

3.2.2.1 Requirements Engineer

The requirements engineer (RE) is responsible for analyzing the software requirements specifications. The requirements engineer specifies the features and the runtime feature dependencies between product line features using requirements specification language (i.e., DOFM).

3.2.2.2 Software Developer

The software developer is responsible for implementing the features and runtime feature dependencies that have been specified by the software requirements engineer. The software developer uses the AO-based design patterns to implement the RTFDs in the source code.

3.2.2.3 Product Line Engineer

One of the responsibilities of the product line engineer is to perform the maintenance of product line artefacts during the DE process. The proposed technique enables the product line engineer to detect inconsistencies in product line assets (i.e., the specified runtime feature dependencies and their respective implementations).

3.3 Problem Overview of Runtime feature dependencies in a Product Line Context

Chapter One motivated the need for an approach to detect the operational and activation inconsistencies in product line assets. The inconsistencies emerge as a result of inability to specify and implement the same information shared among the product line assets (i.e., O&AFDs' specifications and their respective implementations).

When performing the product line maintenance and verification, the product line engineer wants to inquire if the implemented runtime feature dependencies utilizing the AO-design patterns conform to their respective specifications in

order to avoid any inconsistent product behaviour during its execution. This research focuses on verification of RTFDs specified and implemented in a software product line.

Aspect-oriented programming (AOP) and languages were developed to modularize the cross-cutting concerns. Systems developed with AOP have characteristics like flexibility, reusability, adaptability of the elements used to compose the system (Clemente et al., 2003). SPL engineering advocates reuse of platform artefacts. So the product line assets should be designed in such a way that they are reusable. Work by Lee et al. (Lee et al., 2009a) advocates that runtime feature dependencies should be designed in a manner that they are reusable, adaptable and flexible. Lee et al. (Lee et al., 2009a) utilized AOP technique called AspectJ programming to design the cross-cutting features and dependencies among them by utilizing the aspect-oriented design patterns. Lee et al., extended the work on relationship aspects (Pearce and Noble, 2006). The designed AO-design patterns are between a pair of features implementations. The feature implementation could be a component (e.g., in component-based architecture or implementation) or a service (e.g., service-oriented architecture).

Existing work by (Lee et al., 2009a) provided aspect-oriented design patterns to modularize runtime feature dependencies in the source code. This research is complementing the existing work by providing the following, 1) a specification layer on top of the O&AFDs' implementations in the source code, 2) a consistency checking technique to detect the inconsistencies in product line assets (i.e., the A&OFDs' specifications and their respective implementations) and 3) inconsistency detection reporting that enables the product line engineer to achieve that correct aspect-oriented patterns are implemented for the respective specified O&AFDs' type.

3.4 Round-trip Engineering (RTE)

An automatic maintenance process of consistency between multiple associated, changing software artifacts (e.g., specifications, design documents and source code) in software-development environments or tools is commonly referred to as a round-trip engineering (RTE) (Sendall and Küster, 2004). RTE is required when the same information is present in different software assets. And changing one software artefact (e.g., source code) requires related software artefact (e.g., specification document) to be updated consistently to reflect the change.

I have chosen RTE for being an iterative process. One of the important characteristics of RTE is performing an automatic update in response to

the detected inconsistencies in the software artefacts. There are two sub-types of automatic updating. One sub-type called instantaneous, and the other called on-demand update. The instantaneous updating requires immediate change implemented across to all the related software artefacts, whereas on-demand updating requires changes to be implemented at a particular instance of time. In the context of the proposed approach, I have selected on-demand sub-type of automatic updating. Selection of the on-demand type is because the product line engineer wants to perform a consistency checking at a specific point in time in an iterative manner.

3.5 Inconsistency Scenarios Related to Runtime Feature Dependencies Implementations

This section provides discussion on each type of categorized inconsistency scenarios using the running example from Chapter 1 (i.e., Figure 1-4).

The example sketched in Figure 3-1 represents two features, namely “*History*” and “*Off*”. Features are specified within the feature model along with a particular type of runtime feature dependency (i.e., runtime excluded activation). The specified runtime feature dependency in feature model can be of any type discussed in Section 1.4.

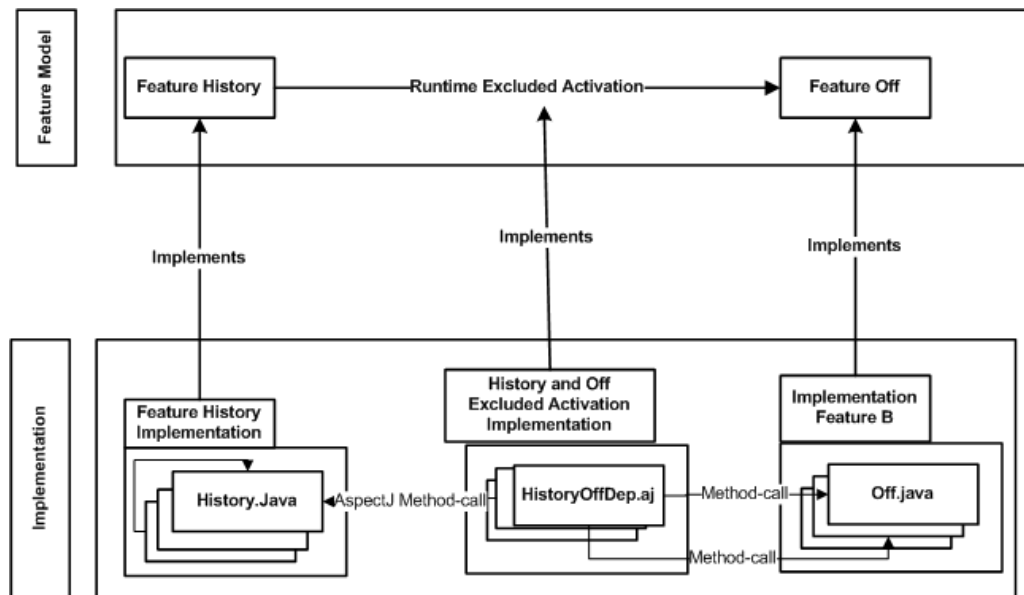


Figure 3-1 An Example to Demonstrate Inconsistency Scenarios in the Implementation of RTFDs

3.5.1 No Runtime Feature Dependencies Implementation

An inconsistency may occur, if a particular type of runtime feature dependency is not implemented as an AspectJ-based pattern. For instance, a runtime excluded activation dependency is specified between two features (i.e., features “*History*” and “*Off*”) but there exists no corresponding AspectJ pattern-based implementation in the source code. This kind of inconsistency scenario applies to all other types of runtime feature dependencies (i.e., modification, required activation, sequential activation and concurrent activation). Figure 3-2 represents an overview of the problematic scenario.

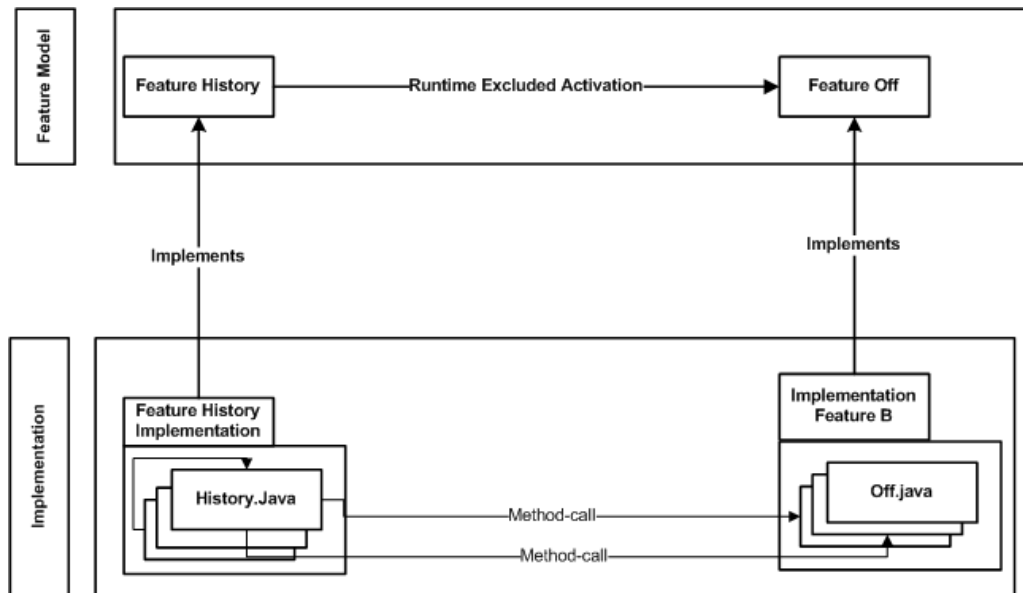


Figure 3-2 No Implementation of Specified Runtime Feature Dependency

3.5.2 Incompatibility between the Specified Type and its Respective Implementation

An inconsistency may occur, if the specified runtime feature dependency between two features “*History*” and “*Off*” is different from the type implemented in the source code. For instance, the requirement's engineer specified a runtime excluded activation dependency between features “*History*” and “*Off*”, whereas the developer implemented the runtime required activation type. This situation causes the wrong behavior being implemented, and it leads towards a faulty end product. This inconsistency scenario applies to all different types (i.e., Modification, Required Activation, Excluded Activation, Sequential Activation and Concurrent Activation).

3.5.3 Incompatible Specified Features Pair and its Respective Implementation Pair

An inconsistency may occur, if the specified type of runtime feature dependency between the feature pairs is not implemented accordingly. For instance, a requirement's engineer specifies runtime required activation between the features (i.e., features “**History**” and “**Off**”), whereas the developer implements it between the wrong pair of features (e.g., features “**History**” and “**ShiftKey**”). From such a situation, one can deduce that the required behavior is not implemented between the intended features pair leading towards a faulty end product. Semantics defined in DOFM model is FeatureHistory—RuntimeExcludedActivation→FeatureOff, whereas the implemented semantics is FeatureHistory—RuntimeRequiredActivation→FeatureShiftKey. In such an inconsistency scenario feature ShiftKey can be a sub-feature implemented as a sub-class in the source code (i.e., implemented behavior semantics is FeatureHistory—RuntimeRequired→FeatureShiftKey inherited-from FeatureOff).

3.5.4 Incomplete Implementation of the Runtime Feature Dependencies

An inconsistency may occur, if the runtime feature dependency is specified between two features (i.e., features “**History**” and “**Off**”), but there is an incomplete pattern-based implementation in the source code. An incomplete pattern-based implementation of excluded activation between features **History** and **Off** means that when feature **Off** is selected the feature **History** is not deactivated or cleared leading towards erroneous end product containing features **History** and **Off**. This kind of inconsistency scenario applies to all other types (i.e., modification, required activation, excluded activation, sequential activation and concurrent activation).

3.5.5 Implementation of Runtime Feature Dependency Not Preserving the Specified Direction

An inconsistency occurs, if the pattern-based implementation of a runtime feature dependency does not comply with the specified direction. The semantics of the direction comes from the DOFM model used to specify the runtime feature dependency. For example, semantics FeatureHistory---runtimeExcludedActivation→FeatureOff depicts that AspectJ pattern-based implementation should exclude feature **History** (i.e., source feature) (Figure 3-3) and not feature **Off** (i.e., target feature). Such type of an inconsistency scenario (i.e., reversal of direction) can occur in all other types (i.e., required activation, excluded activation, concurrent activation and sequential activation).

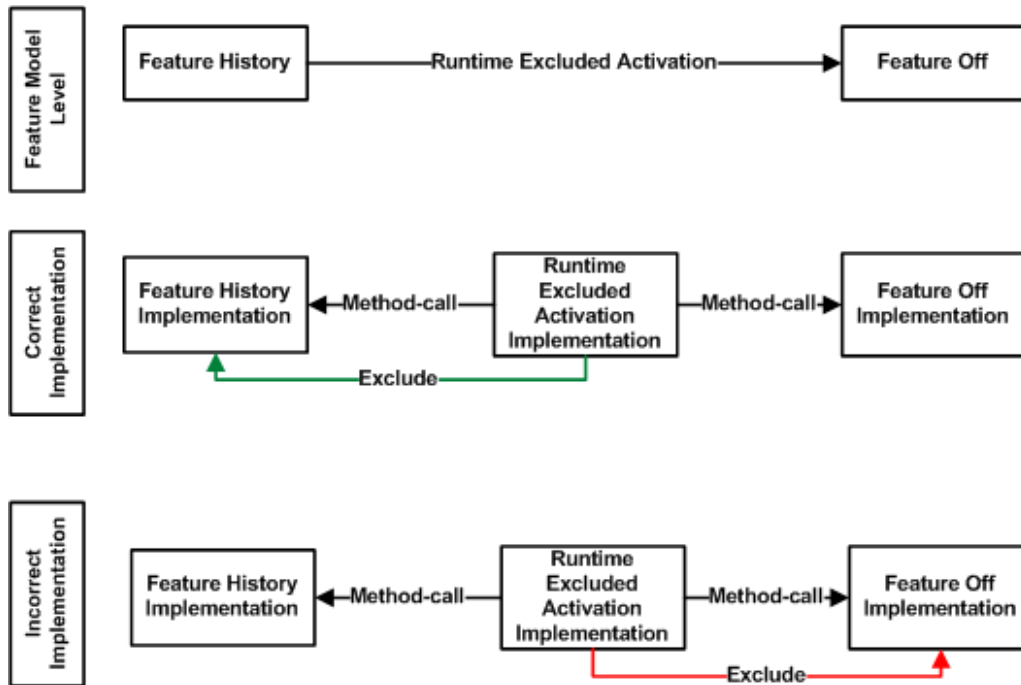


Figure 3-3 Specified Runtime Feature Dependency Order not Preserved in Respective Implementation

3.5.6 Semantic Inconsistency Related to Runtime Feature Dependencies

Semantic inconsistency related to runtime feature dependencies means that the particular parts of the pattern-based implementation (e.g., AspectJ functions Advice, pointcut and inter-type declaration statements) modifying intended parts of the code are not executed accordingly. It also means that the inconsistency can occur at the statement level. For instance, the following part of the code enables the deactivation or exclusion of the *History* feature implementation in the source code when implementation of *Off* feature is executed in the final product.

```

1. public void HistoryFM.terminate(){
2. System.err.println("Excluded Activation Called--
   History.ExcludedAct.OffButton");
3. aspectOf().clearHistory();
   }

```

Statement 3 enables the exclusion of the History that enables to clear the History of the functions being executed saved before the scientific calculator is turned off. But if this statement isn't executed accordingly before turning the scientific calculator off? Leading towards an inconsistency scenario in the final product having implementations of *History* and *Off* features. Such type of

inconsistency can occur as a result of code not being executed accordingly that enables to implement certain behaviour in the final product. This kind of inconsistency is applied to all other types of runtime feature dependencies discussed in Section 1-4 in Chapter 1.

3.5.7 An overview of Inconsistency Scenarios and Classification

In this thesis, inconsistency is broadly defined as any situation in which the implementation of a particular dependency between features does not comply with its specification. In particular, it means any violation of the rules of the relevant aspect-oriented design pattern that should have been applied, since the implementation of runtime feature dependencies is to be performed using these patterns. It must be noted that the prescribed aspect-oriented patterns are part of the solution for modularizing the RTFDs in the source code. Breaking the pattern contributes towards an inconsistency scenario. Inconsistency can be semantic or structural.

Semantic inconsistency is the type of inconsistency scenario found at the statement level. Such a scenario can arise if the implemented feature dependency is not executing the behaviour specified. Examples of such type of inconsistency can be found in Section 3.5.6.

Apart from semantic inconsistency, all other types of inconsistency in this thesis can be classified as structural inconsistencies. Structural inconsistencies are the types of inconsistencies that can arise due to a problematic implementation of the aspect-oriented design patterns (i.e., violation of the rules of the relevant aspect-oriented design pattern). Structural inconsistencies are related to each activation dependency type implemented as an aspect-oriented pattern except the usage dependency and modification dependency, which are implemented using generic aspects. However, semantic inconsistencies are related to both the activation and the operational feature dependencies.

The aspect-oriented pattern to implement a certain type of RTFD can be broken in various ways (e.g., deleting part of a pattern). Such a type of an inconsistency can be categorized as *pattern not detected* or *incomplete pattern implementation*. Examples of this type of inconsistency can be found in Sections 3.5.1 and 3.5.4.

The prescribed aspect-oriented patterns are of a special nature specifying behaviour pair-wise between features. The implementation of an aspect-oriented design pattern results in the parameterization of feature implementations. An inconsistency scenario can arise when parameterization of intended aspect-oriented pattern is *reversed*. Examples of such type of inconsistency can be found in Section 3.5.5.

An inconsistency can arise when the specified pair is *not implemented* accordingly using the prescribed pattern. Examples of such type of inconsistency can be found in Section 3.5.3.

An inconsistency scenario can also arise, when the type of specified type of feature dependency is *not preserved* in the pattern-based implementation. Examples of such type of inconsistency can be found in Section 3.5.2.

3.6 Domain Modelling of the Product Line Assets

In order to develop a model-driven support the domain concepts are modelled first. The domain modelling raises the abstraction level and enables one to manipulate and analyze the concepts of interest. The domain model represents the key concepts and the vocabulary as a structural view of the problem domain.

For the proposed technique, the Eclipse Modelling Framework (EMF) (Gronback, 2009) was used to perform the domain modelling. EMF provides an Ecore meta-modelling technique to define the domain in terms of a UML like class diagram. The EMF provides a modelling language called the Ecore model that allows to define different elements like, 1) **EClass**: it is used to represent a Class with zero or many attributes and zero or many references, 2) **EAttribute**: represents the property of the Class with a name and a type, 3) **EReference**: represents the association between various types (e.g., a containment reference) classes and 4) **EDataType**: represents the type of an attribute (e.g., double, float).

The meta-model enables one to specify, 1) the domain key concepts as classes, 2) the domain concepts' characteristics as attributes and 3) the relationships among the domain concepts as references. The domain model (i.e., Ecore meta-model) can effectively be used among various stakeholders to verify and validated the understanding of the problem domain.

I have used the Ecore modelling for the domain modelling of the product line assets. Three interrelated meta-models, namely the dependency-oriented feature model (DOFM), the implementation model (IM) and the pattern model (PM) were developed. Each of the developed domain models as an interrelated modelling language is discussed as follows.

3.6.1 Dependency-Oriented Feature Model (DOFM)

The DOFM language allows one to specify the product line features and the runtime feature dependencies between the specified features. DOFM provides a specification view for runtime feature dependencies. The DOFM also enables

the mappings between features and their respective runtime feature dependencies to their respective implementations in the implementation model (IM) by providing references to implementation model concepts (e.g., Packages, Classes and Attributes). DOFM language can be used as an extra view for the feature model or an only view for representing the RTFDs.

Figure 3-4 represents an Ecore meta-model developed in EMF for the specification of runtime feature dependencies. The developed meta-model has the following classes and attributes associated with them.

- ***FeatureModel***: An Ecore EClass represents a root of the model and composed of features
- ***Feature***: An Ecore EClass used to specify the features.
- ***Dependency***: An Ecore EClass used to specify the runtime feature dependencies and has the following subclasses, 1) *RuntimeModificationDependency*, 2) *RuntimeIncludedActionDependency*, 3) *RuntimeExcludedActivationDependency*, 4) *RuntimeSequentialActivationDependency* and 5) *RuntimeConcurrentActivationDependency*.

For each dependency type the source(s) and the target(s) are references to the *Feature* Ecore EClass. In order to specify new type(s) of runtime feature dependencies, the developer of the meta-model have to develop a new Ecore EClass in the meta-model and set the references (i.e., the source(s) and the target(s)) to the *Feature* classes.

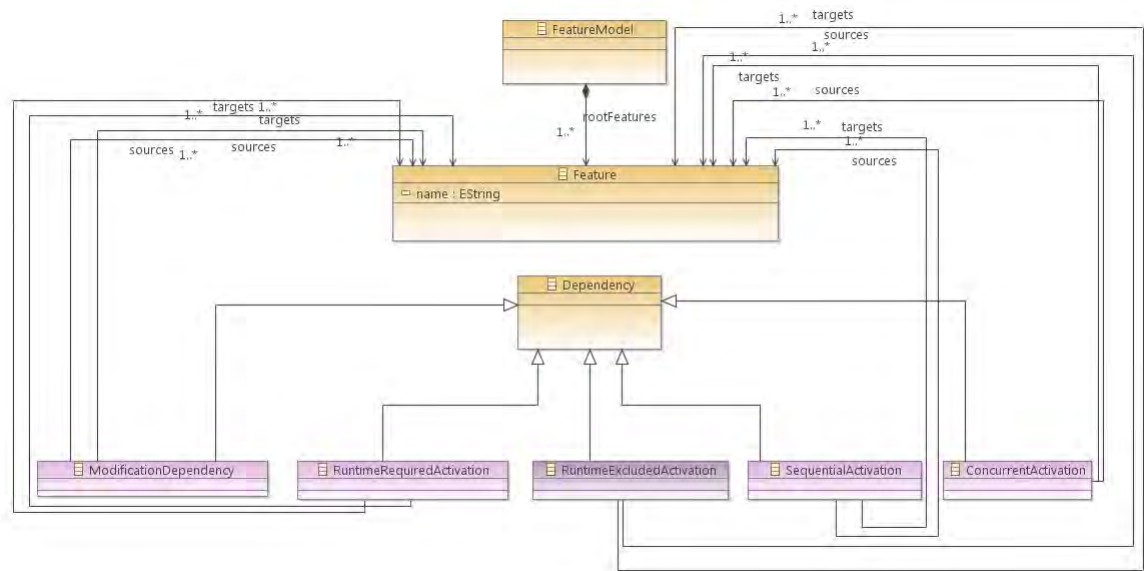


Figure 3-4 Dependency-Oriented Feature Model (DOFM)

An example of a particular specification looks like,

FeatureA—DependencyType→FeatureB, where 1) FeatureA and FeatureB represent the source (i.e., **History**) and the target (i.e., **Off**) features respectively and 2) DependencyType represents a type of dependency (e.g., **runtime excluded activation**) between the features **History** and **Off**.

3.6.2 Implementation Model (IM)

The implementation model provides an abstracted or a structural view on top of the product line Java/AspectJ-based source code implementing the product line features and the dependencies. The IM allows one to specify the Java source code concepts like Java or AspectJ classes (i.e., Java/AspectJ classes) as IM Java and aspect classes, Java packages as packages, Java class methods as IM class methods and references between Java classes (e.g., Java function calls) as IM class method calls. It has to be understood at this point that not all the Java related concepts (e.g., method arguments) can be modelled using the IM modelling language. The IM modelling concepts (e.g., IM Java/AspectJ class, IM Package, etc.) have references to the Pattern Model and DOFM concepts.

Figure 3-5 represents an ecore meta-model developed for representing the commons Java/AspectJ source code concepts (i.e., Project, Package, JavaClass, AspectJClass and Method) at a higher level of abstraction. The implementation model meta-model provides a structure to develop the implementation model

composed of the implementation related concepts. The implementation model has the following hierarchy

- Project consists of Packages and Relationships.
- Packages consists of PackagableElements which has a subclass Component
- Component can be an Aspect or a JavaClass
- Both JavaClass and Aspect are composed of methods
- Method has DirectedRelationships that has methods as sources and targets
- DirectedRelationships had two sub-classes namely AspectJCall and JavaCall

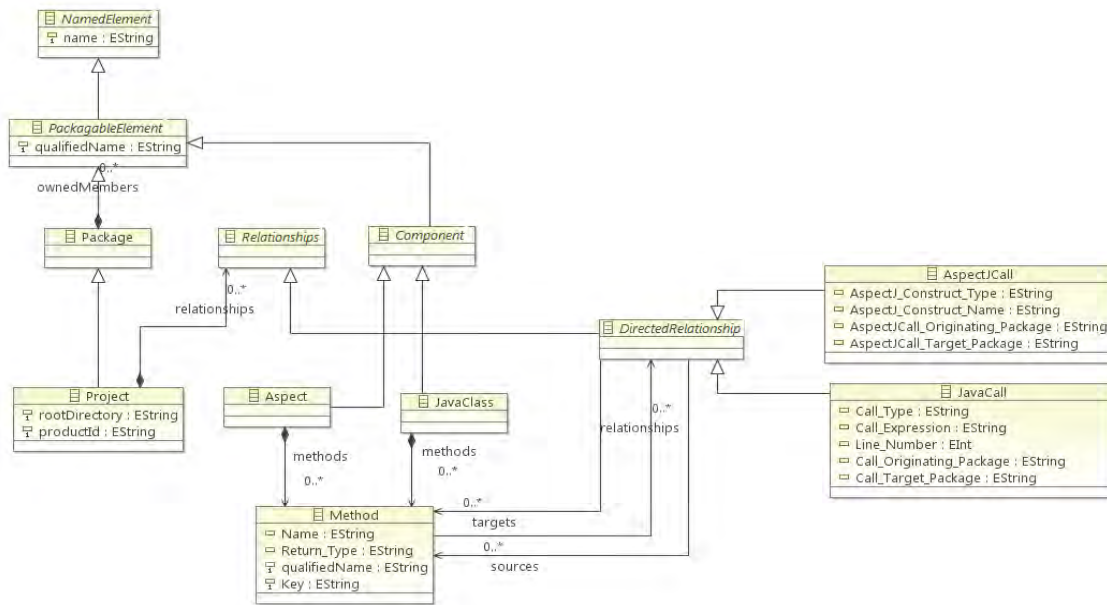


Figure 3-5 Implementation Model (IM) Meta-Model

3.6.3 Pattern Model (PM)

The PM modelling language provides a structural language to specify the AO-patterns established in various works focusing on classifying, designing the implementing the runtime feature dependencies as AspectJ-based design patterns (Lee et al., 2009a, Botterweck and Lee, 2009, Botterweck et al., 2009, Botterweck et al., 2008, Cho et al., 2008). I have used work by Lee et al., dealing with designing and modularization of RTFDs for developing the PM language. The PM language concepts are also mapped to the IM language concepts. The following sections discuss each of the specification languages designed for the AspectJ-based design patterns.

3.6.3.1 Runtime Modification Pattern

Figure 3-6 sketches the meta-model for the *Runtime Modification Pattern*. The sketched runtime modification pattern model has concepts like LeftModule, ModificationModule, RightModule and PossibleRightModule. LeftModule is composed of LeftModuleElement. ModificationModule is composed of ModificationModuleElement. RightModule is composed of RightModuleElement. It is to be noted that LeftModule can also be interpreted as a Source and RightModule can be interpreted Target. LeftModule, RightModule and ModificationModule have references to Java Packages in IM. The Module (Left, Right and Modification) elements have references (i.e., PatternRightUses, PatternAspectJCall and PatternLeftUses) to each other. The module elements have references to Java/AspectJ classes (i.e., type of Component) in IM.

- LeftModule concept represents the Source feature implementation
- RightModule concept represents the Target feature implementation
- ModificationModule concept represents the package containing the AspectJ-based design pattern
- LeftModuleElement represents the set of Java/AspectJ classes implementing the Source feature
- RightModuleElement represents the set of Java/AspectJ classes implementing the Target feature
- ModificationModuleElement represents the AspectJ pattern implementing the modification behaviour by making the calls like PatternAspectJCall (i.e., Advice, Pointcut, Inter-type declaration) to LeftModule (i.e., Source feature implementation) along with Java calls to LeftModule and RightModule (i.e., Target feature implementation)

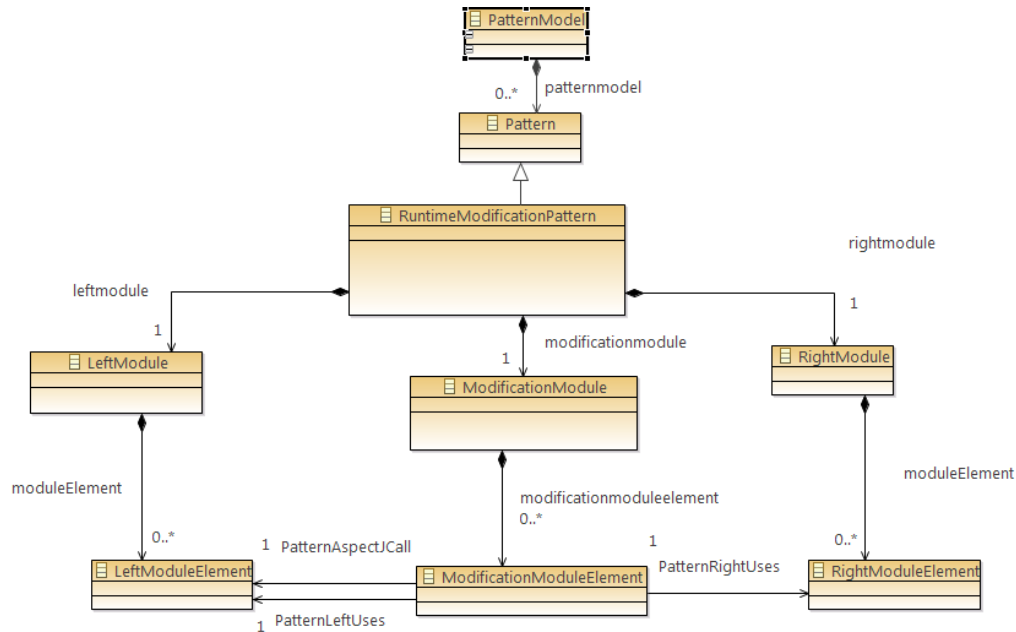


Figure 3-6 Runtime Modification Pattern Meta-Model

3.6.3.2 Runtime Required Activation

Figure 3-7 sketches the meta-model of the *Runtime Required Activation* pattern. The sketched *Runtime Required Activation* pattern is composed of the *Source*, *Target* and *ReqActivationModule* concepts. *Source* class is composed of *SourceElement*. *Target* class is composed of *TargetElement*. *ReqActivationModule* is composed of *ReqActModuleElement*. *ReqActModuleElement* consists of *InterfaceSourceVariable*, *Pointcut*, *InterfaceTerminateFunction*, *InterfaceActiveFunction* and *InterfaceTargetVariable* classes.

Source, *Target* and *ReqActivationModule* classes have a reference towards *Package* class in the IM meta-model. *InterfaceSourceVariable* has a reference towards *Component* class in the IM meta-model. *InterfaceTargetVariable* has a reference towards *Component* class in the IM meta-model. *Pointcut* class has a *Joinpoint* reference to the *Component* class in the IM meta-model. *Pointcut* also have a *PointcutFuncImplemented* property that represents if the *Joinpoint* is implemented in the code or not (i.e., Boolean type).

The *Runtime Required pattern* enables to implement the required activation behaviour between *Source* and *Target* features. It means that if *Target* feature is active then *Source* feature is required to be active. *Source* represents Source feature implementation (e.g., Feature History in Figure 3-1), *Target* represents Target feature implementation (e.g., Feature Off in Figure 3-1) and

ReqActModule represents RTFD implementation of type Runtime Required Activation.

InterfaceTerminateFunction class has a Boolean data type property *TerminateFuncImplemented* that represents if the Terminate function is implemented in the pattern implementation or not. *InterfaceIsActionFunction* has a Boolean data type property *IsActiveFuncImplemented* that represents if the *isActive* function is implemented in the pattern implementation or not.

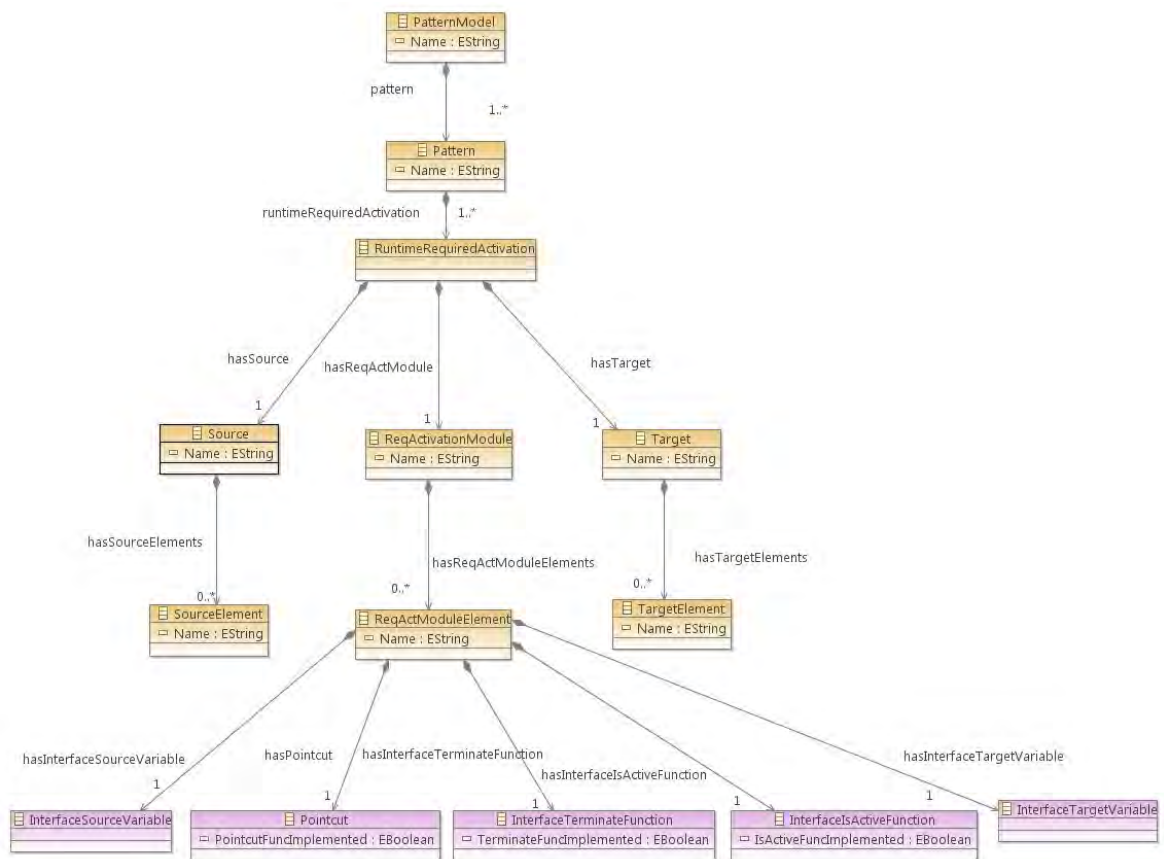


Figure 3-7 Runtime Required Activation Pattern Meta-Model

3.6.3.3 Runtime Excluded Activation

Figure 3-8 in sketches the meta-model of the *Runtime Excluded Activation* pattern. *Runtime Required Activation* and *Runtime Excluded Activation* patterns have identical concepts. However, there is a difference between *Runtime Required Activation* and *Runtime Excluded Activation* patterns. The *Runtime Excluded pattern enables* to implement the exclusion behaviour between *Source* and *Target* features. The main difference is that in *Runtime Excluded Activation* pattern the *ExclActivationModule* checks if the *Target* is running in a

runtime environment by using the *InterfaceIsActiveFunction* and uses the *InterfaceTerminateFunction* concept to terminate the Source concept.

Source, *Target* and *ExclActivationModule* classes have a reference towards *Package* class in the IM meta-model. *InterfaceSourceVariable* has a reference towards *Component* class in the IM meta-model. *InterfaceTargetVariable* has a reference towards *Component* class in the IM meta-model. *Pointcut* class has a *Joinpoint* reference to the *Component* class in the IM meta-model. *Pointcut* also have a *PointcutFuncImplemented* property that represents if the *Joinpoint* is implemented in the code or not (i.e., Boolean type).

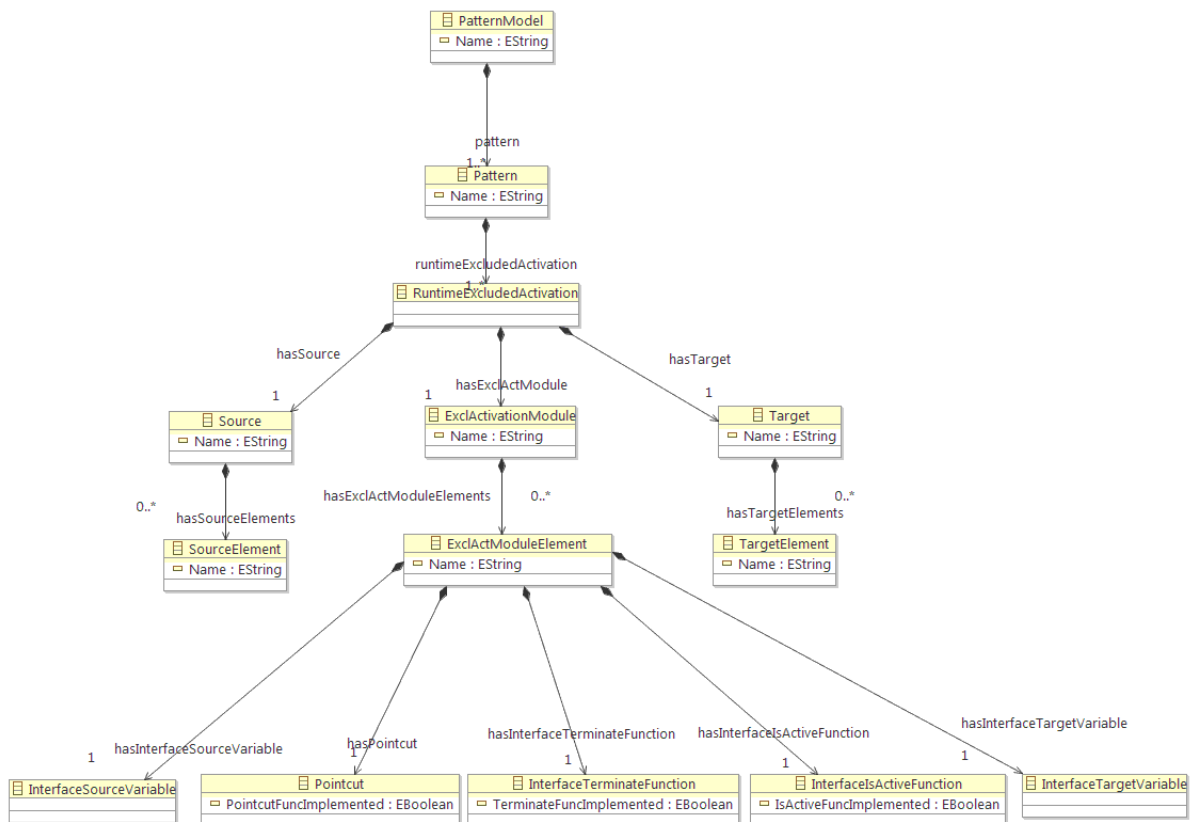


Figure 3-8 Runtime Excluded Activation Pattern Meta-Model

3.6.3.4 Runtime Concurrent Activation

Figure 3-9 sketches the meta-model of *Runtime Concurrent Activation* pattern (Lee et al., 2009a) meta-model. The sketched *Runtime Concurrent Activation* pattern is different than *Runtime Required Activation* and *Runtime Excluded Activation* patterns. Implementation of *Runtime Concurrent Activation* implements the behaviour that *Source* and *Target* features implementations should be active concurrently. The sketched *Runtime Concurrent Activation*

pattern is composed of *Source*, *ConcurrentActModule* and *Target* classes. *Source*, *Target* and *ConcurrentActModule* classes have references towards *Package* class in the IM meta-model. *ConcurrentActModule* class is composed of *ConcActModuleElement* class. *ConcActModuleElement* class is composed of *InterfaceSourceVariable*, *RunFunction* and *Pointcut* classes. *InterfaceSourceVariable* has a reference *SetSource* to *Component* class in the IM meta-model. *RunFunction* class has a Boolean property *RunFunctionImplemented*. *Pointcut* has a reference *Joinpoint* to *Component* class in the IM meta-model. *Pointcut* also has a Boolean property *PointcutImplemented*.

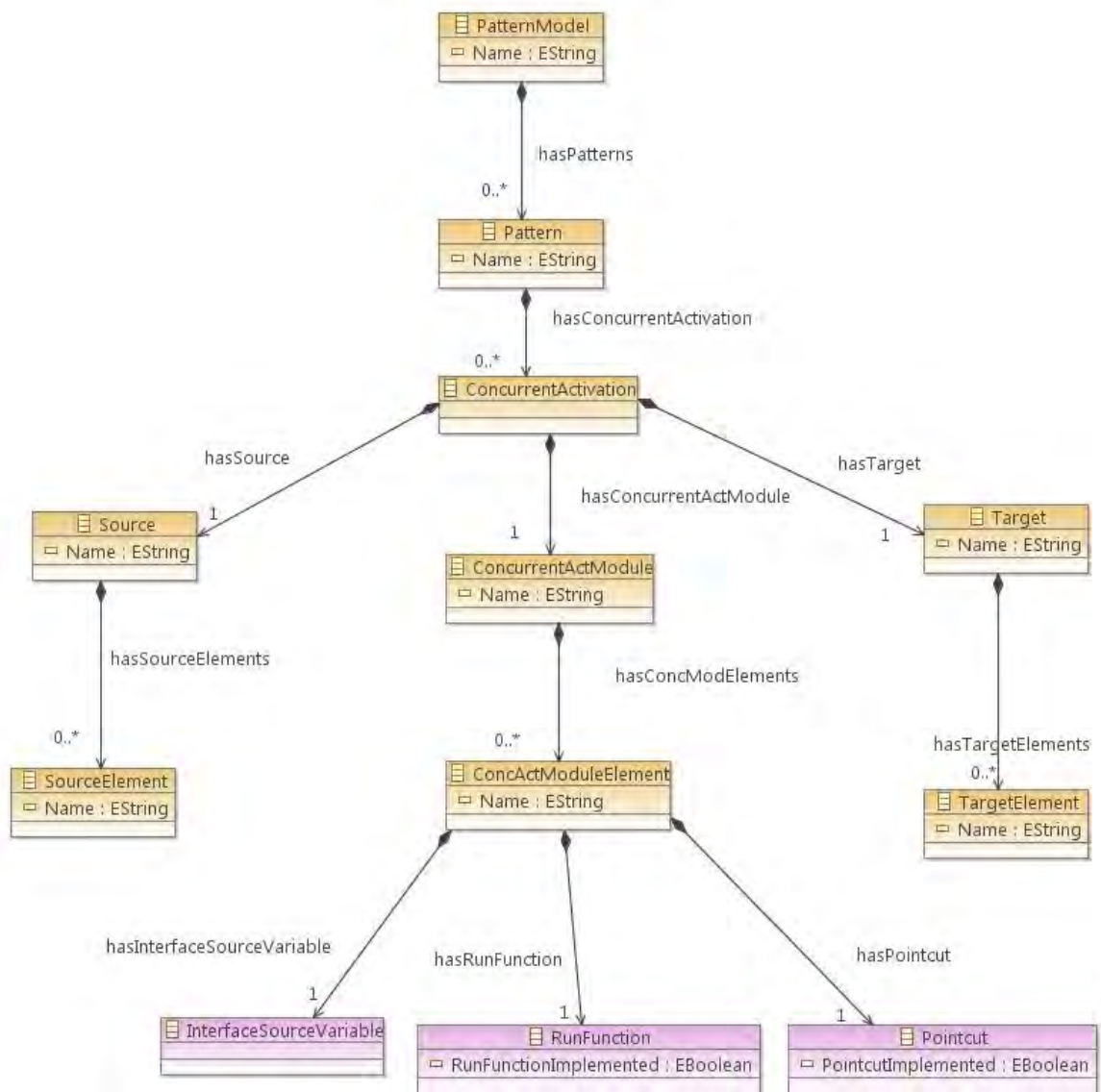


Figure 3-9 Runtime Concurrent Activation Pattern Meta-Model

3.6.3.5 Runtime Sequential Activation

Figure 3-10 sketches the meta-model of *Runtime Sequential Activation* pattern. *The Runtime Sequential Activation pattern* is different from *the Runtime Concurrent Activation pattern*. Implementation of *Runtime Sequential Activation* pattern sequentially activate the *Source* and *Target* features implementations whereas in the implementation of *Concurrent Activation* the *Source* and *Target* features implementations are activated concurrently.

Sequential Activation is composed of *the Source*, *SeqActivationModule* and *Target* classes. *Source*, *Target* and *SequentialActModule* classes have references towards *Package* class in the IM meta-model. *SequentialActModule* class is composed of *SeqActModuleElement* class. *SeqActModuleElement* class is composed of *InterfaceSourceVariable*, *ActivateFunction* and *Pointcut* classes. *InterfaceSourceVariable* class has a reference *SetSource* to *Component* class in the IM meta-model. *ActivateFunction* class has a *Boolean* property *RunFunctionImplemented* that represents if the *Run* Function is implemented or not. *Pointcut* class has a reference *Joinpoint* to *Component* class in the IM meta-model. *Pointcut* class also has a *Boolean* property *PointcutImplemented* that represents if the *Pointcut* is implemented or not.

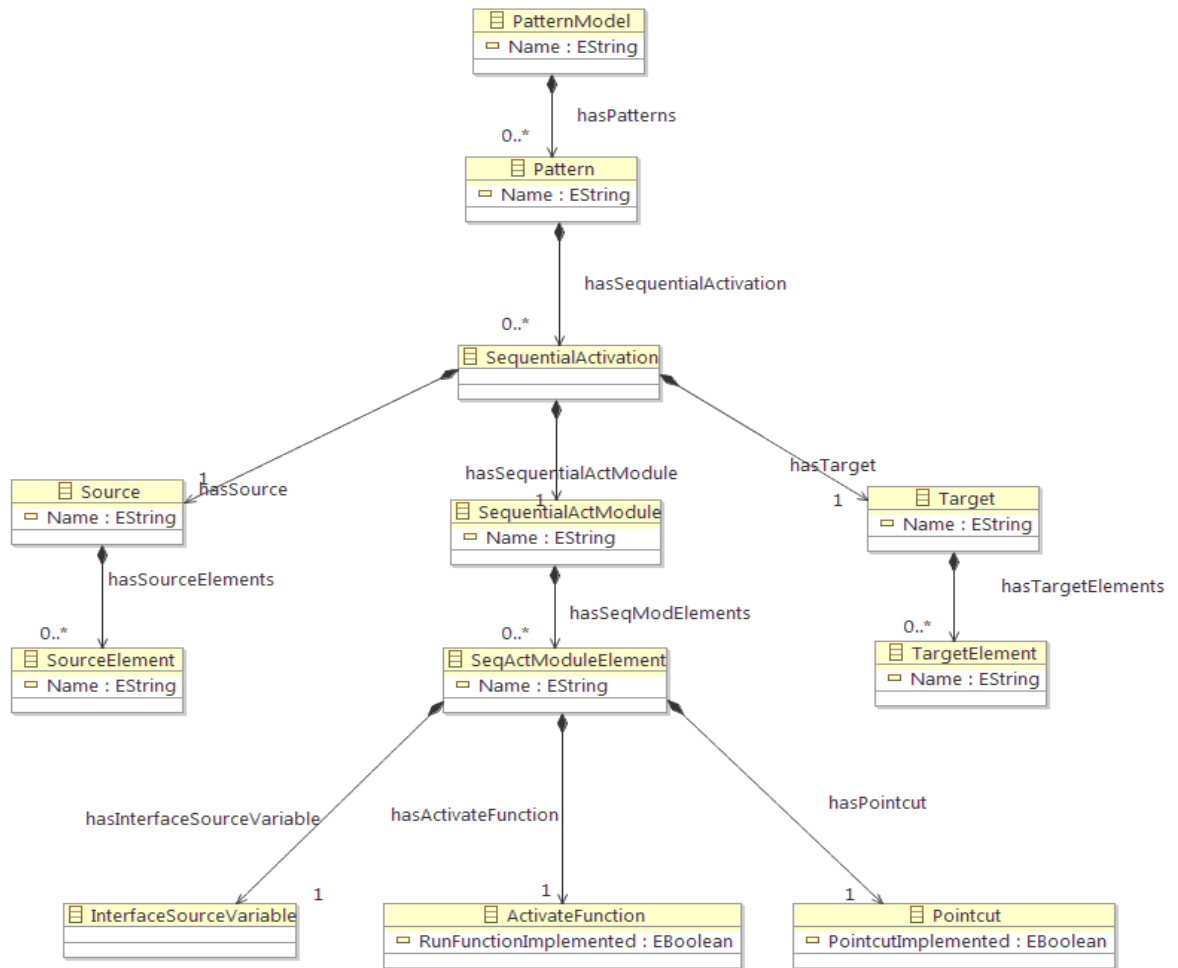


Figure 3-10 Runtime Sequential Activation Pattern Meta-Model

3.7 Process Model of the Proposed Technique

The proposed technique is composed of four main processes. The following sections will discuss in detail each of the processes in the process model (see Figure 3-11).

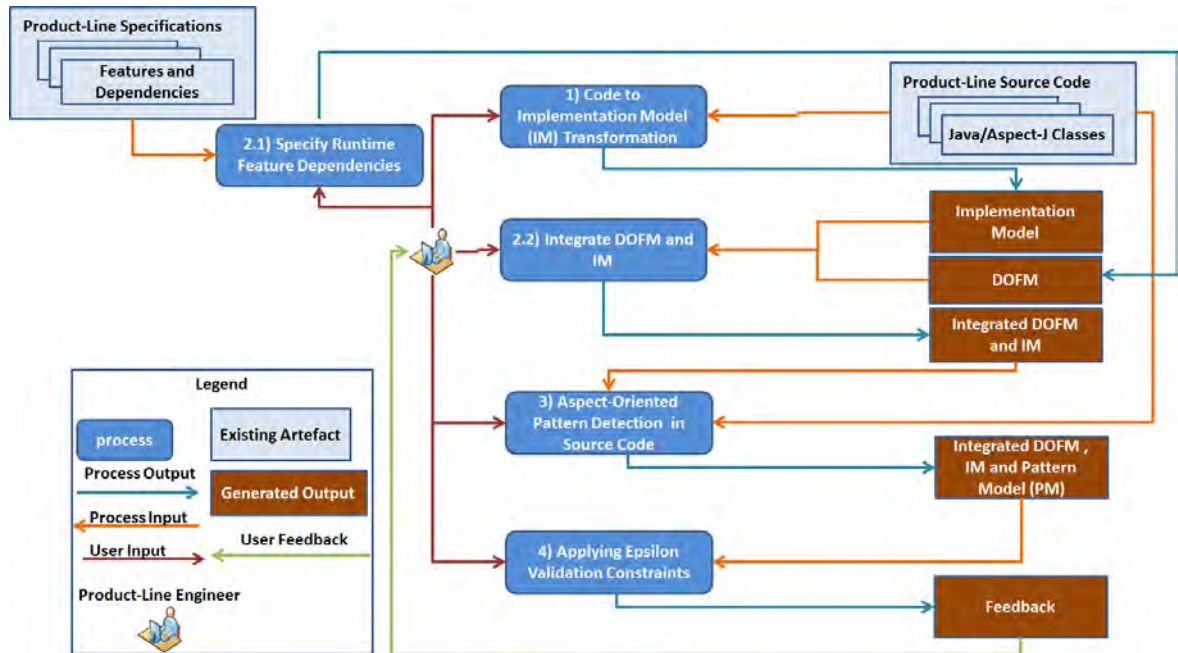


Figure 3-11 Consistency Checking Process Model

3.7.1 Extracting Source Code Concepts/Code to Model Transformation

The Code to Model (C2M) transformation process can be executed automatically by performing the source code parsing utilizing the abstract syntax tree (AST) parsers provided by the Java development tooling (JDT)³. An AST parser provides the implementation IM concepts of interest at a higher level of abstraction as a hierarchical structure to be traversed. The source code of the implementation under study is first parsed then the resulting AST is traversed and used as input to instantiate the IM meta-model as IM. The output of this process is a populated instance of the IM meta-model (Section 3.6.2).

3.7.2 Mapping Dependency-Oriented Feature Model to Implementation Model

The mapping process of the dependency-oriented feature model (DOFM) (Section 3.6.1) concepts to their respective implementations in the implementation model (IM) is performed (e.g., by naming convention) manually by the product line engineer. This process can be subdivided into two, 2.1) specify the RTFDs using the DOFM and 2.2) Integrate DOFM and IM.

³ <http://www.eclipse.org/jdt/>

Specification of RTFDs using the DOFM can be performed in two ways by the product line engineer, 1) By understanding the RTFDs specification document (if available) or 2) By understanding the existing feature model containing the RTFDs specifications (i.e., an extra view focusing only on RTFDs specification). However, the techniques of performing the extraction of RTFDs from either the specifications or feature model are not in the scope of the proposed technique.

The product line engineer manually maps DOFM concepts (i.e., features and dependency types between features) to IM concepts (i.e., Packages, Java/AspectJ classes, etc.). However these mapping techniques are not in the scope of the proposed technique. The process also takes into account the mapping knowledge while performing this process of mapping the DOFM concepts to the IM by the product line engineer. The output of the process is mapped DOFM and IM.

3.7.3 Aspect-Oriented (AO) Pattern-based Implementation Detection

The AO pattern-based implementation detection process is an automated mechanism to detect the AO-patterns established by (Lee et al., 2009a) in the source code implementing the features and the features dependencies of the product line. For the pattern detection, I have developed the AO pattern detection algorithm for each type of runtime feature dependencies. A tool implementing the pattern detection algorithms is developed for the product line engineer to apply in the source code for the detection of the AO-pattern based RTFDs. The detection process takes the source code as an input and generates the populated instance of the PM meta-model (Section 3.6.3) comprised of the detected RTFDs from the source code. The detected RTFDs concepts/parts are automatically mapped to the respective IM concepts. The output from this process is mapped DOFM, IM and PM models.

The developed algorithms perform the static code analysis using the generated AST of the source code (i.e., Java/AspectJ classes) of the product line. The developed algorithms for AO pattern search are the directed search algorithms inspired by Heuzeroth et al. (2003), that allows to detect various design patterns (e.g., observer, mediator, visitor, etc.) in the legacy code by utilizing the static and dynamic analysis and provided algorithms for the design pattern detection. Work by Heuzeroth et al. (2003) developed a static code analysis algorithm that is a directed search algorithm to compute the static design pattern candidates (e.g., observer, mediator and chain of responsibility). The designed algorithms are tree traversal algorithms with depth first search (DFS) characteristics. Work

by Heuzeroth et al. (2003) used both static analysis and dynamic source code analysis techniques to identify the design patterns in the legacy code.

In order to develop the AO pattern detection algorithm inspired by the work of Heuzeroth et al. (2003), I have utilized the same concepts of static code analysis. However, the dynamic analysis techniques for detecting the AO patterns are not in the scope of the proposed technique. For instance, a runtime required activation pattern candidate implementation will be a tuple of method declarations, assignments of interface variables and AspectJ functions (i.e., `Target.isActive()`, `Source.terminate()`, `ServiceJP()`, `Sourceinterfacevariable`, `targetinterfacevariable`). The method declarations, interface variables assignments and AspectJ functions are the concepts in order to identify the tuple the pattern detection algorithm statically analyze the parsed source code and identify the method declarations, interface variable assignments and AspectJ functions (e.g., `pointcut`, `inter-type` declarations, `before()` and `after`). In order to generate the candidates representing the AO patterns the program visits the certain parts (i.e., AspectJ classes implementing RTFDs) of the program responsible for implementing the RTFDs in source code. The time complexity of the developed algorithms is $O(n+m)$.

The static analysis algorithm computes the AO-based design pattern by visiting the generated AST nodes to identify AO-based design pattern parts and provides the result as a set of static candidates, i.e., a set of tuples of IM concepts with the appropriate static structure. The process then populates the Pattern Model (PM) with the detected AO pattern candidates. The output of this process is an integrated DOFM, IM and PM (i.e., PM contains all the detected AO patterns). The detected AO patterns are linked to the implementation concepts/nodes (e.g., IM classes, IM packages, IM methods) in the IM. Detailed discussion on implementation of the AO design pattern detection can be found in Appendix C.

3.7.4 Applying Constraints for Consistency Checking in Product Line Assets

The final process of the proposed technique is to apply off-the-shelf model checking techniques to identify inconsistency scenarios in the product line assets (i.e., Specifications of RTFDs and its respective AO-pattern based implementations). Identified AO patterns in the PM are a set of tuples of PM concepts with the appropriate static structure (i.e., output of process 3).

In order to perform the consistency checking, the process takes the integrated DOFM, IM and PM as an input. I have used Epsilon Validation Language (EVL) (Kolovos et al., 2011) constraints for model validation. EVL constraints

support the inter-model and the intra-model consistency checking, constraint dependency management and specifying fixes that users can invoke to repair identified inconsistencies. Discussion on EVL and working is made in Appendix A. The EVL constraints are designed to detect the inconsistencies discussed earlier in Section 3.5. EVL constraints enable to detect inconsistencies can be categorized in two major groups,

EVL constraints dealing with structural inconsistencies: such types of constraints deal with completeness of specification (i.e., validating DOFM and references to IM), validation the implementation of detected AO-patterns with respect to its specification

EVL constraints dealing with semantic inconsistencies: such types of constraints deal with validation of statement level behaviour with respect to the specification (i.e., DOFM) of a particular type of RTFD. Checking the program parts to prove that what program is doing what it is supposed to do. However, such types of EVL constraints are not in the focus of this research work.

The process takes interrelated DOFM, IM and PM as an input and provides feedback as a set of error markers or an error report to the product line engineer.

However, there is still an element of error with respect to the error feedback. There might be certain possibilities that contribute to element of error. A few of them are as follows,

- The mapping performed between DOFM and IM model is manual and relay on product line engineer knowledge. Hence, the wrong mapping can lead to unintended error markers.
- The tool identifies multiple instances of the same AO patterns in the source code leading to redundancy issues.
- The prototype tool identifies different types of AO patterns for the same pair. For instance, FeatureA—RequiredAct⁴→FeatureB and FeatureA—RequiredExcl⁵→FeatureB.

3.8 Proposed Technique Prerequisites

The following are the pre-requisites for the proposed technique discussed in Section 3.7.

⁴ Runtime Required Activation

⁵ Runtime Excluded Activation

- The requirements engineer is provided with the DOFM modelling language to specify the features and the runtime feature dependencies.
- The DOFM modelling language instance specifying features and RTFDs of the particular product line is provided by the requirements engineer.
- The mapping knowledge for integrating the DOFM and the IM is provided to the product line engineer

3.9 Limitations

The limitations of the proposed approach can be classified as 1) functional limitations and 2) process limitations.

Functional limitations: functional limitations represent a situation, where the proposed approach functionality is limited. Following are the functional limitations

- The proposed technique works in the presence of the AO-patterns implementations in the source code.
- Certain situations/scenarios are not identified contributing towards false negative (i.e., condition not fulfilled but in reality it works). Implemented feature dependency works fine without implementing it using the AO-pattern. In such a situation, my approach will fail to identify that the situation is fine and will raise the false negative alarm.
- The AO-patterns detection is performed within particular packages that contain the AO pattern-based implementations of runtime feature dependencies. Hence, if there is the presence of an implementation that works perfectly fine but not having an AO pattern-based solution in such situation the tool support will fail to identify that there exists an implementation of a particular runtime feature dependency not following a particular AO pattern?

Process Limitations: process limitations represent a situation, where a particular process of a proposed approach depend on or designed with based on a certain data/information. The following are the limitations related to the processes of the proposed technique,

- The AO pattern-based pattern detection algorithms are implemented with the static code analysis process taken into consideration.
- The AO pattern-based design patterns acted as an input for designing the AO pattern detection algorithms.
- The DOFM modelling language does not replace the feature model, which is still required to express the structural and configuration constraints/dependencies between features.

- More formalism is required to specify the RTFDs in DOFM language
- The identified inconsistencies by applying the EVL constraints serve as a set of guidelines to identify the inconsistencies in a particular product line, but it doesn't provide the different solution options to resolve the identified inconsistency related to RTFDs in the product line artefacts (e.g., feature model or in implementation).

3.10 Discussion

There are various works discussed in chapter two that attempted to analyze implementations with respect to their specifications in a product line setup. For instance works (Kim et al., 2008, Apel et al., 2011, Batory et al., 2011, Mosser et al., 2012) focused on the detection of configuration dependencies in the product line and considered dependencies that are not modularized and implemented implicitly in the source code.

Work by (Lee et al., 2009a) provided guidelines to the developers to modularize RTFDs using AspectJ-based design patterns. The AO pattern implemented in a source code provides a modular solution for implementing RTFD. The proposed approach for consistency checking of runtime feature dependencies in product line assets provides the RTE functionality. RTE is followed because the same information (RTFD) is present in multiple product line assets (i.e., specification and implementation), therefore an inconsistency may occur if all the product line assets are not updated consistently to reflect the change. For the proposed technique to work, the developers are advised to follow the guidelines in a form of AO patterns by the work (Lee et al., 2009a).

The proposed technique enables the product line engineer to query for the inconsistency between the AO pattern-based implementation and the corresponding specification in the DOFM. The proposed approach is focusing on a special case of software verification focusing on proof of correctness/program verification (Collofello and Institute, 1988).

In the absence of the proposed approach the product line engineer might have to manually identify the AO pattern-based design patterns in the source code and then later have to manually analyse the specification and the identified AO pattern-based implementation. Manually performing the consistency checking is error prone and time consuming at the same time. Hence, the proposed solution will enable the product line engineer to semi-automatically perform the maintenance task in an effective manner.

The error/warning based feedback provides a list of the problems related to AO pattern-based implementations of runtime feature dependencies in the source

code and their respective specifications. The feedback provides the basis for manual or automatic analysis in order to achieve the correct end product behaviour specified using the DOFM and implemented as intended using the AO pattern-based solution in the source code.

The approach can be interpreted as a static verification challenge focusing on static analysis of the source code. The static analysis of the source code enables the product line engineer to perform structural analysis of implemented AO-design patterns with respect to its specification using the EVL constraints. Figure 3-12 represents an overview of the approach.

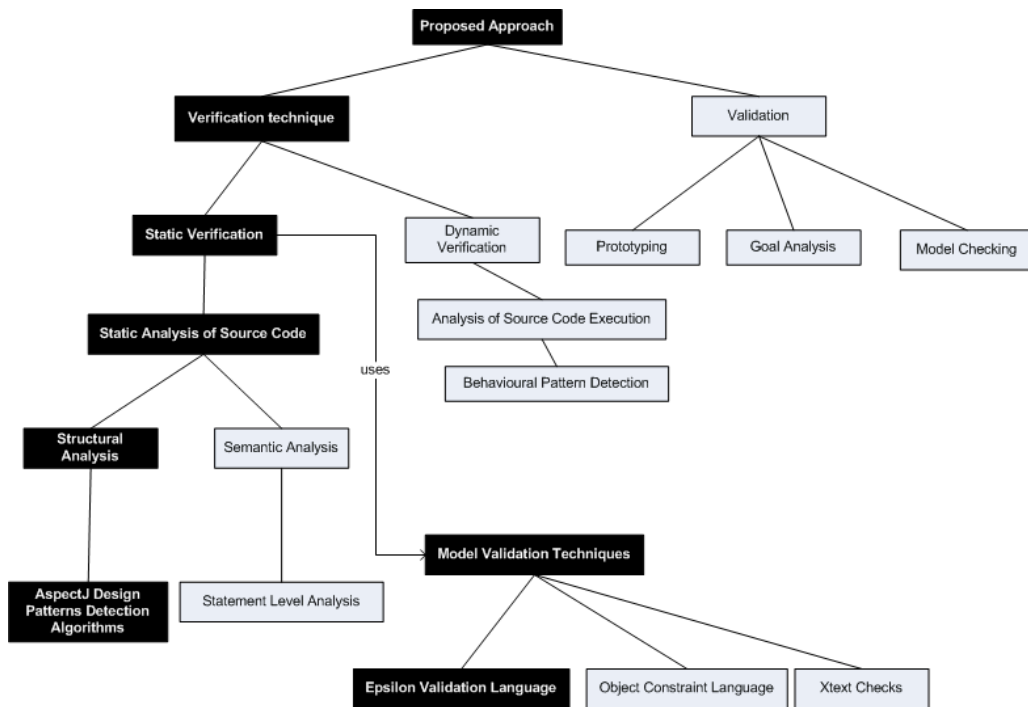


Figure 3-12 Consistency Checking Process Model

The mapping process of the DOFM to IM assumes that there exists the mapping knowledge between the product line features and their respective implementations as well as between the runtime feature dependencies and their respective pattern-based implementations. If certain information is not present to carry out the integration process, then the product line engineer can consult the developer in order to find out which features and their dependencies are implemented and can carry out the integration process.

The developed algorithms perform the static code analysis using the generated AST of the source code (i.e., Java/AspectJ classes) of the product line. The developed algorithms for the detection of AO patterns is the product line

implementation (i.e., AspectJ/Java source code) are inspired by and extension of work by (Heuzeroth et al., 2003). There are approaches like (Birkner, 2007, Collofello and Institute, 1988, Seemann and Gudenberg, 1998, Cichý and Jakubík, 2008, Wendehals and Orso, 2006) in literature that can be used for general pattern detection in Java-based implementation. The proposed approach takes into account RTFDs that are implemented as separate modules using the proposed AO patterns in a product line setup. This limits the scope of the proposed approach because the proposed technique is taking into account the modularized implementations of the specified runtime feature dependencies in the source code.

3.11 Summary

This chapter focuses on providing an overview of the proposed approach. The proposed approach provides a round trip engineering (RTE) in order to check for consistency between the various software artefacts (e.g., specifications and source code) containing the same information (e.g., feature dependencies). The proposed technique performs static verification of AspectJ pattern-based implementations with respect to their specifications and detects structural inconsistencies early in domain engineering phase of a product line to avoid errors during execution of the end product. The meta-modelling languages are discussed in detail. The prerequisites and the assumptions of the approach were also discussed. The proposed technique uses the pre-existing techniques and frameworks for its conceptual framework development. This chapter discussed the stakeholders involved during the development and the deployment of the proposed technique. Limitations of the proposed technique were also discussed.

Chapter Four: Implementation Concepts and Prototype Support

4.1 Overview

A conceptual framework was laid down in Chapter three along with a detailed discussion on the proposed technique for consistency checking of runtime feature dependencies in product line assets. This chapter provides a transition from the conceptual framework to the developed research prototype. The research prototype acts as a proof of concept for the proposed technique and it was implemented by using off the shelf domain modelling techniques, AO-based pattern detection algorithms and the model validation technique. A thorough discussion about the implementation of the processes within the process model is made. The chapter concludes with a discussion and the limitations.

4.2 Consistency Checking Technique - A Conceptual Realization

The proposed technique was implemented as a part manual and a part automatic. For a proof of concept a research prototype is developed. The following sections discuss the conceptual realization of the proposed technique discussed in Chapter 3 Section 3.7.

4.2.1 Code to Model Transformation and AO-Pattern Detection Plug-in

In order to automate the IM generation and AO-pattern detection processes of the proposed technique (Process 1 in Section 3.7), an Eclipse-based plug-in was developed. The developed plug-in is applied on the AspectJ/Java source code. Figure 4-1 represents an instance of the detected runtime required activation (RTRA) pattern and references to the respective IM parts (i.e., Packages and Java/AspectJ classes). This process realizes two steps of the proposed technique (i.e., (1) Code to IM transformation and (3) Aspect-oriented Pattern Detection in source code) represented in Figure 3-11. The implementation model (B) and the detected RTRA (C) in the PM is generated as a result of applying the Code2Model Plugin.

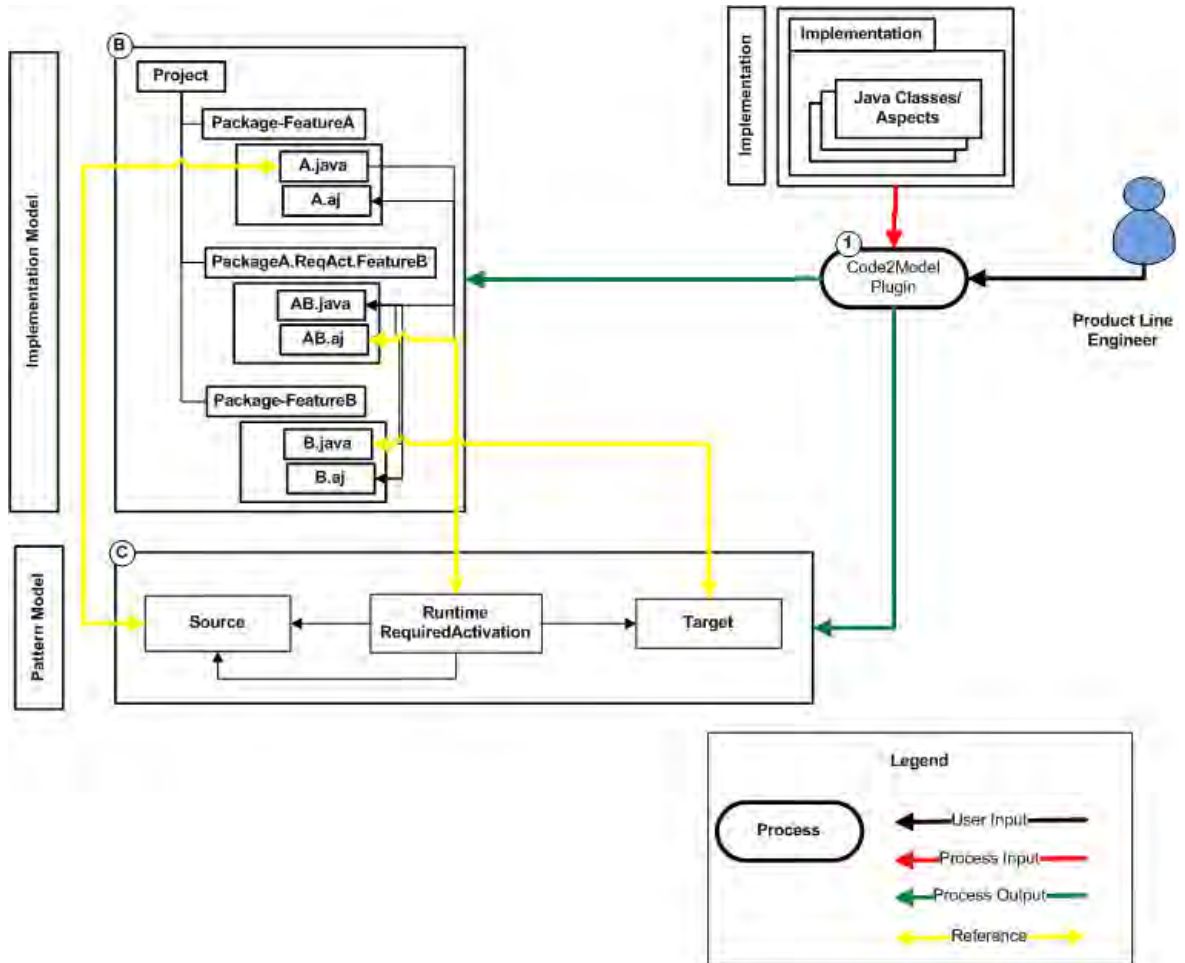


Figure 4-1 Code to IM Transformation and AO Pattern Detection

4.2.2 Mapping the DOFM and the IM

This realized process (i.e., Process 2 discussed in the Section 3.7.2) of the proposed technique described in Figure 3-11 Consistency Checking Process Model is manually performed by the product line engineer. This process can be subdivided into two processes, namely, 2.1) Specify RTFDs using the DOFM and 2.2) integrate the DOFM and the IM (i.e., Figure 4-2, excluding the PM and references to the IM in order to focus only on the integration process between the DOFM and the IM). The specification of RTFDs sub-process can be performed by the product line engineer in various ways 1) by utilizing the specification document or 2) the feature model representing the RTFDs. If the product line engineer utilizes the feature model for developing the DOFM? In this situation the DOFM represents an explicit view that specifies only the RTFDs. The integration of the DOFM (A) specifying RTRA dependency and the IM (B) requires the features and the RTDA to be mapped to the respective

implementation packages. The techniques for the specification of RTFDs and the integration of DOFM and IM are not in the scope of this research work.

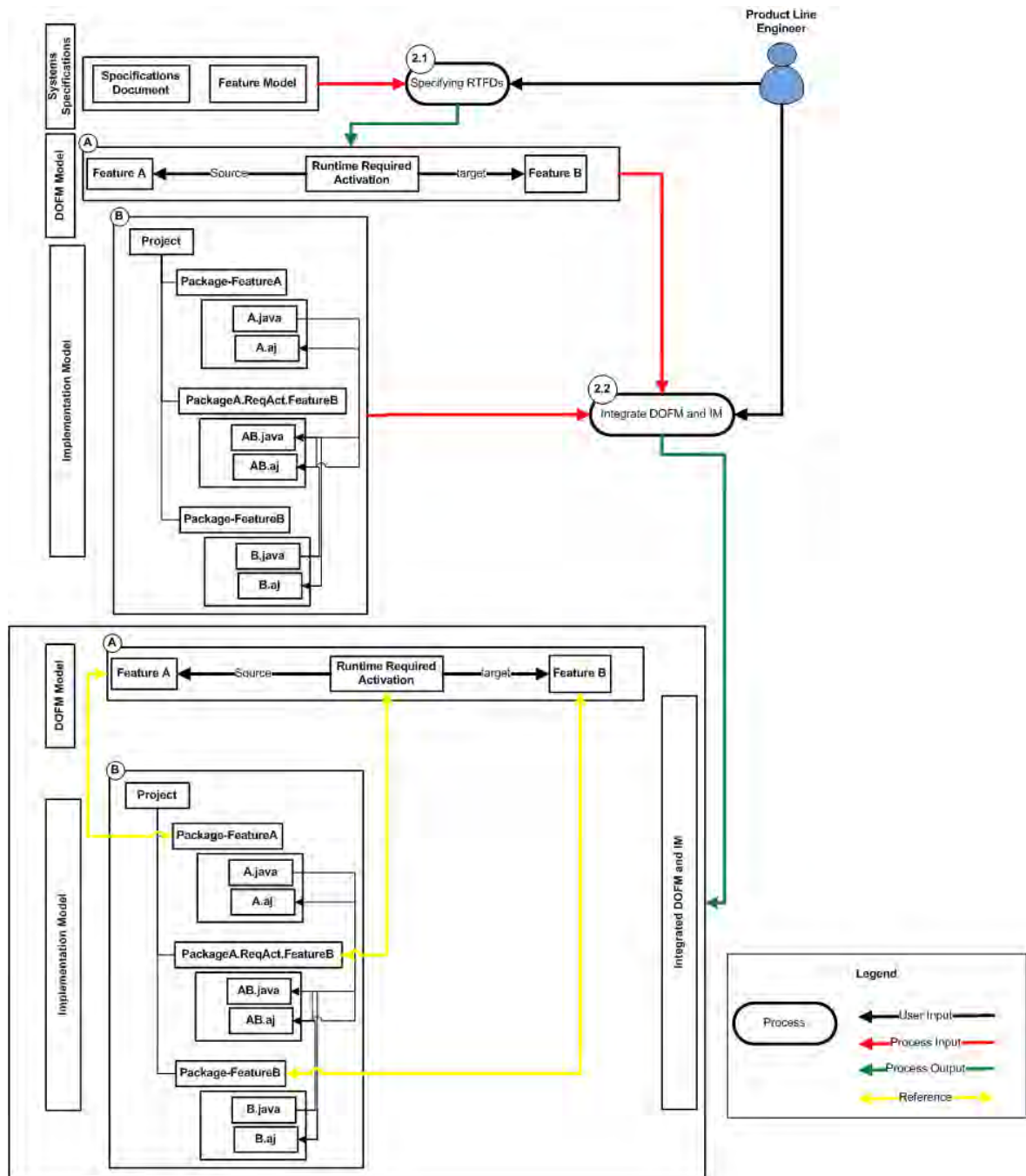


Figure 4-2 Specification of RTFDs and Integration of the DOFM and the IM

4.2.3 Applying the EVL Constraints for the Consistency Checking

In order to verify the DOFM specifications against the detected AO patterns to provide inconsistency feedback to the product line engineer, a verification/validation language is required. Various languages are available for model checking/validation (e.g., object constraint language (Warner and Kleppe, 2003), check language with Xtext framework (Behrens et al., 2010) and EVL (Kolovos et al., 2012b)) by applying constraints on the model elements.

After the generation of the interrelated models, 1) the DOFM, 2) the IM and 3) the IM as an output of the previous process (i.e., Section 4.2.2). The next step is to apply the EVL constraints on the models (Figure 4-3). The process takes the models as an input and generates the error markers to indicate the inconsistencies between the specifications of the RTFDs (i.e., DOFM) and the detected AO pattern-based implementations (i.e., PM). This is an automated process realizing the process 4 (i.e., Section 3.7.4) applied by the product line engineer. The EVL constraints are applied in two stages. Figure 4-3 represents an example of applying the EVL constraints on the RTRA type of RTFDs.

Stage one constraints: Structural constraints are responsible for checking

- If the DOFM specification is complete? And references to the respective IM concepts are set?

Stage two constraints: Once structural constraints are satisfied, stage two constraints are inconsistency constraints that are responsible for validating the direction and type of the RTFD implemented.

Stage two constraints are applied to identify the inconsistency scenarios discussed in Section 3.5.

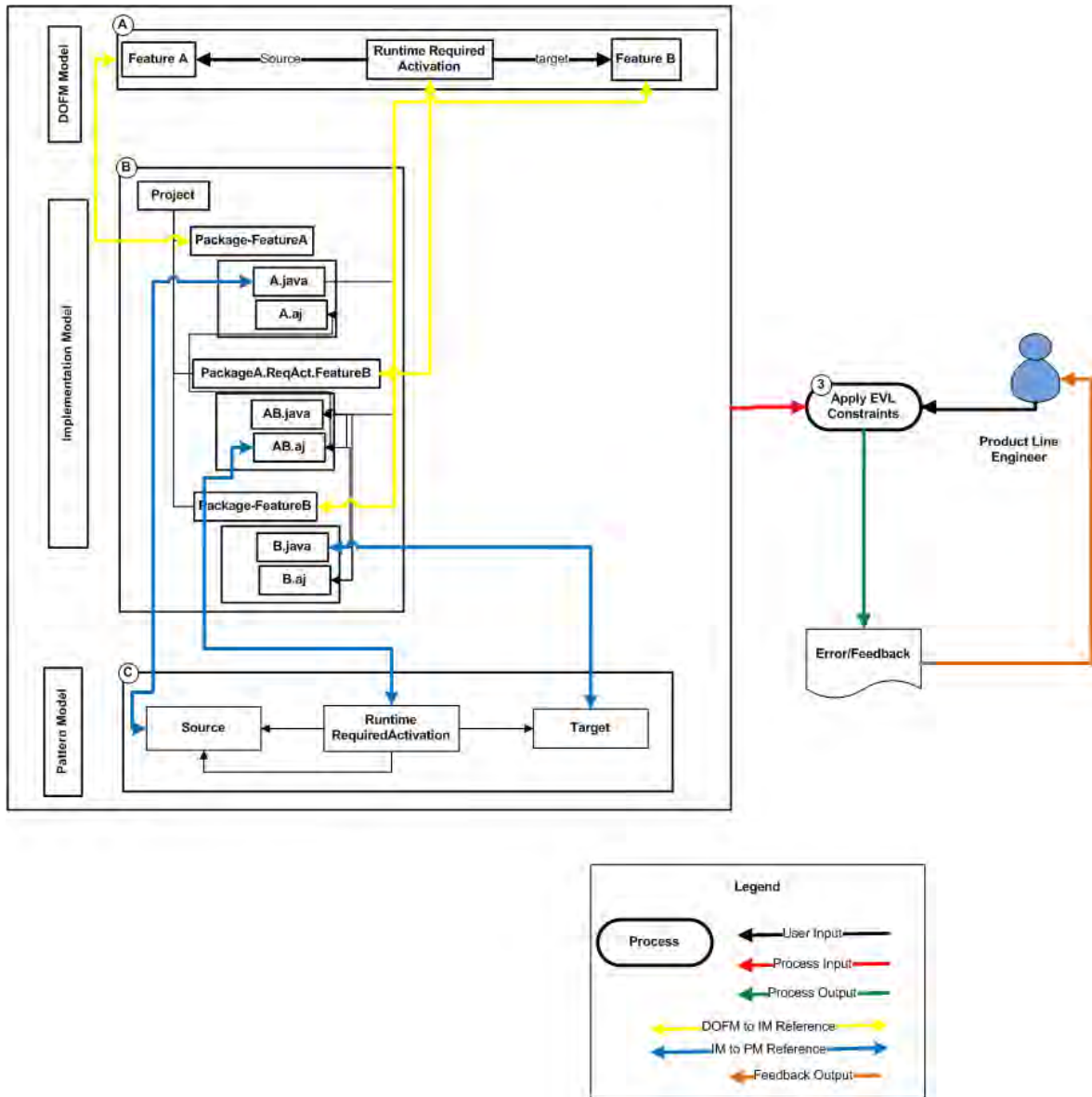


Figure 4-3 Applying the Developed EVL Constraints

4.3 Aspect-oriented Pattern Detection Process – An Implementation Overview

This section discusses the IM generation and AO pattern-based detection in detail. As discussed in Section 4.3.1, the AO pattern detection process was implemented as Code2Model plug-in. Figure 4-4 represents processes in detail for Code2Model plug-in. The process starts by loading the Java/AspectJ project (Process 1) that implements the product line features and the AO pattern-based RTFDs. The loaded project is then input to the source code parser (Process 2). There are two sub-processes of the Process 2 running concurrently, one parsing

the source code for the structural information (sub-process 2.1) and another for parsing the source code to identify the Java/AspectJ implementation concepts (e.g., method declaration, assignment and parameterized type) (sub-process 2.2). The output of the sub-process 2.1 is fed as an input to Process 3 that also takes an IM meta-model as an input and generates an instance of the IM. The output of sub-processes 2.1 and 2.2 is further fed into Process 4 and performs lower level analysis. The lower-level analysis takes into account the structural information and detect the AO pattern parts by parsing the source code (e.g., Source, Target, isActiveFunction(), Pointcut, TerminateFunction(), etc.). Once the any particular type of AO pattern is detected, the identified AO pattern is fed as input to the process (Process 5) that also takes Pattern Model (PM) as input and generates an instance based on the information about the detected AO pattern. It is to be noted here that, the same concepts (i.e., Packages, Java/AspectJ classes) have been used in Process 3 and Process 4 that automatically provides references from the PM parts to the IM.

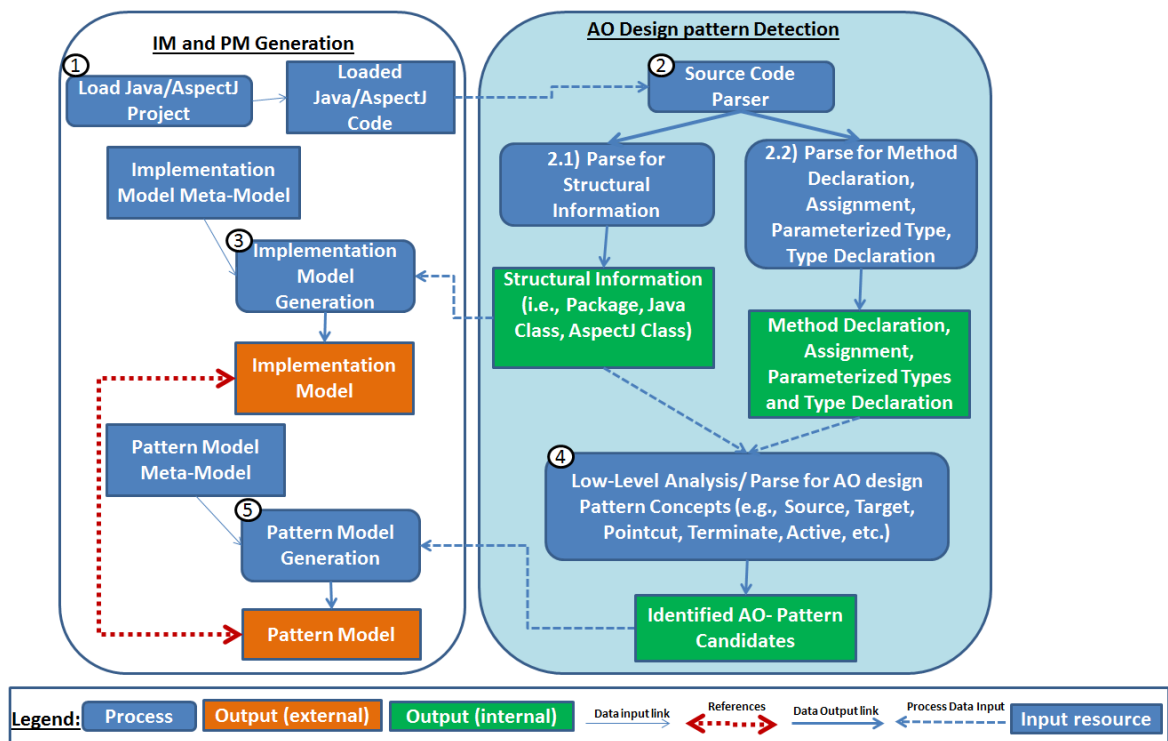


Figure 4-4 Code to IM Generation and AO Pattern Detection Process

Figure 4-5 represents a class diagram of the Code2Model plug-in.

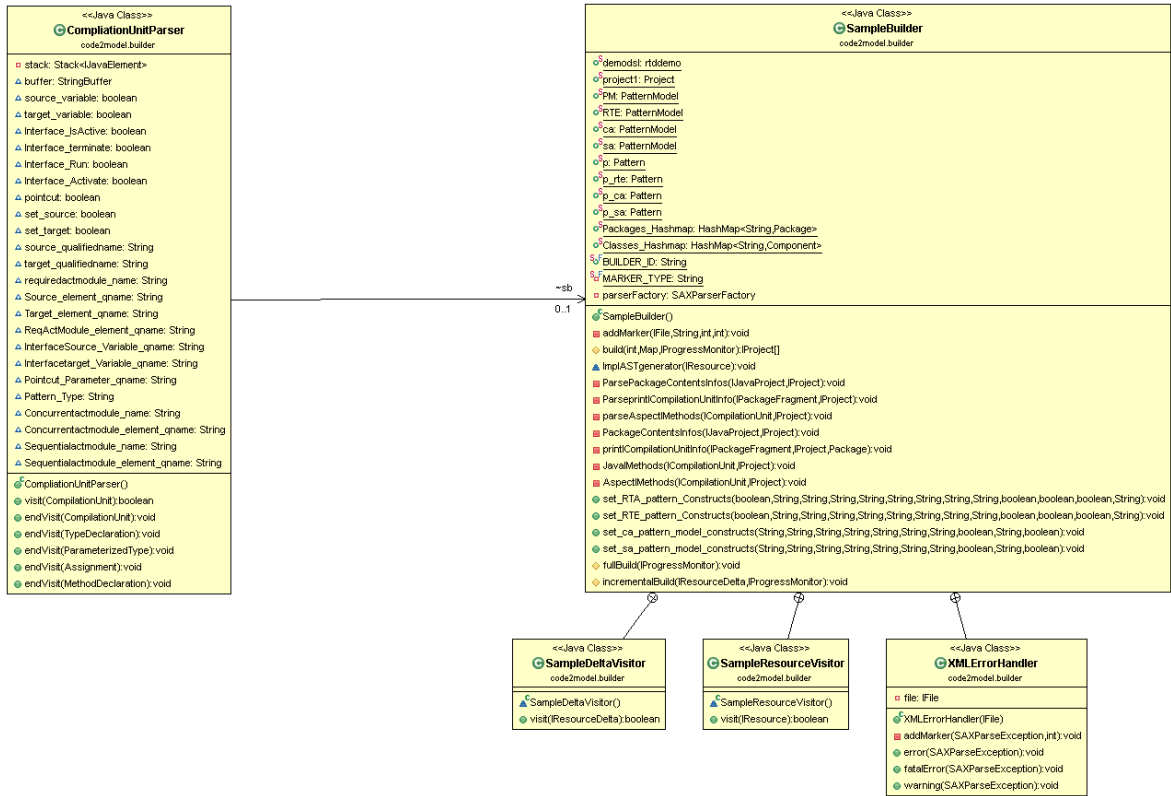


Figure 4-5 Class Diagram of the Code2Model Plug-in

4.4 Aspect-Oriented Pattern Detection Algorithms (Pseudo-code)

The following sections will discuss the aspect-oriented patterns detection algorithms based on the work conducted by (Lee et al., 2009a). As discussed earlier in Section 3.7.3, the AO-pattern detection algorithms are inspired by Heuzeroth et al. (2003) on automatic design pattern detections by utilizing static and dynamic source code analysis in legacy systems. However, the designed algorithms are inspired by the static analysis of the source code in Heuzeroth et al. (2003). The pattern detection algorithms can be classified as tree traversal algorithms with depth first search (DFS) characteristics. The algorithms perform static code analysis of the source code for identifying the AO pattern parts (e.g., *Source*, *Target*, *ReqActModule*) and populate the PM instance. The Java implementation of the designed algorithms is as an Eclipse plug-in (Appendix C)

4.4.1 Runtime Modification Dependency Pattern Detection Algorithm

Step 1: Instantiate empty Runtime Modification Dependency (RMD) pattern instance

Step 2: Populate the RMD pattern instance

For each class C in RMD package

Set ModificationModule= C

For each method m in C do

Set ModificationModuleElement= m

If m is an aspect-oriented method do\ (e.g., After(..), Before(..), Around(..))

Calculate the advises call (AspectJ call)

Set ModificationModuleElement.AdvisesCall= AdvisesCallTarget

Set LeftModuleUses= AdvisesCallTarget class

End if

Within the body of method m

\Calculate the method calls (e.g., AspectJ or Java function calls)

If calculated method call is AspectJ call

Set RightModuleUses= AspectJ call Target class

Else if calculated method call is JavaCall

Set RightModuleUses= JavaCall Target class

End if

End if

End For

Step 3: Save the RMD pattern instance

4.4.2 Runtime Required Activation Pattern Detection Algorithm

Step 1: Instantiate an empty Runtime Activation (RTA) Pattern

Step 2: Populate the empty RTA pattern

For each class C in RTA package do

Calculate the Super-Class of C for set Parameter

Calculate the reference to First Parameterized module type in Super class and identify respective IM package

Set **Source**= Identified IM package

Calculate the reference to Second Parameterized module type in Super class and identify respective IM package

Set **Target**= Identified IM package

Calculate the reference to First Parameterized module type in Super class and identify respective IM Component (i.e., Java/AspectJ)

Set **SourceElement**= Identified IM Component

Calculate the reference to Second Parameterized module type in Super class and identify respective IM Component (i.e., Java/AspectJ)

Set **TargetElement**= Identified IM Component

Calculate the reference to Package in which C is present and find respective IM Package

Set **ReqActivationModule** = IM Package concept in which C is present (i.e., rtareqactmodref= Package)

Find C in respective IM Package (Java/AspectJ class in IM)

Set **ReqActModuleElement**= Identified Java/AspectJ concept in IM (i.e., rtareqactmodelementref=Component)

Check for Method Declaration of m

Start Function

For each method m in C

Calculate methodDeclaration of each m in C

```

if methodDeclaration == ".isActive()"
    set interfaceIsActiveFunction_Found=true;
else if methodDeclaration==" .terminate()"
    set interfaceTerminateFunction_Found=true;
end if

```

Calculate the Returntype of each m in C

```

if returntype of m == "pointcut"
    set pointcut_found=true;
    Calculate the pointcut method parameters
    set joinpoint= calculated pointcut parameter (i.e.,
        joinpoint=Component)

```

End Function

Check for Assignment statements within the body of m

Start Function

```

For each method m in C
if “source” interface variable is set then
    InterfaceSourceVariable.sourcevariable= Component
else if “target” interface variable is set then
    InterfaceTargetVariable.TargetVariable=Component
end if

```

End Function

Step 3: Save the populated RTA pattern instance

4.4.3 Runtime Excluded Activation Pattern Detection Algorithm

Step 1: Instantiate an empty Runtime Excluded Activation (REA) Pattern

Step 2: Populate the empty REA pattern

For each class C in REA package do

Calculate the Super-Class of C for set Parameter

Calculate the reference to First Parameterized module type in Super class and identify respective IM package

Set **Source**= Identified IM package

Calculate the reference to Second Parameterized module type in Super class and identify respective IM package

Set **Target**= Identified IM package

Calculate the reference to First Parameterized module type in Super class and identify respective IM Component (i.e., Java/AspectJ)

Set **SourceElement**= Identified IM Component

Calculate the reference to Second Parameterized module type in Super class and identify respective IM Component (i.e., Java/AspectJ)

Set **TargetElement**= Identified IM Component

Calculate the reference to Package in which C is present and find respective IM Package

Set **ExclActivationModule** = IM Package concept in which C is present

Find C in respective IM Package (Java/AspectJ class in IM)

Set **ExclActModuleElement**= Identified Java/AspectJ concept in IM

Check for Method Declaration of m

Start Function

For each method m in C

Calculate methodDeclaration of each m in C

if methodDeclaration == ".isActive()"

```

        set interfaceIsActiveFunction_Found=true;
    else if methodDeclation=="terminate()"
        set interfaceTerminateFunction_Found=true;
    end if
Calculate the Returntype of each m in C
    if returntype of m == "pointcut"
        set pointcut_found=true;
        Calculate the pointcut method parameters
            set joinpoint= calculated pointcut parameter (i.e.,
                joinpoint=Component)

```

End Function

Check for Assignment statements within the body of m

Start Function

```

    For each method m in C
    if "source" interface variable is set then
        InterfaceSourceVariable.sourcevariable= Component
    else if "target" interface variable is set then
        InterfaceTargetVariable.TargetVariable=Component
    end if

```

End Function

Step 3: Save the populated REA pattern instance

4.4.4 Runtime Sequential Activation Pattern Detection Algorithm

Step 1: Instantiate an empty Runtime Sequential Activation (RSA) Pattern

Step 2: populate the RSA pattern

For each class C in RSA package do

Calculate the Super-Class of C for set Parameter

Calculate the reference to First Parameterized module type in Super class and identify respective IM package

Set **Source**= Identified IM package

Calculate the reference to Second Parameterized module type in Super class and identify respective IM package

Set **Target**= Identified IM package

Calculate the reference to First Parameterized module type in Super class and identify respective IM Component (i.e., Java/AspectJ)

Set **SourceElement**= Identified IM Component

Calculate the reference to Second Parameterized module type in Super class and identify respective IM Component (i.e., Java/AspectJ)

Set **TargetElement**= Identified IM Component

Calculate the reference to Package in which C is present and find respective IM Package

Set **SequentialActModule** = IM Package concept in which C is present

Find C in respective IM Package (Java/AspectJ class in IM)

Set **SeqActModuleElement**= Identified Java/AspectJ concept in IM

Check for Method Declaration of m

Start Function

For each method m in C

Calculate methodDeclaration of each m in C

if methodDeclaration == ".activate()"

```
        set RunFunctionImplemented_Found=true;
    end if
```

Calculate the Returntype of each m in C

```
    if returntype of m == "pointcut"
```

```
        set pointcut_found=true;
```

Calculate the pointcut method parameters

```
        set joinpoint= calculated pointcut parameter (i.e.,
            joinpoint=Component)
```

End Function

Check for Assignment statements within the body of m

Start Function

For each method m in C

if “source” interface variable is set then

```
    InterfaceSourceVariable.sourcevariable= Component
```

end if

End Function

Step 3: Save the populated RSA pattern instance

4.4.5 Runtime Concurrent Activation Detection Algorithm

Step 1: Instantiate an empty Runtime Concurrent Activation (RCA) Pattern

Step 2: populate the RCA pattern

Calculate the Super-Class of C for set Parameter

Calculate the reference to First Parameterized module type in
Super class and identify respective IM package

Set **Source**= Identified IM package

Calculate the reference to Second Parameterized module type in
Super class and identify respective IM package

Set **Target**= Identified IM package

Calculate the reference to First Parameterized module type in Super class and identify respective IM Component (i.e., Java/AspectJ)

Set **SourceElement**= Identified IM Component

Calculate the reference to Second Parameterized module type in Super class and identify respective IM Component (i.e., Java/AspectJ)

Set **TargetElement**= Identified IM Component

Calculate the reference to Package in which C is present and find respective IM Package

Set **ConcurrentActModule** = IM Package concept in which C is present

Find C in respective IM Package (Java/AspectJ class in IM)

Set **ConcActModuleElement**= Identified Java/AspectJ concept in IM

Check for Method Declaration of m

Start Function

For each method m in C

Calculate methodDeclaration of each m in C

if methodDeclaration == ".run()"

set **RunFunctionImplemented_Found**=true;

end if

Calculate the Returntype of each m in C

if returntype of m == "pointcut"

set **pointcut_found**=true;

Calculate the pointcut method parameters

set **joinpoint**= calculated pointcut parameter (i.e.,
joinpoint=Component)

End Function

Check for Assignment statements within the body of m

Start Function

For each method m in C

if “source” interface variable is set then

InterfaceSourceVariable.sourcevariable= Component

end if

End Function

Step 3: Save the populated RCA pattern instance

4.5 Tool Support

The proposed technique is realized as a part manual and a part automatic. To automate the processes of the proposed technique a prototype is implemented in Eclipse development environment (Gronback, 2009). The following sections will give a brief overview of the prototype provided for the proof of concept.

4.5.1 Code to Model Transformation and AO-Pattern Detection Plug-in

In order to automate the code to model transformation and AO-pattern detection processes of the proposed technique, an Eclipse-based plug-in is developed. The developed plug-in is applied on the Java/AspectJ implementation. After applying the plug-in, the IM meta-model and the PM meta-model instances are created. Figure 4-6 represents applying the Code2ImplModel and AOP-Pattern detection transformation plug-in on an example Java/AspectJ implementation project. The developed plug-in realizes processes (1) and (3) in Figure 3-11. Appendix C provides an implementation source code for the developed code2model plug-in.

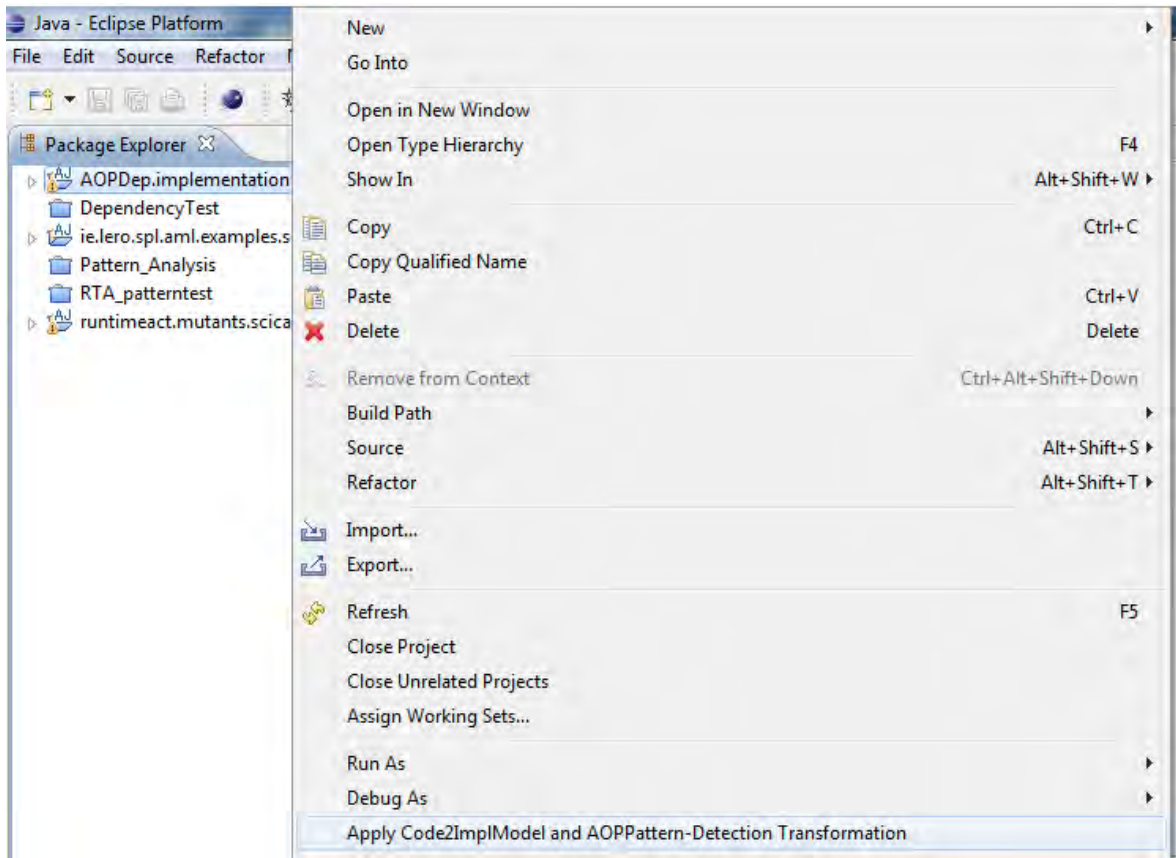


Figure 4-6 Code2IM and AOP-Pattern Detection Transformation Plug-in

Figure 4-7 shows the generated instances of IM and Pattern Model meta-models.

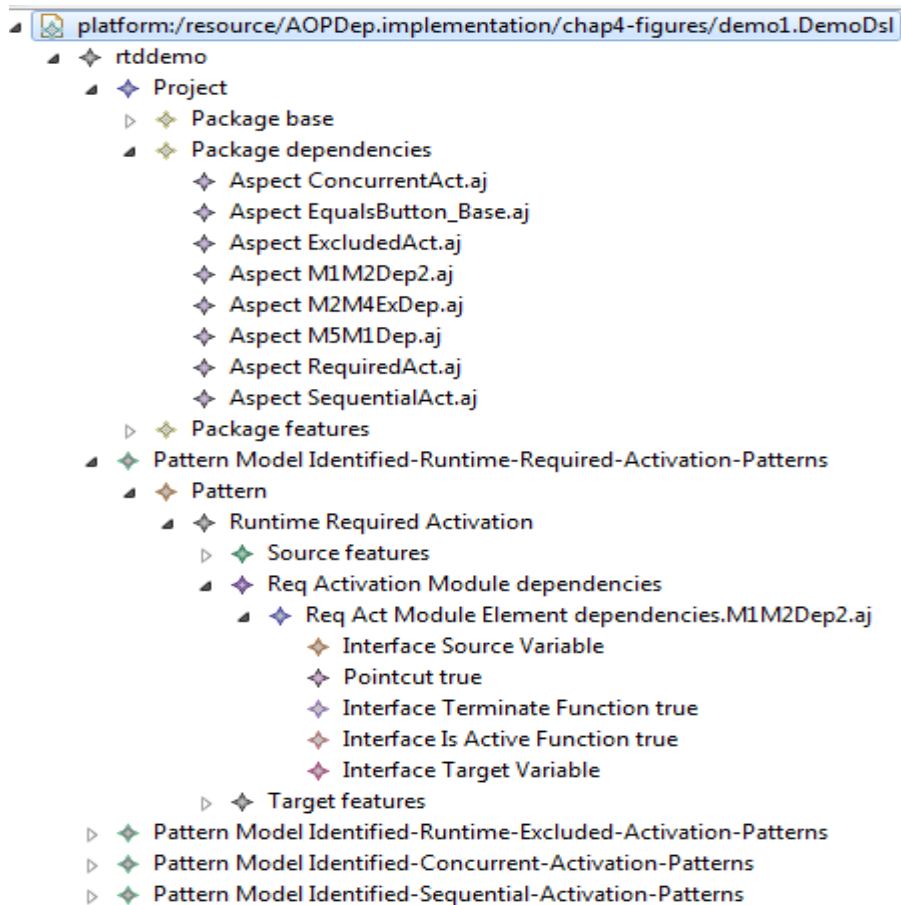


Figure 4-7 Generated IM and PM Meta-Model Instances

4.5.2 Mapping Specified Concepts in DOFM to Respective Implementation in IM

This realized process (Process 2) of the proposed technique described in Figure 3-11 is manually performed by the product line engineer. Figure 4-8 represents an instance of the feature model meta-model instance developed by the product line engineer. Various concepts like Feature and specific types of runtime feature dependencies are specified using the provided DOFM meta-model. Lower part of the Figure 4-8 represents the properties of the specified concepts of the runtime sequential activation dependency.

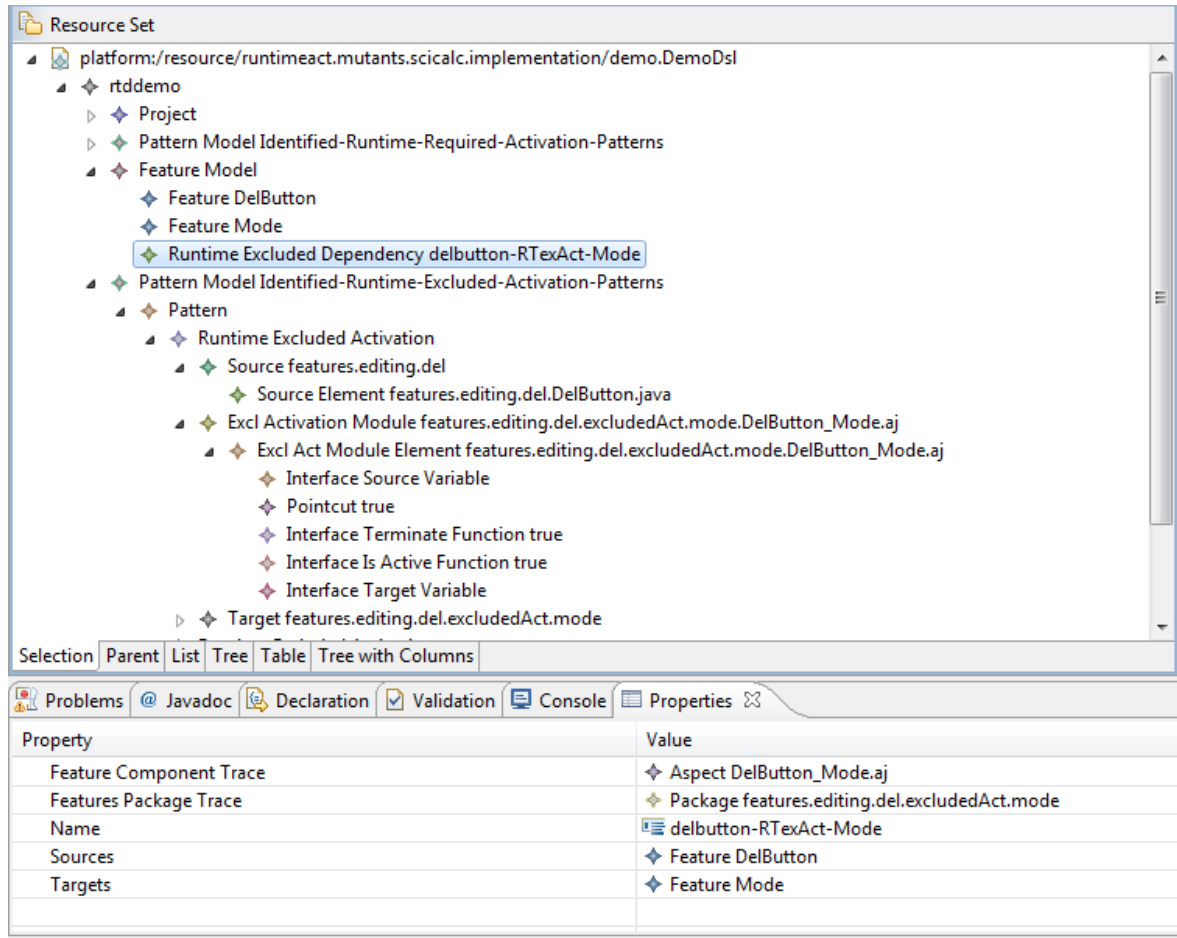


Figure 4-8 Manually Mapping the DOFM and the IM

4.5.3 Applying EVL Constraints

After the generation of the PM meta-model instance, the next step is to apply EVL constraints on the detected AO-patterns along with the DOFM and the IM in order to find any inconsistency between the DOFM and the PM. This realizes Process 3 discussed in Section 3.7.4. Figure 4-9 demonstrates the application of the EVL constraints. The error markers (e.g., Figure 4-10 represents an example) are generated in the Eclipse error view after the application of the EVL constraints. Appendix E represents the implementation of the EVL constraints.

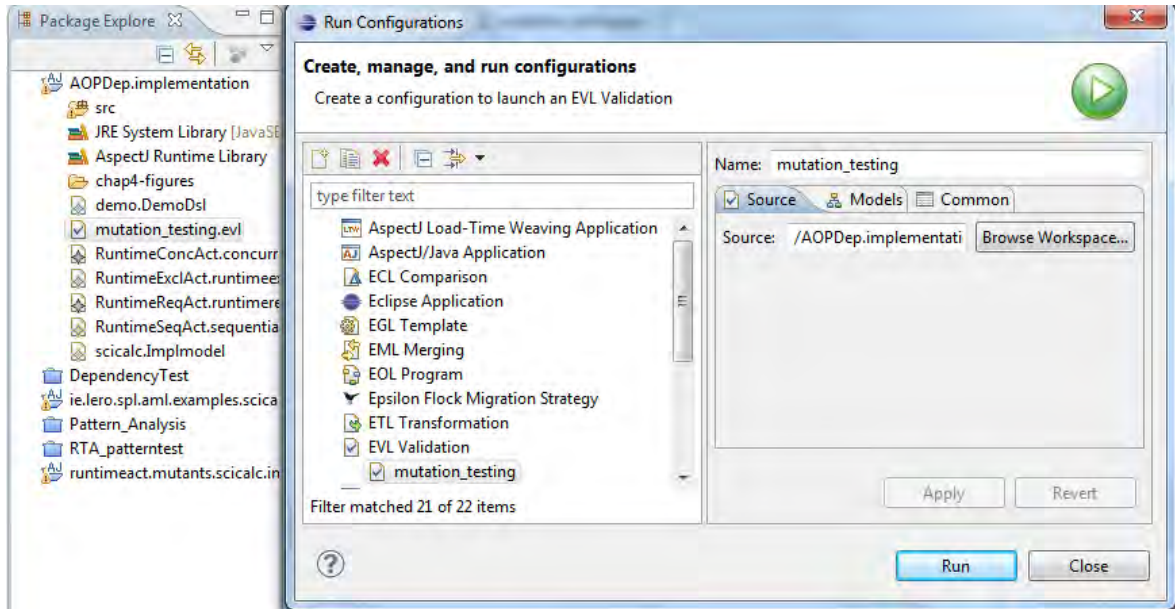


Figure 4-9 Applying Developed EVL Constraints

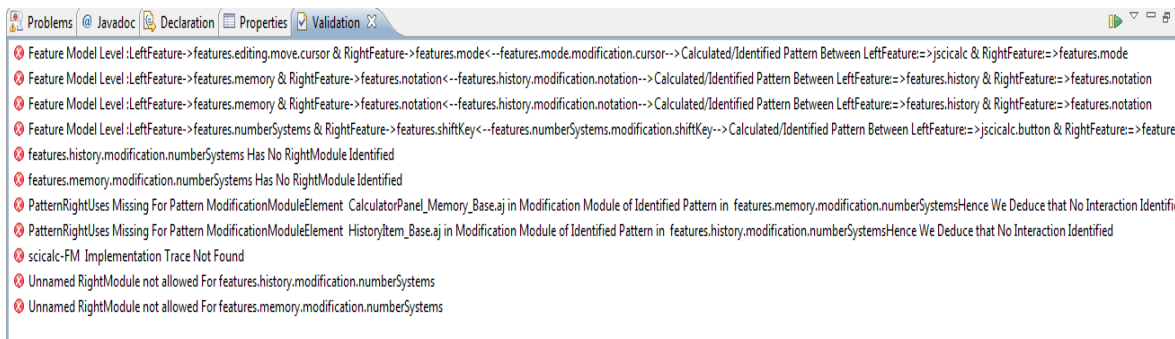


Figure 4-10 Example of the Generated Error Markers

The EVL error markers are the feedback to the product line engineer. The structural constraints give the product line engineer to identify if the specification is not complete (e.g., Unnamed RightModule not allowed for feature.history.modification.numberSystems). Inconsistency constraints produce error markers that represent inconsistency to the product line engineer that the Calculated/Identified Pattern between Source and Target features implementation do not conform to the specified pair at the Feature Model level (i.e., inconsistency discussed in Section 3.5.3). The EVL error markers also direct the product line engineer to a particular package not implementing the specified RTFD type at the feature model.

4.6 Discussion

The proposed technique is intended for identifying the inconsistencies in product line assets (i.e., the specified runtime feature dependencies and their respective AO pattern-based implementations). The AO pattern-based implementations were inspired by the pre-existing work by Pearce and Noble in (Pearce and Noble, 2006). However, work by (Lee and Kang, 2004) focused on dynamic relationships rather on the static relationships. The proposed technique composed of a process model (i.e., Chapter Three) that acts as a set of guidelines to perform the inconsistency detection. As discussed earlier in Chapter Three (i.e., Section 3.7), the proposed technique provides a round-trip engineering functionality.

There are various commercial and research prototypes providing RTE functionality in order to synchronize between models developed in the Unified Modeling Language (UML) and corresponding source code. FUJABA⁶ provides a set of plug-ins is one of the research prototypes providing the RTE functionality. But FUJABA provides a modelling tool for UML models and generates the code (forward engineering) and then maintains the model to code consistency. In order to provide the source code to architecture consistency by providing the pattern detection in the source code using the reclipse plug-in⁷. JITTAC (Buckley et al., 2013) research prototype is like FUJABA but only provides a reverse engineering facility in order to extract design patterns from source code. JITTAC provides real time inconsistency information while implementing the design patterns in the source code. However, FUJABA and JITTAC are not providing the source code to specification consistency while taking a particular type of design pattern (e.g., AO design patterns) consistency checking like the research prototype developed in this thesis work.

This chapter discussed the plug-in based tool support for the proposed technique. A meta-model developed using the EMF enables to raise the abstraction level and model the product line concepts. The implementation model meta-model enables the code to model transformation for further static analysis of the abstracted implementation concepts. The plug-in based tool support is flexible in terms developing a specific technique for the automation of any of the processes of the proposed technique (discussed in Section 3.7). For instance, when applying the constraints on the detected patterns, one can

⁶ http://www.fujaba.de/no_cache/home.html

⁷ http://www.fujaba.de/no_cache/projects/reengineering/reclipse.html

use OCL constraints or Xtext Check language constraints rather than using EVL constraints. Another flexible option can be to perform code to model transformation by using Java component by MoDisco Project⁸.

The research prototype has limitations that are as follows,

- The tool support works in the presence of AO pattern-based implementation in the source code of a product line.
- The AO pattern-based pattern detection algorithms are implemented with the static code analysis process taken into consideration and not considering the dynamic analysis of the source code.
- The AO patterns detection is performed within particular packages that contain the AO pattern-based implementations of runtime feature dependencies. Hence, if there is the presence of an implementation that works perfectly fine but not having an AO pattern-based solution in this situation the tool support will fail to identify that there exists an implementation of a particular runtime feature dependency not following a particular AO pattern?
- The identified inconsistencies by applying the EVL constraints serve as guidelines to identify the inconsistencies in a particular product line, but it doesn't provide the different solution options (e.g., quick fixes in Eclipse IDE) to resolve the identified inconsistency related to runtime feature dependencies in the product line artefacts (e.g., feature model or in implementation).

4.7 Summary

This chapter provided discussion on the implementation choices along with the research prototype realizing the conceptual framework discussed in Chapter three. The implementation of each of the processes was discussed. The domain modelling utilizing the ECore modelling language was performed and discussed in detail. The algorithms for AO-pattern detection are initially sketched (i.e., as a pseudo code) and then a detailed discussion was made on the working of AO-pattern detection algorithms. An off-the-shelf model checking language was utilized to verify/validate the specified behaviour (i.e., utilizing the DOFM) the implemented runtime behaviour using AO- design patterns. The chapter ended

⁸ <http://www.eclipse.org/MoDisco/>

with the discussion on the possible implementation choices that can be made for the implementation of the proposed technique and the limitations of the currently developed research prototype.

Chapter Five: Validation and Evaluation

5.1 Overview

The aims of this chapter are to describe the validation of the tool supported solution presented in previous chapters and to demonstrate that the proposed technique works in practice. An experimental validation approach inspired by “Mutation Testing” (Offutt, 1994) is applied to validate the solution. The proposed validation technique seemed appropriate, due to the scarcity of suitable examples of software product lines implemented in Java and AspectJ. The mutation-based approach allows the generation of multiple test cases that can be used as an input to the proposed approach for consistency checking of runtime feature dependencies in the product line assets. The validation results are discussed in detail. Formal evaluation of the proposed approach is performed by interviewing various experts in software research and development industry and discussion is made of the feedback obtained from the formal interviewing process. Comparative evaluation is also made with respect to the current state-of-art (i.e., Chapter Two). Qualitative analysis of the proposed technique is also made.

5.2 Validation

The proposed process model is realized as a prototype and discussed in Chapter Four. The prototype is developed for proof of the concept/ the conceptual process model (Section 3.7). In order to validate the prototype, the prototype has to be applied on a case study. The following sections will discuss the validation strategy, the experimentation case study and presents the validation results.

5.2.1 Validation Strategy and Hypothesis

This section discusses the validation strategy in order to validate the proposed technique. The only way to prove the usefulness of the proposed solution is to apply the validation technique to the case study and to gather and present the collected data. Due to scarcity of case studies validation purposes Mutation Testing-inspired approach is applied for the validation purposes on the “Scientific Calculator” (SciCalc) product line case study. The main idea of using the mutation testing-inspired approach in the thesis research work is to mutate the SciCalc product line and build various different mutants. The mutants are validated using the proposed technique.

In Mutation Testing (MT) (Offutt, 1994), the program under test is mutated. In practice, mutation testing used to evaluate a set of test cases. In theory, it can also be used to evaluate a (new) testing technique using the mutated source code, to see whether the new technique can find more errors than previous techniques. In the context of this research, the validation approach is similar to the latter use. Using mutated versions of the SciCalc product line, the purpose is to evaluate a static analysis technique, not a dynamic testing technique.

In the scope of the thesis, research both the specification of runtime feature dependencies and their respective pattern-based implementations can change leading towards a set of mutated versions of the SciCalc product line. The mutated versions of SciCalc provided me with a set of tests that needs to be evaluated using the proposed static analysis technique. The choice of changing the specification along with implementation is made because of the fact that the specifications and their respective implementations are dealt with by various stakeholders in a distributed environment. Specification change is called a specification mutant and implementation change is called an implementation mutant. Following hypothesis have been established for validation purposes:

Validation Hypothesis: The solution enables the product line engineer to identify inconsistencies in product line assets (i.e., the specified runtime feature dependencies and their respective aspect-oriented pattern based implementations).

In order to examine the correctness (i.e., true/false) of the hypothesis, a formal evaluation process is performed by interviewing the experts (i.e., software engineers, researchers and architects) to obtain detailed feedback to formally evaluate the approach (i.e., Section 5.3).

The aim of the validation is to provide the prototype working in the context of case study. Validation results provide results that are further used to demonstrate the usability of the prototype and the proposed technique to the interviewees.

5.2.2 Mutation Testing-Inspired Approach for Validation

The challenge addressed by this thesis work focuses on the heterogeneous product line assets developed by various stakeholders. The artefacts can be changed independently by the stakeholders. An example of a specification mutant can be generated by changing the specification of a particular runtime dependency type with one known problem or inconsistency. On the other hand, an example of implementation mutant can be generated by commenting out one

part of the AspectJ patterns implemented in the source code of a respective runtime feature dependency type. A mutant has a mutant ID and one known problem or inconsistency.

Mutant in the context of my research is a deliberate error made in either the specification or implementation of the RTFDs. A developed mutant (i.e., specification or implementation) represents one problem (e.g., specification incompleteness or code breaking) that can be related to inconsistency scenarios discussed in Section 3.5.

Once the mutants were created for each type of runtime feature dependencies (i.e., specification and implementation), the proposed technique and prototype were then validated using the generated mutants. If the inconsistencies in the mutants (i.e., specification mutants or implementation mutants) are detected then the mutants are said to be killed and validation is said to be adequate. Mutation testing is assessed by mutation coverage technique. Mutation coverage is equal to the number of mutants killed. In the context of validation of the proposed technique, the technique was applied on the mutants created for the SciCalc product line case study.

A mutant is classified as being killed, if the proposed technique is able to identify the deliberate change made either in the specification (i.e., DOFM) or in the AspectJ-based pattern implementation. The more the proposed technique is able to detect the change/mutation the more is proposed technique is considered to be successful.

5.2.3 Experimentation- Scientific Calculator Product Line Case Study

The SciCalc case study contains twelve mandatory variable features and thirteen common features. The scientific calculator product line is capable of generating different products with various functionalities that includes different types of scientific functions (e.g., Sin, Cos and Tan) number systems (e.g., Hexadecimal, Binary and Octal). The implementation of calculator product line is in Java development language. There are 67 packages with 316 Java and AspectJ classes in total are used to implement the calculator product line. There are 19 packages (28 %) with 28 AspectJ classes used to implement the runtime modification, 7 packages (10 %) with 14 AspectJ classes to implement the runtime required activation, 7 packages (10 %) with 17 AspectJ classes to implement the runtime excluded activation and 2 packages (3 %) with 2 AspectJ classes to implement the runtime sequential activation dependencies between various features implementations. Figure 5-1 represents the calculator

product line with mandatory variable and common features as a feature model excerpt.

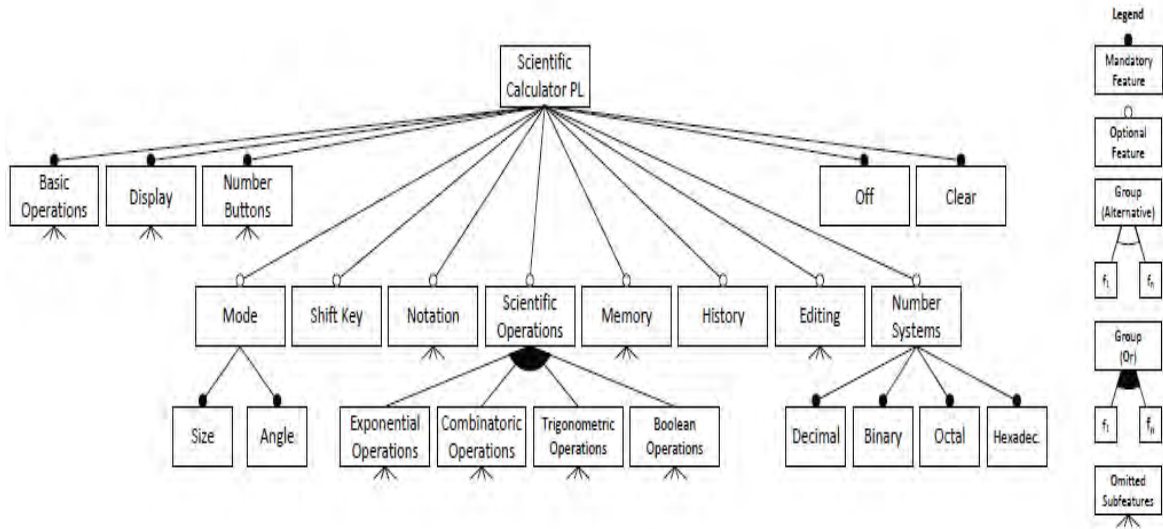


Figure 5-1 Scientific Calculator Product Line Feature Model (an excerpt)
(Lee et al., 2009a)

5.2.4 Validation Example from Scientific Calculator Implementation of Runtime Required Activation Dependency (RTRAD)-An Example Case

This section discusses a validation example from the SciCalc product line case study. The section discusses the specification of the example, respective pattern-based implementation code and mutation testing-inspired validation results in a tabular form. The generated mutants (i.e., both the specification and implementation mutants) as examples provided a sufficient coverage of the scenarios discussed in Section 3.5.

5.2.4.1 Specification

The developed domain-specific language (DSL) (please consider Section 3.6.1) is used for specifying the features model that enables the requirements engineer to specify the RTRAD between the features “**And**” and “**HexPanel**”. Figure 5-2 shows the DOFM for specifying the runtime required activation between the features “**And**” and “**HexPanel**” along with the property view for specifying the RTRAD attributes.

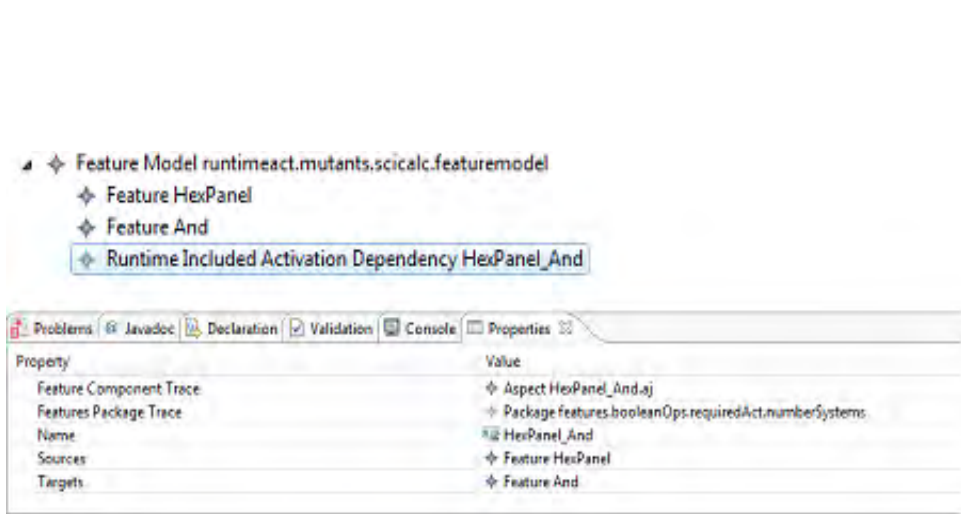


Figure 5-2 Specification of Runtime Required Activation between features “And” and “HexPanel”

Figure 5-3 represents a XMI format of the specified runtime Required Activation between features “And” and “HexPanel”.

```
1. <featuremodel
   name="requiredact.mutants.scicalc.implementation.featuremodel">
2. <rootFeatures name="HexPanel"
   FeaturesPackageTrace="//@implmodel.0/@ownedMembers.48"
   FeatureComponentTrace="//@implmodel.0/@ownedMembers.48/@ownedMembers.0"
   />
3. <rootFeatures name="And"
   FeaturesPackageTrace="//@implmodel.0/@ownedMembers.1"
   FeatureComponentTrace="//@implmodel.0/@ownedMembers.1/@ownedMembers.0"
   />
4. <rootFeatures
   xsi:type="featuresmodel:RuntimeIncludedActivationDependency"
5. name="HexPanel-RTA-And"
6. FeaturesPackageTrace="//@implmodel.0/@ownedMembers.2"
7. sources="//@featuremodel.0/@rootFeatures.1"
8. targets="//@featuremodel.0/@rootFeatures.0"/>
9. FeatureComponentTrace="//@implmodel.0/@ownedMembers.2/@ownedMembers.8"
10. </featuremodel>
11. </DemoDsl:rtddemo>
```

Figure 5-3 XMI-based representation of Feature Model (DOFM) with Runtime Required Activation between features “And” and “HexPanel”

5.2.4.2 Implementation Code

Figure 5-4 shows the AspectJ-based implementation (Colyer et al., 2004) of the RTRAD between the features “And” and “HexPanel”. The AspectJ implementation follows the RTRAD pattern (Lee et al., 2009a). The same code was deliberately altered for generation of implementation mutants for the proposed technique validation.

```

1. package features.booleanOps.requiredAct.numberSystems;
2. import features.numberSystems.requiredAct.buttons.HexPanel;
3. import features.booleanOps.And;
4. public privileged aspect HexPanel_And extends
   RequiredAct5<HexPanel,And>
5. {
6. after(And a): execution(And.new()) && this(a)
7. {
8. isAndActive=true;
9. target=a;
10. }
11. boolean isAndActive = false;
12. public boolean And.isActive()
13. {
14. return aspectOf().isAndActive;
15. }
16. public void HexPanel.terminate(){}
17. before(HexPanel hp): execution(* setButtons()) && target(hp)
18. {
19. hp.buttons().elementAt(8).setPObject(new And());
20. System.out.println("HexPanel is Active");
21. source=hp;
22. }
23. pointcut serviceJP(HexPanel m1):
24. execution(* setButtons()) && target(m1);}

```

Figure 5-4 Actual Implementation of Runtime Required Activation Dependency between “HexPanel” and “And” Implementations

Figure 5-5 shows an extracted RTRAD pattern model instance by applying the runtime required activation pattern detection algorithm.

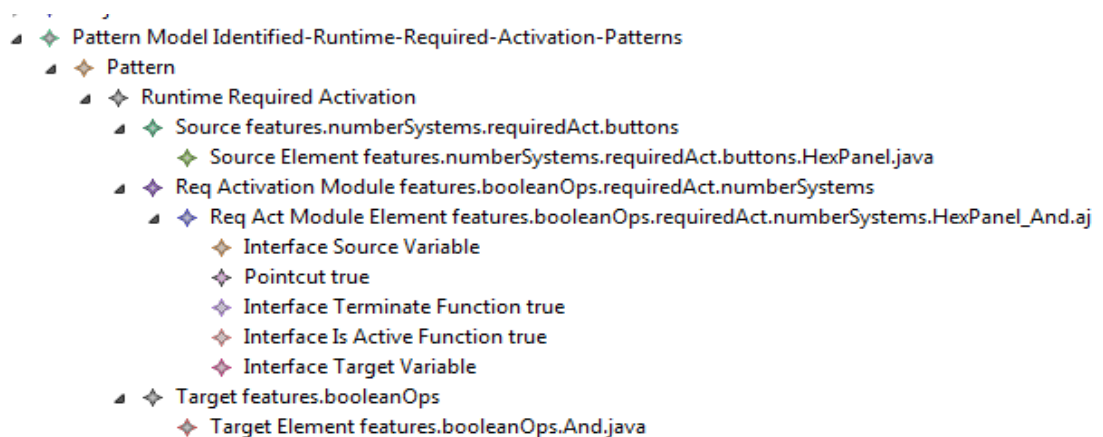


Figure 5-5 Detected Runtime Required Activation Pattern Model Instance

5.2.4.3 Validation Results for the example RTRAD between "And" and "HexPanel" features

An example of the specification mutant is generated as a result of change in the feature model (DOFM). For instance, Sources feature not specified in RTRAD relationship between features HexPanel and And (i.e., reference to HexPanel.java not specified in DOFM Runtime Included/Required Activation). There is a wide range of mutation operators possible for the generation of the specification and implementation mutants. For instance,

- Value mutation (i.e., changing the values of constants)
- Decision mutations (i.e., modifying conditions to reflect errors in the code)
- Statement mutations (i.e., deleting certain lines of code)
- Access Control mutation (example, Access modifier change)
- Inheritance change mutation (example, super keyword deletion)
- Polymorphism mutation (example, new method call with child class type)
- Overloading mutation (example, Argument order change)
- Java specific features mutation (example, this keyword deletion)
- Common programming mistakes (example, reference assignment and content assignment change)

An example of an implementation mutant is breaking various parts of the implemented runtime required activation pattern in the source code. For instance, an implementation mutant can be generated by applying the statement mutation mechanism to delete certain lines of AspectJ pattern implementation between And.java and HexPanel.java classes (e.g., deleting the source part of the pattern).

At specification model level, I applied the value mutation to change the Feature model parts (e.g., "FeaturesComponentTrace" reference to the Implementation model part). At the source code level, I have applied statement mutation, Overloading mutation, Access modifier mutation, Value mutation, Java specific features and common programming mistakes mutation for validation purposes

of the runtime required activation dependency implementation between And.java and HexPanel.java classes.

Validation is performed on an example RTRAD between the “And” and “HexPanel” features. Table 5-1 shows the validation results for implementation mutants. Table 5-2 shows the validation results of the specification mutants. The expected inconsistency column represents what to expect as a result of the change. If the response generated by the proposed technique is equal to expected response, the mutant is said to be killed. Implementation mutants are generated by applying the various mutation operators in the example AspectJ based code (please consider Figure 5-4). Specification mutants were generated by applying the mutation operators on the example specification (i.e., Figure 5-3).

From tables 5-1 and 5-2, I deduce that 22 (implementation and specification mutants) out of total 24 mutants (91.6%) were killed by the proposed solution. It provides 91.6 percent of the mutation coverage.

Mutant ID	Mutant Implementation	Expected Inconsistency	Actual Implementation	Response
IMPL1	Deleting the source part(line-21)	Pattern not found	source=hp	Pattern not found between HexPanel and And
IMPL2	Deleting the target part(line-9)		target=a	
IMPL3	Terminate() not implemented (line-16)		Public void HexPanel.terminate() {}	Pattern no found between HexPanel and And. Java compiler also gives an error that Terminate function is not implemented for HexPanel.java
IMPL4	Pointcut ServiceJp() part not implemented (lines 23-24)		pointcut serviceJP(HexPanel m1): execution(* setButtons()) && target(m1);	Pattern no found between HexPanel and And. Java compiler also gives an error that Pointcut is not implemented.
IMPL5	IsActive() not implemented (lines 12-15)		public boolean And.isActive() {return aspectOf().isActive;};	Pattern no found between HexPanel and And. Java compiler also gives an error that isActive function should be implemented for And.java
IMPL6	Reverse the Parameters and Implementation RequiredAct5<And,HexPanel> Where Source =And and Target=Hexpanel	Pattern found but specified and implemented concepts contradicts	RequiredAct5<HexPanel,And>W here Source=HexPanel and Target=And	Pattern found but contradiction between specified and implemented concepts also Java Compiler gives an error
IMPL7	Replace target=a to source=a (line-9)	Pattern not found	target=a	Pattern not found but already caught by the Java compiler
IMPL8	Replace source=hp to target=hp (line-21)		source=hp	Pattern not found also Java Compiler caught the error upfront
IMPL9	Adding Terminate() for Target And.Terminate()	Pattern found but	Implementation contains only	Pattern found. But solution didn't able

	along with existing HexPanel.terminate() (i.e., two terminate() implementations in the pattern)	ambiguous implementation	HexPanel.Terminate()	to find the ambiguity.
IMPL10	Adding HexPanel.isActive() function in the implementation (i.e., HexPanel.isActive() {return true;})	Pattern found but ambiguous implementation	Implementation contains only And.isActive() function.	Pattern found. Solution didn't able to find ambiguity.
IMPL11	Adding pointcut function advising And.java not HexPanel.java class (i.e., pointcut serviceJP(And m2): execution (new())&& target(m2);)	Pattern found but pointcut defined on And (target) and not on HexPanel(source).	Implementation contains pointcut for HexPanel.java.pointcut serviceJP(HexPanel m1): execution(* setButtons()) && target(m1);	Pattern found. Solution able to detect that pointcut is defined on And.java and not HexPanel.java class. Compiler also gives an error because of intertype declaration conflict.
IMPL12	Adding lines Or or= new Or(); source=or; and commenting source=hp; (i.e., //source=hp)	Pattern found but source variable is not pointing towards Hexpanel.java class else it is pointing to Or.java class	source=hp	Pattern found. Java compiler gives an error cannot convert from Or to RequiredAct5.S1Interface. Error given by the solution that source variable is not pointing to HexPanel.java but pointing to Or.java, please set source variable appropriately in the implementation.
IMPL13	Adding lines Or or= new Or(); Target=or; and commenting target=a (i.e., target=a)	Pattern found but target variable is not point towards And.java class but it is pointing to Or.java class	target=a	Pattern found. No Java compiler error. Error given by the solution that target variable is not pointing to And.java but pointing to Or.java class. Please set target appropriately in implementation.

IMPL14	Changing the target Parameters for implementation of RequiredAct5<HexPanel,Or>	Pattern found between HexPanel and Or but not between "HexPanel" and "And"	RequiredAct5<HexPanel,And>	Pattern found. Java compiler also gives a TypeMismatch error. Solution gives an error that target element is Or.java and not And.java.
IMPL15	Changing the source parameters for implementation of RequiredAct5<Or,And>	Pattern found between "Or" and "And" features and not between HexPanel and And	RequiredAct5<HexPanel,And>	Pattern found, Java compiler also gives a TypeMismatch error. Solution gives an error that source element is Or.java and not HexPanel.java

Table 5-1 Validation of Implementation Mutants for RTRAD Relationship Example

Mutant ID	Mutant Specification	Expected Inconsistency	Actual Specification	Response
SPEC1	Source not specified in Runtime Required Activation Dependency	Source not Specified in the Runtime Required Activation Dependency	sources=//@featuremodel.0/@rootFeatures.1 (line-7 in Figure 5-3)	Source Feature not specified in RuntimeRequiredActivation
SPEC2	Target not specified in Runtime Required Activation Dependency	Target not Specified in the Runtime Required Activation Dependency	targets=//@featuremodel.0/@rootFeatures.0"/> (line-8 in Figure 5-3)	Target Feature not specified in RuntimeRequiredActivation
SPEC3	RuntimeRequiredActivation reference to Implementation Package not specified	RuntimeRequiredActivation reference missing to Implementation	FeaturesPackageTrace=//@implmodel.0/@ownedMembers.2 (line-7 in Figure 5-3)	RuntimeRequiredActivation to Implementation Package Trace not specified
SPEC4	RuntimeRequiredActivation reference to Implementation Component (AspectJ or Java Classes) not specified	Missing reference from RuntimeRequiredActivation to Implementation Component (AspectJ or Java class)	FeatureComponentTrace=//@implmodel.0/@ownedMembers.2/@ownedMembers.8 (line-9 in Figure 5-3)	Reference from RuntimeRequiredActivation to Implementation Component not specified
SPEC5	RuntimeRequiredActivation name not specified	RuntimeRequiredActivation name not specified	name="HexPanel-RTA-And" (line-5 in Figure 5-3)	RuntimeRequiredActivation name is not specified
SPEC6	RuntimeRequiredActivation sources and targets reversed	RuntimeRequiredActivation dependency pattern Identified	sources= "//@featuremodel.0/@rootFeatu	Specified Pattern is Implemented in Reverse Order (i.e., Specified

	(source=HexPanel and target=And), specification reversed	but contradiction between specified and Implemented dependency	res.0"/> targets="//@featuremodel.0/@rootFeatures.1	Dependency Target=Identified Pattern Source and Specified Dependency Source= Identified Pattern Target)
SPEC7	RuntimeRequiredActivation specified sources changed	RuntimeRequiredActivation specified source is different than identified pattern source	sources="//@featuremodel.0/@rootFeatures.1 (line-7 in Figure 5-3)	Solution is able to identified specified source is different than identified pattern source
SPEC8	RuntimeRequiredActivation specified target changed	RuntimeRequiredActivation specified target is different than identified pattern target	targets="//@featuremodel.0/@rootFeatures.0"/> (line-8 in Figure 5-3)	Solution is able to detect that identified specified target is different than identified pattern target
SPEC9	Specified Dependency is not RuntimeRequiredActivation (e.g., SequentialActivationDependency)	Specified Dependency is RuntimeSequentialDependency whereas identified is RuntimeRequiredActivation	<rootFeatures xsi:type="featuresmodel:RuntimeIncludedActivationDependency "...>	Solution is able to detect that there is a contradiction between specified dependency type contradicts with identified dependency type

Table 5-2 Specification Mutants Validation for RTRAD Relationship Example

5.3 Evaluation based on Formal Interviews

In order to formally evaluate the proposed technique, a series of formal interviews was conducted. Various stakeholders (i.e., researchers, software engineers and software architects/designers in software research and industry) were selected and interviewed to get their feedback.

5.3.1 Candidate Selection

Candidates were selected based on their experiences in software engineering systems design and implementation. A total of eight candidates were selected for the formal interview sessions. Out of eight candidates there were four candidates that are serving in various industrial projects (e.g., Epsilon⁹, Siemens¹⁰ and Ericsson¹¹) with aim of providing the model-driven solutions to solve the complex industrial challenges. Two have major experiences in designing and implementing the software systems for the major telecommunication industry. The remaining two have research experiences in designing software systems in the cloud computing and multimedia domain. The idea of selecting candidates not related to MDD domain was to get an independent feedback and check the feasibility of the proposed approach in their respective domains of interest (i.e., telecommunication and cloud computing).

5.3.2 Evaluation Package Formulation

Before conducting the formal interview process, an evaluation package was prepared for the interviewees. The evaluation package consists of the following contents;

- A set of slides providing an overview of the PhD research
- A demo video¹² providing step by step guide to execute the prototype and map to the processes of the proposed process model
- A set of inconsistency scenarios discussed in the prototype demo
- A questionnaire formulated to elicit the feedback from the interviewees

⁹ <https://www.eclipse.org/epsilon/>

¹⁰ <http://www.siemens.com/innovation/en/>

¹¹ <http://labs.ericsson.com/>

¹² Demo video was uploaded using the YouTube (www.youtube.com) at the URL: <https://www.youtube.com/watch?v=ncEmblSx7Fc>

5.3.3 Questionnaire Formulation

A set of open questions was formulated to get feedback on various aspects (e.g., usability, efficiency and adaptability) of the proposed approach. These may be found in Appendix D. It was designed in the nature of formal interview rather than survey with the intension of performing face to face interviews where possible.

5.3.4 Formal Evaluation Process

The evaluation process is divided into three steps. The following sections discuss each of the steps to perform the formal evaluation process.

5.3.4.1 Evaluation Pre-Requisites

In order to get the feedback from the interviewees the following steps were taken,

- Formal interview invitation were sent to the selected candidates
- Date, time and venue were decided
- The evaluation package along with the guidelines as a training material were sent in advance to the selected candidates
- Queries (if any) made by the interviewee were answered
- Data collection process was defined

5.3.4.2 In the Interview

Once the selected candidates agreed to provide their feedback for the formal evaluation of the proposed approach, the next step performed was to conduct the formal interviews. Candidates were asked the questions from the questionnaire giving them enough time to comprehend the questions easily. (Any ambiguities related to the questions could have been resolved before the formal interview session but it did not arise.)

5.3.4.3 Collection of Interview Data

The formal interviews were recorded with the consent of the interviewees. A separate confidentiality clause was explicitly included in the questionnaire in order to protect the data provided by the interviewee. The answers to the questions were also typed up from the audio recordings after performing the interviews. Some of the participants were not available for face to face interview and therefor answered the questionnaire by email.

5.3.4.4 Validation of the Interview Data

The typed transcript of each interview was emailed to the relevant interviewee for validation. Some interviewees made small corrections in response. The resulting transcripts can be found in Appendix G.

5.3.5 Key Findings of Evaluation Results Discussion

The feedback obtained from the formal interview sessions is discussed in this section.

Most important factors addressed by my research based on the interviewee's feedback are as follows, 1) Inconsistency detection between specifications of RTFDs and respective pattern-based implementations, 2) RTFDs implementations based on AO-design patterns detection, 3) Domain specific language (DSL) for the product lines with the focus on RTFDs, 4) Consistency checker as an Eclipse plug-in and set of EVL constraints.

Both the structural and the behavioural types of inconsistencies in a software product line domain are important and required to be taken into account. However, a few of the interview candidates prioritized detection of behavioural inconsistencies detection above structural inconsistencies earlier in the domain engineering to avoid causing problems during product assembly and product derivation.

All of the interview candidates agreed that getting earlier feedback as error markers related to inconsistencies of RTFDs is important during the product line asset development in domain engineering. Such an activity of performing consistency checking of RTFDs would help them identify bugs/errors related to RTFDs earlier during the development phase of the product line assets and would facilitate them to perform the maintenance of the product line assets.

All of the participants agreed that the processes in the process model (Section 3.7) were easy to follow and the tooling/prototype (Section 4.5) would benefit the product line engineer/software engineer to identify inconsistencies during the development and maintenance of the product line assets (i.e., specification and implementations of RTFDs).

It was easy for the participants to understand and comprehend the error markers generated by the EVL constraints. A few candidates also mentioned that not all possible types of the inconsistencies were covered in their feedback. However, all candidates agreed that the error markers were useful for the covered inconsistency scenarios (Section 3.5).

Static verification is considered by the interviewees to be one of the best practices in order to perform the validation and verification (V&V) of software systems. All of them agreed that the proposed approach performed well when verifying the implementations of the RTFDs to their respective specifications. A few of them are confident that the developed prototype has a potential to be a commercial tool after performing necessary improvement steps. However, a few candidates recommended a scalability check for the proposed approach by applying it to a large set of product line assets, as a possible step towards commercialization of the prototype.

5.4 Discussion of Validation Results and Lessons Learned

This Section discusses the validation results of the proposed technique applied to the SciCalc product line case study. Tables 5-1 and 5-2 show the validation results by applying the mutation testing-inspired approach. Both the implementation mutants and the specification mutants are created for each type of runtime feature dependencies and validated using the proposed solution. Table 5-3 represents a total of 100 mutants (both the specification and implementation mutants).

The validation and formal evaluation process both helped me to identify the strengths and limitations of the proposed approach, as well as to answer the correctness (e.g., true/false) of the hypothesis set earlier, that is

“The solution enables the product line engineer to identify inconsistencies between the specified runtime feature dependencies and their respective aspect-oriented pattern based implementations”.

The positive feedback from the interviewees also supports the set hypothesis that the provided model-based solution can help the product line engineer to identify the inconsistencies earlier in the development phase of the product line assets. The fair amount of coverage and positive feedback becomes a success criteria of the proposed technique.

One of the major strengths of the proposed approach is the simplicity of following its processes and the ease of use to perform consistency checking to identify inconsistencies related to RTFDs. Other strengths are that it is easy to comprehend the inconsistencies based on the generated error markers/feedback, that the AspectJ pattern detection algorithms are implemented as a Java plug-in and particularly the static analysis of the source code.

On the other hand, one of the limitations identified during evaluation processes is its scalability. However, based on the formal evaluation, 75% of the interviewees agreed that the proposed approach has the potential to be

of commercial use. Other limitations are its focus on certain types of design patterns (AspectJ-based design patterns) and Java-based implementations.

One of my findings is that the work on the classification, aspectual separation and modularization of runtime feature dependencies started by Lee et al. (Botterweck and Lee, 2009, Lee et al., 2009a, Lee et al., 2009c, Lee and Kang, 2004) is incomplete with respect to its validation. One step towards completion was to implement the AspectJ-based patterns in the SciCalc implementation. It was essential to do this before applying the proposed technique. I prepared the experimentation case study myself by applying the prescribed AspectJ-based patterns in the implementation before validating them.

As discussed earlier that due to lack of the available case studies only SciCalc product line case study was selected for validation purposes. In order to fulfil the need of having multiple case studies mutation testing-inspired validation technique was adopted to perform the validation of the proposed technique and provided tool support. Mutants (both specification and implementation) were created and the proposed technique was applied in order to kill the mutant. Killing a mutant means that the proposed technique when applied on the mutant is able to identify the problem.

Tables 5-1 and 5-2 show the validation results. One of the noticeable observations was that certain types of mutants that were easily killed by the Java compiler were not taken into consideration for the validation. Only known mutants, these are based on inconsistency scenarios, were generated for validation. It might be the case that other unknown mutants still could exist.

In the case of the modification dependency which is implemented as a generic AspectJ class, 12 of the known mutants were created and killed by the proposed technique. Hence the coverage is 100%. In the case of mutants representing runtime required and excluded activation dependencies (both specification and implementation), 22 out of 24 mutants were killed. The ones that are not killed have multiple implementations of `isActive()` and `terminate()` in the respective pattern-based implementation.

For runtime excluded activation please consider Appendix B. The reason that it is unable to identify the ambiguity is that the pattern detection algorithm is designed in a way to detect if the `isActive()` and `terminate()` functions are implemented or not in the source code. This type of ambiguity can be dealt with by improving the pattern detection algorithm for detecting multiple `isActive()` and `terminate()` functions implementation in the

respective runtime required/excluded activation pattern-based implementation.

In the case of runtime sequential activation dependency, 20 known types of mutants were generated for the validation purposes. For runtime excluded activation, please consider Appendix B. Out of 20 known mutants 18 were successfully killed. Hence mutation coverage or validation of 91.6% is achieved. The ones that are not killed by the proposed technique have multiple implementations of `activate()` and `pointcut()` functions in the pattern-based implementation. The reason again is the same, that the pattern detection algorithm is designed in a way that it only detects the presence of the implementations of `activate()` and `pointcut()` functions and not the presence of multiple implementations of them. The issue can be resolved by improving the pattern detection algorithm in order to detect multiple `activate()` and `pointcut()` functions in the sequential implementation.

In the case of runtime concurrent activation dependency, again 20 known types of mutants are generated for the validation purposes. For validation results of runtime concurrent activation dependency, please consider Appendix B. Here I would like to mention that the SciCalc product line does not contain any examples of runtime concurrent implementation examples. Hence, I created a demo example implementation between M5 and M1 features and performed that validation on the demo example. Out of 20 known mutants, 18 were killed by technique, the hence the coverage is 91.6%. The ones that are not killed by applying the proposed technique have multiple implementations of `run()` and `pointcut()` in the pattern-based implementation. The reason why these kinds of mutants are not killed is because the pattern detection algorithm is not designed and implemented for detecting multiple `run()` and `pointcut()` implementations. It can be resolved by improving the concurrent activation pattern detection algorithm for detecting the multiple `run()` and `pointcut()` functions in implementation. The pattern detection algorithms need to be refined so that respective EVL constraints can kill the remaining mutants (i.e., which are very few).

After analyzing the results obtained from the validation, I can deduce that the proposed technique is applicable to other contexts and case studies that share the same characteristics as the case study investigated in terms of using Java implementations and AspectJ design patterns.

The validation of the proposed technique (i.e., set of inconsistency scenarios generated based on mutants) was also discussed in the demo video (i.e., part of the evaluation package) with the interviewees. Based on their feedback it was deduced that not all types of mutants were taken into consideration for the validation of the proposed technique. It means that only known types of

mutants, based on the inconsistency scenarios, were generated for the validation.

After analyzing the validation results obtained, one can deduce that the hypothesis is correct but with certain factors to be taken into account, namely,

1. Only certain known types of mutants are taken into consideration, and
2. Pattern detection algorithms still need improvement.

Type of Pattern	Number of Mutants Created	Number of Mutants Killed	Mutation Coverage/Validation
Modification Dependency	12	12	100%
Runtime Required Activation Dependency	24	22	91.6 %
Runtime Excluded Activation Dependency	24	22	91.6%
Runtime Sequential Activation Dependency	20	18	90.0%
Runtime Concurrent Activation Dependency	20	18	90.0%

Table 5-3 Validation Results for “Scientific Calculator” Case Study

5.5 Comparative Evaluation of the Proposed Approach with State of the Art

Chapter Two (Section 2.3) provided an overview of the product line approaches (with or without tool support) that supports the feature dependency/interaction detection and analysis. For each of the approach/technique we discussed how they are trying to resolve the feature dependency detection and analysis process.

Table 5-4 summarizes the comparison by showing how well the techniques/approaches support the feature dependency identification, detection and resolution of the inconsistencies caused by the contradiction between specified and implemented, focus on runtime feature dependencies, tool support and focus on particular product line stakeholders (e.g., Developer, Product Line Engineer).

Capability/ Technique or Tool	Ferber et al.	Kang et al.	Ye et al.	Zhang et al.	Lee et al.	Savolainen et al.	Silveira et al.	Kim et al.	Parra et al.	Apel et al.	Batory et al.	Mosser et al.	Proposed Technique
Focus on Runtime Feature Dependencies	--	++	--	--	--	+	--	--	+	--	--	--	++
Contradiction Analysis between Specified Feature Dependencies and Respective Implementation	--	--	--	--	-	-	+	+	+	+	++	++	++
Tool-Support	-	--	--	--	-	--	--	++	+	+	+	+	++
Runtime Feature Dependency Detection	--	++	--	--	--	--	--	--	--	--	--	--	++
Resolution of Inconsistencies Due to Contradictions	--	-	--	--	--	-	-	--	--	+	+	+	+
Focus on Any Product Line Role (e.g., Product Line Engineer, Developer.)	--	--	--	--	--	--	+	--	++	+	--	++	++

Table 5-4 Existing SPLE Approaches/Tools Supporting Identification and Resolution of Runtime Feature Dependencies Related Inconsistencies (-- Not Supported, - Some Supported, + Supported, ++ Very Well Supported)

By analysing the Table 5-4, I can deduce that not many techniques are focusing on the runtime behaviour of the feature dependencies. However, many of the techniques/approaches are focusing on structural and configuration dependencies detection and analysis. Also contradiction analysis between specified structural and configuration dependencies (e.g., requires and excludes dependencies) are of main interest for many of the techniques. Many of the techniques of interest discussed in Section 2.3 are also not focusing on any particular product line stakeholders (e.g., product line engineer, developer and requirements engineer).

5.6 Qualitative Analysis of the Proposed Solution

This section presents a qualitative analysis of the proposed technique in terms of the properties of the framework discussed in Chapter Three. I have identified four factors (i.e., extensibility, scalability, usability and variability).

5.6.1 Extensibility

Extensibility means that the proposed approach has a capability to be extended into new domains and can also consider new product line artefacts. The proposed approach is given a support for extensibility to an extent. The proposed approach is based on MDD methodology. Models are considered as a first-class for the development and adaptation of the different artefacts. Extensibility is achieved by adding new meta-models, transformations and models. The extensibility is achieved at domain engineering level by modification of the core assets (i.e., developed meta-models, code to model transformation, pattern detection algorithms and constraint language).

Extending the domain represents a major challenge since it requires modifying the product line assets accordingly. An example of extensibility at domain engineering level is to change the implementation language (e.g., C++, Python) other than Java and AspectJ. Such type of extensibility can be achieved by adding meta-models for new platform language by the developers (in the same way meta-models for Java/AspectJ language is provided). It also requires changes made in code to implementation model transformation. In order to extend the specifications to add specifications for the new types of the runtime feature dependencies the feature modelling needs to be extended by the developer of the feature model (i.e., DOFM DSL in Chapter 4).

Figure 5-6 shows the extensibility when new platform language is introduced to the product line core artefacts. The elements that are added or modified as a result of adding a new platform language could be; In the dashed line box (1) the meta-model for the new platform language, the code to model transformation.

When modifying/adding new specification the elements to be modified are; in the dashed line box (2) the DOFM modified meta-model. Adding new specifications for dependencies results in adding/modifying new patterns in the pattern model (dashed line box (3)). Both adding/modifying the DOFM and Pattern model requires addition/modification of respective constraints applied on.

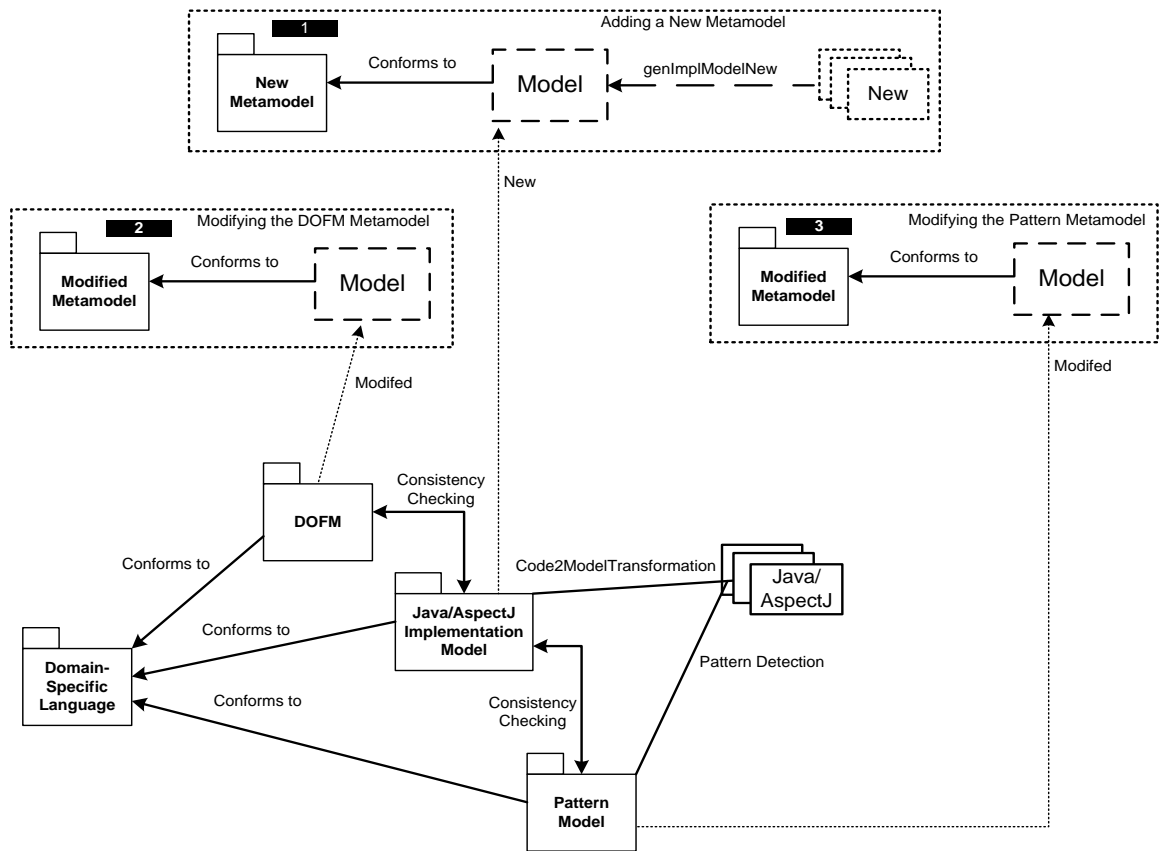


Figure 5-6 Extensibility of the Domain

5.6.2 Scalability

Scalability is a capability of the software to handle future growth in an efficient manner. One of the important outcome of the formal evaluation based on the interviews is to scale the propose approach on a large product line. Scalability is one of the very important challenges of the proposed technique to taken in to consideration the exponential numbers of runtime feature dependencies. By applying the AspectJ-based patterns in the source code the implementation became modular and easy to understand. Since the proposed technique is only considering the AspectJ-based patterns. It is not considering the whole product line source code but only focusing on the AspectJ pattern-based implementations in the source code. Hence focusing on certain parts of the implementation makes the proposed technique scalable. In the SciCalc product line case study 61 AspectJ classes in 48 Java Packages were considered for detecting runtime dependencies in the total of 324 Java/AspectJ classes.

The specification of the dependencies using the DOFM modelling language however requires a human effort to specify number of features and dependencies among them. The number of AO patterns implemented in the product line may affect the performance of the pattern detection process (i.e., it might get slow with large number of patterns in the implementation). The scalability of Epsilon validation language (EVL) constraints applying on the inter-connected feature, implementation and pattern models depend on the number of the identified patterns in the pattern model.

5.6.3 Usability

Usability can be defined as an ease of use of a particular human made object that also contributes towards learning that particular object that can serve a particular need. During the formal evaluation process, one of the important questions was to get a feedback on the usability of the proposed approach. I managed to identify from the feedback given by the interviewees that the proposed solution is usable. The proposed technique provides a domain-specific language for specifying the runtime feature dependencies as DOFM, source code concepts in Java/AspectJ platform language as Implementation model, runtime feature dependencies concepts as pattern model and EVL constraints to detect the inconsistencies between the runtime feature dependencies specifications and their respective implementations.

DOFM or dependency-oriented feature model is a very simple model for the specification of the features and the types of dependencies between them. DOFM is not considered to be a full feature model as it doesn't facilitate to specify the mandatory, optional, feature groups using the DOFM. I am aware of the fact that the simplicity of the DOFM also implies that it lacks expressiveness. However my aim was to develop a language that can only specify the features and the dependencies among them along with the references to the implementation concepts in the implementation model. On the other side DOFM helps to focus on and query about certain parts of the product line (i.e., feature dependencies).

Implementation model is a language developed to extract and represent source code concepts at a higher level of abstraction. Implementation model is generated as a result of applying the code2model transformation plug-in on the product line features implementation source code. Implementation model is developed with the aim to extract information from the source code related to AspectJ-based patterns. Implementation model is considered to be very vital in inconsistency detection process.

Pattern model is a language developed to extract and represent the detected patterns in the source code. Pattern model language is developed with the aim to model each and every part of the pattern. Doing so helps to represent the specific pattern implementation on the model level. It is the pattern model which derives the inconsistency detection process as the detected pattern needs to be analyzed for possible contradictions with the specified type of pattern between a certain pair of features by application of EVL constraints.

5.6.4 Variability

Variability represents choices that one can make in order to perform a certain action. The proposed technique tool-supported solution can have various implementation choices. For instance when transforming the code into implementation model one can use Atlas Transformation Language¹³ (ATL) or Epsilon Object Language¹⁴ (EOL) despite of using Java abstract syntax tree parsing and pattern detection plug-in. When getting the feedback in terms of the error markers the proposed approach can use the Object Constraint Language¹⁵ (OCL) as a replacement to apply EVL constraints. Hence the approach is very flexible towards various choices that one can make to apply the proposed technique.

5.7 Threats to Validity

My validation is inspired by mutation testing approach showed that the proposed approach has supported the hypothesis (Section 5.2.1). However, there are a few threats to validity regarding my research work,

The required capabilities of the proposed approach are identified based on the research gap identified while performing the literature review. However, there might be some factors that need more importance than others.

Before applying the proposed approach it is mandated that the AO-design patterns is to be used in the product line feature implementation base for modularization of O&AFDs. This serves as a limitation of the proposed approach but at the same time extends the existing research work started by Lee et al.

Validation of the proposed approach is inspired by the mutation testing approach. It is due to the scarcity of the available case studies in research

¹³ <http://www.eclipse.org/atl/>

¹⁴ <http://www.eclipse.org/epsilon/doc/eol/>

¹⁵ <http://www.omg.org/spec/OCL/>

literature. Hence, the SciCalc case study was mutated in order to validate the proposed approach. It resolved the issue of having multiple case studies for the validation.

Literature Review strongly depends on a research question (i.e., approaches focusing on detection and analysis techniques of feature dependencies in the context of software product lines) and discussed according to factors identified in Section 2.4. Not much literature is available specially focusing on the consistency checking of runtime feature dependencies in the context of software product lines. It is possible to mitigate this threat to validity by providing a more systematic literature review (e.g., a systematic literature review). In order to make sure that the selection set is as big as possible yet focused, I have used terms like features interactions detection and management that facilitate me while searching for the approaches of interest.

5.8 Summary

This chapter presented validation and formal evaluation of the proposed technique. The chapter started by giving the validation strategy, set the validation hypothesis, provides an overview of the SciCalc case study and discuss the mutation testing-inspired validation approach. The chapter further argues why the validation approach is inspired by the mutation testing is adopted. The validation of each type of the runtime feature dependencies is performed and the results are discussed with the use of examples taken from the SciCalc product line. Discussion of the validation results is made with the recommendations to improve the proposed technique. Evaluation of the proposed approach is performed based on the formal interviewing of various experts in the software research and development. Formal evaluation process is discussed in detail along with the feedback generated by the interviewee candidates. Comparative evaluation of the proposed technique is made with respect to existing technique for analyzing and detecting the feature interactions discussed in Chapter Two. The chapter concludes with the qualitative analysis of the proposed technique with respect to the identified quality factors.

Chapter Six: Conclusions and Future Directions

6.1 Overview

The chapter summarizes the thesis, presents the key contributions of this thesis and limitations of the proposed technique (i.e., both theoretical and implementation). The chapter ends with an overview of the future work.

6.2 Summary of the Dissertation

In feature-oriented product lines, the functionalities of a product line are implemented as a set of features. Product line features are usually not independent in nature. They may depend on each other in order to provide a particular functionality. Understanding feature dependencies in product lines can help in understanding the composition of the product line features and behaviour when they are composed together.

During the verification of product line functionalities, feature dependencies should be taken into account for verification. Understanding feature dependencies is a critical challenge in product line asset development and maintenance, they have not been analyzed and documented explicitly (Silveira et al., 2007). Inability to manage feature dependencies at different levels of abstraction (i.e., between feature model and implementation levels) can lead to erroneous product configuration and derivation.

Structural and configuration dependencies are important but not sufficient for developing reusable and adaptable product line components (Lee and Kang, 2004). There are other types of feature dependencies (e.g., runtime dependencies and operational dependencies) that are responsible for implementing the behaviour of a derived product in a runtime environment (Lee and Kang, 2004).

Runtime feature dependencies are important, as they are responsible for implementing the runtime behaviour between the features implementations.

For functionally correct products, implemented RTFDs should also comply with their respective specifications apart of having various types of feature dependencies (i.e., implementations of structural and configuration feature dependencies) complying with their respective specifications. Work done by Lee et al., (Botterweck and Lee, 2009, Lee et al., 2009a, Lee et al., 2009c, Lee and Kang, 2004, Lee et al., 2006) categorized runtime feature dependencies and provided modular implementations by using a particular

type of design pattern called aspect-oriented design (AO) patterns using the AspectJ technology (Colyer et al., 2004).

This thesis extends the work started by Lee et al., by introducing the specification language (i.e., DOFM) and implementation model layers on top of the source code containing the AO pattern-based implementations. This research work was conducted in the context of the product line maintenance and evolution, where a product line engineer wants to inquire if a particular type of runtime feature dependency is implemented as intended (i.e., using a particular design pattern) in the source code and according to the respective specification.

The research focused on maintenance of the domain engineering assets. The following aspects are outside the scope of this research,

- Structural and configuration feature dependencies
- Application engineering activities like the product configuration, the assembly and the deployment
- Visualization techniques for model-driven product lines

In order to identify the research gap, I have conducted a literature review with the focus on identifying the current state-of-art for consistency checking of runtime feature dependencies in product line assets and managed to find 12 relevant approaches. The approaches were investigated with respect to factors of interest. The findings are discussed indicating that such types of runtime feature dependencies require a consistency checking approach to enable the product line engineer to understand them in the product line asset base. Moreover, many of the identified approaches are focusing on structural and configuration dependencies. The approaches like (Apel et al., 2011, Kaestner et al., 2007, Leich et al., 2005) deal with the feature interactions by encoding them and applying off-the-shelf model checking technology. However, they are not focusing on runtime feature dependencies.

The research prototype identifies semi-automatically check the important inconsistency scenarios discussed in Section 3.5. These include, 1) No runtime feature implementation identified, 2) Incompatibility between specified type and its respective implementation, 3) Incompatible specified features pair and its respective implementation(s), 4) Incomplete implementation of specification for a particular runtime feature dependency type, and 5) Implementation of runtime feature dependency not preserving the specified direction.

The thesis provides a model-driven engineering solution to detect inconsistencies between the specified runtime feature dependencies and their respective AO patterns-based implementations in the source code. The proposed technique follows the processes of, 1) code to implementation model transformation along with the pattern detection, 2) specification of runtime feature dependencies using the DOFM, and 3) applying a constraint language to identify inconsistencies along with providing the feedback to the product line engineer in the form of an error report.

The prototype developed in the Eclipse IDE supports the proposed technique of inconsistency detection of RTFDs in product line assets. The proposed technique can be classified as a static verification technique that performs static analysis of source code to perform the AO patterns detection and then performs model validation to identify inconsistencies related to RTFDs. The Eclipse IDE-based prototype tool comprises the following 1) a Domain-Specific Language (DSL) for modelling the product line artefacts, 2) a set of EMF plug-in to detect the AO pattern-based implementations in the source code, and 3) Epsilon Validation Language (Kolovos et al., 2011) constraints to provide a feedback related to detected inconsistencies as error markers.

Scarcity of suitable case studies to validate the proposed technique led mutation testing (Offutt, 1994) inspired technique to validate the proposed technique. In Mutation Testing (MT) the program under test is mutated. Mutation testing in practice evaluates the tests relevant to a particular source code. In theory, MT can be used to evaluate a new testing technique. In this research, it was used in parallel to the latter except the focus here is on evaluating the proposed technique (e.g., static analysis technique) against the mutated versions of the SciCalc product line. In the scope of thesis research, both the specifications of RTFDs and their respective pattern-based implementations can change. The choice of changing the specification along with implementation was made because of the fact that the specifications and their respective implementations are dealt with by various stakeholders in a distributed environment. Specification change is called a specification mutant and implementation change is called an implementation mutant. Following hypothesis have been established for validation purposes:

Validation Hypothesis: The solution sufficiently enables the product line engineer to identify inconsistencies in the domain engineering assets of a product line (i.e., the specified runtime feature dependencies and their respective aspect-oriented pattern based implementations).

Validation of the proposed technique was performed by applying it to the mutated versions of the SciCalc product line case study. The success of the proposed technique is its ability to detect the change in both the specification and implementation of runtime feature dependencies (i.e., definition of mutant is killed) the more is proposed technique is considered to be successful. A total number of 100 mutants (both specification and implementation mutants) were created and used to validate the proposed technique reaching an average mutation coverage of 91.3%. It is to be noted that only known mutants are used for validation purposes. The mutation coverage may drastically reduce with the introduction of new complicated mutants in the validation process. The qualitative analysis of the proposed technique is performed with respect to the quality factors (i.e., extensibility, scalability, usability and variability factors).

To identify the correctness of the set hypothesis a formal evaluation process was performed and series of interviews were conducted. Feedback was obtained on my research work from various experts, researchers, software engineers and architects from software research and development industry. A total of eight very highly skilled interviewees were selected from various parts of software industry (i.e., experts/project leaders in MDD, architects in telecommunication industry and software engineers working in cloud computing and multimedia research and development).

6.3 Contributions

The main contributions of the thesis are as follows,

- An extension of the work started by Lee et al., (Botterweck and Lee, 2009, Lee et al., 2009a, Lee et al., 2009c, Lee and Kang, 2004, Lee et al., 2006) on modularizing the runtime feature dependencies in the context of software product lines.
- A comprehensive literature review of state-of-art attempting to detect and resolve feature dependencies related inconsistencies.
- A validation of the runtime feature dependencies using AO patterns-based on AspectJ technology in the context of a "*Scientific Calculator*" (Lee et al., 2009a, Cho et al., 2008) product line was done.
- A set of runtime feature dependencies detection algorithms is developed.
- A modelling language using EMF technology for runtime feature dependencies pattern along with the DOFM modelling language to specify runtime feature dependencies was provided.

- A set of validation rules developed in Epsilon validation language (EVL) in order to facilitate the product line engineer in managing of the domain engineering assets by identifying important inconsistency scenarios.

6.4 Limitations

This section discusses limitations of the proposed technique.

The proposed static analysis technique is focusing on the structural analysis of the AspectJ pattern-based implementations against their respective specifications. The technique is not taking into account how the implementation will AspectJ implementation executes during the runtime but statically checking for the patterns and run analysis against their respective specifications. However, I am confident that if the approach is able to detect the pattern it can facilitate the product line to manage a certain RTFD in the source code.

The proposed technique works in the presence of the AO pattern-based solutions implemented in the product line. I am aware that if there exists a legacy code with no AspectJ pattern-based implementation the proposed technique might not work.

The DOFM modelling language does not replace the feature model, which is still required to express the structural and configuration constraints between features. Using a DOFM modelling language can enable the product line engineer to focus on the relevant parts/views of the product line assets and provides an ease to extend the DOFM language.

The theoretical contributions serve as a set of guidelines to allow the product line engineer towards semi-automatically analyzing runtime feature dependencies in a particular product line. The proposed approach provides a RTE functionality that enables the product line engineer to maintain the RTFDs against their respective specifications.

The approach is validated utilizing the single case study “Scientific Calculator Product Line” established in the work by Lee et al. (Botterweck and Lee, 2009, Lee et al., 2009a, Lee et al., 2009c, Lee and Kang, 2004, Lee et al., 2006). However, the tests generated by mutation of the SciCalc product line provided tried to mitigate this limitation.

The AO patterns detection was performed within particular packages that contain the AO pattern-based implementations of runtime feature dependencies. Hence, in the presence of an AO pattern-based implementation the pattern detection algorithms work fine but not having an

AO pattern-based solution in this situation the tool support will fail to identify a particular AO pattern.

The identified inconsistencies by applying the EVL constraints serve as guidelines to identify the inconsistencies in a particular product line, but it doesn't provide the different solution options (e.g., quick fixes in Eclipse IDE) to resolve the identified inconsistency related to runtime feature dependencies in the product line artefacts (e.g., feature model or in implementation).

The AO pattern-based patterns acted as an input for designing the AO pattern detection algorithms. The algorithms are following the static analysis of the source code technique.

6.5 Future Work

This section discusses the future work and possibilities of improving the proposed approach

6.5.1 Further Case studies

Only Scientific calculator product line was utilized for the proposed technique validation inspired by the mutation testing technique. It is on the agenda to investigate further case studies and validate the approach. Based on the feedback I plan to improve the proposed technique.

6.5.2 Improving Pattern Detection Algorithms

One potential future work direction is to improve the AO-patterns detection algorithms to reduce the false positives and false negatives.

6.5.3 Dynamic Analysis of RTFDs

The proposed technique can be classified as a static verification technique. It performs the structural analysis to identify inconsistency scenarios discussed in Section 3.5. At the moment the proposed technique does not execute the RTFD implementations and perform the semantic inconsistency identification. One potential future direction is to perform inconsistency after executing the RTFDs implementation, perform the pattern detection and then identify inconsistency related to RTFDs.

6.5.4 Scalability of the Approach

Improving the proposed approach scalability is also a potential future work. At the moment the proposed technique is validated using a simple case study. Hence, scalability of the approach is still an open question.

6.5.5 Providing Graphical User Interfaces

A future work direction could be to apply better visualization techniques to improve the proposed technique by visually viewing the inconsistencies on the model level. At the moment the proposed technique has structural editor views to specify the runtime feature dependencies, view the implementation and pattern models. In future I try to explore the visualization techniques to improve the proposed technique efficiency.

6.5.6 Model to Code Inconsistency Traceability and Vice Versa

One potential future direction could be to identify the inconsistency on the model level and then trace the inconsistency to source code part (i.e., classes, functions, etc.) having the inconsistency and vice versa.

6.5.7 Comparison with new Approaches

The proposed approach tries to fill the gap identified as a result of comprehensive literature review of the state-of-the-art approaches try to detect and resolve the inconsistencies related to feature dependencies in the product lines domain. It is on the agenda to compare with the research work on architecture consistent with the tool support called JITTAC started by Buckley et al. (2013). In future further approaches will be investigated for comparison purposes that will help to improve the proposed technique.

6.5.8 Traceability and Reverse Engineering

Traceability is the domain yet to be explored. One possible future direction could be to explore that how much efficiency can be gained as a result of improving the traceability between the modelled product line artefacts. The proposed technique is using the reverse engineering mechanism to develop the implementation model and pattern detection. Identifying a scalable reverse engineer mechanism to generate industrial size implementation models from the large source code.

6.5.9 Extending the Domain

For the implementation of the proposed technique only runtime feature dependencies were selected. One potential future direction could be to apply the proposed technique in other domains (e.g., telecommunication) to check the feasibility of the approach.

REFERENCES

- ABGAZ, Y., JAVED, M. & PAHL, C. 2012. Dependency Analysis in Ontology-Driven Content-Based Systems. *In: RUTKOWSKI, L., KORYTKOWSKI, M., SCHERER, R., TADEUSIEWICZ, R., ZADEH, L. & ZURADA, J. (eds.) Artificial Intelligence and Soft Computing*. Springer Berlin Heidelberg.
- APEL, S., SPEIDEL, H., WENDLER, P., RHEIN, A. V. & BEYER, D. 2011. Feature-Aware Verification. *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Lawrence, Kansas: IEEE Computer Society.
- BATORY, D., HOFNER, P. & KIM, J. 2011. Feature interactions, products, and composition. *Proceedings of the 10th ACM international conference on Generative programming and component engineering*. Portland, Oregon, USA: ACM.
- BEHRENS, H., CLAY, M., EFFTINGE, S., EYSHOLDT, M., FRIESE, P., KÖHNLEIN, J., WANNHEDEN, K., ZARNEKOW, S. & CONTRIBUTORS 2010. Validation with the Check Language. *Xtext User Guide*.
- BENCOMO, N. 2008. *Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability*. PhD, Lancaster University.
- BIRKNER, M. 2007. *Object-Oriented Design Pattern Detection Using Static and Dynamic Analysis in Java Software*. Masters Master Thesis, University of Applied Sciences Bonn-Rhein-Sieg Sankt Augustin, Germany.
- BOTTERWECK, G. & LEE, K. 2009. Feature Dependencies have to be Managed Throughout the Whole Product Life-cycle. *Software Engineering (Workshops)*.
- BOTTERWECK, G., LEE, K. & THIEL, S. Automating product derivation in software product line engineering. *Software Engineering (SE 2009)*, 2009 Kaiserslautern, Germany.
- BOTTERWECK, G., THIEL, S., NESTOR, D., ABID, S. B. & CAWLEY, C. 2008. Visual Tool Support for Configuring and Understanding Software Product Lines. *International Software Product Lines Conference (SPLC'08)*. Limerick, Ireland: IEEE Computer Society.
- BUCKLEY, J., MOONEY, S., ROSIK, J. & ALI, N. 2013. JITTAC: a just-in-time tool for architectural consistency. *Proceedings of the 2013 International Conference on Software Engineering*. San Francisco, CA, USA: IEEE Press.
- CAWLEY, C., BOTTERWECK, G., HEALY, P., ABID, S. B. & THIEL, S. 2009. A 3D Visualisation to Enhance Cognition in Software Product Line Engineering. *5th International Symposium on Visual Computing (ISVC09)*. Las Vegas, Nevada, USA.
- CHO, H., LEE, K. & KANG, K. C. 2008. Feature Relation and Dependency Management: An Aspect-Oriented Approach. *12th International Software Product Line Conference (SPLC'08)*, 2008 Limerick, Ireland. 3-11.

- CICHÝ, M. & JAKUBÍK, J. 2008. Design Patterns Identification Using Similarity Scoring Algorithm with Weighting Score Extension. *Proceedings of the 2008 conference on Knowledge-Based Software Engineering: Proceedings of the Eighth Joint Conference on Knowledge-Based Software Engineering*. IOS Press.
- CLASSEN, A. 2007. Problem-Oriented Feature Interaction Detection in Software Product Lines. *Ninth International Conference on Feature Interactions in Software and Communication Systems (ICFI'07)*. Grenoble, France.
- CLEMENTE, M. P. J., HERNÁNDEZ, J., MURILLO, J. M., PÉREZ, M. A. & SÁNCHEZ, F. 2003. Component based system design an composition: an aspect oriented approach. In: LAU, K.-K. (ed.) *Component based software development: Case Studies*. University of Manchester: World Scientific.
- CLEMENTS, P. & NORTHROP, L. M. 2002. *Software Product Lines: Practices and Patterns*, Boston, Addison-Wesley.
- COLLOFELLO, J. S. & INSTITUTE, C.-M. U. S. E. 1988. *Introduction to Software Verification and Validation*, Carnegie Mellon University, Software Engineering Institute.
- COLYER, A., CLEMENT, A., HARLEY, G. & WEBSTER, M. 2004. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison-Wesley Professional.
- CZARNECKI, K. 1998. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD.
- DAIZHONG LUO, S. D. 2009. Feature Dependency Modeling for Software Product Lines. *International Conference on Computer Engineering and Technology (ICCET'09)*. Chengdu, Sichuan, China.
- DASHOFY, E. M., HOEK, A. V. D. & TAYLOR, R. N. 2002. An infrastructure for the rapid development of XML-based architecture description languages. *Proceedings of the 24th International Conference on Software Engineering*. Orlando, Florida: ACM.
- FERBER, S., HAAG, J. & SAVOLAINEN, J. Feature Interaction and dependencies: Modeling Features for Reengineering a Legacy Product Line. Second International Conference on Software Product Lines (SPLC'02), 2002 San Diego, CA, USA. 235 - 256.
- FILHO, R. S. S. & REDMILES, D. F. 2005. Striving for versatility in publish/subscribe infrastructures. *Proceedings of the 5th international workshop on Software engineering and middleware*. Lisbon, Portugal: ACM.
- GIBSON, J. P. 1997. *Feature Requirements Models: Understanding Interactions*, Montreal, Canada, IOS Press.
- GRONBACK, R. C. 2009. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, Addison-Wesley Professional.
- HASSELBRING, W. 2011. Reverse engineering of dependency graphs via dynamic analysis. *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. Essen, Germany: ACM.

- HEUZEROTH, D., HOLL, T., HÖGSTRÖM, G. & LÖWE, W. 2003. Automatic Design Pattern Detection. *Proceedings of the 11th IEEE International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society.
- JAYARAMAN, P. K., WHITTLE, J., ELKHODARY, A. M. & GOMAA, H. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. 10th International Conference on Model Driven Engineering Languages and Systems(MODELS'07), 2007 Nashville, USA. 151-165.
- JÉZÉQUEL, J.-M. 2012. Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering*, 2012, 24.
- KAESTNER, C., APEL, S. & BATORY, D. A Case Study Implementing Features Using AspectJ. 11th International Software Product Line Conference 2007 (SPLC'07), 2007 Washington, DC, USA. 223-232.
- KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E. & PETERSON, A. S. 1990. Feature Oriented Domain Analysis (FODA) Feasibility Study. *In: INSTITUTE, S. E. & UNIVERSITY, C. M. (eds.)*. Pittsburgh, Pennsylvania.
- KANG, K. C., LEE, J. & DONOHOE, P. 2002. Feature Oriented Product Line Software Engineering: Principles and Guidelines. *IEEE Transactions on Software Engineering*, 19, 58-65.
- KIM, C. H. P., KEASTNER, C. & BATORY, D. 2008. On the modularity of feature interactions. *Proceedings of the 7th international conference on Generative programming and component engineering*. Nashville, TN, USA: ACM.
- KITCHENHAM, B., BRERETON, O. P., BUDGEN, D., TURNER, M., BAILEY, J. & LINKMAN, S. 2009. Systematic literature reviews in software engineering - A systematic literature review. *Inf. Softw. Technol.*, 51, 7-15.
- KNODEL, J., MUTHIG, D., NAAB, M. & LINDVALL, M. 2006. Static Evaluation of Software Architectures. *Software Maintenance and Reengineering (CSMR'06)*. Bari, Italy.
- KOLOVOS, D., ROSE, L. & PAGE, R. 2011. Epsilon Validation Language (EVL). *The Epsilon Book*.
- KOLOVOS, D., ROSE, L., PAIGE, R. & GARCIA-DOMINGUEZ, A. 2012a. The Epsilon Object Language (EOL). *The Epsilon Book*.
- KOLOVOS, D., ROSE, L., PAIGE, R. & GARCIA-DOMINGUEZ, A. 2012b. The Epsilon Validation Language (EVL). *The Epsilon Book*.
- LEE, K., BOTTERWECK, G. & THIEL, S. Aspectual Separation of Feature Dependencies for Flexible Feature Composition. 33rd Annual IEEE International Computer Software and Applications Conference, 2009a Seattle, Washington, USA. 45-52.
- LEE, K., BOTTERWECK, G. & THIEL, S. Feature-Modeling and Aspect-Oriented Programming: Integration and Automation. SNPD, 2009b. 186-191.
- LEE, K., BOTTERWECK, G. & THIEL, S. 2009c. Feature-Modeling and Aspect-Oriented Programming: Integration and Automation. *10th ACIS International Conference on Software Engineering, Artificial*

- Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD '09.* Daegu.
- LEE, K. & KANG, K. 2004. *Feature Dependency Analysis for Product Line Component Design* Berlin Heidelberg, Springer-Verlag Berlin Heidelberg.
- LEE, Y., YANG, C., ZHU, C. & ZHAO, W. An approach to Managing Feature Dependencies for Product Releasing in Software Product Lines. 9th International Conference on Software Reuse (ICSR'06), 2006 Turin, Italy. 127-141.
- LEICH, T., APEL, S., ROSENMUELLER, M. & SAAKE, G. Handling Optional Features in Software Product Lines. 2nd Managing Variabilities Consistently in Design and Code (MVCD2) at 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), 2005.
- LIENHARD, A., GREEVY, O. & NIERSTRASZ, O. Tracking Objects to Detect Feature Dependencies. 15th IEEE International Conference on Program Comprehension, 2007(ICPC '07), 2007 Banff, Alberta, BC. 59 - 68.
- LUO, D. & DIAO, S. Feature Dependency Modeling for Software Product Line. International Conference on Computer Engineering and Technology, 2009 Riverview Hotel, Singapore. 256-260.
- MENS, T. & STRAETEN, R. V. D. 2007. Incremental Resolution of Model Inconsistencies. In: SCIENCE, L. N. I. C. (ed.) *Recent Trends in Algebraic Development Techniques*. Springer Berlin/Heidelberg.
- MOSSER, S., PARRA, C., DUCHIEN, L. & BLAY-FORNARINO, M. 2012. Using domain features to handle feature interactions. *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. Leipzig, Germany: ACM.
- MUÑOZ, H. J. 2009. A Systematic Review on Feature Interaction in Software Product Line. In: LABS, R. (ed.). *Workshop in RiSE Labs: Reuse in Software Engineering (RiSE)*, Brazil.
- MURPHY, G. C., NOTKIN, D. & SULLIVAN, K. 1995. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20, 18-28.
- OFFUTT, A. J. A practical system for mutation testing: help for the common programmer. Test Conference, 1994. Proceedings., International, 1994. IEEE, 824-830.
- PARRA, C., CLEVE, A., BLANC, X. & DUCHIEN, L. 2010. Feature-based composition of software architectures. *Proceedings of the 4th European conference on Software architecture*. Copenhagen, Denmark: Springer-Verlag.
- PASSOS, L., TERRA, R., VALENTE, M. T., DINIZ, R. & MENDONCA, N. D. C. 2010. Static Architecture-Conformance Checking: An Illustrative Overview *IEEE Software*, 22, 7.
- PEARCE, D. J. & NOBLE, J. 2006. Relationship aspects. *Proceedings of the 5th international conference on Aspect-oriented software development*. Bonn, Germany: ACM.

- POHL, K., BÖCKLE, G. & LINDEN, F. V. D. 2005. *Software Product Line Engineering: Foundations, Principles and Technique*, New York.
- QAMAR, A. 2013. *Model and Dependency Management in Mechatronic Design*. degree of Doctor in Technology, KTH Royal Institute of Technology.
- SATYANANDA, T. K., LEE, D. & KANG, S. Formal Verification of Consistency between Feature Model and Software Architecture in Software Product Line. International Conference on Software Engineering Advances (ICSEA 2007), 2007 Cap Esterel, France. 1-10.
- SAVOLAINEN, J., OLIVER, I., MYLLÄRNIEMI, V. & MÄNNISTÖ, T. 2007. Analyzing and Re-structuring Product Line Dependencies. *31st Annual International Computer Software and Applications Conference*. Beijing, China.
- SEEMANN, J. & GUDENBERG, J. W. V. 1998. Pattern-based design recovery of Java software. *SIGSOFT Softw. Eng. Notes*, 23, 10-16.
- SENDALL, S. & KÜSTER, J. Taming Model Round-Trip Engineering. OOPSLA & GPCE Workshop on Best Practices for Model-Driven Software Development, 2004 Vancouver, Canada.
- SILVEIRA, R., FILHO, S. & REDMILES, D. F. 2007. Managing Feature interactions by documenting and enforcing dependencies in software product lines. *Feature Interactions in Software and Communication Systems IX (ICFI'07)*. Grenoble, France.
- WARMER, J. & KLEPPE, A. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison-Wesley Longman Publishing Co., Inc.
- WENDEHALS, L. & ORSO, A. 2006. Recognizing behavioral patterns atruntime using finite automata. *Proceedings of the 2006 international workshop on Dynamic systems analysis*. Shanghai, China: ACM.
- XUE, Y., XING, Z. & JARZABEK, S. Detection and Reconciliation of Feature Inconsistencies in a Family of Product Variants. 14th International Conference on Software Product Lines (SPLC'10), 2010 Jeju Island, South Korea. 1-15.
- YE, H. & LIU, H. 2005. Approach to modelling feature variability and dependencies in software product lines. *IEE Proceedings*, 152, 101-109.
- ZHANG, W., MEI, H. & ZHAO, H. 2005. A Feature-Oriented Approach to Modeling Requirements Dependencies. *Proceedings of the 13th IEEE International Conference on Requirements Engineering*. IEEE Computer Society.
- ZHANG, W., MEI, H. & ZHAO, H. 2006. Feature-driven requirement dependency analysis and high-level software design. *Requir. Eng.*, 11, 205-220.

APPENDIX A: Abstract Concepts of Epsilon Validation Language

The following sections will discuss some of the important EVL concepts of the applied constraints on the detected AO patterns.

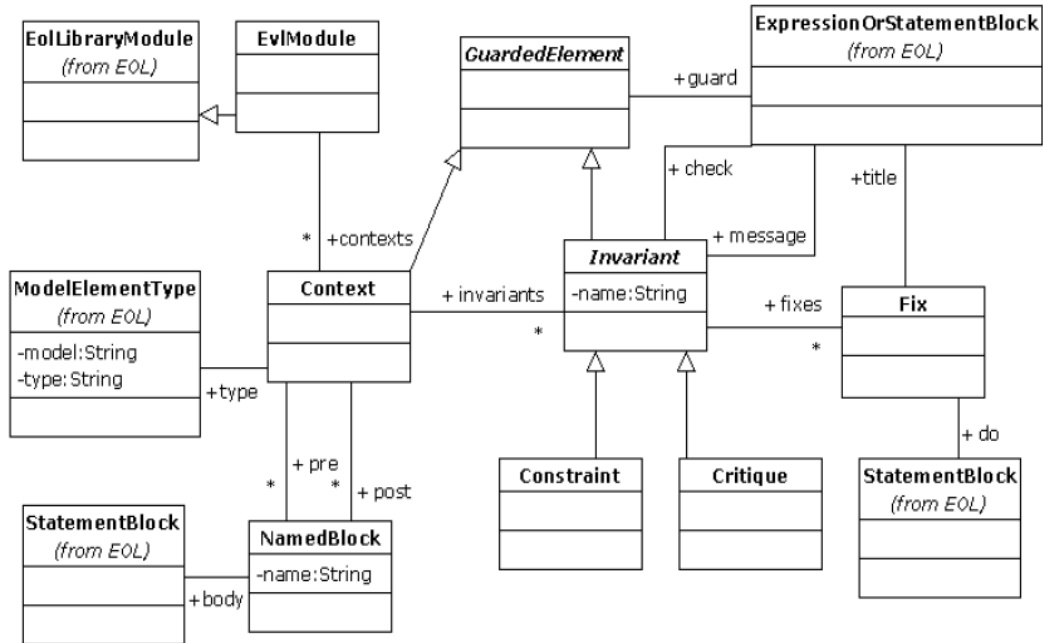


Figure A-1 Abstract Syntax of EVL (Kolovos et al., 2012b)

Abstract Syntax Concepts of EVL

Figure A-1 represents that EVLModule is “specialization of” EolLibraryModule which means that it can contain user-defined operations and import other epsilon object language (EOL) (Kolovos et al., 2012a) and EVL modules.

Context: “A context specifies the kind of instances on which the contained invariants (i.e., constraint or critique) will be evaluated. Each context can optionally define a guard which limits its applicability to narrow subset of instances of its specified type.” (Kolovos et al., 2012b). Figure B-2 sketches the concrete syntax of an EVL context.

```

context name {
    (guard (:expression) | ({statementBlock}))?
    (invariant)*
}

```

Figure A-2 Concrete Syntax of an EVL Context (Kolovos et al., 2012b)

Invariant: As in OCL, each of EVL invariant defines a name and a body (check). Figure A-3 sketches the concrete syntax of EVL invariant.

```

(@lazy)?
(constraint|critique) name {
    (guard (:expression) | ({statementBlock}))?
    (check (:expression) | ({statementBlock}))?
    (message (:expression) | ({statementBlock}))?
    (fix)*
}

```

Figure A-3 Concrete Syntax of an EVL Invariant (Kolovos et al., 2012b)

Guard: “Guards are used to limit the applicability of invariants. It can be achieved on two levels. At context level it limits the applicability of all invariants of the context and at the invariants level it limits the applicability of a specific invariant” (Kolovos et al., 2012b).

Fix: “A fix defines a title using an *ExpressionOrStatementBlock* instead of a static String to allow users to specify context-aware titles (e.g. Rename class customer to Customer instead of a generic Convert first letter to upper case). Moreover, the do part is a statement block which fixing function can be defined using EOL” (Kolovos et al., 2012b). Figure A-4 sketches the concrete syntax of an EVL fix.

```
fix {  
  (guard (:expression) | ({statementBlock}))?  
  (title (:expression) | ({statementBlock}))?  
  do {  
    statementBlock  
  }  
}
```

Figure A-4 Concrete Syntax of an EVL Fix (Kolovos et al., 2012b)

Constraint: “Constraints in EVL are used to capture critical errors that invalidate the model” (Kolovos et al., 2012b).

Critique: “Unlike Constraints, Critiques are used to capture non-critical situations that do not invalidate the model, but should nevertheless be addressed by the user to enhance the quality of the model” (Kolovos et al., 2012b)

APPENDIX B: Mutation Testing-Inspired Validation

This Appendix provides a remaining validation testing results continued from Chapter Five for the proposed technique discussed in Chapter Four. Examples of each type of runtime feature dependencies (excluding the runtime required activation which is already discussed in Chapter Five) are discussed, mutants are created and validation is performed.

Mutation Testing of Runtime Excluded Activation Dependency- An Example Scenario

This section discusses the validation of the example between DelButton and ModeFM features. Figure B-1 shows an instance of the pattern model for modeling Runtime Excluded Activation Pattern that is detected by applying the pattern detection algorithm for detecting the runtime excluded activation pattern in the source code.

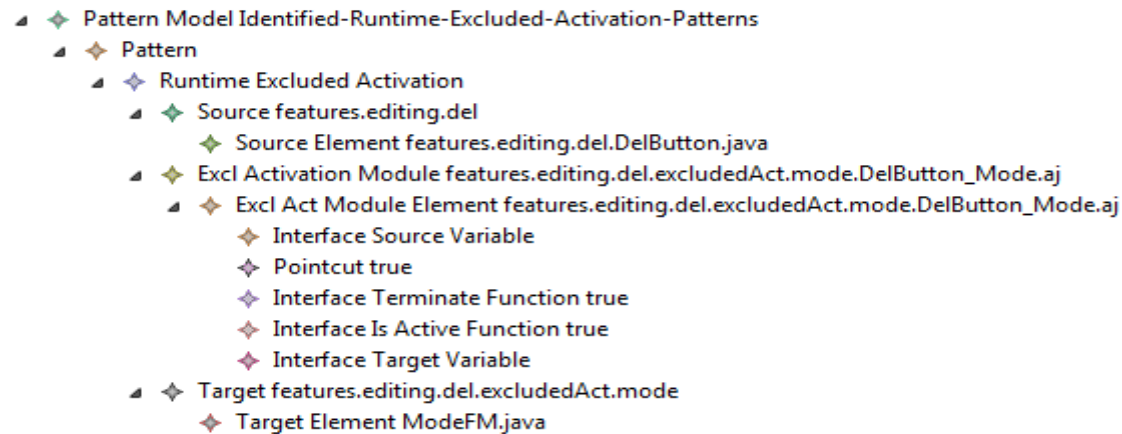


Figure B-1 Identified Runtime Ex-Act Dependency between “DelButton” and “ModeFM” Features

Figure B-2 shows a specified runtime excluded activation dependency between features DelButton and ModeFM in an XMI-form.

```
1. <featuremodel>
2. <rootFeatures name="DelButton" FeaturesPackageTrace="//@implmodel.0/@ownedMembers.5
3. FeatureComponentTrace="//@implmodel.0/@ownedMembers.5/@ownedMembers.2"/>
4. <rootFeatures name="Mode" FeaturesPackageTrace="//@implmodel.0/@ownedMembers.26
5. FeatureComponentTrace="//@implmodel.0/@ownedMembers.26/@ownedMembers.5"/>
6. <rootFeatures xsi:type="featuresmodel:RuntimeExcludedDependency" name="delbutton-RTexAct-Mode"
7. FeaturesPackageTrace="//@implmodel.0/@ownedMembers.6
8. FeatureComponentTrace="//@implmodel.0/@ownedMembers.6/@ownedMembers.0
9. sources="//@featuremodel.0/@rootFeatures.0"
10. targets="//@featuremodel.0/@rootFeatures.1"/>
11. </featuremodel>
```

Figure B-2 XMI-Based Feature Model Instance for Specifying “DelButton” and “ModeFM” Features and their Excluded Activation Dependency

```

1. package features.editing.del.excludedAct.mode;
2. import features.editing.del.DelButton;
3. import features.mode.ModeFM;
4. privileged aspect DelButton_Mode extends ExcludedAct<DelButton,ModeFM> {
5. before(DelButton db): execution (* actionPerformed(..) && target(db) && within (DelButton)
6. {source=db;}
7. pointcut serviceJP(DelButton d): execution(* actionPerformed(..) && this(d) && within
8. (DelButton);
9. before(ModeFM mf): execution (ModeFM.new()) && target(mf)
10. {target=mf;}
11. public void DelButton.terminate(){
12. System.out.println("Excluded Activation Called--DelButton.ExcludedAct.Mode");
13. getApplet().requestFocusInWindow();
14. return;}
15. public boolean ModeFM.isActive(){return (getMode()!=0); }
16. }

```

Figure B-3 Actual Runtime Excluded Activation Pattern-Based Implementation between “DelButton.java” and “ModeFM.java”

Mutant ID	Mutant Implementation	Expected Inconsistency	Actual Implementation	Response
IMPL1	Deleting the source part (line-6 in Figure B-3)	Pattern not found	Source=db	Pattern not found
IMPL2	Deleting the target part (line-9 in Figure B-3)		Target=mf	
IMPL3	DelButton.Terminate() not implemented (lines 10-13 in Figure B-3)		public void DelButton.terminate() {getApplet().requestFocusInWindow(); return;}	

IMPL4	ModeFM.isActive() not implemented (line-14 in Figure B-3)		public Boolean ModeFM.isActive() {return (getMode() != 0); }	
IMPL5	Pointcut ServiceJP not implemented (lines 7-8 in Figure B-3)		pointcut serviceJP(DelButton d): execution(* actionPerformed(..)) && this(d) && within (DelButton);	
IMPL6	Swap the parameters and Implementation ExcludedAct<ModeFM, DelButton>	Pattern found but Specified and implemented concepts contradicts	ExcludedAct<DelButton,ModeFM>	Pattern found but contradiction between specified and implemented concepts also Java Compiler gives an error
IMPL7	Replace target=mf to source=mf	Pattern not found	Target=mf	Pattern not found but already caught by the Java compiler
IMPL8	Replace source=db to target=mf	Pattern not found	Source=db	Pattern not found also Java compiler caught the error
IMPL9	Adding Terminate() for target (i.e., ModeFM.terminate()) in the implementation	Pattern found but ambiguous implementation	Implementation only contains DelButton.terminate()	Pattern found. But Solution didn't able to find the ambiguity
IMPL10	Adding isActive() for source (i.e., DelButton.isActive()) in the implementation	Pattern found but ambiguous implementation	Implementation only contains ModeFM.isActive()	Pattern found but Solution didn't able to find the ambiguity
IMPL11	Adding pointcut function advising ModeFM and not DelButton	Pattern found but pointcut defined on ModeFM and not DelButton	Implementation only contains pointcut for DelButton (lines 7-8)	Pattern found. Solution is able to detect that pointcut is defined on ModeFM and not DelButton. Compiler also gives an error

				because of intertype declaration conflicts.
IMPL12	Adding lines Or or= new Or(); Source=or; and commenting source=db (line-6 in Figure B-3)	Pattern found but source variable is not pointing towards DelButton.java else it is pointing to Or.java	Source=db	Pattern found. Java compiler gives an error: cannot convert from Or to ExcludeAct.S1 Interface.
IMPL13	Adding lines Or or= new Or(); Target=or; and commenting the target=mf (line-10 in Figure B-3)	Pattern found but target variable is not pointing towards ModeFM.java else it is pointing to Or.java	Target=fm	Pattern found. No Java Compiler error given. Error given by the solution is that target variable is point towards Or.java please check the pattern implementation.
IMPL14	Changing the target Parameters for Implementation ExcludedAct<DelButton,Or>	Pattern found between DelButton and Or	ExcludedAct<DelButton,ModeFM>	Pattern found. Java compiler also gives a TypeMismatch error. Solution gives an error that target element is Or.java and not ModeFM.java
IMPL15	Changing the source parameters for implementation ExcludedAct<Or,ModeFM>	Pattern found between Or and ModeFM	ExcludedAct<DelButton,ModeFM>	Pattern found. Java compiler also gives a TypeMismatch error. Solution gives an error that source element is Or.Java and not DelButton.java
Mutant ID	Mutant Specification	Expected Inconsistency	Actual Specification	Response

SPEC1	Source not specified in RuntimeExcludedActivation Dependency	Source not specified	sources="//@featuremodel.0/@rootFeatures.0" (line-9 in Figure B-2)	Source Feature not specified in RuntimeExcludedActivation
SPEC2	Target not specified in RuntimeExcludedActivation Dependency	Target not specified	targets="//@featuremodel.0/@rootFeatures.1"/> (line-10 in Figure B-2)	Target Feature not specified in RuntimeExcludedActivation
SPEC3	RuntimeExcludedActivation class reference to implementation Package not specified	RuntimeExcludedActivation reference missing to implementation	FeaturesPackageTrace="//@implmodel.0/@ownedMembers.6 (line-7 in Figure B-2)	RuntimeExcludedActivation to Implementation Package Trace not specified
SPEC4	RuntimeExcludedActivation class reference to Implementation Component (AspectJ or Java classes) not specified	Missing reference from RuntimeExcludedActivation to Implementation Component (AspectJ or Java classes)	FeatureComponentTrace="//@implmodel.0/@ownedMembers.6/@ownedMembers.0 (line-8 in Figure B-2)	Reference from RuntimeExcludedActivation to Implementation Component not specified
SPEC5	RuntimeExcludedActivation name not specified	RuntimeExcludedActivation name not specified	name="delbutton-RTexAct-Mode" (line-6 in Figure B-2)	RuntimeExcludedActivation dependency name is not specified
SPEC6	RuntimeExcludedActivation dependency source and targets are reversed (Source=ModeFM and Target=DelButton)	RuntimeExcludedActivation dependency pattern identified but contradiction between specified and implemented dependency source and target	sources="//@featuremodel.0/@rootFeatures.0 targets="//@featuremodel.0/@rootFeatures.1"/> (lines 9-10 in Figure B-2)	Specified Pattern is Implemented in Reverse Order

SPEC7	RuntimeExcludedActivation specified sources changed	RuntimeExcludedActivation on dependency source is different than identified pattern source	sources="//@featuremodel.0/@rootFeatures.0" (line-9 in Figure B-2)	Solution is able to detect specified source is different from identified pattern source
SPEC8	RuntimeExcludedActivation specified targets changed	RuntimeExcludedActivation onDependency target is different than identified pattern target	targets="//@featuremodel.0/@rootFeatures.1"/> (line-10 in Figure B-2)	Solution is able to detect specified target contradicts with the identified pattern target
SPEC9	Specified Dependency is not RuntimeExcludedActivation (e.g., Runtime Sequential Activation Dependency is specified)	Specified dependency is RuntimeSequential Activation whereas Identified dependency is RuntimeExcludedActivation	type="featuresmodel:RuntimeExcludedDependency" (line-6 in Figure B-2)	Solution is able to detect specified dependency type is different than identified dependency type

Table B-1 Validation of the Example Runtime Excluded Activation Example between DelButton and ModeFM

Mutation Testing of Runtime Sequential Activation Dependency- An Example Scenario

This section discusses the validation of the example between EqualsButton and BaseFM features. Figure B-4 shows an instance of the pattern model for modeling Runtime Sequential Activation Pattern that is detected by applying the pattern detection algorithm for detecting the runtime sequential activation pattern in the source code.

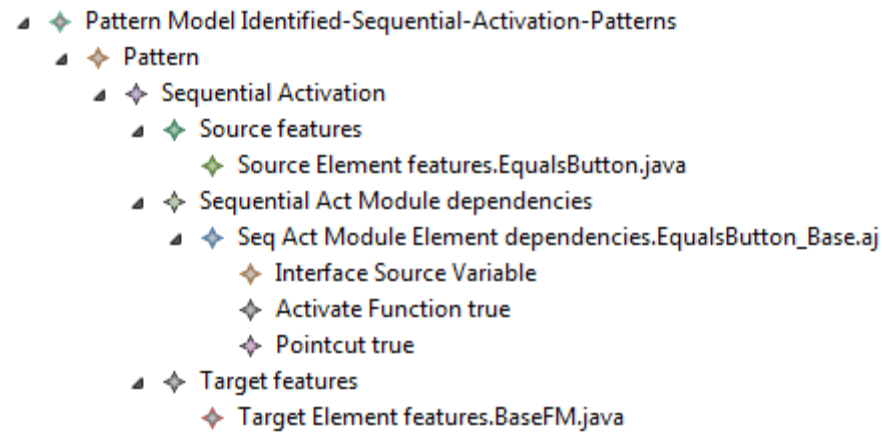


Figure B-4 Identified Runtime Sequential Activation Dependency between “EqualsButton” and “BaseFM” Features

Figure B-5 shows specification of the runtime sequential activation dependency between EqualsButton and BaseFM features using the DOFM language in a XMI-based format.

```
1. <featuremodel>
2. <rootFeatures name="EqualsButton" FeaturesPackageTrace="//@implmodel.0/@ownedMembers.2
3. FeatureComponentTrace="//@implmodel.0/@ownedMembers.2/@ownedMembers.3"/>
4. <rootFeatures name="BaseFM" FeaturesPackageTrace="//@implmodel.0/@ownedMembers.2
5. FeatureComponentTrace="//@implmodel.0/@ownedMembers.2/@ownedMembers.2"/>
6. <rootFeatures xsi:type="featuresmodel:RuntimeSequentialActivationDependency" name="EqualsButton-Seq Act-BaseFM"
7. FeaturesPackageTrace="//@implmodel.0/@ownedMembers.1"
8. FeatureComponentTrace="//@implmodel.0/@ownedMembers.1/@ownedMembers.1"
9. sources="//@featuremodel.0/@rootFeatures.0"
10. targets="//@featuremodel.0/@rootFeatures.1"/>
11. </featuremodel>
```

Figure B-5 XMI-Based Feature Model Instance for Specifying “EqualsButton” and “BaseFM” Features and their Sequential Activation Dependency

```
1. package dependencies;
2. import features.BaseFM;
3. import features.EqualsButton;
4. public aspect EqualsButton_Base extends SequentialAct<EqualsButton,BaseFM>{
5.   after(EqualsButton e): execution(EqualsButton.new()) && target(e) {
6.     source = e;}
7.   public void EqualsButton.activate() {
8.     EqualsButtonService();}
9.   pointcut completed(BaseFM BFM):execution(void BaseFMService()) && target(BFM);}
```

Figure B-6 Actual Runtime Sequential Activation Pattern-Based Implementation between “EqualsButton.java” and “BaseFM.java”

Mutant ID	Mutant Implementation	Expected Inconsistency	Actual Implementation	Response
IMPL1	Deleting the Source part of the pattern (line-6 in Figure B-6)	Pattern not identified	source=e;	Pattern not identified
IMPL2	EqualsButton.activate() not implemented (lines 7-8 in Figure B-6)		Public void EqualsButton.activate() {EqualsButtonService();}	
IMPL3	Pointcut Completed not implemented (line-9 in Figure B-6)		pointcut completed(BaseFM BFM):execution(void BaseFMService()) && target(BFM);}	
IMPL4	Swapping the Implementation Parameters in SequentialAct<BaseFM,EqualsButton>	Pattern Identified but different then specified source and target	SequentialAct<EqualsButton,BaseFM >	Pattern identified but contradicts with source and targets. Pattern is Implemented in oppositely.
IMPL5	Adding lines of code BaseFM b=new BaseFM(); and commenting source=e; (line-6 in Figure B-6)	Pattern Identified but different source element.	source=e;	Pattern identified but source element points towards BaseFM.java and not EqualsButton.java. Java compiler error Description Type mismatch: cannot convert from BaseFM to SequentialAct.C1Interface

IMPL6	Adding new line BaseFM b= new BaseFM(); and not commenting source=e;	Pattern Identified between EqualsButton.java and BaseFM.java. There is an ambiguity as there are two pointers to source elements EqualsButton.java and BaseFM.java	source=e;	Pattern identified between EqualsButton.java and BaseFM.java. Source element is different. Java compiler error Description Type mismatch: cannot convert from BaseFM to SequentialAct.C1Interface
IMPL7	Adding new ModeFM.activate() implementation	Pattern Identified but ambiguity as there are two implementations for activate() functions	Only EqualsButton.activate() implementation present	Pattern Identified between EqualsButton.java and ModeFM.java but unable to find ambiguity.
IMPL8	Adding pointcut for EqualsButton pointcut complete(EqualsButton eb): execution(void EqualsButtonService()) && target(eb);	Pattern identified but ambiguity as there are two pointcuts implemented for BaseFM and EqualsButton.	Only pointcut implementation for BaseFM.java is present	Pattern Identified between EqualsButton.java and BaseFM.java.
IMPL9	Changing source parameters (i.e., SequentialAct<M1,BaseFM>)	Pattern identified but between M1 and BaseFM and not between EqualsButton and BaseFM	SequentialAct<EqualsButton,BaseFM>	Pattern Identified between EqualsButton.java and M1.java. Specified source is different than identified pattern source.
IMPL10	Changing target parameters (i.e., SequentialAct<EqualsButton,M1>)	Pattern identified but between EqualsButton and M1 and not between EqualsButton and	SequentialAct<EqualsButton,BaseFM>	Pattern identified between EqualsButton and M1. Specified target is different than identified

		BaseFM		target.
IMPL11	Implementation different type of dependency type (i.e., RequiredAct<EqualsButton,BaseFM>	Pattern identified but there is a contradiction between specified and implemented dependency type.	SequentialAct<EqualsButton,BaseFM>	Pattern identified but of different type. Specified type is SequentialAct whereas implemented type is RequiredAct
Mutant ID	Mutant Specification	Expected Inconsistency	Actual Specification	Response
SPEC1	Name not specified (line-6 in Figure B-5)	Name not specified	name="EqualsButton-SeqAct-BaseFM"	Name not specified for runtime sequential activation dependency
SPEC2	Source not specified in SequentialActivation Dependency	Source not specified	sources="//@featuremodel.0/@rootFeatures.0"	Source is not specified for SequentialActivation dependency
SPEC3	Target not specified in SequentialActivation Dependency	Target not specified	targets="//@featuremodel.0/@rootFeatures.1"/>	Target is not specified for SequentialActivation dependency
SPEC4	SequentialActivation class reference to implementation Package not specified	SequentialActivation reference missing to implementation	FeaturesPackageTrace="//@implmodel.0/@ownedMembers.1"	SequentialActivation dependency reference to Package is missing.
SPEC5	SequentialActivation class reference to Implementation Component (AspectJ or Java classes) not specified	Missing reference from SequentialActivation to Implementation Component (AspectJ or Java classes)	FeatureComponentTrace="//@implmodel.0/@ownedMembers.1/@ownedMembers.1"	Reference from SequentialActivation to Implementation Component not specified
SPEC6	SequentialActivation dependency source and targets are reversed (Source=BaseFM and	SequentialActivation dependency pattern identified but contradiction between	sources="//@featuremodel.0/@rootFeatures.0 targets="//@featuremodel.0/@rootFea	Specified Pattern is Implemented in Reverse Order

	Target=EqualsButton)	specified and implemented dependency source and target	tures.1"/>	
SPEC7	SequentialActivation specified sources changed	SequentialActivation dependency source is different than identified pattern source	sources="//@featuremodel.0/@rootFeatures.0"	Solution is able to detect specified source is different from identified pattern source
SPEC8	SequentialActivation specified targets changed	SequentialActivationDependency target is different than identified pattern target	targets="//@featuremodel.0/@rootFeatures.1"/>	Solution is able to detect specified target contradicts with the identified pattern target
SPEC9	Specified Dependency is not SequentialActivation (e.g., Runtime Required Activation Dependency is specified)	Specified dependency is RuntimeRequiredActivation whereas Identified dependency SequentialActivation	xsi:type="featuresmodel:RuntimeSequentialActivationDependency"	Solution is able to detect specified dependency type is different than identified dependency type

Table B-2 Validation of the Example Runtime Sequential Activation Example between EqualsButton and BaseFM

Mutation Testing of Runtime Concurrent Activation Dependency- An Example Scenario

This section discusses the mutation testing example between M5 and M1 features. Figure B-7 shows an instance of the pattern model for modeling Runtime Concurrent Activation Pattern that is detected by applying the pattern detection algorithm for detecting the runtime concurrent activation pattern in the source code.

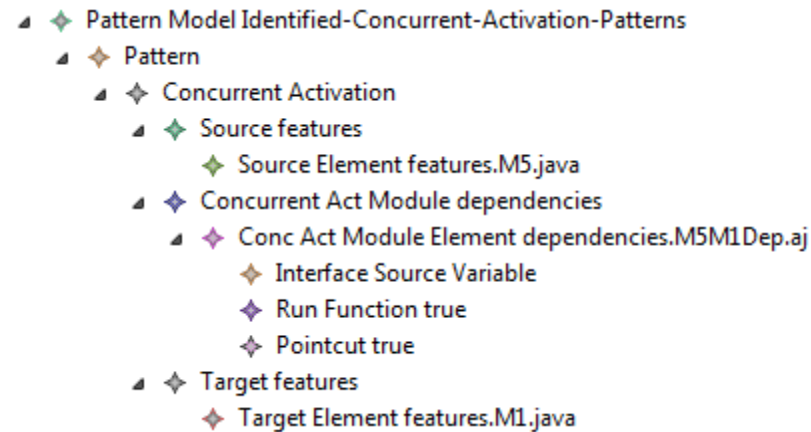


Figure B-7 Identified Runtime Concurrent Activation Dependency between “M5” and “M1” Features

Figure B-8 shows specification of the concurrent activation dependency between M5 and M1 features using the DOFM language discussed earlier in a XMI-based format.

```

1. <featuremodel>
2. <rootFeatures name="M1" FeaturesPackageTrace="//@implmodel.0/@ownedMembers.2
3. FeatureComponentTrace="//@implmodel.0/@ownedMembers.2/@ownedMembers.6"/>
4. <rootFeatures name="M5" FeaturesPackageTrace="//@implmodel.0/@ownedMembers.2
5. FeatureComponentTrace="//@implmodel.0/@ownedMembers.2/@ownedMembers.5"/>
6. <rootFeatures xsi:type="featuresmodel:RuntimeConcurrentActivationDependency" name="M5-ConAct-M1"
7. FeaturesPackageTrace="//@implmodel.0/@ownedMembers.1
8. FeatureComponentTrace="//@implmodel.0/@ownedMembers.1/@ownedMembers.5
9. sources="//@featuremodel.0/@rootFeatures.4"
10. targets="//@featuremodel.0/@rootFeatures.3"/>
11. </featuremodel>

```

Figure B-8 XMI-Based Feature Model Instance for Specifying “M5” and “M1” Features and their Concurrent Activation Dependency

```

1. package dependencies;
2. import features.M5;
3. import features.M1;
4. public aspect M5M1Dep extends ConcurrentAct<M5,M1>{
5. after(M1 m1):execution(M1.new()) && target(m1){
6. source = new M5();}
7. public void M5.run() {
8. service5();}
9. pointcut activate(M1 m1) :execution(void service1()) && target(m1);}

```

Figure B-9 Actual Runtime Concurrent Activation Pattern-Based Implementation between “M5.java” and “M1.java”

Mutant ID	Mutant Implementation	Expected Inconsistency	Actual Implementation	Response
IMPL1	Deleting the Source part of the pattern (line-6 in Figure B-9)	Pattern not identified	source=new M5();	Pattern not identified
IMPL2	M5.run() not implemented (lines 7-8 in Figure B-9)		public void M5.run() {service5();}	
IMPL3	Pointcut Completed not implemented (line-9 in Figure B-9)		pointcut activate(M1 m1) :execution(void service1()) && target (m1);}	
IMPL4	Swapping the Implementation Parameters in SequentialAct<M1,M5>	Pattern Identified but different then specified source and target	SequentialAct<M5,M1>	Pattern identified but contradicts with source and targets. Pattern is Implemented oppositely.
IMPL5	Adding lines of code M3 m=new M3(); and commenting source=new M5(); (line-6 in Figure B-9) source= <u>b</u> ;	Pattern Identified but different source element.	source=new M5();	Pattern identified but source element points towards M3.java and not M5.java. Java compiler error Type mismatch: cannot convert from M3 to SequentialAct.C1Interface
IMPL6	Adding lines of code M3 m=new M3(); and not commenting source=new M5(); (line-6 in Figure B-9)	Pattern Identified between M5.java and M1.java. There is an ambiguity as there are two pointers to source elements M5.java	source=new M5();	Pattern identified between M5.java and M1.java. Source element is different. Java compiler error Description Type mismatch: cannot convert from

		and M1.java		M3 to SequentialAct.C1Interface
IMPL7	Adding public void M1.run() {}	Pattern Identified but ambiguity as there are two implementations for run() functions	public void M5.run() {service5();}	Pattern Identified between EqualsButton.java and ModeFM.java but unable to find ambiguity.
IMPL8	pointcut Activate(M5 m5) :execution(void service5()) && target(m5);	Pattern Identified but ambiguity as there are two implementations for pointcut	pointcut activate(M1 m1) :execution(void service1()) && target(m1);}	Pattern Identified between M5 and M1 but unable to find ambiguity
IMPL9	Changing source parameters (i.e., ConcurrentAct<M4,M1>)	Pattern identified but between M4 and M1 and not between M5 and M1	ConcurrentAct<M5,M1>	Pattern Identified between M4.java and M1.java.Specified source is different than identified pattern source.
IMPL10	Changing target parameters (i.e., ConcurrentAct<M5,M2>)	Pattern identified but between M5 and M2 and not between M5 and M1	ConcurrentAct<M5,M1>	Pattern Identified between M5.java and M2.java.Specified source is different than identified pattern source.
IMPL11	Implementation different type of dependency type (i.e.,RequiredAct<M5,M1>)	Pattern identified but there is a contradiction between specified and implemented dependency type.	ConcurrentAct<M5,M1>	Pattern identified but of different type. Specified type is ConcurrentAct whereas implemented type is RequiredAct
Mutant ID	Mutant Specification	Expected Inconsistency	Actual Specification	Response
SPEC1	Name not specified (line-6 in Figure B-8)	Name not specified	name="M5-ConAct-M1"	Name not specified for runtime concurrent activation dependency

SPEC2	Source not specified in ConcurrentActivation Dependency	Source not specified	sources="//@featuremodel.0/@root Features.4"	Source is not specified for ConcurrentActivation dependency
SPEC3	Target not specified in ConcurrentActivation Dependency	Target not specified	targets="//@featuremodel.0/@root Features.3"/>	Target is not specified ConcurrentActivation dependency
SPEC4	ConcurrentActivation class reference to implementation Package not specified	ConcurrentActivation reference missing to implementation	FeaturesPackageTrace="//@implmodel.0/@ownedMembers.1	ConcurrentActivation dependency reference to Package is missing.
SPEC5	ConcurrentActivation class reference to Implementation Component (AspectJ or Java classes) not specified	Missing reference from ConcurrentActivation to Implementation Component (AspectJ or Java classes)	FeatureComponentTrace="//@implmodel.0/@ownedMembers.1/@ownedMembers.5	Reference from ConcurrentActivation to Implementation Component not specified
SPEC6	ConcurrentActivation dependency source and targets are reversed (Source=M1 and Target=M5)	ConcurrentActivation dependency pattern identified but contradiction between specified and implemented dependency source and target	sources="//@featuremodel.0/@root Features.4 targets="//@featuremodel.0/@root Features.3"/>	Specified Pattern is Implemented in Reverse Order
SPEC7	ConcurrentActivation specified sources changed	ConcurrentActivation dependency source is different than identified pattern source	sources="//@featuremodel.0/@root Features.4"	Solution is able to detect specified source is different from identified pattern source
SPEC8	ConcurrentActivation	ConcurrentActivation dependency target is	targets="//@featuremodel.0/@root	Solution is able to detect specified target

	specified targets changed	different than identified pattern target	Features.3"/>	is different from identified pattern target
SPEC9	Specified Dependency is not ConcurrentActivation (e.g., Runtime Required Activation Dependency is specified)	Specified dependency is RuntimeRequiredActivation whereas Identified dependency ConcurrentActivation	xsi:type="featuresmodel:RuntimeConcurrentActivationDependency"	Solution is able to detect specified dependency type is different than identified dependency type

Table B-3 Validation of the Example Runtime Concurrent Activation Dependency between “M5” and “M1” Features

Mutation Testing of Runtime Modification Dependency- An Example Scenario

This section discusses the validation of an example between OffButton and ShiftKey features. Figure B-10 shows an instance of the pattern model for modeling Runtime Modification Pattern that is detected by applying the pattern detection algorithm for detecting the runtime modification pattern in the source code.

- ▲ ◆ Pattern runtime feature dependency patterns
 - ▲ ◆ Runtime Modification Pattern
 - ▲ ◆ Left Module jscicalc.button
 - ◆ Left Module Element OffButton.java
 - ▲ ◆ Modification Module features.shiftKey.modification.off
 - ◆ Modification Module Element OffButton_Shift.aj
 - ▲ ◆ Right Module features.shiftKey
 - ◆ Right Module Element ShiftFM.aj

Figure B-10 Identified Runtime Modification Dependency between “OffButton” and “ShiftFM” Features

Figure B-11 shows specification of runtime modification dependency between ShiftKey and OffButton features using the DOFM language in a XMI-based format.

```
1. <featureModels name="scicalc-FM">
2. <rootFeatures name="ShiftKey" FeaturePackageTrace="//@implementationModels.0/@ownedMembers.48"/>
3. <rootFeatures name="OffButton" FeaturePackageTrace="//@implementationModels.0/@ownedMembers.58"/>
4. <rootFeatures xsi:type="spdsl.features:RuntimeModificationDependency" name="Off.modification.ShiftKey"
5. FeaturePackageTrace="//@implementationModels.0/@ownedMembers.53"
6. sources="//@featureModels.0/@rootFeatures.1"
7. targets="//@featureModels.0/@rootFeatures.0"/>
8. </featureModels/>
```

Figure B-11 XMI-Based Feature Model Instance for Specifying “ShiftKey” and “OffButton” Features and their Runtime Modification Dependency

```
1. package features.shiftKey.modification.off;
2. import jscicalc.button.OffButton;
3. import features.shiftKey.ShiftFM;
4. public privileged aspect OffButton_Shift {
5. before(OffButton button) :call(* updateDisplay(..) && this(button) && within(OffButton)
6. {ShiftFM.aspectOf().setShift(false);})
```

Figure B-12 Actual Runtime Modification Pattern-Based Implementation between “OffButton.java” and “ShiftFM.java”

Mutant ID	Mutant Implementation	Expected Inconsistency	Actual Implementation	Response
IMPL1	No AspectJ construct (i.e., before(OffButton)) implemented	No Pattern Found	before(OffButton button) :call(* updateDisplay(..) && this(button) && within(OffButton) {ShiftFM.aspectOf().setShift(false);})	No Pattern Found
IMPL2	setShift(false) not in the before(OffButton button).....{}		ShiftFM.aspectOf().setShift(false);	
IMPL3	Adding a new line ModeFM m= new ModeFM(); and commenting ShiftFM.aspectOf().setShift(false); (line-6 in Figure B-12)	Pattern Found but between OffButton and ModeFM implementations and not between OffButton and ShiftKey	ShiftFM.aspectOf().setShift(false);	Pattern Found but between OffButton and ShiftKey
IMPL4	Deleting before(OffButton button):.....{.....} and implementing before(ShiftFM shiftfm):.....{.....}	Pattern not found	Before(OffButton button):.....{.....}	Pattern not found Since identified Source and target implementations are pointing towards the same AspectJ class (ShiftFM.aj)
Mutant ID	Mutant Specification	Expected Inconsistency	Actual Specification	Response
SPEC1	Pattern Name not specified. name="" (line-4 in Figure B-11)	Name not specified	name="Off.modification.ShiftKey" (line-4 in Figure B-11)	Name not specified for RuntimeModification Dependency

SPEC2	RuntimeModificationDependency FeaturePackageTrace not defined (i.e., FeaturePackageTrace=")	RuntimeModificationDependency Reference to Package in the Implementation missing	FeaturePackageTrace="//@implementationModels.0/@ownedMembers.53" (line-5 in Figure B-11)	FeaturePackageTrace missing for "Off.modification.Shiftkey" dependency
SPEC3	Source not specified in RuntimeModificationDependency (i.e., sources=")	Source is not specified in the RuntimeModificationDependency	sources="//@featureModels.0/@rootFeatures.1" (line-6 in Figure B-11)	Source not specified in 'Off.modification.shiftkey' dependency
SPEC4	Target not specified in RuntimeModificationDependency (i.e., targets=")	Target is not specified in the RuntimeModificationDependency	targets="//@featureModels.0/@rootFeatures.0"/> (line-7 in Figure B-11)	Target not specified in 'Off.modification.shiftkey' dependency
SPEC5	Swapping the source and target specification (i.e., source=ShiftFM and target=OffButton)	Pattern Identified but implemented in opposite direction.	sources="//@featureModels.0/@rootFeatures.1 targets="//@featureModels.0/@rootFeatures.0"/> (lines 6-7 in Figure B-11)	Pattern Identified but implemented oppositely as specified source is equals to identified target and specified target is equals to identified source.
SPEC6	Specifying wrong type of dependency other than runtime modification (e.g., RuntimeRequiredActivation dependency)	Specified pattern Type is contradicts with identified pattern type	xsi:type="spldsl.features:RuntimeModificationDependency" (line-4 in Figure B-11)	Specified Type is runtime required activation whereas identified type is runtime modification pattern
SPEC7	Change the source feature reference to some other	Pattern identified between Or and ShiftFM hence	sources="//@featureModels.0/@rootFeatures.1" (line-6 in Figure B-11)	Source specified in 'Off.modification.shiftkey' is

	feature other than OffButton (e.g., set source reference to Or.java)	contradiction between specified source in RuntimeModification and identified source in the detected pattern.		different than identified source in detected RuntimeModification dependency Pattern.
SPEC8	Change the target feature reference to some other feature Other than ShiftFM (e.g., set target reference to Or.java)	Pattern identified between OffButton and Or hence contradiction between specified target in RuntimeModification and identified target in the detected pattern.	targets="//@featureModels.0/@rootFeatures.0"/> (line-7 in Figure B-11)	target specified in 'Off.modification.shiftkey' is different than identified target in detected RuntimeModification dependency Pattern (i.e., ShiftFM).

Table B-4 Validation of the Example Runtime Modification Dependency between "Shiftkey" and "OffButton"

Appendix C: Pattern Detection Algorithms Implementation for Parsing of AspectJ Implementation Using Java (Source Code)

```
package code2model.builder;

import java.util.Iterator;

import java.util.Stack;

import org.eclipse.emf.ecore.EObject;

import org.eclipse.jdt.core.IJavaElement;

import org.eclipse.jdt.core.dom.ASTNode;

import org.eclipse.jdt.core.dom.ASTVisitor;

import org.eclipse.jdt.core.dom.AnonymousClassDeclaration;

import org.eclipse.jdt.core.dom.ArrayAccess;

import org.eclipse.jdt.core.dom.Assignment;

import org.eclipse.jdt.core.dom.ClassInstanceCreation;

import org.eclipse.jdt.core.dom.CompilationUnit;

import org.eclipse.jdt.core.dom.ConstructorInvocation;

import org.eclipse.jdt.core.dom.Expression;

import org.eclipse.jdt.core.dom.ExpressionStatement;

import org.eclipse.jdt.core.dom.FieldAccess;

import org.eclipse.jdt.core.dom.IBinding;

import org.eclipse.jdt.core.dom.IMethodBinding;

import org.eclipse.jdt.core.dom.ITypeBinding;

import org.eclipse.jdt.core.dom.IVariableBinding;

import org.eclipse.jdt.core.dom.InstanceOfExpression;

import org.eclipse.jdt.core.dom.MethodDeclaration;

import org.eclipse.jdt.core.dom.MethodInvocation;

import org.eclipse.jdt.core.dom.MethodRef;

import org.eclipse.jdt.core.dom.Name;

import org.eclipse.jdt.core.dom.ParameterizedType;

import org.eclipse.jdt.core.dom.ParenthesizedExpression;

import org.eclipse.jdt.core.dom.QualifiedName;
```

```

import org.eclipse.jdt.core.dom.SimpleName;
import org.eclipse.jdt.core.dom.SimpleType;
import org.eclipse.jdt.core.dom.SingleVariableDeclaration;
import org.eclipse.jdt.core.dom.SuperConstructorInvocation;
import org.eclipse.jdt.core.dom.SuperFieldAccess;
import org.eclipse.jdt.core.dom.SuperMethodInvocation;
import org.eclipse.jdt.core.dom.ThisExpression;
import org.eclipse.jdt.core.dom.TypeDeclaration;
import org.eclipse.jdt.core.dom.TypeLiteral;
import org.eclipse.jdt.core.dom.TypeParameter;
import org.eclipse.jdt.core.dom.VariableDeclaration;
import org.eclipse.jdt.core.dom.VariableDeclarationExpression;
import org.eclipse.jdt.core.dom.VariableDeclarationFragment;
public class CompilationUnitParser extends ASTVisitor {
    private Stack<IJavaElement> stack = new Stack<IJavaElement>();
    StringBuffer buffer = new StringBuffer();
    boolean source_variable=false;
    boolean target_variable=false;
    boolean Interface_IsActive=false;
    boolean Interface_terminate=false;
    boolean Interface_Run=false;
    boolean Interface_Activate=false;
    boolean pointcut=false;
    boolean set_source=false;
    boolean set_target=false;
    String source_qualifiedname="";
    String target_qualifiedname="";
    String requiredactmodule_name="";
    String Source_element_qname="";
    String Target_element_qname="";
    String ReqActModule_element_qname="";
    String InterfaceSource_Variable_qname="";

```

```

String Interfacetarget_Variable_qname="";
String Pointcut_Parameter_qname="";
String Pattern_Type="";
String Concurrentactmodule_name="";
String Concurrentactmodule_element_qname="";
String Sequentialactmodule_name="";
String Sequentialactmodule_element_qname="";
SampleBuilder sb= new SampleBuilder();

public boolean visit(CompilationUnit node) {
    // TODO: This may not have a resource associated (unlikely though)!
    IJavaElement element = node.getJavaElement();
    stack.push(element);
    return true;
}

public void endVisit(CompilationUnit node) {
    stack.pop();

    if((source_variable==true) &&(target_variable==true)&& (Interface_IsActive==true)&&
(Interface_terminate==true)&& (pointcut==true) && (set_source==true) &&
(set_target==true))
    {
        requiredactmodule_name=node.getJavaElement().getParent().getElementName().toString()
;
        ReqActModule_element_qname=node.getJavaElement().getParent().getElementName().to
String()+"."+node.getJavaElement().getElementName().toString();
        if(Pattern_Type.contains("RequiredAct")==true)
        {
            sb.set_RTAPattern_Constructs(true,Source_element_qname,source_qualifiedname,target
_qualifiedname,requiredactmodule_name,Target_element_qname,ReqActModule_element
_qname,InterfaceSource_Variable_qname,Interfacetarget_Variable_qname,Interface_IsActi
ve,Interface_terminate,pointcut,Pointcut_Parameter_qname);

            System.out.println("Runtime Required Activation Pattern is Identified");
        }
        else if(Pattern_Type.contains("ExcludedAct")==true)
        {
            sb.set_RTEPattern_Constructs(true,Source_element_qname,source_qualifiedname,target
_qualifiedname,requiredactmodule_name,Target_element_qname,ReqActModule_element

```

```

qname,InterfaceSource_Variable_qname,Interfacetarget_Variable_qname,Interface_IsActiv
e,Interface_terminate,pointcut,Pointcut_Parameter_qname);

System.out.println("Runtime Excluded Activation Pattern is Identified");

}}

if(Pattern_Type.contains("ConcurrentAct")==true)

{ if((source_variable==true) && (pointcut==true) && (set_source==true) &&
(set_target==true) && (Interface_Run==true))

{

Concurrentactmodule_name=node.getJavaElement().getParent().getElementName().toStrin
g();

Concurrentactmodule_element_qname=node.getJavaElement().getParent().getElementNam
e().toString()+ "." +node.getJavaElement().getElementName().toString();

sb.set_ca_pattern_model_constructs(source_qualifiedname,target_qualifiedname,Concurren
tactmodule_name,Source_element_qname,Target_element_qname,Concurrentactmodule_el
ement_qname,InterfaceSource_Variable_qname,pointcut,Pointcut_Parameter_qname,Interf
ace_Run);

System.out.println("Runtime Concurrent Activation Pattern is Identified);

}}

else if(Pattern_Type.contains("SequentialAct")==true)

{

if((source_variable==true)&&(pointcut==true)&&(set_source==true)&&(set_target==tru
e)&&(Interface_Activate==true))

{

Sequentialactmodule_name=node.getJavaElement().getParent().getElementName().toString
();

Sequentialactmodule_element_qname=node.getJavaElement().getParent().getElementNam
e().toString()+ "." +node.getJavaElement().getElementName().toString();

sb.set_sa_pattern_model_constructs(source_qualifiedname, target_qualifiedname,
Sequentialactmodule_name, Source_element_qname, Target_element_qname
,Sequentialactmodule_element_qname, InterfaceSource_Variable_qname, pointcut,
Pointcut_Parameter_qname, Interface_Activate);

System.out.println("Runtime Sequential Activation Pattern is Identified);

}}}}

public void endVisit(TypeDeclaration node)

{

if((node.resolveBinding().getSuperclass()!=null)&&(node.resolveBinding().getSuperclass()
.getName().equalsIgnoreCase("Object")!=true))

{

```

```

Pattern_Type=node.resolveBinding().getSuperclass().getName().toString();
}}
public void endVisit(ParameterizedType node){
int arg_size=node.typeArguments().size();
if(node.typeArguments().size()!=0)
{
for (int i=0; i<arg_size;i++)
{ SimpleType st_0=(SimpleType) node.typeArguments().get(i);
if(i==0)
{ set_source=true;
// Putting if statement for correctly identifying patterns in scical.implementation otherwise
not needed for AOPDep.implementation project hence I can delete it without any issues
if(st_0.resolveBinding().getJavaElement().getResource()!=null) {
source_qualifiedname=st_0.resolveBinding().getPackage().getName().toString();
Source_element_qname=st_0.resolveBinding().getQualifiedName()+"."+st_0.resolveBinding().getJavaElement().getResource().getFileExtension().toString();
}}
if(i==1)
{ set_target=true;
target_qualifiedname=st_0.resolveBinding().getPackage().getName().toString();
Target_element_qname=st_0.resolveBinding().getQualifiedName()+"."+st_0.resolveBinding().getJavaElement().getResource().getFileExtension().toString();
}}}}
public void endVisit(Assignment node)
{
if(node.getLeftHandSide().toString().equalsIgnoreCase("Source"))
{
source_variable=true;
InterfaceSource_Variable_qname=node.getRightHandSide().resolveTypeBinding().getQualifiedName()+"."+node.getRightHandSide().resolveTypeBinding().getJavaElement().getResource().getFileExtension();
}
if(node.getLeftHandSide().toString().equalsIgnoreCase("target"))
{target_variable=true;

```

```

Interface target_Variable_qname=node.getRightHandSide().resolveTypeBinding().getQualifiedName()+"."+node.getRightHandSide().resolveTypeBinding().getJavaElement().getResource().getFileExtension();

}

Expression expr = node.getRightHandSide();

if(expr.resolveTypeBinding()!=null)

{}

public void endVisit(MethodDeclaration node) {

    SimpleName a;

    a=node.getName();

    if (a.toString().replace('$', '!').endsWith("isActive"))

    {

        Interface_IsActive=true; }

    if (a.toString().replace('$', '!').endsWith("terminate"))

        {

            Interface_terminate=true; }

    if(a.toString().replace('$', '!').endsWith("run"))

        {

            Interface_Run=true; }

    if(a.toString().replace('$', '!').endsWith("activate"))

        {

            Interface_Activate=true; }

    if(node.getReturnType2()!=null)

        {

            if(node.getReturnType2().toString().equalsIgnoreCase("pointcut"))

            {

                pointcut=true;

                if(node.parameters().size()>0)

                {

                    for (int itr=0; itr<node.parameters().size(); itr++)

                    {

                        SingleVariableDeclaration svd=(SingleVariableDeclaration) node.parameters().get(itr);

                        IVariableBinding variableBinding = svd.resolveBinding().getVariableDeclaration();

                        if(variableBinding.getType().getJavaElement()!=null)

                        {

                            Pointcut_Parameter_qname=variableBinding.getType().getQualifiedName()+"."+variableBinding.getType().getJavaElement().getResource().getFileExtension().toString();

                        }

                    }

                }

            }

        }

}

```

Sample Builder Code

```
package code2model.builder;

import DemoDsl.DemoDslFactory;

import DemoDsl.DemoDslPackage;

import DemoDsl.rtddemo;

import DemoDsl.concurrentact.ConcurrentactFactory;

import DemoDsl.implmodel.Component;

import DemoDsl.implmodel.ImplmodelFactory;

import DemoDsl.implmodel.ImplmodelPackage;

import DemoDsl.implmodel.Package;

import DemoDsl.implmodel.Project;

import DemoDsl.implmodel.JavaClass;

import DemoDsl.implmodel.Aspect;

import DemoDsl.runtimeexcludedactivation.ExclActModuleElement;

import DemoDsl.runtimeexcludedactivation.RuntimeExcludedActivation;

import DemoDsl.runtimeexcludedactivation.RuntimeexcludedactivationFactory;

import DemoDsl.runtimerequiredactivation.InterfaceIsActiveFunction;

import DemoDsl.runtimerequiredactivation.InterfaceSourceVariable;

import DemoDsl.runtimerequiredactivation.InterfaceTargetVariable;

import DemoDsl.runtimerequiredactivation.InterfaceTerminateFunction;

import DemoDsl.runtimerequiredactivation.Pattern;

import DemoDsl.runtimerequiredactivation.Pointcut;

import DemoDsl.runtimerequiredactivation.ReqActModuleElement;

import DemoDsl.runtimerequiredactivation.ReqActivationModule;

import DemoDsl.runtimerequiredactivation.RuntimerequiredactivationFactory;

import DemoDsl.runtimerequiredactivation.RuntimerequiredactivationPackage;

import DemoDsl.runtimerequiredactivation.PatternModel;

import DemoDsl.runtimerequiredactivation.RuntimeRequiredActivation;

import DemoDsl.runtimerequiredactivation.Source;

import DemoDsl.runtimerequiredactivation.SourceElement;

import DemoDsl.runtimerequiredactivation.Target;

import DemoDsl.runtimerequiredactivation.TargetElement;
```

```
import DemoDsl.sequentialactivation.SequentialActivation;
import DemoDsl.sequentialactivation.SequentialactivationFactory;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.xml.parsers.SAXParserFactory;
import org.eclipse.jdt.core.ICompilationUnit;
import org.eclipse.jdt.core.IJavaProject;
import org.eclipse.jdt.core.IPackageFragment;
import org.eclipse.jdt.core.IPackageFragmentRoot;
import org.eclipse.jdt.core.JavaCore;
import org.eclipse.jdt.core.JavaModelException;
import org.eclipse.jdt.core.dom.AST;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.ASTParser;
import org.eclipse.core.resources.IFile;
import org.eclipse.core.resources.IMarker;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.resources.IResourceDelta;
import org.eclipse.core.resources.IResourceDeltaVisitor;
import org.eclipse.core.resources.IResourceVisitor;
import org.eclipse.core.resources.IncrementalProjectBuilder;
import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
```

```

import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;

import org.xml.sax.SAXException;

import org.xml.sax.SAXParseException;

import org.xml.sax.helpers.DefaultHandler;

public class SampleBuilder extends IncrementalProjectBuilder {

public static rtddemo demodsl= DemoDslFactory.eINSTANCE.creatertddemo();

public static Project project1 = ImplmodelFactory.eINSTANCE.createProject();

public static PatternModel PM=
RuntimeRequiredactivationFactory.eINSTANCE.createPatternModel();

public static DemoDsl.runtimeexcludedactivation.PatternModel RTE=
RuntimeexcludedactivationFactory.eINSTANCE.createPatternModel();

public static DemoDsl.concurrentact.PatternModel ca=
ConcurrentactFactory.eINSTANCE.createPatternModel();

public static DemoDsl.sequentialactivation.PatternModel sa=
SequentialactivationFactory.eINSTANCE.createPatternModel();

public static Pattern p= RuntimeRequiredactivationFactory.eINSTANCE.createPattern();

public static DemoDsl.runtimeexcludedactivation.Pattern p_rte=
RuntimeexcludedactivationFactory.eINSTANCE.createPattern();

public static DemoDsl.concurrentact.Pattern p_ca=
ConcurrentactFactory.eINSTANCE.createPattern();

public static DemoDsl.sequentialactivation.Pattern p_sa=
SequentialactivationFactory.eINSTANCE.createPattern();

public static      HashMap<String, Package> Packages_Hashmap = new
HashMap<String, Package>();

public static      HashMap<String, Component> Classes_Hashmap = new
HashMap<String, Component>();

class SampleDeltaVisitor implements IResourceDeltaVisitor {

public boolean visit(IResourceDelta delta) throws CoreException {

IResource resource = delta.getResource();

        switch (delta.getKind()) {

                case IResourceDelta.ADDED:

                        // handle added resource

                        ImplASTgenerator(resource);

                        break;

                case IResourceDelta.REMOVED:

                        //handle removed resource

```

```

        break;
    case IResourceDelta.CHANGED:
        //handle changed resource
        ImplASTgenerator(resource);
        break;
    }
    //return true to continue visiting children.
    return true;
}
}

class SampleResourceVisitor implements IResourceVisitor {
    public boolean visit(IResource resource) {
        try {
            ImplASTgenerator(resource);
        } catch (JavaModelException e) {
            //TODO Auto-generated catch block
            e.printStackTrace();
        } catch (CoreException e) {
            //TODO Auto-generated catch block
            e.printStackTrace();
        }
        return true;
    }
}

class XMLErrorHandler extends DefaultHandler {
    private IFile file;
    public XMLErrorHandler(IFile file) {
        this.file = file;}

    private void addMarker(SAXParseException e, int severity) {
        SampleBuilder.this.addMarker(file, e.getMessage(), e.getLineNumber(), severity);
    }

    public void error(SAXParseException exception) throws SAXException {
        addMarker(exception, IMarker.SEVERITY_ERROR);}

    public void fatalError(SAXParseException exception) throws SAXException {

```

```

addMarker(exception, IMarker.SEVERITY_ERROR);}

public void warning(SAXParseException exception) throws SAXException {
    addMarker(exception, IMarker.SEVERITY_WARNING);
}
}

public static final String BUILDER_ID="Code2Model.sampleBuilder";
private static final String MARKER_TYPE ="Code2Model.xmlProblem";
private SAXParserFactory parserFactory;

private void addMarker(IFile file, String message, int lineNumber,int severity) {
    try {
        IMarker marker = file.createMarker(MARKER_TYPE);
        marker.setAttribute(IMarker.MESSAGE, message);
        marker.setAttribute(IMarker.SEVERITY, severity);
        if (lineNumber ==-1){
            lineNumber =1;
        }
        marker.setAttribute(IMarker.LINE_NUMBER, lineNumber);
    } catch (CoreException e){
    }
}

protected IProject[] build(int kind, Map args, IProgressMonitor monitor)
    throws CoreException{
    if (kind == FULL_BUILD){
        fullBuild(monitor);
    } else{
        IResourceDelta delta = getDelta(getProject());
        if (delta == null) {
            fullBuild(monitor);
        } else{
            incrementalBuild(delta, monitor);
        }
    }
    return null;
}

```

```

    }

void ImplASTgenerator(IResource resource) throws CoreException, JavaModelException
{if (resource instanceof IProject && resource.getName().endsWith(".implementation")) {
    IProject proj = (IProject) resource;

    //Note: Check if the project has the Java nature so that we can work on Java Ast
    if (proj.isNatureEnabled("org.eclipse.jdt.core.javanature")) {
        System.out.println("project name is :"+proj.getName());

        IJavaProject javaProject = JavaCore.create(proj);

        PackageContentsInfos(javaProject, proj);

        //Calling the parse functions - can be deleted

        ParsePackageContentsInfos(javaProject, proj);

        System.out.println("Saving Implementation Model");

        //saving new file

        ResourceSet resourceSet1 = new ResourceSetImpl();

resourceSet1.getResourceFactoryRegistry().getExtensionToFactoryMap().put(Resource.Fa
ctory.Registry.DEFAULT_EXTENSION,new XMIResourceFactoryImpl());

resourceSet1.getPackageRegistry().put(DemoDsl.implmodel.ImplmodelPackage.eNS_URI,
DemoDsl.implmodel.ImplmodelPackage.eINSTANCE);

System.out.println("Saving Runtime-Required-Activation Pattern Model");

//Saving new Pattern Model file

ResourceSet rs= new ResourceSetImpl();

rs.getResourceFactoryRegistry().getExtensionToFactoryMap().put(Resource.Factory.Regis
try.DEFAULT_EXTENSION, new XMIResourceFactoryImpl());

rs.getPackageRegistry().put(DemoDsl.runtimerequiredactivation.Runtimerequiredactivati
onPackage.eNS_URI,
DemoDsl.runtimerequiredactivation.RuntimerequiredactivationPackage.eINSTANCE);

System.out.println("Saving Runtime-Excluded-Activation Pattern Model");

ResourceSet rs_rte= new ResourceSetImpl();

rs_rte.getResourceFactoryRegistry().getExtensionToFactoryMap().put(Resource.Factory.R
egistry.DEFAULT_EXTENSION, new XMIResourceFactoryImpl());

rs_rte.getPackageRegistry().put(DemoDsl.runtimeexcludedactivation.Runtimeexcludedacti
vationPackage.eNS_URI,

DemoDsl.runtimerequiredactivation.RuntimerequiredactivationPackage.eINSTANCE);

System.out.println("Saving Runtime-Concurrent-Activation Pattern Model");

ResourceSet rs_ca= new ResourceSetImpl();

```

```

rs_ca.getResourceFactoryRegistry().getExtensionToFactoryMap().put(Resource.Factory.Re
gistry.DEFAULT_EXTENSION, new XMIRResourceFactoryImpl());

rs_ca.getPackageRegistry().put(DemoDsl.concurrentact.ConcurrentactPackage.eNS_URI,
DemoDsl.concurrentact.ConcurrentactPackage.eINSTANCE);

System.out.println("Saving Runtime-Sequential-Activation Pattern Model");

ResourceSet rs_sa= new ResourceSetImpl();

rs_sa.getResourceFactoryRegistry().getExtensionToFactoryMap().put(Resource.Factory.Re
gistry.DEFAULT_EXTENSION, new XMIRResourceFactoryImpl());

rs_sa.getPackageRegistry().put(DemoDsl.sequentialactivation.SequentialactivationPackage.
eNS_URI, DemoDsl.sequentialactivation.SequentialactivationPackage.eINSTANCE);

        //Saving demodsl model

System.out.println("Saving DemoDsl Model");

        //Saving new DemoDsl Model

ResourceSet ddrs= new ResourceSetImpl();

ddrs.getResourceFactoryRegistry().getExtensionToFactoryMap().put(Resource.Factory.Re
gistry.DEFAULT_EXTENSION, new XMIRResourceFactoryImpl());

ddrs.getPackageRegistry().put(DemoDsl.DemoDslPackage.eNS_URI,
DemoDsl.DemoDslPackage.eINSTANCE);

try {

String path =ResourcesPlugin.getWorkspace().getRoot().getLocation().toString();

Resource r = resourceSet1.createResource(URI.createURI("SciCalc.Implmodel"));

        r.getContents().add(project1);

String path1= ResourcesPlugin.getWorkspace().getRoot().getLocation().toString();

Resource
R=rs.createResource(URI.createURI("RuntimeReqAct.runtimerequiredactivation"));

        R.getContents().add(PM);

String path_rte= ResourcesPlugin.getWorkspace().getRoot().getLocation().toString();

Resource
R_RTE=rs_rte.createResource(URI.createURI("RuntimeExclAct.runtimeexcludedactivatio
n"));

Resource r_ca=rs_ca.createResource(URI.createURI("RuntimeConcAct.concurrentact"));

Resource
r_sa=rs_sa.createResource(URI.createURI("RuntimeSeqAct.sequentialactivation"));

String path2= ResourcesPlugin.getWorkspace().getRoot().getLocation().toString();

Resource DDR=ddrs.createResource(URI.createURI("demo.DemoDsl"));

demodsl.getImplmodel().add((Project)project1);

```

```

demodsl.getRtamodel().add(PM);

demodsl.getRtemodel().add(RTE);

demodsl.getCamodel().add(ca);

demodsl.getSamodel().add(sa);

DDR.getContents().add(demodsl);

File f= new File(resource.getLocation().toString()+"/scialc.Implmodel");

File pm_f= new
File(resource.getLocation().toString()+"/RuntimeReqAct.runtimerequiredactivation");

File rte_f= new
File(resource.getLocation().toString()+"/RuntimeExclAct.runtimeexcludedactivation");

File f_ca= new File(resource.getLocation().toString()+"/RuntimeConcAct.concurrentact");

File f_sa= new
File(resource.getLocation().toString()+"/RuntimeSeqAct.sequentialactivation");

File dd_f=new File(resource.getLocation().toString()+"/demo.DemoDsl");

If
((!f.exists())||(!pm_f.exists())||(!dd_f.exists())||(!rte_f.exists()))||(!f_ca.exists())||(!f_sa.exists()
))
{

        FileOutputStream fos = new FileOutputStream(f);

        r.save(fos,null);

        FileOutputStream pm_fos = new FileOutputStream(pm_f);

        R.save(pm_fos,null);

        FileOutputStream rte_fos= new FileOutputStream(rte_f);

        R_RTE.save(rte_fos, null);

        FileOutputStream ca_f=new FileOutputStream(f_ca);

        r_ca.save(ca_f,null);

        FileOutputStream sa_f= new FileOutputStream(f_sa);

        r_sa.save(sa_f, null);

        FileOutputStream dd_fos = new FileOutputStream(dd_f);

        DDR.save(dd_fos,null);

    }

    else        {

                f.deleteOnExit();

                pm_f.deleteOnExit();

```

```

        dd_f.deleteOnExit();
    }
} catch(IOException exception){
    exception.printStackTrace();    }}}}
private void ParsePackageContentsInfos(IJavaProject javaProject,IProject proj)
    throws JavaModelException, org.eclipse.jdt.core.JavaModelException {
    IPackageFragment[] packages = javaProject.getPackageFragments();
    for (IPackageFragment mypackage : packages) {
        if (mypackage.getKind() == IPackageFragmentRoot.K_SOURCE) {
            if (mypackage.getChildren().length!=0)
                {ParseprintICompilationUnitInfo(mypackage,proj);}}}}
private void ParseprintICompilationUnitInfo(IPackageFragment mypackage, IProject Proj)
    throws JavaModelException, org.eclipse.jdt.core.JavaModelException {
    for (ICompilationUnit unit : mypackage.getCompilationUnits()) {
        if (unit.getResource().getFileExtension().endsWith("java"))
            {}
        else if (unit.getResource().getFileExtension().endsWith("aj"))
            {parseAspectIMethods(unit,Proj);}}
private void parseAspectIMethods (ICompilationUnit unit,IProject proj) throws
JavaModelException, org.eclipse.jdt.core.JavaModelException
{
    ASTParser parser = ASTParser.newParser(AST.JLS4);
    parser.setBindingsRecovery(true);
    parser.setResolveBindings(true);
    parser.setStatementsRecovery(true);
    parser.setSource(unit);
    ASTNode ast = parser.createAST(null);
    ast.accept(new CompliationUnitParser());
}
private void PackageContentsInfos(IJavaProject javaProject,IProject proj)
    throws JavaModelException, org.eclipse.jdt.core.JavaModelException
{
    IPackageFragment[] packages = javaProject.getPackageFragments();
    for (IPackageFragment mypackage : packages) {

```

```

        if (mypackage.getKind() == IPackageFragmentRoot.K_SOURCE) {
System.out.println("Package " + mypackage.getElementName());
        if (mypackage.getChildren().length!=0)
        { Package pkg= ImplmodelFactory.eINSTANCE.createPackage();
            pkg.setName(mypackage.getElementName().toString());
            pkg.setQualifiedName(mypackage.getElementName());
            project1.getOwnedMembers().add(pkg);
Packages_Hashmap.put(mypackage.getElementName().toString().trim(), pkg);
        printICompilationUnitInfo(mypackage,proj,pkg);}}}}
private void printICompilationUnitInfo(IPackageFragment mypackage, IProject Proj,
Package pkg)
        throws JavaModelException, org.eclipse.jdt.core.JavaModelException {
    for (ICompilationUnit unit : mypackage.getCompilationUnits()) {
        if (unit.getResource().getFileExtension().endsWith("java"))
        {
            JavaClass jc= ImplmodelFactory.eINSTANCE.createJavaClass();
            jc.setName(unit.getElementName());
            jc.setQualifiedName(pkg.getQualifiedName()+ "." +unit.getResource().getName());
            pkg.getOwnedMembers().add(jc);
Classes_Hashmap.put(pkg.getQualifiedName()+ "." +unit.getResource().getName(),jc);}
        else if (unit.getResource().getFileExtension().endsWith("aj"))
        {
            Aspect a= ImplmodelFactory.eINSTANCE.createAspect();
            a.setName(unit.getElementName());
            a.setQualifiedName(pkg.getQualifiedName()+ "." +unit.getResource().getName());
            pkg.getOwnedMembers().add(a);
Classes_Hashmap.put(pkg.getQualifiedName()+ "." +unit.getResource().getName(), a);
            AspectIMethods(unit,Proj);}}}}
public void set_RTA_pattern_Constructs(boolean a,String src_Element_qname, String
src_qname,String trgt_qname,String reqactmodule_qname, String trgt_Element_qname,
String ReqActModule_Element_qname, String interface_src_var_qname,String
interface_trgt_var_qname, boolean isActive_Implemented, boolean
terminate_Implemented,boolean pointcut_Implemented,String pointcut_parameter_qname)
{
    PM.getPattern().add(p);
    PM.setName("Identified-Runtime-Required-Activation-Patterns");

```

```

RuntimeRequiredActivation
rra=RuntimeRequiredActivationFactory.eINSTANCE.createRuntimeRequiredActivation();

p.getRuntimeRequiredActivation().add(rra);

//Adding Source Concept to the runtime required Activation pattern instance

    Source S= RuntimeRequiredActivationFactory.eINSTANCE.createSource();

    S.setName(src_qname);

    //System.out.println("Find Source in
Packages_HashMap"+Packages_Hashmap.values());

    S.setRtasrcref(Packages_Hashmap.get(src_qname));

    rra.setHasSource(S);

// Adding Target Concept to the runtime required Activation pattern instance

    Target T= RuntimeRequiredActivationFactory.eINSTANCE.createTarget();

    T.setName(trgt_qname);

    T.setRtatrgtref(Packages_Hashmap.get(trgt_qname));

    rra.setHasTarget(T);

//Adding Runtime Required Module in the runtime required Activation pattern instance

ReqActivationModule ram=
RuntimeRequiredActivationFactory.eINSTANCE.createReqActivationModule();

    ram.setName(reqactmodule_qname);

    ram.setRtareqactmodref(Packages_Hashmap.get(reqactmodule_qname));

    rra.setHasReqActModule(ram);

//Adding Source element in the Runtime Req Activation pattern

SourceElement src_e=
RuntimeRequiredActivationFactory.eINSTANCE.createSourceElement();

    src_e.setName(src_Element_qname);

    src_e.setRtasrcelementref(Classes_Hashmap.get(src_Element_qname));

    S.getHasSourceElements().add(src_e);

//Adding Target Element in the Runtime Req Activation pattern

TargetElement trgt_e=
RuntimeRequiredActivationFactory.eINSTANCE.createTargetElement();

    trgt_e.setName(trgt_Element_qname);

    trgt_e.setRtatrgtelementref(Classes_Hashmap.get(trgt_Element_qname));

    T.getHasTargetElements().add(trgt_e);

//Adding ReqActModuleElement in the runtime required activation pattern

```

```

ReqActModuleElement rame=
RuntimeRequiredActivationFactory.eINSTANCE.createReqActModuleElement();

    rame.setName(ReqActModule_Element_qname);

rame.setRtreqactmoduleelementref(Classes_Hashmap.get(ReqActModule_Element_qname));

    ram.getHasReqActModuleElements().add(rame);

//Adding interfacesourcevariable

InterfaceSourceVariable ISV=
RuntimeRequiredActivationFactory.eINSTANCE.createInterfaceSourceVariable();

ISV.setSourcevariable((Component)Classes_Hashmap.get(interface_src_var_qname));

    rame.setHasInterfaceSourceVariable(ISV);

//Adding InterfacetargetVariable

InterfaceTargetVariable
ITV=RuntimeRequiredActivationFactory.eINSTANCE.createInterfaceTargetVariable();

ITV.setTargetvariable((Component) Classes_Hashmap.get(interface_trgt_var_qname));

rame.setHasInterfaceTargetVariable(ITV);

//Adding InterfaceIsActiveFunction

    InterfaceIsActiveFunction
IAF=RuntimeRequiredActivationFactory.eINSTANCE.createInterfaceIsActiveFunction();

    IAF.setIsActiveFuncImplemented(isActive_Implemented);

    rame.setHasInterfaceIsActiveFunction(IAF);

//Adding InterfaceTerminateFunction

InterfaceTerminateFunction ITF=
RuntimeRequiredActivationFactory.eINSTANCE.createInterfaceTerminateFunction();

ITF.setTerminateFuncImplemented(terminate_Implemented);

rame.setHasInterfaceTerminateFunction(ITF);

//Adding pointcut

Pointcut pc= RuntimeRequiredActivationFactory.eINSTANCE.createPointcut();

    pc.setPointcutFuncImplemented(pointcut_Implemented);

    pc.setJoinpoint(Classes_Hashmap.get(pointcut_parameter_qname));

    rame.setHasPointcut(pc);}

public void set_RTE_pattern_Constructs(boolean a,String src_Element_qname, String
src_qname,String trgt_qname,String reqactmodule_qname, String trgt_Element_qname,
String ReqActModule_Element_qname, String interface_src_var_qname,String
interface_trgt_var_qname, boolean isActive_Implemented, boolean
terminate_Implemented,boolean pointcut_Implemented,String pointcut_parameter_qname)

{

```

```

RTE.getPattern().add(p_rte);

RTE.setName("Identified-Runtime-Excluded-Activation-Patterns");

RuntimeExcludedActivation
rea=RuntimeexcludedactivationFactory.eINSTANCE.createRuntimeExcludedActivation();

p_rte.getRuntimeExcludedActivation().add(rea);

//Adding Source Concept to the runtime excluded Activation pattern instance

DemoDsl.runtimeexcludedactivation.Source S=
RuntimeexcludedactivationFactory.eINSTANCE.createSource();

S.setName(src_qname);

    S.setRtesrcref(Packages_Hashmap.get(src_qname));

    rea.setHasSource(S);

//Adding Source element in the Runtime Excl Activation pattern

DemoDsl.runtimeexcludedactivation.SourceElement src_e=
RuntimeexcludedactivationFactory.eINSTANCE.createSourceElement();

    src_e.setName(src_Element_qname);

    src_e.setRtesrcelementref(Classes_Hashmap.get(src_Element_qname));

    S.getHasSourceElements().add(src_e);

//Adding Target Concept to the runtime excluded Activation pattern instance

DemoDsl.runtimeexcludedactivation.Target T=
RuntimeexcludedactivationFactory.eINSTANCE.createTarget();

    T.setName(trgt_qname);

    T.setRtetrgtref(Packages_Hashmap.get(trgt_qname));

    rea.setHasTarget(T);

//Adding Target Element in the Runtime Excluded Activation pattern

DemoDsl.runtimeexcludedactivation.TargetElement trgt_e=
RuntimeexcludedactivationFactory.eINSTANCE.createTargetElement();

    trgt_e.setName(trgt_Element_qname);

    trgt_e.setRtetrgttelementref(Classes_Hashmap.get(trgt_Element_qname));

    T.getHasTargetElements().add(trgt_e);

//Adding ReqActModule in the runtime excluded activation pattern

DemoDsl.runtimeexcludedactivation.ExclActivationModule rem=
RuntimeexcludedactivationFactory.eINSTANCE.createExclActivationModule();

    rem.setName(ReqActModule_Element_qname);

    rem.setRtereqactmodref(Packages_Hashmap.get(reqactmodule_qname));

```

```

        rea.setHasExclActModule(rem);

//Adding ReqExclModuleElement in the runtime required excluded pattern
DemoDsl.runtimeexcludedactivation.ExclActModuleElement reme=
RuntimeexcludedactivationFactory.eINSTANCE.createExclActModuleElement();

        reme.setName(ReqActModule_Element_qname);
reme.setRtareqactmodelelementref(Classes_Hashmap.get(ReqActModule_Element_qname));
        rem.getHasExclActModuleElements().add(reme);

//Adding interfacesourcevariable
DemoDsl.runtimeexcludedactivation.InterfaceSourceVariable ISV=
RuntimeexcludedactivationFactory.eINSTANCE.createInterfaceSourceVariable();

        ISV.setSourcevariable((Component)Classes_Hashmap.get(interface_src_var_qname));

        reme.setHasInterfaceSourceVariable(ISV);

//Adding InterfacetargetVariable
DemoDsl.runtimeexcludedactivation.InterfaceTargetVariable
ITV=RuntimeexcludedactivationFactory.eINSTANCE.createInterfaceTargetVariable();

ITV.setTargetvariable((Component) Classes_Hashmap.get(interface_trgt_var_qname));

        reme.setHasInterfaceTargetVariable(ITV);

//Adding InterfaceIsActiveFunction
DemoDsl.runtimeexcludedactivation.InterfaceIsActiveFunction
IAF=RuntimeexcludedactivationFactory.eINSTANCE.createInterfaceIsActiveFunction();

        IAF.setIsActiveFuncImplemented(isActive_Implemented);

        reme.setHasInterfaceIsActiveFunction(IAF);

//Adding InterfaceTerminateFunction
DemoDsl.runtimeexcludedactivation.InterfaceTerminateFunction ITF=
RuntimeexcludedactivationFactory.eINSTANCE.createInterfaceTerminateFunction();

        ITF.setTerminateFuncImplemented(terminate_Implemented);

        reme.setHasInterfaceTerminateFunction(ITF);

//Adding pointcut
DemoDsl.runtimeexcludedactivation.Pointcut pc=
RuntimeexcludedactivationFactory.eINSTANCE.createPointcut();

        pc.setPointcutFuncImplemented(pointcut_implemented);
        pc.setJoinpoint(Classes_Hashmap.get(pointcut_parameter_qname));

        reme.setHasPointcut(pc);
}

```

```

public void set_ca_pattern_model_constructs(String src_qname, String trgt_qname,String
camodule_qname,String src_element_qname,String trgt_element_qname,

String ca_modelement_qname,String interface_src_variable, boolean
pointcut_implemented, String pointcut_parameter_qname,boolean run_implemented)
{
    ca.getHasPatterns().add(p_ca);

    ca.setName("Identified-Concurrent-Activation-Patterns");

    DemoDsl.concurrentact.ConcurrentActivation concat=
    ConcurrentactFactory.eINSTANCE.createConcurrentActivation();

    p_ca.getHasConcurrentActivation().add(concat);

    //Adding Source Concept

    DemoDsl.concurrentact.Source S= ConcurrentactFactory.eINSTANCE.createSource();

    S.setName(src_qname);

    S.setCasrcref(Packages_Hashmap.get(src_qname));

    concat.setHasSource(S);

    //Adding Target Concept

    DemoDsl.concurrentact.Target T=
    ConcurrentactFactory.eINSTANCE.createTarget();

    T.setName(trgt_qname);

    T.setCatrgtref(Packages_Hashmap.get(trgt_qname));

    concat.setHasTarget(T);

    //Adding concurrentModule Concept

    DemoDsl.concurrentact.ConcurrentActModule camod=
    ConcurrentactFactory.eINSTANCE.createConcurrentActModule();

    camod.setName(camodule_qname);

    camod.setCamoduleref(Packages_Hashmap.get(camodule_qname));

    concat.setHasConcurrentActModule(camod);

    //Adding source element concept

    DemoDsl.concurrentact.SourceElement ca_src_element=
    ConcurrentactFactory.eINSTANCE.createSourceElement();

    ca_src_element.setName(src_element_qname);

    ca_src_element.setCasrcelementref(Classes_Hashmap.get(src_element_qname));

    S.getHasSourceElements().add(ca_src_element);

    //Adding target element concept

```

```

DemoDsl.concurrentact.TargetElement ca_trgt_element=
ConcurrentactFactory.eINSTANCE.createTargetElement();

    ca_trgt_element.setName(trgt_element_qname);

    ca_trgt_element.setCatrgtelementref(Classes_Hashmap.get(trgt_element_qname));

    T.getHasTargetElements().add(ca_trgt_element);

//Adding concurrentModule Element Concept

DemoDsl.concurrentact.ConcActModuleElement ca_mod_element=
ConcurrentactFactory.eINSTANCE.createConcActModuleElement();

    ca_mod_element.setName(ca_modelement_qname);

ca_mod_element.setCamodelementref(Classes_Hashmap.get(ca_modelement_qname));

    camod.getHasConcModElements().add(ca_mod_element);

//Adding interfacesourcevariable concept

DemoDsl.concurrentact.InterfaceSourceVariable ca_ISV=
ConcurrentactFactory.eINSTANCE.createInterfaceSourceVariable();

    ca_ISV.setSetSource((Component)Classes_Hashmap.get(interface_src_variable));

    ca_mod_element.setHasInterfaceSourceVariable(ca_ISV);

//Adding Pointcut concept

DemoDsl.concurrentact.Pointcut ca_pointcut=
ConcurrentactFactory.eINSTANCE.createPointcut();

    ca_pointcut.setPointcutImplemented(pointcut_implemented);

    ca_pointcut.setJoinpoint(Classes_Hashmap.get(pointcut_parameter_qname));

    ca_mod_element.setHasPointcut(ca_pointcut);

//Adding run function concept

DemoDsl.concurrentact.RunFunction ca_RunFunc=
ConcurrentactFactory.eINSTANCE.createRunFunction();

    ca_RunFunc.setRunFunctionImplemented(run_implemented);

    ca_mod_element.setHasRunFunction(ca_RunFunc);

}

public void set_sa_pattern_model_constructs(String src_qname, String trgt_qname,String
samodule_qname,String src_element_qname,String trgt_element_qname,String
sa_modelement_qname,String interface_src_variable, boolean pointcut_implemented,
String pointcut_parameter_qname,boolean activate_implemented)

{

    sa.getHasPatterns().add(p_sa);

    sa.setName("Identified-Sequential-Activation-Patterns");

```

```

DemoDsl.sequentialactivation.SequentialActivation sa=
SequentialactivationFactory.eINSTANCE.createSequentialActivation();

    p_sa.getHasSequentialActivation().add(sa);

//Adding Source Concept

DemoDsl.sequentialactivation.Source S=
SequentialactivationFactory.eINSTANCE.createSource();

    S.setName(src_qname);

    S.setSasrcref(Packages_Hashmap.get(src_qname));

    sa.setHasSource(S);

//Adding Target Concept

DemoDsl.sequentialactivation.Target T=
SequentialactivationFactory.eINSTANCE.createTarget();

    T.setName(trgt_qname);

    T.setSatrgtref(Packages_Hashmap.get(trgt_qname));

    sa.setHasTarget(T);

//Adding Sequentialactivation module Concept

DemoDsl.sequentialactivation.SequentialActModule SAMod=
SequentialactivationFactory.eINSTANCE.createSequentialActModule();

    SAMod.setName(samodule_qname);

    SAMod.setSamoduleref(Packages_Hashmap.get(samodule_qname));

    sa.setHasSequentialActModule(SAMod);

//Adding SourceElement Concept

DemoDsl.sequentialactivation.SourceElement sa_src_element=
SequentialactivationFactory.eINSTANCE.createSourceElement();

    sa_src_element.setName(src_element_qname);

    sa_src_element.setSasrcelementref(Classes_Hashmap.get(src_element_qname));

    S.getHasSourceElements().add(sa_src_element);

//Adding TargetElement Concept

DemoDsl.sequentialactivation.TargetElement sa_trgt_element=
SequentialactivationFactory.eINSTANCE.createTargetElement();

    sa_trgt_element.setName(trgt_element_qname);

    sa_trgt_element.setSatrgtelementref(Classes_Hashmap.get(trgt_element_qname));

    T.getHasTargetElements().add(sa_trgt_element);

//Adding SequentialActivation Module element Concept

```

```

DemoDsl.sequentialactivation.SeqActModuleElement sa_MOD_element=
SequentialactivationFactory.eINSTANCE.createSeqActModuleElement();

sa_MOD_element.setName(sa_modelement_qname);

sa_MOD_element.setSamodelementref(Classes_Hashmap.get(sa_modelement_qname));

    SAMod.getHasSeqModElements().add(sa_MOD_element);

//Adding interfacesourcevariable concept

DemoDsl.sequentialactivation.InterfaceSourceVariable ISV=
SequentialactivationFactory.eINSTANCE.createInterfaceSourceVariable();

    ISV.setSetSource((Component)Classes_Hashmap.get(interface_src_variable));

    sa_MOD_element.setHasInterfaceSourceVariable(ISV);

//Adding pointcut Concept

DemoDsl.sequentialactivation.Pointcut sa_pointcut=
SequentialactivationFactory.eINSTANCE.createPointcut();

sa_pointcut.setPointcutImplemented(pointcut_implemented);

sa_pointcut.setJoinpoint((Component)Classes_Hashmap.get(pointcut_parameter_qname));

    sa_MOD_element.setHasPointcut(sa_pointcut);

//Adding activative Concept

DemoDsl.sequentialactivation.ActivateFunction sa_activate=
SequentialactivationFactory.eINSTANCE.createActivateFunction();

    sa_activate.setRunFunctionImplemented(activate_implemented);

    sa_MOD_element.setHasActivateFunction(sa_activate);

}protected void fullBuild(final IProgressMonitor monitor)throws CoreException {

    try {getProject().accept(new SampleResourceVisitor());

    } catch (CoreException e) {}

protected void incrementalBuild(IResourceDelta delta,

IProgressMonitor monitor) throws CoreException {delta.accept(new
SampleDeltaVisitor());}

```

Appendix D: Questionnaire for the Formal Interviews

Interview Questionnaire for PhD work title “Consistency Checking of Runtime Feature Dependencies in Software Product Lines”

Interviewee Details

Name:

Designation:

Contact Details:

Venue:

Time:

Interviewee Consent of Giving Feedback/Evaluation (Yes/No):

Evaluation Questions

Question 1: What are the important factors in your opinion are addressed in my research work?

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Data Protection Statement

I Saad Bin Abid would like to confirm that the data extracted during the interview and the recordings made will be treated as confidential and will solely be used for the evaluation of my PhD research purpose only.

Appendix E: EVL Constraints

```
context RuntimeExcludedDependency
{
constraint Name_Not_Specified
{
check: self.name.isDefined()
message: 'Name of '+self.eClass().name + ' Not Specified'
}
constraint Source_Specification_Check
{
check: not self.sources.isEmpty()
message: ' Source Feature Not Specified in ' + self.name
}
constraint Target_Specification_Check
{
check: not self.targets.isEmpty()
message: 'Target Feature Not Specified in ' + self.name
}
constraint Feature_Pkg_Trace_Check
{
check: not self.FeaturesPackageTrace.isEmpty()
message: 'RuntimeExcludedActivation to Implementation Package Trace
Not Specified in ' + self.name
}
constraint Feature_Component_Trace_Check
{
check: not self.FeatureComponentTrace.isEmpty()
message: 'Feature Component Trace Not Specified in' + self.name
}
constraint reverse_pattern_check
{
guard: self.satisfies("Feature_Component_Trace_Check")
check:
self.sources.get(0).FeaturesPackageTrace.get(0).equals(RuntimeExclu
dedDependency.getAllofKind().get(0).eContents().get(2))==true
and
self.targets.get(0).FeaturesPackageTrace.get(0).equals(RuntimeExclu
dedDependency.getAllofKind().get(0).eContents().get(0))==true
and
self.sources.get(0).FeatureComponentTrace.get(0).equals(RuntimeExcl
udedDependency.getAllofKind().get(0).eContents().get(2).eContents()
.get(0))==true
and
self.targets.get(0).FeatureComponentTrace.get(0).equals(RuntimeExcl
udedDependency.getAllofKind().get(0).eContents().get(0).eContents()
.get(0))==true
message: 'Specified Pattern----'+self.name+ '-----is Implemented in
Reverse Order (i.e., Specified Dependency Target=Identified Pattern
Source and Specified Dependency Source= Identified Pattern Target)'
+self.name
}}
context RuntimeConcurrentActivationDependency
{
constraint Name_Not_Specified
{
```

```

check: self.name.isDefined()
message: 'Name of '+self.eClass().name + ' Not Specified'
}
constraint Source_Specification_Check
{
check: not self.sources.isEmpty()
message: ' Source Feature Not Specified in ' + self.name
}
constraint Target_Specification_Check
{
check: not self.targets.isEmpty()
message: 'Target Feature Not Specified in ' + self.name
}
constraint Feature_Pkg_Trace_Check
{
check: not self.FeaturesPackageTrace.isEmpty()
message: 'RuntimeConcurrentActivation to Implementation Package
Trace Not Specified in ' + self.name
}
constraint Feature_Component_Trace_Check
{
check: not self.FeatureComponentTrace.isEmpty()
message: 'Feature Component Trace Not Specified in' + self.name
}
constraint reverse_pattern_check
{
guard: self.satisfies("Feature_Component_Trace_Check")
check:
self.sources.get(0).FeaturesPackageTrace.get(0).equals(RuntimeConcu
rrentActivationDependency.getAllofKind().get(0).eContents().get(2))
==true
and
self.targets.get(0).FeaturesPackageTrace.get(0).equals(RuntimeConcu
rrentActivationDependency.getAllofKind().get(0).eContents().get(0))
==true
and
self.sources.get(0).FeatureComponentTrace.get(0).equals(RuntimeConc
urrentActivationDependency.getAllofKind().get(0).eContents().get(2)
.eContents().get(0))==true
and
self.targets.get(0).FeatureComponentTrace.get(0).equals(RuntimeConc
urrentActivationDependency.getAllofKind().get(0).eContents().get(0)
.eContents().get(0))==true
message: 'Specified Pattern----'+self.name+ '-----is Implemented in
Reverse Order (i.e., Specified Dependency Target=Identified Pattern
Source and Specified Dependency Source= Identified Pattern Target)'
+self.name
}}
context RuntimeSequentialActivationDependency
{
constraint Name_Not_Specified
{
check: self.name.isDefined()
message: 'Name of '+self.eClass().name + ' Not Specified'
}
constraint Source_Specification_Check
{
check: not self.sources.isEmpty()

```

```

message: ' Source Feature Not Specified in ' + self.name
}
constraint Target_Specification_Check
{
check: not self.targets.isEmpty()
message: 'Target Feature Not Specified in ' + self.name
}
constraint Feature_Pkg_Trace_Check
{
check: not self.FeaturesPackageTrace.isEmpty()
message: 'RuntimeSequentialActivation to Implementation Package
Trace Not Specified in ' + self.name
}
constraint Feature_Component_Trace_Check
{
check: not self.FeatureComponentTrace.isEmpty()
message: 'Feature Component Trace Not Specified in' + self.name
}
constraint reverse_pattern_check
{
guard: self.satisfies("Feature_Component_Trace_Check")
check:
self.sources.get(0).FeaturesPackageTrace.get(0).equals(RumtimeSeque
ntialActivationDependency.getAllofKind().get(0).eContents().get(2))
==true
and
self.targets.get(0).FeaturesPackageTrace.get(0).equals(RumtimeSeque
ntialActivationDependency.getAllofKind().get(0).eContents().get(0))
==true
and
self.sources.get(0).FeatureComponentTrace.get(0).equals(RumtimeSequ
entialActivationDependency.getAllofKind().get(0).eContents().get(2)
.eContents().get(0))==true
and
self.targets.get(0).FeatureComponentTrace.get(0).equals(RumtimeSequ
entialActivationDependency.getAllofKind().get(0).eContents().get(0)
.eContents().get(0))==true
message: 'Specified Pattern----'+self.name+ '-----is Implemented in
Reverse Order (i.e., Specified Dependency Target=Identified Pattern
Source and Specified Dependency Source= Identified Pattern Target)'
+self.name
}}

```


Appendix F: List of Publications

- Goetz Botterweck, Steffen Thiel, Daren Nestor and **Saad bin Abid** (2008). “Visual Tool Support for Configuring and Understanding Software Product Lines”, 12th International Software Product Line Conference (SPLC 2008), 8-12 September 2008, Limerick, Ireland. (**Conference publication**)
- **Saad bin Abid** and Goetz Botterweck: Resolving Product Derivation Tasks using Traceability in Software Product Lines, 5th Traceability workshop (ECMDA-TW) held in conjunction with European Conference of Model Driven Architectures 2009 (ECMDA'09), Twente, The Netherlands. (**Workshop publication**)
- **Saad bin Abid**: Resolving Traceability Issues of Product Derivation for Software Product Lines (Short paper), 4th International Conference for Software and Data Technologies (ICSOFT09), Sofia, Bulgaria (38% acceptance rate). (**Conference publication**)
- Ciarán Cawley, Goetz Botterweck, Patrick Healy, **Saad Bin Abid**, Steffen Thiel: “A 3D Visualisation to Enhance Cognition in Software Product Line Engineering,” 5th International Symposium on Visual Computing (ISVC09), Las Vegas, Nevada, USA, November 30-December 2, 2009. (**Conference publication**)
- **Saad bin Abid**: "Resolving Feature Dependency Implementations Inconsistencies during Product Derivation", 6th Traceability workshop (ECMFA-TW) held in conjunction with ECMFA 2010, Paris, France. (**Conference publication**)
- Dimitrios Kolovos, Louis, M.Rose, **Saad bin Abid**, Richard Paige, Fiona A.C Polack and Goetz Botterweck: "Taming EMF and GMF Using Model Transformation", ACM/IEEE 13th International Conference on Model Driven Engineering and Languages and Systems (MoDELS 2010), Oslo, Norway, Accepted at Foundation Track, October 3-8, 2010 (20.8% acceptance rate only). (**Conference publication**)
- **Saad bin Abid**: "Tracing Aspect-Oriented Patterns for identifying feature dependency inconsistencies in Software Product Lines", 15th International Workshop on Aspect-Oriented Modeling (AOM-15) in conjunction with ACM/IEEE 13th International Conference on Model Driven Engineering and Languages and Systems (MoDELS 2010), Oslo, Norway. (**Workshop publication**)
- **Saad bin Abid**: “Analyzing Runtime feature dependency Relationships in Product Line Assets”, 20th Asian-Pacific Software Engineering Conference (APSEC 2013), 3-5 December, Bangkok, Thailand. (**Conference publication**)

Appendix G: Interview Transcripts

Participant-1

Evaluation Questions and Answers

Question 1: What are the important factors in your opinion are addressed in my research work?

Answer: Very often there is a mismatch between the feature descriptions in the problem space and their implementation in the solution space, this thesis addresses this issue and proposes a solution of managing consistency.

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Answer: Not all the different kinds of dependencies can be tracked, detected or resolved automatically. Here is a summary of the inconsistencies described in the research and my comments:

Inconsistency	Importance (my view)	Comments
No Runtime Feature Dependencies Implementation	Medium	Maybe possible to detect, if there is a matching element in the solution space
Incompatibility between the Specified Type and its Respective Implementation	High	Not easy to detect, as this would mean an automated analysis of the behaviour of the implemented code
Incompatible Specified Features Pair and its Respective Implementation Pair	Low	Easy to detect
Incomplete Implementation of the RTFD	high	Not easy to detect, as the feature descriptions are typically requirements in a high level and there is no way of detecting whether some code

		fulfils ALL of those requirements
Implementation of RTFD Not Preserving the Specified Direction	Low	Easy to detect

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Answer: Because evolution is inevitable and the chances of mismatch are high

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Answer: The process seems easy to follow – but requires some knowledge in model-based engineering, the available tools and technologies.

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Answer: Yes, the error markers are a good way of displaying errors. Although in some cases, it was not clear where in the model the actual inconsistencies occurred (as they were displayed in the problem view only).

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

Answer: Yes, but only for some kinds of inconsistencies. Some inconsistencies require an analysis of the behaviour of the code, which is not easy to achieve only through static analysis. See table in question 2.

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Answer: Yes, this is good work in the right direction. But of course, there are many things to be done before these results can be put in real-world practice.

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Answer: A set of design patterns, best practice guides or a collection of DOs and DONTs is always helpful for practitioners.

Participant-2

Evaluation Questions and Answers

Question 1: What are the important factors in your opinion are addressed in my research work?

Answer: The research aims to support a product line engineer in (semi)-automatically checking the consistency of runtime feature dependencies between their specification and respective implementations

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Answer: you have listed 4-5 types of inconsistencies, and it is hard to rank them, since all seem important.

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Answer: It is important to check the implemented behaviour with the specified behaviour because one does not want unreliable piece of software. New features may be introduced later in the development phase leading towards behavioural challenges.

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Answer: it seems easy from the simple example demonstrated, although I would not be able to intuitively perform the steps – unless I receive clear instructions on the steps needed to perform each process. (But I guess it is less important to have a good UI for this project.) Another question is if this approach is practically scalable with many features and their implementations to choose from.

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Answer: Yes, if you are used to the Eclipse environment (which I am). Is it possible to highlight the errors in the models as well?

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

Answer: Of course.

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Answer: Yes. If the expected results are to check/find all inconsistencies, then this would need further validation? Will this approach be scalable in a real case?

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Answer: If I understood it correctly, the design patterns are defined manually, and the inconsistency checking is based on these patterns? If that is correct, then the whole solution depends on having the appropriate patterns in place.

Participant-3

Evaluation Questions and Answers

Question 1: What are the important factors in your opinion are addressed in my research work?

Answer: Automate or semi-automate inconsistency checking

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Answer: Two types of inconsistencies are most important: *Specified Type and its Respective Implementation* and *Specified Features Pair and its Respective Implementation Pair*

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Answer: It is important because we need to ensure the application/product is working as we defined.

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Answer: Yes, it is easy to perform the process

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Answer: Yes, it is very useful. The description in the demo is good.

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

I don't think the static analysis is good enough for the verification challenge

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Answer: Yes, I think it can

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Answer: It is important as the design patterns can make the inconsistency checking easier and more efficient.

Participant-4

Evaluation Questions and Answers

Question 1: What are the important factors in your opinion are addressed in my research work?

Answer: Supporting checking the consistency of feature dependencies between features in feature models (i.e., specifications) and the implemented features in source code at runtime with a tool well-integrated into the Eclipse IDE and making use of existing solutions (AOP, EVL, ...).

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Answer: Dependency in model but not in code, Dependency in code but not in model, Incompatible type of dependency (one type in model, the other in code).

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Answer: Because otherwise specification and implementation will become more and more incompatible over time as changes are made to the code and the whole sense of having an abstract representation of features and their dependencies in an more easy to perceive form as looking at the actual code will get lost when the product line evolves/changes over time.

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Answer: Performing the checks is really easy and straightforward. What is missing in my opinion are “quick fixes”, i.e., allowing the user to select one of the inconsistencies in the error view and apply one of possibly several possible quick fixes. For example, when the inconsistency points towards a missing dependency on specification level (that dependency exists in code but not in the model), then it should be possible to automatically create the missing dependency in one click without requiring the user to do this manually.

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Answer: Yes, they help to understand and resolve the inconsistencies. I’m not sure all kinds of possible inconsistencies are covered but for the ones that are covered, the error markers are really useful.

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

Answer: Definitely, several researchers are working in this direction, especially in the Feature Oriented Software Development (FOSD) community, static code analysis becomes a more and more important tool these days.

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Answer: Well, the key gap here is that the solution has not (yet) been applied to a real industrial application, so it is not possible to evaluate whether it will scale for a real life problem. One suggestion to overcome this issue is to perform simulations, i.e., generate large numbers of different features and dependencies among them and seed many defects and then see whether the approach still performs well.

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Answer: Design patterns are helpful to identify possible dependencies which in turn can be checked for inconsistencies.

Participant-5

Evaluation Questions and Answers

Question 1: What are the important factors in your opinion are addressed in my research work?

Answer: Consistency checking of RTFDs implementations with respect to their specifications using the following, 1) developed meta-models/DSL, 2) pattern detection and 3) constraints applied for errors/bugs identification

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Answer: Both Structural and Behavioural inconsistencies

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Answer: Not checking inconsistencies during the development phase will lead towards more effort in finding errors late in the development life cycle. Not checking for inconsistencies earlier also lead towards mismatch of implementation with respect to the architecture of the software system.

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Answer: Yes, the process is easy to follow and the covered inconsistency scenarios were elaborated well in the demo.

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Answer: Markers were effective in identifying and comprehending the erroneous scenarios discussed in the dissertation.

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

Answer: There are other ways to analyse the source code. For example, control flow graphs (CFGs) and sometimes executing the test suites can provide some insights to the source code. But usually its static analysis that is performed as compared to executing the source code.

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Answer: Although the MDD solution is a good effort towards understanding certain behavioural inconsistency scenarios, the approach requires some advancement in terms of automating the mapping process of the specifications and the implementations.

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Answer: Design patterns are templates to solve problems that are occurring every now and then. As a software architect, I feel that they are very important in designing a robust software system architecture. Identifying software design patterns can help understand the software architecture and perform consistency checking of software system architecture with respect to their implementation.

Participant-6

Evaluation Questions and Answers

Question 1: What are the important factors in your opinion are addressed in my research work?

Answer: Checking the constancies of runtime feature dependencies in a Software Produce Line.

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Answer: Structural inconsistencies and behavioural inconsistencies.

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Answer: Checking consistencies can help the product line engineer to detect potential faults in product development using a software product line.

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Answer: Yes. The demo is clear and easy to understand.

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Answer: Yes. The error markers are very informative and provide a clear guide to resolve the inconsistencies.

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

Answer: Yes. It is a good approximation of the actual aspect-oriented pattern based implementation in the source code.

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Answer: Yes.

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Answer: Very important. The product line engineer can check if a particular type of runtime feature dependency is implemented using a particular design pattern.

Participant-7

Evaluation Questions and Answers

Question 1: What are the important factors in your opinion are addressed in my research work?

Answer: Consistency checking code should allow more errors to be detected at compile time rather than at runtime, thus saving maintenance cost for the developer and downtime costs for the customer. Making this checker an IDE plugin is of great benefit to the developer, who can run the checks within the code/compile/test cycle.

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Answer: Behavioural inconsistencies can cause huge problems in running software systems. They are typically not caught at compile time, but exhibit strange systems behaviour that is difficult to reproduce. In some cases they can cause system outages and cost customers large revenue loss.

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Answer: It is essential to perform these consistency checks as the behaviour must match both the interface specification and the functional specification, in order to have a correctly functioning system. Without such checks the system will behave in strange ways that are difficult to debug, or in the worst case scenario it will crash, if no runtime binding can be found for the implementation.

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Answer: Yes, the visualization model of the software seems very easy to use. It is well structured and easy to navigate between the different elements.

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Answer: Yes, they highlight the specific problem adequately

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

Answer: Yes, I think this is the best way to do it. You want to highlight the problems to the developer within the IDE, so that they can be fixed within the code/compile/test cycle.

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Answer: Yes, modelling the software allows all of its attributes and mandatory relationships to be checked and any deficiencies highlighted.

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Answer: Yes, I think this work depends on knowing the design patterns apriori, as this is the basis on which to test for completeness of the specification.

Participant-8

Evaluation Questions and Answers

Question 1: What are the important factors in your opinion are addressed in my research work?

Answer: In my view, the most important contributions of this work are the consistency checking and pattern detection facilities, and the developed product lines DSL.

Question 2: What are the types of inconsistencies that are most important to be taken care of in your opinion in my research work?

Answer: Both structural and behavioural inconsistencies are important.

Question 3: Why do you think it is important to perform consistency checking of behaviour with respect to its specification?

Answer: To obtain early feedback about any inconsistencies that may have been introduced during the development process.

Question 4: Do you find it easy to perform the processes (i.e., please consider the Demo video) in order to check for inconsistencies related to the runtime feature dependencies?

Answer: Yes. The tooling would benefit from a bit of polishing but most of the necessary building blocks appear to be in place.

Question 5: In your opinion, do the error markers helped you to understand and can serve as guidelines to you to resolve the inconsistencies?

Answer: Yes

Question 6: In your opinion, does static analysis of source code prove to be a good candidate for performing the verification challenge addressed in my research work?

Answer: Yes

Question 7: Do you think that the designed and implemented Model-Driven solution in my research work can provide the expected results?

Answer: The solution has the potential to be used in practice.

Question 8: In your opinion, how much important is it to identify the design patterns to perform the inconsistency checking in my research work?

Answer: This is quite important.