REPRESENTING VARIABILITY IN SOFTWARE ARCHITECTURE

UMAIMA HAIDER

A thesis submitted in partial fulfilment of the requirements of the University of East London for the degree of Doctor of Philosophy

April 2016

Abstract

Software Architecture is a high level description of a software intensive system that enables architects to have a better intellectual control over the complete system. It is also used as a communication vehicle among the various system stakeholders. Variability in software-intensive systems is the ability of a software artefact (e.g., a system, subsystem, or component) to be extended, customised, or configured for deployment in a specific context. Although variability in software architecture is recognised as a challenge in multiple domains, there has been no formal consensus on how variability should be captured or represented.

In this research, we addressed the problem of representing variability in software architecture through a three phase approach. First, we examined existing literature using the Systematic Literature Review (SLR) methodology, which helped us identify the gaps and challenges within the current body of knowledge. Equipped with the findings from the SLR, a set of design principles have been formulated that are used to introduce variability management capabilities to an existing Architecture Description Language (ADL). The chosen ADL was developed within our research group (ALI) and to which we have had complete access. Finally, we evaluated the new version of the ADL produced using two distinct case studies: one from the Information Systems domain, an Asset Management System (AMS); and another from the embedded systems domain, a Wheel Brake System (WBS).

This thesis presents the main findings from the three phases of the research work, including a comprehensive study of the state-of-the-art; the complete specification of an ADL that is focused on managing variability; and the lessons learnt from the evaluation work of two distinct real-life case studies.

Table of Contents

Abstract ... i
List of Figures ... vi
List of Tables ... viii
Acronyms ... ix
Acknowledgment ... x
Dedication ... xi

Part I: INTRODUCTION ... 1
Chapter 1: Introduction ... 2
1.1 Motivation ... 2
1.2 Problem Statement: Research Questions ... 5
1.3 Contributions ... 5
1.4 Organisation of Thesis ... 8

Chapter 2: Research Methodology ... 12
2.1 Introduction ... 12
2.2 Systematic Literature Review (SLR) ... 13
2.2.1 SLR Review Protocol ... 14
2.2.2 SLR Research Questions ... 15
2.2.3 Search Strategy ... 15
2.2.4 Study Selection ... 18
2.2.5 Quality Assessment Criteria ... 20
2.2.6 Data Extraction and Synthesis ... 21
2.3 Language and Framework Design ... 23
2.4 Case Study Research ... 25
2.5 Summary ... 26

Part II: STATE-OF-THE-ART ... 28
Chapter 3: Background ... 29
3.1 Introduction ... 29
3.2 Concepts and Terminology ... 29
3.2.1 Software Architecture ... 29
3.2.2 Variability ... 31
3.3 Architecture Description Languages (ADLs) ... 32
3.3.1 Analysis of existing ADLs ... 34
3.3.2 Limitations in existing ADLs ... 42
3.4 Conclusion ... 44

Chapter 4: Systematic Literature Review 46
4.1 Introduction 46
4.2 Data and Analysis 47
4.2.1 Demographic Data 47
4.2.2 Geographical Distribution 49
4.3 Discussion of SLR Research Questions 50
4.3.1 SLR.RQ1: *What approaches have been proposed to represent variability in software architecture?* 50
4.3.2 SLR.RQ2: *What is the quality of the research conducted in the reported approaches?* 55
4.3.3 SLR.RQ3: *What is the context and areas of research of the studies employing variability in software architecture?* 58
4.3.4 SLR.RQ4: *What are the limitations of the existing approaches to represent variability in software architecture?* 63
4.4 Threats to Validity & Limitations 63
4.5 SLR Update: Work beyond Search Period 66
4.6 Conclusion 67

Part III: ALI 69
Chapter 5: ALI Initial Version 70
5.1 Introduction 70
5.2 Rationale 71
5.2.1 Flexible interface description 72
5.2.2 Architectural pattern description 73
5.2.3 Formal syntax for capturing meta-information 75
5.2.4 Linking the feature and architecture spaces 76
5.3 ALI Constructs and Notations 76
5.3.1 Meta Types 77
5.3.2 Interface Types 79
5.3.3 Connector Types 81
5.3.4 Component Types 84
5.3.5 Pattern Templates 91
5.3.6 Features 94
5.3.7 System 96
5.4 Limitations in original version 98
5.4.1 Architectural artefact reusability 98
5.4.2 Limited support for behavioural description 99
5.4.3 Lack of support for graphical representation 100
5.4.4 Lack of support for architectural views 101
5.5 Summary 102

Chapter 6: ALI V2 104
6.1 Introduction 104
6.2 Design Principles 105
6.3 Conceptual Model 109
6.4 Textual Constructs and Notations 111
6.4.1 Meta Types 112
6.4.2 Features 114
6.4.3 Interface Templates 115
6.4.4 Interface Types 118
6.4.5 Connector Types 119
6.4.6 Component Types 121
6.4.7 Pattern Templates 125
6.4.8 Product Configurations 126
6.4.9 Events 127
6.4.10 Conditions 127
6.4.11 Scenarios 128
6.4.12 Transaction Domains 128
6.4.13 Viewpoints 132
6.4.14 System 132
6.5 Graphical Constructs and Notations 133
6.5.1 Structural Notation 133
6.5.2 Behavioural Notation 136
6.6 Semantics 140
6.6.1 Structural Semantics 141
6.6.2 Behavioural Semantics 142
6.7 Summary and Changes to ALI Initial Version 146

Part IV: CASE STUDIES 150
Chapter 7: Case Study:Asset Management System 151
7.1 Introduction 151
7.2 Description of the AMS Case Study 152
7.3 AMS Architecture Representation Using ALI V2 154
7.3.1 AMS Meta Types 154
7.3.2 AMS Features 155
7.3.3 AMS Interface Templates 156
7.3.4 AMS Interface Types 157
7.3.5 AMS Connector Types 159
7.3.6 AMS Component Types 160
7.3.7 AMS Product Configurations 165
7.3.8 AMS Events 166
7.3.9 AMS Conditions 167
7.3.10 AMS Scenarios 168

7.3.11 AMS Transaction Domains 169
7.3.12 AMS Viewpoint 176
7.3.13 Asset Management System (AMS) 176
7.4 AMS Evaluation 178
7.5 Discussion 184
Chapter 8: Case Study: Wheel Brake System 186
8.1 Introduction 186
8.2 Description of the WBS Case Study 187
8.3 WBS Architecture Representation Using ALI V2 189
8.3.1 WBS Meta Types 189
8.3.2 WBS Features 190
8.3.3 WBS Interface Template 191
8.3.4 WBS Interface Types 192
8.3.5 WBS Component Types 193
8.3.6 WBS Product Configuration 197
8.3.7 WBS Events 197
8.3.8 WBS Conditions 198
8.3.9 WBS Scenarios 199
8.3.10 WBS Transaction Domain 199
8.3.11 WBS Viewpoint 206
8.3.12 Wheel Brake System (WBS) 206
8.4 WBS Evaluation 209
8.5 Discussion 215
Part V: CONCLUSION 217
Chapter 9: Conclusion and Future Perspectives 218
9.1 Summary and Conclusion 218
9.2 Future Perspectives 223
9.2.1 Short Term 223
9.2.2 Long Term 225
References 228
Appendices 242
Appendix A: Systematic Literature Review (SLR) A-1
Appendix B: ALI V2 Event Traces Notation Comparison B-1
Appendix C: ALI V2 BNF C-1
Appendix D: AMS Case Study D-1
Appendix E: WBS Case Study E-1

List of Figures

Figure 1: Organisation of thesis ____ 9

Figure 2: SLR review protocol ____ 14

Figure 3: The search and selection process ____ 20

Figure 4: ALI redesign process ____ 24

Figure 5: Publications per year ____ 47

Figure 6: Publication outlet ____ 48

Figure 7: Highly occurring publication venues ____ 49

Figure 8: Primary study distribution per country ____ 50

Figure 9: Quality assessment scores of studies (overall) ____ 56

Figure 10: Overall quality assessment scores per question ____ 57

Figure 11: Research context ____ 59

Figure 12: Research relevance ____ 61

Figure 13: Breakdown of primary studies over research areas ____ 62

Figure 14: An example architecture of a simple web service (Bashroush et al., 2006) 73

Figure 15: A simple architecture assembled from a number of components using two pattern templates: PipesAndFilters and ClientServer (Bashroush et al., 2006) ____ 74

Figure 16: ALI V2 conceptual model ____ 110

Figure 17: Graphical structural representation for a system ____ 135

Figure 18: Graphical behavioural representation of transaction domain TransactionDomain1 ____ 138

Figure 19: Component Comp1 interactions in transaction domain TransactionDomain1 ____ 140

Figure 20: AMS component type Internal_EquityData ____ 162

Figure 21: Graphical behavioural representation of transaction domain PortfolioValuation ____ 173

Figure 22: Graphical structural representation of transaction domain PortfolioValuation ____ 174

Figure 23: AMS component Portfolio_GUI interactions in transaction domain PortfolioValuation ____ 175

Figure 24: Wheel Brake System (ARP4761, 1996) ____ 188

Figure 25: WBS component type Aircraft_BrakePedal ____ 194

Figure 26: Graphical behavioural representation of transaction domain WheelDecelerationOnGround ____ 204

Figure 27: WBS component Electrical_Pedal interactions in transaction domain WheelDecelerationOnGround ____ 205

Figure 28: WBS graphical structural notation ____ 208

List of Tables

Table 1: Manually searched conferences and workshops ______ 17

Table 2: Data extraction form ______ 23

Table 3: Variability representation approaches ______ 51

Table 4: Quality assessment scores of studies (per question) ______ 57

Table 5: Research context with study identifier ______ 59

Table 6: Research relevance with study identifier ______ 60

Table 7: Breakdown of primary studies over research areas ______ 62

Table 8: ALI V2 transaction domain textual notation ______ 129

Table 9: ALI V2 graphical structural notation ______ 134

Table 10: ALI V2 event traces notation ______ 137

Table 11: ALI initial version Vs ALI V2 ______ 149

Table 12: List of acronyms for AMS component types interfaces ______ 163

Table 13: AMS interface templates notations ______ 163

Table 14: List of acronyms for AMS connector interfaces ______ 165

Table 15: AMS evaluation ______ 183

Table 16: List of acronyms for WBS component types interfaces ______ 195

Table 17: WBS evaluation ______ 210

Table 18: Case studies criteria ______ 222

Acronyms

AADL Architecture Analysis Description Language

ADL Architecture Description Language

ADLARS Architecture Description Language for Real-time Systems

ALI Architecture Description Language for Industrial Applications

AMS Asset Management System

BNF Backus Naur Form

CASE Computer Aided Software Engineering

CSP Communicating Sequential Processes

EBNF Extended Backus Naur Form

IoT Internet of Things

IS Information System

JavaCC Java Compiler Compiler

SLR Systematic Literature Review

SOA Service Oriented Architecture

SPL Software Product Lines

UCM Use Case Maps

UI User Interface

UML Unified Modelling Language

WBS Wheel Brake System

WSDL Web Services Description Language

Acknowledgment

This project is by far the most significant accomplishment in my life. Various people deserve my sincere thanks for their immeasurable help to me throughout the course of this thesis. I must offer my profoundest gratitude to my thesis advisor and director of study, ***Dr Rabih Bashroush***. He has inspired me to become an independent researcher and helped me realise the power of critical reasoning. He also demonstrated what a brilliant and hard-working scientist can accomplish. His unreserved help and guidance has led me to finish my thesis step by step. His words can always inspire me and bring me to a higher level of thinking. What I learnt from him is not just how to write a thesis to meet the graduation requirement, but how to view this world from a new perspective. Without his kind and patient instruction, it was impossible for me to finish this thesis.

My sincere thanks must also go to the member of my thesis advisory ***Dr Usman Naeem.*** He generously gave his time to offer me valuable comments toward improving my work.

I also want to give gratitude to ***Dr John McGregor*** from Clemson University, South Carolina, USA, who helped and offered me his inspiring suggestions for my research work.

I would also like to express my thankfulness to the School of Architecture, Computing and Engineering (ACE) and Graduate School at University of East London (UEL) for their support in my research.

Last but not least, I would like to express my gratitude to my parents for their unfailing emotional support and also, thank for heart-warming kindness.

Dedication

I would like to dedicate this Doctoral dissertation to my lovely parents. There is no doubt in my mind that without their constant loves, endless support and encouragements I could not have completed process.

Part I

INTRODUCTION

Chapter One

1 Introduction

"The secret of getting ahead is getting started."

--Mark Twain

1.1 Motivation

Within the software engineering community, the concept of software architecture started to emerge as a distinct discipline in 1990 (Kruchten, Obbink and Stafford, 2006) which led to an explosion of interest during the 1990s and 2000s, referred to as the "Golden Age of Software Architecture" (Shaw and Clements, 2006). Today, software architecture has moved towards the point of growing from its adolescence in research laboratories to the responsibilities of maturity, which was predicted by Shaw (Shaw, 2001) over a decade ago. However this does not mean that the time for research, innovation, and enhancement is a thing of the past. In fact, it brings an additional responsibility to show not just that ideas are promising (adequate grounds to continue research) but also that they are effective (indispensable grounds to move into practice) (Shaw, 2001). In other words, it is a coupling between ongoing research and practical application to make new ideas practical. For this reason, software architecture has drawn considerable attention from both academia and industry.

The increasing complexity of software and the critical nature of its use are driving a rapid maturation of the field of software architecture. According to Garlan (Garlan, 2014), a critical issue in the design and construction of any complex software system is its architecture: that is, its organization as a collection of interacting elements – modules, components, services, etc. Thus, a well-designed architecture ensures the quality and

longevity of a software system. A number of approaches exist that can describe a software architecture, ranging from formal notations (e.g. ADLs), semi-formal (e.g. UML) and informal (e.g. boxes and lines, videos, etc.).

Architecture Description Languages (ADLs) are currently considered to be viable tools for formally representing the architectures of systems at a reasonably high level of abstraction to enable better intellectual control over the systems (Bass, Clements and Kazman, 2012). An ideal ADL is considered to be both human-readable and machine readable. An ADL must be simple, understandable, encompassed by multiple architectural views and syntactically flexible. With regards to this, Lago et al. (Lago *et al.*, 2015) presented a general framework of requirements for the next generation architectural languages by taking into account current architectural needs of both the academic and industrial worlds.

Over the past two decades, a vast number of ADLs have been developed as compared to the number of ADLs reported in (Clements, 1996; Medvidovic and Taylor, 2000) but the majority of the problems still remain the same. Among those, the most common problem is that ADLs have gained wide acceptance in the research community as a means of describing system designs but their current industrial adoption level is still reported to be as low as before with some exceptions, for example, in the embedded systems domain (Bashroush *et al.*, 2005; Cuenot *et al.*, 2010; Feiler, Gluch and Hudak, 2006; Ommering *et al.*, 2000). This could be due to a number of reasons identified in (Bashroush *et al.*, 2006; Malavolta *et al.*, 2013; Woods and Hilliard, 2005), including the mismatch between their strengths and the needs of practitioners.

Many existing ADLs tend to focus on a specific aspect of a system (e.g. system structure), or are geared towards a particular application domain (e.g. embedded systems). While domain specific notations can be well tailored to serve particular application area

needs, todays systems (and systems of systems) cross traditional design boundaries, where software persists across various layers (e.g. Cyber-physical systems, Smart Cities systems, etc.). Thus, to be able to use an ADL in such domains, it would need to have the flexibility and expressiveness that allows it to stretch beyond a single application domain.

Moreover, there has recently been an increase in the usage of variability mechanisms at the architectural level (e.g. to represent product families or runtime system adaptation). Variability management allows a) the development and evolution of different versions of software and product variants, b) planned reuse of software artefacts, and c) well-organized instantiation and assessment of architecture variants (Galster *et al.*, 2014). An ADL with the capability to capture and express such complex variability exhibited in software systems would empower architects to build and model more sophisticated systems.

To overcome these aforementioned limitations, there is a need for an ADL which will be designed as a comprehensive language, suited for different types of systems, from individual systems, to product lines, and system-of-systems. A major goal of that ADL should be to provide a blend of flexibility and formalism. Flexibility is based on ease of use and be informative enough to convey the needed information to the stakeholders involved in the architecting phase. Formalism, on the other hand, paves the way for developing better tool support and automated analysis. Lastly, most importantly, the design of an ADL is to be highly customisable to provide support for a wide range of application domains.

1.2 Problem Statement: Research Questions

The goal of this thesis is to answer the following research questions:

RQ1. What approaches have been proposed to represent the variability in software architecture and what are the limitations of these approaches?

RQ2. How can variability be represented formally throughout the architectural description? Furthermore, how will this representation assist in addressing the system's stakeholder concerns, particularly in large-scale industrial systems?

RQ3. Which architectural description constructs (textual and graphical) are required to best capture system behaviour, while maintaining support for variability?

RQ4. How can ADLs be extended to support system modelling that spans multiple application domains?

This thesis proposes an approach to formally designing a system architecture that helps in answering these research questions, with a focus on its applicability in multi-scale industrial projects.

1.3 Contributions

The primary contribution of this research is a novel approach to designing the architecture of a software system, by adapting a formal process which must be both valuable and practical. More specifically, this thesis details a software architecture description technique, which has been formalised through an ADL, in order to design a system that conforms to the needs of practitioners. Furthermore, the proposed software

architecture description also focuses on natively representing the system's variability. The contributions of this thesis are as follows:

- **A comprehensive review on representing variability in software architecture.** To conduct this review in a formal way, a Systematic Literature Review (SLR) methodology has been adopted in order to have a credible, repeatable and fair evaluation of the available studies in this area. This review captures and summarises the state-of-the-art in representing variability in software architecture in a manner accessible to practitioners working in this area. This allows practitioners to choose the best approach to describe variability that fits into their system, and assists researchers in identifying areas requiring further research. Furthermore, this review assesses the quality of the literature and the nature of the different approaches used to represent variability in software architecture.

- **Identification of flexible ADL design principles to facilitate representation of today's multi-domain systems of systems.** The design principles satisfy the current industrial requirements of practitioners that are required when designing an architectural language. In addition, principles have been designed that consider both the structural and behavioural architectural descriptions of the system.

- **Enrichment of an existing ADL from the perspective of its industrial adoption.** ALI (**A**rchitecture Description **L**anguage for **I**ndustrial Applications), an academia originated ADL has been enhanced (referred as ALI V2 in this thesis) by considering the needs of the industrial practitioners in the following ways:

    - **Strengthened variability representation in the architecture description** accommodates a variety of products, including a product with

variable features. Variability has been considered as a first-class element that is treated equally in both the structural and behavioural descriptions of the language. Furthermore, variability is taken into consideration from the initial requirements stage through to the architectural design stage, as a collection of features and conditions.

- **A high-level description of the architectural language** in the form of a conceptual model. This is designed to demonstrate the relationship between the structural and behavioural constructs of the language.

- **Introduction of a new behaviour description section** including: events, transactions and transaction domains. This aspect of the architectural description is given detailed exposition, with a clear separation of concern from the structural description (in terms of both textual and graphical representation) of the system, while maintaining consistency and completeness.

- **Introduction of graphical notation for representing architectural behaviour,** which has been designed so that it can be easily understood by different system stakeholders, such as management and technical stakeholders. The notation has been designed and configured as an event trace that demonstrates a particular functional behaviour of the system. Along with this, components involved in a particular functional behaviour have been visualised with their own interactions in a well-defined sequential manner.

- **Formal modelling of the notation semantics** have been provided for both the structural and behavioural descriptions of the language, in a stand-alone fashion. The language semantics have been defined using a formal

language, as it affords precise and unambiguous semantics. The formal languages used were: mathematical set theory to define structural semantics, and CSP (Communicating Sequential Processes) to define behavioural semantics of the language.

- **Evaluation of the ADL in multiple domains using two case studies.** The proposed ADL (i.e. ALI V2) has been applied to two different case studies which have different natures and sizes. They are: 1) an Asset Management System (AMS) which is a generic information system that helps in managing investment decisions of a large-scale investment portfolio for a bank, and 2) a Wheel Brake System (WBS) which is an embedded system that stops/decelerates the wheels of a commercial aircraft.

1.4 Organisation of Thesis

This thesis is structured into nine chapters, each devoted to describing a specific aspect of the research, and the structure is illustrated in Figure 1.

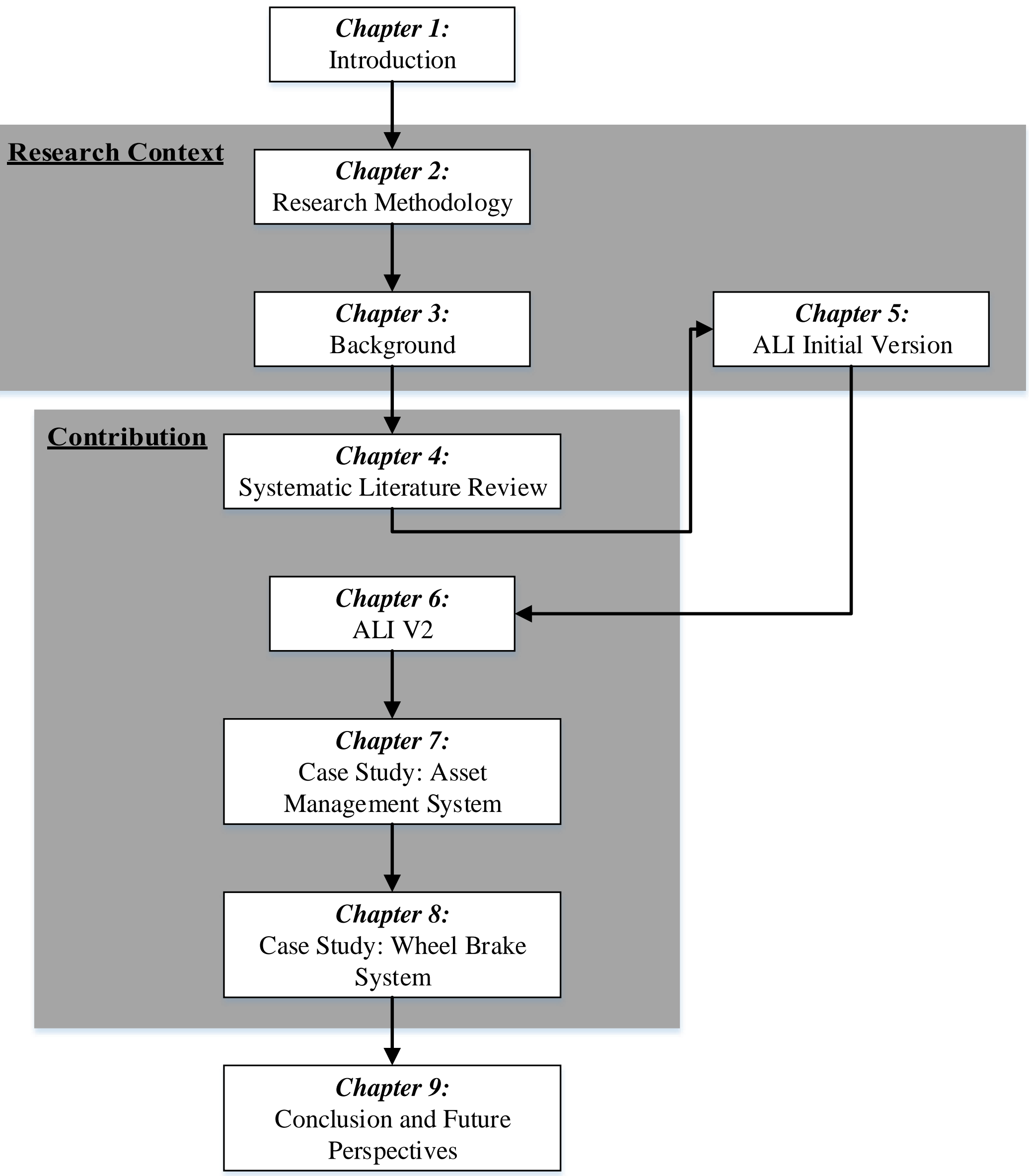


Figure 1: Organisation of thesis

Chapter 2 presents the research methods used to carry out this research work. Here, the current state-of-the-art on representing variability in software architecture is captured through a SLR. Then, the ALI language is redesigned to meet the current industrial requirements, and is subsequently evaluated on two real-life case studies.

Chapter 3 provides background information on the research areas of this thesis. Specifically, the concepts of software architecture, variability and ADL re described. In addition to this, a critical analysis of existing ADLs is presented, which focuses on the areas that are relevant to this thesis, along with a discussion of their limitations.

Chapter 4 presents a detailed analysis of the primary studies identified in the previous chapter, regarding the representation of variability in software architecture. The nature of the different approaches used to represent variability in software architecture, the quality of the work conducted, the research context and area, and the limitations within the studies, are all assessed.

Chapter 5 describes the original form of ALI, which was the version prior to the commencement of this research. The chapter discusses its rationale and basic language constructs. Furthermore, this chapter reveals the limitations that exist within this version, which are identified by considering the current challenges in architectural languages, especially from an industrial perspective.

Chapter 6 introduces the latest form of ALI (referred as ALI V2), which is the current at the time of publication of this thesis. This chapter presents the design principles on which the ALI V2 is based on and how they have been leveraged in order to tackle the research problems addressed in this thesis. This chapter also describes the conceptual model, as well as the language constructs (structural and behavioural both) of the ALI V2 and its formal semantic definition.

Chapter 7 presents a case study where the proposed software architectural language, ALI V2, has been applied. The case study is called the Asset Management System (AMS), part of the Information System (IS) domain, and it is a system that supports decision-making and executing investment decisions for a large-scale investment portfolio in an investment bank. At the end of this chapter, the AMS has been evaluated with respect to the limitations identified in the current ADL literature in Chapter 2 and how it addresses the ALI V2 design principles explained in Chapter 6. Finally, the results obtained from the AMS architectural design have been discussed.

Chapter 8 presents a second case study where the proposed software architectural language, ALI V2, has been applied. The case study is called the Wheel Brake System (WBS), part of the embedded system domain, and it is a system that controls braking the wheels of a commercial aircraft. As in Chapter 7, this chapter concludes with an evaluation of WBS with respect to the limitations identified in the current ADL research literature and how the system addresses the ALI V2 design principles explained in Chapter 6. The chapter concludes with a discussion of the results obtained from the WBS architectural design.

Chapter 9 concludes this thesis by providing a comprehensive summary of the proposed approach for developing a system architecture, with a focus on its industrial adoption. This chapter also discusses the future directions for this research.

Chapter Two

2

Research Methodology

"Highly organized research is guaranteed to produce nothing new."

--Frank Herbert, Dune

2.1 Introduction

This chapter details the research methodology employed in this thesis, which is guided towards representing variability in software architecture. Furthermore, the research methodology is outlined in such a way that it addresses the research questions (described in Section 1.2) for this thesis.

Three main research methods were determined for this thesis. Firstly, the objective was to provide a snapshot of the state-of-the-art on representing variability in software architecture while assessing the quality of work conducted and the nature of the different approaches. A systematic literature review (SLR) was conducted to achieve this objective. Subsequently, grounded theory was used to conduct the analysis and to draw conclusions from the data, thus minimising threats to validity.

Secondly, Architecture Description Language (ADL), a formal architecture-description technique used to represent variability in software architecture was adopted as a result of the SLR. This was done by redesigning an existing ADL –ALI (Bashroush *et al.*, 2008) that captures the architectural description (both structural and behavioural), while maintaining the support for variability. The language was also designed with the intention of meeting the current industrial requirements, which can then easily be applied to any system irrespective of their size.

Lastly, the proposed language has been evaluated via its implementation in two different real-life case studies. Both the case studies comprise distinct characteristics that demonstrate the broader scope of the proposed language.

The following section describes how the SLR was conducted. Section 2.3 presents the strategy followed to design the language, while the methodology to evaluate the language via case studies is described in Section 2.4. Finally, Section 2.5 summarises the defined research methods used to carry out the work in this thesis.

2.2 Systematic Literature Review (SLR)

The main objective of the proposed research methodology was to identify, summarize and analyse all approaches that have been proposed or used to represent variability in software architecture. To achieve this, a SLR referred to as systematic review or review hereafter is conducted. A systematic review is a well-defined and methodical way to identify, evaluate, and synthesize the available evidence concerning a particular technology to understand the current direction and status of research or to provide background in order to identify research challenges (Kitchenham and Charters, 2007). This method was chosen because of the requirement to have a credible, repeatable and fair evaluation of the available studies on representing variability in software architectures.

In this section, SLR review protocol is defined in Section 2.2.1 and subsequently, its steps (Section 2.2.2 – 2.2.6) that are used to identify the current literature on representing variability in software architecture.

2.2.1 SLR Review Protocol

A significant step of the systematic literature review process is the development of the protocol (Figure 2). The protocol specifies all of the steps and procedures followed by researchers during a review to neutralize author bias and minimize threats to validity (discussed in Chapter 4). The review protocol is one of the main aspects that differentiate SLRs from conventional literature reviews. The protocol adopted for this work was reviewed by an independent researcher.

Identify research questions (Section 2.2.2)
Define search strategy (Section 2.2.3)
Define study selection criteria (Section 2.2.4)
Define quality criteria (Section 2.2.5)
Define data extraction and synthesis (Section 2.2.6)

Figure 2: SLR review protocol

The protocol starts by defining the research questions, followed by a definition of the search strategy process to be followed (Sections 2.2.2 and 2.2.3). Then, inclusion and exclusion criteria are developed to provide a systematic way of selecting among identified primary studies (Section 2.2.4). Clear criteria for assessing the quality of studies are then identified (Section 2.2.5). Finally, the data elements to be extracted from the primary studies to help answer the research questions are identified (Section 2.2.6). Once the data is extracted, grounded theory is used to help analyse and draw conclusions to minimize threats to validity (discussed in Chapter 4).

2.2.2 SLR Research Questions

We aim at research questions important not only to researchers, but also to practitioners. Therefore, SLR covers the following research questions:

SLR.RQ1: What approaches have been proposed to represent variability in software architecture?

SLR.RQ2: What is the quality of the research conducted in the reported approaches?

SLR.RQ3: What is the context and areas of research of the studies employing variability in software architecture?

SLR.RQ4: What are the limitations of the existing approaches to represent variability in software architecture?

SLR.RQ1 is motivated by the need to describe the state-of-the art of how existing approaches represent variability. In order to understand the overall quality of the research conducted in the domain, SLR.RQ2 was formulated. SLR.RQ3 helps better understand the applicability of each of the identified approaches, and to analyse any recurring patterns in different domain, while helping practitioners navigate through the reviewed approaches. We pose SLR.RQ4 to provide an overview of existing challenges in order to provide the directions for further research.

2.2.3 Search Strategy

The search string used in this review was constructed using the following strategy and criteria:

- Derive main terms based on the topics being researched and research questions;

- Determine and include synonyms, related terms, and alternative spelling for major terms;
- Check the keywords in all relevant papers that the researchers were already aware of and using initial searches on the relevant databases;
- Include other relevant terms where there is a possibility of identifying further material related to the topic.
- Incorporate alternative spellings and synonyms using Boolean "OR";
- Link main terms using Boolean "AND";
- Pilot different combinations of the search terms.

Following this strategy, and after a series of test executions and reviews, the search string was constructed which is defined below:

<< (Variability OR Variabilities) AND (reference architecture OR software architecture OR architectural) >>

The primary studies in seven digital sources (1. IEEExplore; 2. ACM Digital library; 3. Citeseer; 4. SpringerLink; 5. Google Scholar; 6. ScienceDirect and 7. SCOPUS) were searched. As an indication of inclusiveness, the results were checked against relevant literature the researchers were aware of, and all of the papers checked were found in the identified primary studies. Papers that were not able to access online were acquired by contacting the relevant authors via email.

As an additional measure to ensure the comprehensiveness of the review, a manual check was conducted of the proceedings of the major conferences and workshops that the researchers were aware of that published relevant papers. Table 1 presents the list of conferences and workshops that were searched manually. SATURN conference were also

considered due to its relevance; however, as the conference only publishes presentations rather than full research papers, it was excluded for failing to meet one of our inclusion criteria (discussed in the next section).

SOURCE	ACRONYM	YEAR
International Conference on Software Engineering	ICSE	1991-2015
Foundation of Software Engineering	FSE	1991-2014
Working IEEE/IFIP Conference on Software Architecture	WICSA	2004-2015
Workshop on Variability Modeling of Software-Intensive Systems	VaMoS	2007-2015
Quality of Software Architecture	QoSA	2005-2015
European Conference on Software Architecture	ECSA	2007-2014
Systems and Software Product Line Conference	SPLC	1996-2015

Table 1: Manually searched conferences and workshops

The publication lists of known researchers publishing in the area were also checked manually. Finally, for the primary studies identified, forward and backward reference checking was conducted. For backward reference checking, the reference list of the papers searching for any potential primary studies that had been missed were examined. Similarly, for forward reference checking, search engines to identify citations to the primary studies that could be relevant to the review were used. This process helped to identify a number of additional potential primary studies. In terms of timeline, the primary studies published between January 1991 and July 2015 were searched. The start date was set to be as early as possible (the earliest relevant primary studies identified were published in 2002). The search stage of this SLR was concluded in July 2015 (hence the end date), after that, the data extraction stage commenced.

2.2.4 Study Selection

The outcome from the different initial searches on digital libraries, manual searches, and known author searches, produced 1045 primary studies. After initial screening of this SLR based on title, abstract and keywords and excluding papers that were irrelevant or duplicates, 131 primary studies were selected. These remaining primary studies were subject to a more detailed review (of the full papers) where each paper was checked thoroughly. This process resulted in 25 papers being excluded. Of the remaining 106 primary studies, forward references (papers citing the primary study) and backward references (papers cited in the primary study) were followed which helped to identify a further 11 studies. The resulting 117 papers were then reviewed by applying the following inclusion and exclusion criteria:

- **Inclusion criteria:**

IC1: The primary study proposes or uses an approach to represent variability in software architecture;

IC2: When several reports of the same study existed in different sources, the most complete and recent version of the study was included in the review.

- **Exclusion criteria:**

EC1: The primary study addresses variability but not in software architecture domain.

EC2: The primary study is in the domain of software architecture, but does not consider variability. A paper that does not address variability along with software architecture has no value to answer our research questions.

EC3: Lack of enough details about representing variability in software architecture to make any useful contribution towards addressing research questions.

EC4: The primary study is a short (less than 3000 words) or symposium paper, abstract, keynote, opinion, tutorial summary, panel discussion, technical report, presentation slides, compilation of work (for instance, from a conference or workshop or special issue) or a book chapter. Books/book chapters were only included if they were conference/workshop proceedings (e.g., as part of the LNCS or LNBIP series) and are available through data sources are included in our review.

This led to the exclusion of 59 papers leaving us with 58 primary studies. The study selection process is summarized in Figure 3.

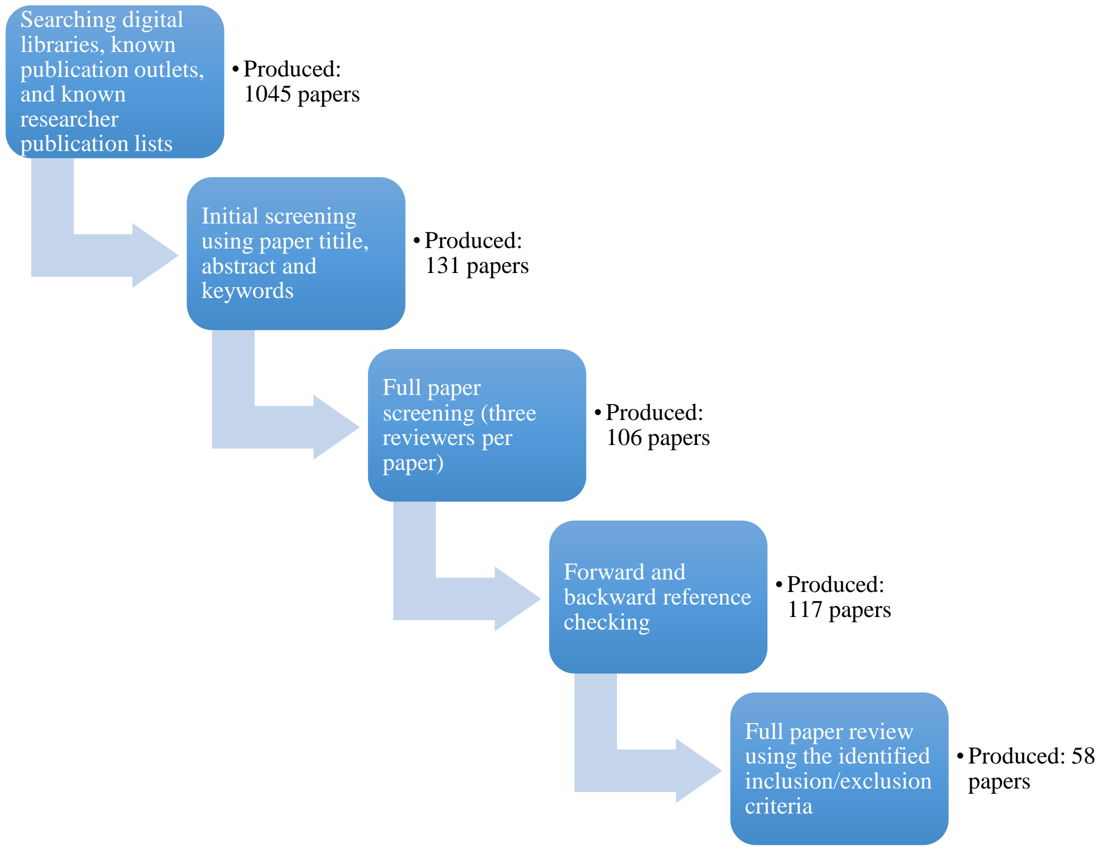


Figure 3: The search and selection process

2.2.5 Quality Assessment Criteria

We adopted the quality assessment strategy defined by (Kitchenham and Charters, 2007) where each primary study was assessed using the following quality criteria:

QA.Q1. Is there a rationale for why the study was undertaken?

QA.Q2. Is there an adequate description of the context (e.g. industry, laboratory setting, products used, etc.) in which the research was carried out?

QA.Q3. Did the paper present sufficient detail about the software architecture variability approach to allow it to be understood and assessed?

QA.Q4. Did the case study (if exist) employ a single or multiple case research design?

QA.Q5. Did the case study consider construction validity, internal validity, external validity, and reliability to the study?

QA.Q6. Is there a description and justification of the research design, including a statement of what the result should be (e.g. a construct, a model, a method, or an instantiation)?

QA.Q7. Is there a clear statement of findings with sufficient data to support any conclusions?

QA.Q8. Do the authors discuss the credibility of their findings?

QA.Q9. Are the limitations of the study discussed explicitly?

A ternary (''Yes'', ''Partially'' or ''No'') scale was used to grade the reviewed studies on each element of the quality assessment criteria. By including ''Partially'' in the scale is to make sure that statements where authors only provided limited information to answer the quality assessment questions were not totally neglected. To quantify the results, these values: 1 to Yes, 0.5 to Partially, and 0 to No were assigned. Then, a quality assessment score was given to each study by aggregating the scores of all questions.

The quality assessment criteria were used for synthesis purposes and not for filtering papers. The calculated quality scores were used as one of the factors to validate all of the primary studies that were reviewed. This assessment is also used to answer SLR.RQ2 and the results are provided in Chapter 4.

2.2.6 Data Extraction and Synthesis

On completion of the search, selection and quality assessment steps, data extraction was then conducted on the selected 58 primary studies to help answer the research questions defined in Section 2.2.2. Appendix A1 shows the complete list of the primary

studies that were included in this systematic literature review. Data was extracted using a data extraction form whose fields are shown in Table 2. In addition, Table 2 shows the mapping between the data extraction questions and the research questions (excluding SLR.RQ2 which is solely quality assessment question, discussed in prior section) that they help to answer.

During data extraction, information related to the paper synopsis (DE.Q5) to define the identified approach more elaborately, variability approach (DE.Q8) and the limitations (DE.Q10) were also captured. Every effort was made to capture as much information as possible, but at the same time, kept the data as succinct as possible in order to avoid any potential influence of a taxonomic or classification framework on our results.

GoogleDocs was used to collect the extracted data from the different researchers and the aggregated results were made available in Excel spreadsheets for analysis. Finally, sanity checks are performed on the results and the differences were reconciled collaboratively.

	DATA FIELD	RELATED CONCERN/RESEARCH QUESTION
DE.Q1	Paper title	Documentation
DE.Q2	Year of publication	Documentation
DE.Q3	Type of publication (e.g. Journal, Conference, etc.)	Reliability of review
DE.Q4	Publication outlet (conference name, etc.)	Reliability of review
DE.Q5	Brief description (synopsis)	SLR.RQ1
DE.Q6	Research Context (e.g. industry, academic, etc.)	SLR.RQ3
DE.Q7	Research Area (e.g. SPL, SOA, etc.)	SLR.RQ3
DE.Q8	Proposed approach for representing variability in software architectures (please provide category [UML, ADL, etc.] and an example/sample if possible)	SLR.RQ1
DE.Q9	Relevance (Research/ Practice/Both)	SLR.RQ3
DE.Q10	Research limitations as reported in the paper	SLR.RQ4

Table 2: Data extraction form

2.3 Language and Framework Design

Different approaches were identified via an SLR that represents variability in software architecture (as reported in Chapter 4). Of these, ADL (after UML, which is a semi-formal notation) is the most common formal architectural description notation in the existing literature that is used to represent variability in software architecture.

Therefore, ADL was chosen as an approach to represent variability in software systems at the architectural level. For this, ALI ADL (Bashroush *et al.*, 2008), which was initially designed within our research group, was adopted due to its strengths (such as a flexible

way to design architectural elements, meta-information and so on), and its applicability for industrial systems.

Redesigning the ALI ADL (named as ALI V2 in this thesis) was done in such a way that it overcame the current limitations (such as limited support for variability management, restrictive syntax and the like) that exists in the architectural languages, as discussed in the next chapter. In particular, it addresses the current challenges faced by industrial practitioners when designing the architecture of large-scale systems; these challenges were not addressed in the initial versions (in other words, versions that existed before this research work for this thesis started), as explained in Chapter 5. These include limited support for behavioural descriptions and multiple architectural views.

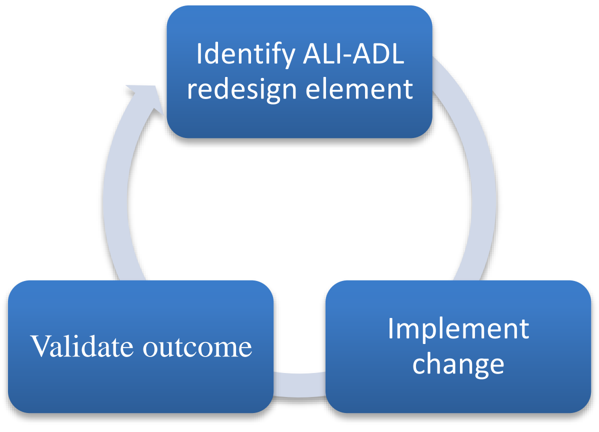


Figure 4: ALI redesign process

An agile approach was adopted for redesigning the language, whereby changes were introduced in small increments and the ALI V2 was then tested and validated using a snippet of the Asset Management System's (AMS) case study, which was used as one of the real-life case studies to evaluate ALI V2. This ensured that we avoided any major surprises in the final evaluation stage. Thus, the redesigning of the ALI V2 language is a recursive process, as demonstrated in Figure 4.

In addition, the ALI V2 framework was enhanced in the form of a conceptual model that demonstrates the high-level (abstract) description of the language, which was not considered previously in its original version (described in Chapter 5).

2.4 Case Study Research

After the completion of the language and the design of the framework, the final version of ALI V2 was evaluated using the two real-life case studies to serve as a benchmark. The two case studies were chosen to demonstrate the broader scope of the ALI V2 language due to their distinct characteristics.

The first case study corresponded to the information system (IS) domain; namely, an Asset Management System (AMS) that described how a portfolio for a financial instrument (equity) was managed by the fund manager (or fund management team) within an investment bank. The operational description of AMS was obtained by conducting a detailed interview with some of the finance personnel from the leading investment banks. In addition, while designing the AMS architecture using the ALI V2 language, the finance personnel analysed each complete aspect in segments. This was to ensure its computational correctness, and to make sure that it fulfilled the real-life AMS requirement in terms of managing the portfolio, as well as to avoid any major consequences at the end.

Another case study corresponded to the embedded system domain, namely the Wheel Brake System (WBS), which described how brakes can be applied to decelerate/stop the wheels of commercial aircraft during landing or parking. WBS is a standardised case study that was obtained from the SAE Standard Aerospace Recommended Practice (ARP) 4761, *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* (ARP4761, 1996).

In addition to this, a number of selection criteria were applied for deciding these two best case studies, including: distinct application domains (to demonstrate cross domain modelling capabilities); existence of inherent variability in the application domain; varying types of connectivity between components; different complexity levels (information overload); varied emphasis on behavioural versus structural descriptions; potential for artefact reusability within the case study; and last but not least, access to full technical details.

The detailed description of both case studies, together with their architectural implementation using ALI V2 language has been described thoroughly in Chapter 7 and Chapter 8. These sections also present an evaluation in accordance with the ALI V2 design principles and the results obtained after the implementation.

2.5 Summary

The research methodology described in this chapter addresses the research questions (see Chapter 1) determined for this thesis, which demonstrate how variability can be represented in software systems at the architectural level. RQ1 has been addressed through an SLR, while language and the framework design addresses RQ2 and RQ3. The case study research method addresses RQ4.

In the SLR research method, five protocols were developed to identify the current state-of-the-art on representing variability in software architecture. Of these, the search strategy and the selection criteria to identify the existing papers in the research literature were defined. Following the search strategy, and considering the pre-defined inclusion and exclusion criteria, 58 papers (termed primary studies in SLR) were selected. Moreover, to assess the quality of each primary study, nine quality assessment questions

were defined. Subsequently, ten different questions were defined to extract the data to answer the research questions that were described as part of the review protocol.

Based on the SLR findings, ALI V2 language was redesigned in such a way that it provided a flexible method of representing variability conjunction with its other properties in order for practitioners to use it to model their systems with ease. To validate it, ALI V2 was evaluated via two distinct, real-life case studies.

In the next section, Chapter 3, the theoretical background to the terminology used in this research, as well as the analysis of the existing ADLs and their limitations (particularly from industrial perspectives) is presented. A detailed analysis of the primary studies identified via the SLR research method is provided in Chapter 4.

Part II

STATE-OF-THE-ART

Chapter Three

Background

3

"There is nothing so practical as a good theory."

--Ludwig Boltzman

3.1 Introduction

This chapter gives an overview of the basic concepts and related work that are most closely related to this thesis work. Firstly, software architecture and variability concepts are outlined in Section 3.2. This is followed by a brief overview of the Architecture Description Language (ADL) in Section 3.3, which also includes detailed analysis of the existing ADLs followed by their limitations. Section 3.4 concludes the information analysed from the current research literature.

3.2 Concepts and Terminology

This section discusses some conceptual background information this thesis work is based on. Basic concepts about software architecture and variability are presented in this section.

3.2.1 Software Architecture

The field of software architecture addresses notations and methodologies that can help abstract large-scale systems in order to enable better intellectual control over the system as a whole (Bass *et al.*, 2012). In a simpler way, software architecture acts as a skeleton for the software development, usually designed at the early stages of the software

development lifecycle once initial requirements are understood. Then the whole development process rotates around this skeleton, keeping into account the constraints and facilities implied by the software architecture. Nowadays, it has been observed that software architecture is widely visible as an important and explicit design activity in software development. Typically, it plays a key role as a bridge between requirements and implementation (Garlan, 2014).

There are several published definitions for Software Architecture, such as those of the Software Engineering Institute's architecture practice site. For example, one definition states that:

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them"* (Bass *et al.*, 2012).

Kruchten et al. (Kruchten *et al.*, 2006) elucidates that software architecture seizures and preserves designers' intentions about system structure and behaviour, thereby providing a resistance against design decay as a system ages. In addition to this, it involves two things: 1) the structure and organization of the modern system components and subsystems by which they interact to form systems, and 2) the properties of systems that can be best designed along with the analysis at the system level (Kruchten *et al.*, 2006). A software architecture can also be defined as the "blueprint" of a system at the highest level of abstraction, describing the main components and their important interconnections (Yao *et al.*, 2010).

Basically, software architecture is the 'first cut' at solving a problem and designing a system. The importance of software architecture lies in its ability to: a) represent earliest design decisions; b) abstract system details to provide a holistic view of the system (the

big picture); and c) allow for systematic reuse (e.g. reuse of large components and frameworks into which components can be integrated).

A precise description of the software architecture of a system provides considerable benefits for the system's stakeholders. For instance, the system allows an early analysis of whether the system can meet its requirements from the description of a software architecture; it may be used as a centre of discussion by system stakeholders; and it allows for reasoning on the system from the very early stages of its development life-cycle.

3.2.2 Variability

Variability in software-intensive systems is commonly understood as the ability of a software artefact (e.g., a system, subsystem, or component) to be changed for deployment in a specific context (Galster *et al.*, 2014).In addition to this, variability is often understood as "anticipated" change, i.e., change that is mostly foreseen, with predefined points of potential change and adaptation, as well as options for how to adapt software systems (Galster and Avgeriou, 2011b). Variability management helps organise the commonalities and differences amongst software systems. More specifically, variability management allows for a) the development and evolution of different versions of software and product variants, b) planned reuse of software artefacts, c) well-organized instantiation and assessment of architecture variants, and d) runtime adaptations of deployed systems (Galster *et al.*, 2014).

Variability is pervasive, thus, architects need adequate support for dealing with it. Therefore, it is essential for the architect to have suitable methods and tools for handling (i.e., representing, managing and reasoning about) variability (Galster and Avgeriou, 2011a). As discussed in (Galster and Avgeriou, 2011a), software architecture considers

variability in a broader scope and acknowledges that variability is a concern of different stakeholders, and in turn affects other concerns.

There are several mechanisms that can accommodate variability such as software product lines, variant management tools, configuration tools, configuration interfaces of software components, or the dynamic runtime composition of web services (Galster *et al.*, 2014). So far, variability has primarily been studied in the software product line (SPL) domain. But as compared to software architectures, product line architectures have a limited scope with regard to variability (Chen *et al.*, 2009). Bachmann and Bass raised two causes of variability in the software architecture of a product line: (1) at design time many alternatives may exist and need to be captured, and (2) software product line architectures comprises of a collection of different alternatives that must be resolved during product configuration (Bachmann and Bass, 2001).

3.3 Architecture Description Languages (ADLs)

Architecture Description Languages (ADLs) proliferated in the 1990s as a formal modelling notation to describe the architecture of the software systems. It provides the embodiment of early design decisions prior to the detailed design and implementation of a system.

According to the (ISO/IEC/IEEE 42010, 2011), an architectural language is:

"Any form of expression for use in architecture descriptions".

In theory ADLs differ from requirements languages, because ADLs are rooted in the solution space, whereas requirements define problem spaces. Moreover, they also differ from programming languages, because ADLs do not bind architectural abstractions to specific point solutions.

Basically, ADLs result from a linguistic (informal, such as box-and-line) approach to the formal representation of architectures but still its designing intends to be readable to both human and machines. It permits analysis and assessment of architectures, for completeness, consistency, ambiguity, and performance. Also, it can support automatic generation of software systems. Despite of several advantages of the ADLs, there is still a consensus in the research community on what an ideal ADL is and what aspects of an architecture should be modelled in an ADL, especially when it comes to its applicability into large-scale industrial systems.

Following are some of the common definitions given by different researchers which have been usually taken into account by ADL creators while designing the language:

> *"An ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module"* (Vestal, 1993).
>
> *"Architecture description languages (ADLs) are formal languages that can be used to represent the architecture of a software-intensive system"* (Clements, 1996).

In addition to above definitions, Software Engineering Institute (SEI) defined ADLs as:

> *"A language (graphical, textual, or both) for describing a software system in terms of its architectural elements and the relationships among them"* (Software Engineering Institute (SEI) / Carnegie Mellon University (CMU)).

The above SEI definition can be interpreted as ADLs provide abstractions for representing architectures through architectural elements (components and connectors) and their configurations. In that, components represent software functionalities, connectors are communication elements, and configurations describe the relationship between components and connectors (Medvidovic and Taylor, 2000).

The ADL community also generally agrees that software architecture is a set of components and the connections among them conforming to a set of constraints. Thus, it means components, connectors and architectural configurations are the basic building block for the architectural designing of a system.

There are several ADLs designed by researchers and practitioners that had made an attempt to address the problems of modelling a system architecture in some way. Those ADLs have been discussed and analysed in detail in the following section:

3.3.1 Analysis of existing ADLs

Since the early 90's, a thread of research on formal architecture description languages (ADLs) has evolved. Numerous ADLs have been proposed in the literature for modelling architectures both within a particular domain, and general-purpose architecture modelling notations.

All the classical ADLs (also considered first generation ADLs (Oquendo, 2004)) compared and analysed by Medvidovic and Taylor (Medvidovic and Taylor, 2000) were conceptually based on structural architecture modelling features (components, connectors, interfaces and architectural configuration) and tool support. Another ADL survey was conducted by Clements (Clements, 1996) in the same era. Some of the second generation ADLs have been compared in (Yao *et al.*, 2010) but it covers a very limited number of characteristics of the languages.

Looking at the existing literature, it was interesting to note that very few ADLs were originated in industry. The main three are described here.

Architecture Analysis & Design Language (AADL) (Feiler, Gluch and Hudak, 2006) derived from the MetaH (Binns *et al.*, 1996) ADL, is a SAE standard formal modelling

language for describing software and hardware system architectures and uses a component-based notation for the specification of task and communication. It provides precise execution semantics for system components, such as threads, processes, memory, and buses. All external interaction points of a component are defined as *features.* Data and events *flow* through and across multiple components. The AADL *Behavioural annex* describes nominal component behaviour and the *Error annex* describes flows in the presence of errors.

Koala (Ommering *et al.*, 2000) is a component oriented ADL based on key concepts from Darwin (Magee and Kramer, 1996). Basically, it was designed with the aim of achieving a strict separation between component and configuration development in order to reuse software components in many different configurations for different product variants, while controlling cost and complexity.

EAST-ADL (Cuenot *et al.*, 2010) defines an approach for describing automotive electronic systems through an information model that captures engineering information in a standardized form, provides separation of concerns and embraces the de-facto architecture of automotive software – AUTOSAR (Qureshi *et al.*, 2011). It covers a variety of aspects -functions, requirements, variability, software components, hardware components and communication.

Although these ADLs come from different industries, they all relate to the embedded systems domain. AADL and EAST-ADL emerged from the avionics and automotive industries and are currently widely used in their respective domains. Koala, on the other hand, was developed within the consumer electronics domain, though its use hasn't seen the same proliferation as the previous two.

On the academic side, a large number of ADLs have been proposed, each characterised by slightly different conceptual architectural elements; different syntax or semantics;

varying emphasis on a single view (structural or behavioural) or operational domain such as embedded system; or for specific analysis techniques.

Below are some of the main ADLs developed in academia:

- ACME (Garlan, Monroe and Wile, 1997) is a general purpose ADL proposed as an architectural interchange language.
- Darwin is a *declarative* ADL which is intended to be a general purpose notation for specifying the structure of distributed systems composed from diverse component types using diverse interaction mechanisms (Magee and Kramer, 1996)
- UniCon (Shaw *et al.*, 1995) creates a useful, pragmatic and extensible test-bed that would allow the architectural abstractions used by practitioners (such as pipes, filters, objects, clients and servers) to be captured and reasoned about in a systematic manner.
- xADL(Dashofy, van der Hoek and Taylor, 2005), an XML based architecture description language, is defined as a set of XML schemas and has been designed to use the standard XML infrastructure and to be easily extensible using standard XML-Schema extension mechanisms.
- C2 is a component- and message-based ADL which simplifies the definition of architectures following the Chiron-2 (“C2”) style (Medvidovic, Taylor and Whithead, 1996).
- Rapide (Luckham *et al.*, 1995) is an event-based concurrent object-oriented language specifically designed for prototyping architectures of distributed systems.
- WRIGHT (Allen and Garlan, 1997) is designed with an emphasis on analysis of communication protocols and provides formal semantics for an entire architectural description by extending CSP. Wright has been extended, termed Dynamic

WRIGHT (Allen, Douence and Garlan, 1998), with the ability to handle foreseen dynamic reconfiguration aspects of architecture.

Apart from the ADLs mentioned above, we examined a number of other ADLs with varying degree of maturity.

According to the ANSI/IEEE 1471-2000 standard, structural and behavioural viewpoints are the two most important and frequently used viewpoints for architectural description. The specification of each viewpoint with their entities is elucidated in (ISO/IEC/IEEE 42010, 2011; Oquendo, 2004). A great challenge for an ADL is being able to describe static and dynamic software architectures from structural and behavioural perspectives.

ADLs like ACME (Garlan, Monroe and Wile, 1997), Aesop (Garlan, Allen and Ockerbloom, 1994), Aspectual-ACME (Garcia *et al.*, 2006), Darwin (Magee and Kramer, 1996), Koala (Ommering *et al.*, 2000), MontiArc^{HV} (Haber *et al.*, 2011c), UniCon (Shaw *et al.*, 1995), Weaves (Gorlick and Razouk, 1991) and xADL (Dashofy, van der Hoek and Taylor, 2005) were focused largely on the structural concerns of software architecture. On the other hand, some ADLs covered both behavioural and structural specifications, including: AADL (Feiler, Gluch and Hudak, 2006), ABC/ADL (Mei *et al.*, 2002), ADLARS (Bashroush *et al.*, 2005), ADML (Wang *et al.*, 2012), C2 (Medvidovic, Taylor and Whithead, 1996), CBabel (Rademaker, Braga and Sztajnberg, 2005), EAST-ADL (Cuenot *et al.*, 2010), LEDA (Canal, Pimentel and Troya, 1999), MetaH (Binns *et al.*, 1996), PrimitiveC (Magableh and Barrett, 2010), PRISMA (Perez *et al.*, 2003), Rapide (Luckham *et al.*, 1995), SOADL (Xiangyang *et al.*, 2007), xADL (Dashofy, van der Hoek and Taylor, 2005), XYZ/ADL (Zhang, Shi and Rong, 2011), vADL (Zhang, Xiang and Wang, 2005), WRIGHT (Allen, Douence and Garlan, 1998; Allen and Garlan, 1997), Zeta (Alloui and Oquendo, 2002), π-ADL (Oquendo, 2004) and π-SPACE (Chaudet and

Oquendo, 2000). While some only covered behavioural aspects, such as Monterey Phoenix (Auguston, 2009).

Most of these languages (except (Allen and Garlan, 1997; Magee and Kramer, 1996)) define structural elements using their own bespoke notation. Some ADLs (such as AADL and ADLARS) used their own structural notation to describe the behavioural architecture. Some used different processes to define the behavioural description. For example, Rapide describes behaviour through partially ordered event sets (or "posets"); Wright uses CSP with minor extensions; LEDA, PRISMA, SOADL, vADL, π-ADL and π-SPACE use the π-calculus. It is useful to mention that despite the presence of a π –calculus model for Darwin's structural descriptions, it does not provide an adequate basis for analysis of the behaviour of an architecture.

Generally, the overall architectural structure of ADLs focuses on the basic component, connector and system paradigm. All ADLs that have been analysed so far treat components as first class citizens, but in some languages (Bashroush *et al.*, 2005; Canal, Pimentel and Troya, 1999; Cassou *et al.*, 2009; Chang and Seongwoon, 1999; Feiler, Gluch and Hudak, 2006; Haber *et al.*, 2011; Klien, 2010; Luckham *et al.*, 1995; Magee and Kramer, 1996; Ommering *et al.*, 2000; Binns *et al.*, 1996; Poizat and Royer, 2006; Pinto, Fuentes and Troya, 2003; Faulkner and Kolp, 2003; Zhang, Xiang and Wang, 2005) there is no notion of connectors as first class citizens. Connectors are not even defined. This does not mean that we cannot create a useful language without first class connectors. There are viable and potentially useful architectural languages that have been created without them, like (Feiler, Gluch and Hudak, 2006; Luckham *et al.*, 1995; Magee and Kramer, 1996; Ommering *et al.*, 2000). (Alloui and Oquendo, 2002; Canal, Pimentel and Troya, 1999; Chang and Seongwoon, 1999; Su, De Fraine and Vanderperren, 2005; Gorlick and Razouk, 1991; Klien, 2010; Ubayashi, Nomura and Tamai, 2010; Wang *et al.*, 2012; Zhang, Xiang and Wang, 2005) do not support an architectural configuration

as a first class element. Neither connector nor architectural configuration was considered first class citizens in (Canal, Pimentel and Troya, 1999; Chang and Seongwoon, 1999; Klien, 2010; Zhang, Xiang and Wang, 2005).

There are few second generation academic ADLs that focus mainly on the behavioural modelling in a slightly different way as compared to traditional ADLs. Monterey Phoenix (Auguston, 2009) is an ADL in which behaviour of the system is defined as a set of events (event trace) with two basic relations: precedence and inclusion. Different types of patterns (such as alternative, optional, etc.) are defined in the form of an event trace that occurs in a transaction. But they lack the unique visual notation for each of these event patterns. A schema is defined as a set of transactions that includes all possible event traces. It can be tedious to understand (especially visually) and sometimes becomes more complicated when it is encapsulated with several pattern types in a single schema, particularly, in case of large-scale and complex systems.

PrimitiveC-ADL (Magableh and Barrett, 2010) is a component-based language that modifies the application architecture by subdividing components into subsystems of static and dynamic elements. A *design pattern* typically shows relationships and interactions between components' dynamic behaviour parts. The *decision policy* proposes the use of a *design* pattern and the application of the *decision policy* depends on a *scenario*. The main problem in (Magableh and Barrett, 2010) and other ADLs (Oquendo, 2004; Zhang, Xiang and Wang, 2005) is that while they define the behaviour of the system within a component or in their configuration, behavioural elements are not explicitly defined. In other words, it provides a single view of the system which is not suitable for a large-scale industrial system where component behaviour varies enormously. In that case, component definition becomes complex and it is difficult to differentiate static and dynamic parts.

AspectLEDA (Navasa, Pérez-Toledano and Murillo, 2009) is an ADL that provides behavioural specification of the system using the UML use case and activity diagrams by adopting the Aspect-Oriented (AO) approach. Each use case diagram represents a component that constitutes the system and its interactions are expressed in the form of sequence diagrams. Subsequently, a sequence diagram for every use case contains by default an aspect component as each use case is extended with an aspect. In other words, it describes the interactions among components visually via a UML sequence diagram with its dependency on an AO approach. Looking at this, component interactions need to be more elaborative in a sense by considering component interfaces (or ports) that are involved in the interaction which would be helpful to design complex systems.

Another major element that needs attention with regards to ADLs, is the concept of variability. This is a very important and critical area when it comes to its use in the architectural description, especially in large-scale industrial applications (Bashroush *et al.*, 2005; Svahnberg and Bosch, 2000). Variability is the ability to design for a planned set of changes for deployment in specific contexts (Galster *et al.*, 2014a). It facilitates the development of different versions of a system architecture. Variability is largely taken into account in the architecture and design phase of software engineering (Galster *et al.*, 2014b). Although there are several ADLs where variability has been studied, variation is specific to describing a set of related products as in a software product line (SPL). Among the ADLs are: PL-AspectualACME (Barbosa *et al.*, 2011), ADLARS, EAADL (Oh *et al.*, 2007), LightPL-ACME (Silva *et al.*, 2013), vADL and the recently DSOPL (Adjoyan and Seriai, 2015). Other ADLs that consider variability as a separate entity are: MontiArc^{HV} and Δ-MontiArc (Haber *et al.*, 2013).

Software architecture typically plays a key role as a bridge between requirements and implementation (Garlan, 2014). In terms of ADLs, a challenge in bridging this gap is how to trace feature (requirements) into the architecture description particularly, into each

architectural element. So far, in the research literature, ADLARS and LightPL-ACME are the only two ADLs that made an attempt to capture the relationship between the system's features and the architectural structures. Both assumed a feature model as a precursor to the architecture design process and were limited to specifying a product line.

It is worth mentioning that there are few ADLs that try to represent different aspects and domains in the architecture by presenting it in the form of different versions. Each focuses on a particular aspect/domain. For instance, ACME has been extended to AspectualACME with its descendant PL-AspectualACME, LightPL-ACME, Cloud-ADL (Cavalcante, Medeiros and Batista, 2013) and ADML; MontiArc (Haber, Ringert and Rumpe, 2012) to MontiArc^{HV}, Δ-MontiArc and MontiArcAutomaton (Ringert, Rumpe and Wortmann, 2013).

There is a framework known as ByADL (Build Your ADL) (Ruscio *et al.*, 2010) that supports a software architecture team in defining their own ADL by allowing software architects to (i) extend existing ADLs with domain specificities, new architectural views, or analysis aspects, (ii) integrate an ADL with development processes and methodologies, and (iii) customise an ADL. Basically, it takes the meta-model of the ADL to be extended as an input.

Overall, a common pitfall for the discussed ADLs is their limited ability to support large-scale real-life applications. Some possible reasons behind this are discussed in (Bashroush *et al.*, 2006; Malavolta *et al.*, 2013). Limitations are further discussed in the next section.

3.3.2 Limitations in existing ADLs

After critically analysing the existing ADL literature, particularly around scalability and uptake (industrial adoption), it was evident that only ADLs that were originated in industry saw some level of industrial adoption. This has been attributed to potential misalignment between practitioner needs and the academic focus (Malavolta *et al.*, 2013).

Below, we summarise some of the main limitations identified in ADLs that emerged from academic research, but failed to achieve any notable industrial adoption:

L1: *Limited support for variability management*

To manage the size and complexity of industrial systems, and with the current trend of delaying architectural decisions as much as economically feasible (and the shift of variability from hardware to software), it is valuable to have the capability of modelling variability adequately in the architecture design.

L2: *No explicit mechanism to link requirements to architectural artefacts*

Requirements traceability has emerged as a main objective in industry. Yet, without the support for capturing such relationships at the architecture description stage, the link between requirements and implementation becomes difficult to establish and maintain. For example, although AADL does not support modelling such relationships natively, tools such as AADL's OSATE provide such mechanisms outside the ADL.

L3: *Domain dependency*

As can be seen from the previous section, many ADLs are tailored for a particular domain, with embedded systems having the majority of such systems. However, given the way today's systems are evolving with Systems-of-Systems, Cyber-Physical Systems, Smart Cities Systems, etc., for an ADL to be capable of modelling a complete solution, it needs to cross cut multiple domains.

L4: *Restrictive syntax*

Many ADLs impose a strict syntax and design principles on the architect (e.g. layered model, network model, etc.). Building ADLs in such a way allows the ADL designer to provide various automated architectural analysis. However, from a practitioner perspective, the last thing needed is to be forced to reason about the system in a specific way, or end up writing code twice.

L5: *Lack of support for architectural artefact reusability*

Existing ADLs have been designed to support the abstraction of details; however, support for architectural artefact reuse across multiple projects is lacking. While architecture reuse has seen some success in specific domains, e.g. the automotive domain using AADL (Feiler, Gluch and Hudak, 2006), the granularity of reuse remains relatively small. In order to support large-scale reuse, ADL's would need to provide mechanisms to capture some degree of variability in the description of artefacts (and their interfaces) to enable redeployment in multiple contexts.

L6: ***Overloaded architectural views***

Given that one of the main benefits of having an overall system architecture description is to use it as a communication vehicle among the various stakeholders, not all the information captured within the architecture tend to relate to every stakeholder. Accordingly, ADLs providing one or two architectural views tend to suffer from information overload. The importance of having multiple architectural views has also been highlighted in (ISO/IEC/IEEE 42010, 2011).

L7: ***Focus on structure more than behavioural architectural aspects***

The structural description of a system changes less frequently compared to the behavioural description because systems can serve different objectives with the same structural description. In other words, the structural description can encapsulate more than one behavioural description. Yet, it can be said that most ADLs still overlook the importance of behavioural description. While it is viewed as a major construct in some (Feiler, Gluch and Hudak, 2006; Luckham *et al.*, 1995), it is not covered in many (Allen and Garlan, 1997; Gorlick and Razouk, 1991; Shaw *et al.*, 1995), with fewer ADLs supporting the representation of behavioural architectural knowledge graphically (Brown *et al.*, 2006).

3.4 Conclusion

Software architecture is widely visible as an important and explicit design activity to develop a software system. However, the changing face of technology raises a number of challenges for software architecture. Among those, the most important challenge is how

to capture a higher degree of variability in the software architecture (as observed in this chapter).

Representing variability in software architecture not only allows for large grain reuse of artefacts, but also permits tracing requirements into the system implementation and deployment. So it raises a critical question for software architects regarding how to describe the architectural description for the system (particularly, large-scale industrial systems) that captures variability and fulfils other requirements of practitioners.

Ideally, architectural descriptions should express their design intent clearly to others and also require low overhead to create and maintain the system architecture. For this, ADL provides both a conceptual framework and a concrete syntax for formal modelling of software architectures.

A number of different ADLs exist, largely within academia (as analysed in Section 3.3.1). However, during the increasingly in-depth study and wide application of ADLs, there is a gradual recognition that conventional ADLs lack various concepts, which restricts their uptake into real-life industrial applications. Some of those concepts are: support for managing variability as an integral part of the system; domain dependency; restrictive syntax; and architectural artefact reusability (as explained in Section 3.3.2).

The current state-of-the-art that has been identified through an SLR (in the previous chapter) on representing variability in software architecture is analysed in detail in the next chapter.

Chapter Four

Systematic Literature Review

4

"A man should look for what is, and not for what he thinks should be."

--Albert Einstein

4.1 Introduction

Over the last 15 years, a lot of work has been reported that addresses the representation of variability in software architecture in different domains. Some approaches have defined variability in software architecture as a way of representing and reasoning about alternative system implementations (Bachmann and Bass, 2001; Galster and Avgeriou, 2011). Similarly, a number of different mechanisms have been used to represent variability at the architecture level (e.g. Software Product Lines (SPL), Service-oriented architecture (SOA)). Although it is generally agreed that variability representation is a key step of the development process, which can affect the success or failure of a system or a product line (Bashroush, 2010), there seems to be little consensus on how the representation is best conducted.

In this chapter, a Systematic Literature Review (SLR) is presented which is conducted to summarize the current state-of-the-art in representing variability in software architecture. The analysis in this chapter is based on the data collected from the quality assessment and data extraction phases described in the research methodology chapter, Section 2.2.5 and 2.2.6 respectively, through a SLR review protocol (see Section 2.2.1).

The presentation of the work in this chapter will benefits practitioners working in the area who are looking to choose the best variability approach that fits their design needs, as well as researchers trying to identify areas that require further investigation.

The rest of the chapter is organised as follows: Section 4.2 provides the data and its analysis of the selected primary studies in relation to their publication type, venues and trends, and their geographical distribution. Section 4.3 provides the analysis and discussion of the collected data in order to answer the research questions set for this SLR. Threats to the validity of the data and limitations of this SLR is presented in Section 4.4. The most recent work on representing variability in software architecture after the data analysis were summarised in Section 4.5. Finally, outcome of the analysed data for representing variability in software architecture is concluded in Section 4.6.

4.2 Data and Analysis

Once the data extraction phase has been completed, data synthesis and analysis was conducted on the collected information. This section provides an analysis of the 58 selected primary studies (listed in Appendix A1) in relation to their publication type, venues and trends, and their geographical distribution.

4.2.1 Demographic Data

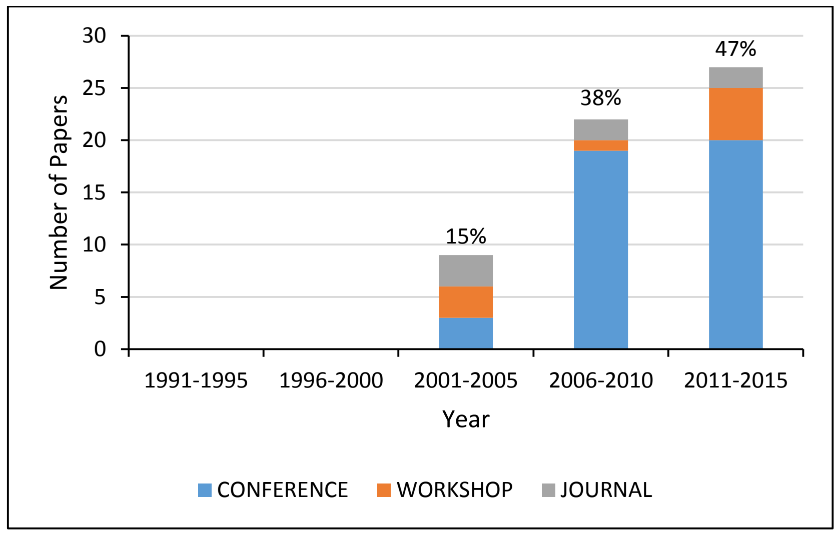


Figure 5: Publications per year

Although our search period is set to start from January 1991 but unfortunately no studies were found in the 90s decade, the earliest primary studies identified were published in 2002. This could be due to the timing of the first major paper on the topic of Software Architecture by Shaw et al. (Shaw *et al.*, 1995) in mid 90's. Figure 5 shows the number of primary studies identified, along with the breakdown of numbers of papers published via each publication outlet type (Conference, Journal or Workshop). The data presented shows papers bundled in 5 year brackets to smooth the effect of conference frequency (e.g. some conferences happen every 18 months, while others every 12 months) and public funding call trends (e.g. EU funded research projects addressing a specific challenge tend to start and end during the same time frame leading to increased paper publications in the area around the end of the funding period). Looking at the chart, it can be seen that there is an uptrend in research publications relating to variability in software architecture. It is worth mentioning here that the primary studies identified in 2015 only covered the ones published up until July, when the search and selection process of this study was completed (thus 2015 is partially covered).

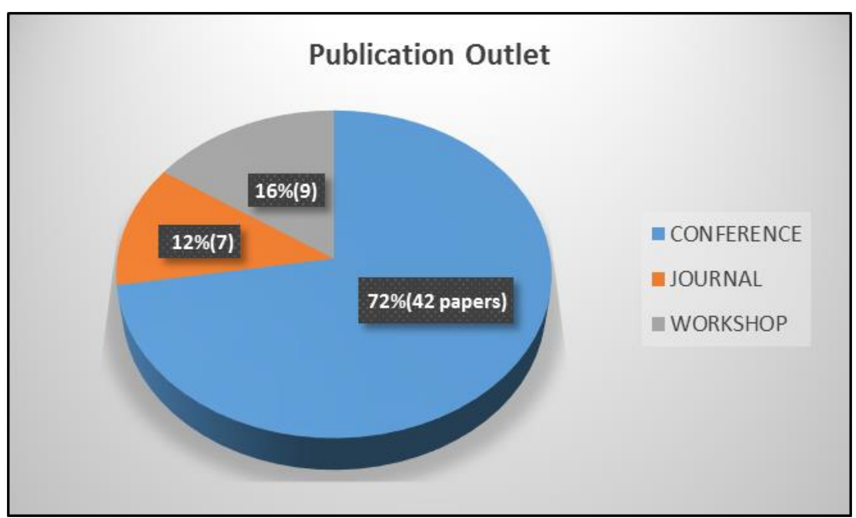


Figure 6: Publication outlet

Figure 6 shows a pie chart of the publication outlet of the selected primary studies. As can be seen, the majority of the primary studies were published in the proceedings of conferences, followed by Workshops, and then Journals.

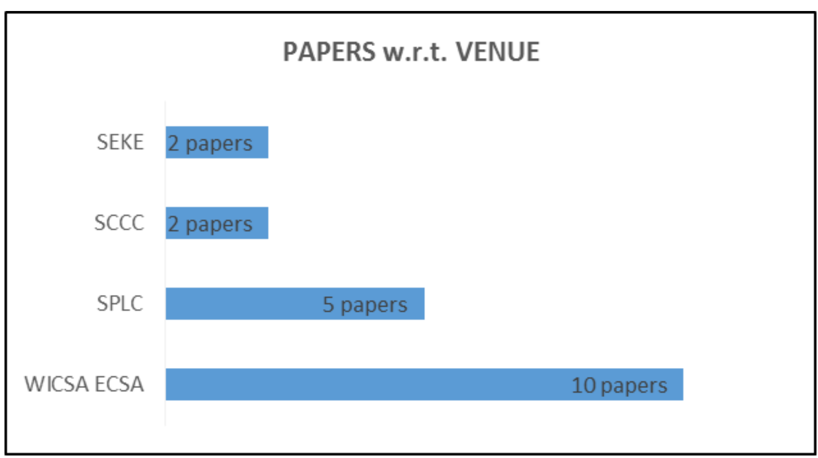


Figure 7: Highly occurring publication venues

Venues identified in Figure 7 encapsulates 32% (19 papers) of the total primary studies (and were the only venues with more than one primary study published). The primary studies were most commonly found in the proceedings of conferences such as WICSA/ECSA (17%) and SPLC (8%). The reason for amalgamation of WICSA/ECSA is because these conferences were co-located twice (in 2009 and 2012). A tabular form of the data with their acronyms can be found in Appendix A2.

4.2.2 Geographical Distribution

A detailed list of publications per country is provided in Figure 8. Countries of all authors named on a primary study were accounted (hence the discrepancy between the number of papers and number of papers per country). The data is plotted in Figure 8, which shows Germany and Brazil as the most popular countries in terms of research on capturing variability in software architecture.

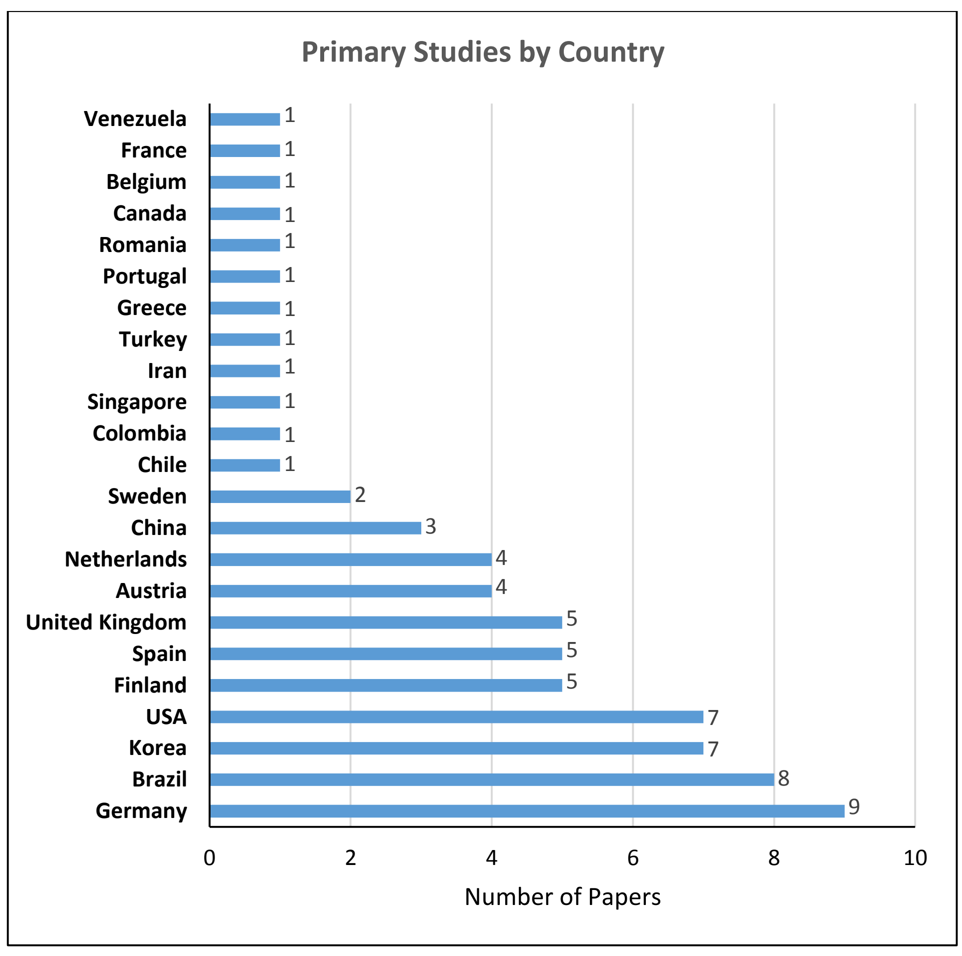


Figure 8: Primary study distribution per country

4.3 Discussion of SLR Research Questions

This section attempts to answer the SLR research questions (defined in the Chapter 2) by synthesizing and analysing the data extracted from the 58 selected primary studies listed in Appendix A1.

4.3.1 SLR.RQ1: *What approaches have been proposed to represent variability in software architecture?*

Two major approaches for representing variability in software architecture were identified in the primary studies: (1) defining variability using Unified Modelling Language (UML) or one of its extension in the form of another method, domain-ontology

etc., and (2) using an ADL with explicit variability representation mechanisms. A detailed classification can be found in Table 3.

From the 58 selected primary studies, 45% (26 papers) of the primary studies presented various variability through UML, in which 21% (12 papers) used a form of meta-model based on UML class diagram. While 16% (9 papers) represented variability using other UML diagrams such as component diagram (e.g. S12, S17, S23, S29, S51), activity diagram (e.g. S38, S51) and sequence diagram (e.g. S58). Finally, 9% (5 papers) extended the UML notation into UML PLUS (Product Line UML based Software Engineering) method (S48, S50); Kumbang (S13, S41), a modelling language and an ontology for modelling variability in software product line architectures from feature and component points of view; and KumbangSec (S57).

Notation	Total Papers	Percentage	Study Identifier
UML			
Class Diagram	12	21%	S1, S10, S15, S19, S25, S26, S27, S30, S31, S38, S43, S49
Other (Component, Activity etc.)	9	16%	S12, S17, S23, S26, S29, S31, S32, S51, S58
Extension (PLUS, Kumbang etc.)	5	9%	S13, S41, S48, S50, S57
	26	45%	
ADL	14	24%	S3 - S5, S14, S18, S20, S24, S34, S36, S37, S39, S40, S42, S54
OVM	4	7%	S17, S26, S45, S53
XML	2	3%	S31, S41
Other (CVL, LISA etc.)	20	34%	S2, S6 - S9, S11, S16, S21, S22, S25, S28, S29, S33, S35, S44, S46, S47, S52, S55, S56

Table 3: Variability representation approaches

24% (14 papers) of the selected primary studies described how to represent variability using an ADL, with a number of different ADLs adopted. The ADLs used for addressing variability were:

xADL 2.0: S3 uses xADL 2.0 together with several tools to express variability in xADL (MÉNAGE) and to select a particular system instance out of product line architecture (SELECTORDRIVER). S24 uses xADL 2.0 describing operators and process for merging reference architecture and application architecture. The result embodies all the application differences by new variation points, which makes it possible to synchronize application and component architectures.

vADL: S4 is an ADL that extends the framework of traditional ADL, and provides variability mechanisms, such as: Customized Interface, Variable Instance, Guard Condition, Variant Mapping, etc. vADL is able to describe the assembly of variability in product line architecture.

ADLARS: S5 presents the ADL "ADLARS", a 3-view description of software architecture. This is an ADL with first class support for embedded systems product lines. It captures the relationship with explicit support for variability between the system's feature model and the architectural structures (using keywords like “supported”, “unsupported” and “otherwise” in the description).

ACME: S14 describes two modelling notations, Forfamel for feature models and ACME (Garlan, Monroe and Wile, 1997) for the architecture model. They are evaluated using the Formal Concept Analysis (FCA) technique, using a tool that generates a concept lattice graph that defines a mapping relationship between feature and architecture components.

ALI: S18 presents an ADL called "ALI" (a descendent of "ADLARS" (S5)) that aims to support product line engineering (and therefore also variability) as well as non-variant and individual system architectures.

Darwin: S20 presents a framework with the Darwin ADL (with elements borrowed from one of its extensions, Koala (Ommering *et al.*, 2000)). The paper proposes a decision-making process to generate a generic software design that can accommodate the full space of design alternatives from a goal model with high variability in configurations.

MontiArc: an ADL designed to model architectures for asynchronously communicating logically distributed systems. Two studies present extension to MontiArc: (1) delta-modelling to represent variability - Δ-MontiArc in S36 and S40, and (2) using hierarchical variability modelling - MontiArc^{HV} in S39. The given examples were difficult to extend if one is not using MontiArc, but the proposed variability modelling techniques were not new.

PL-AspectualACME: S37 presents PL-AspectualACME (an extension to AspectualACME (Garcia *et al.*, 2006)) with a graphical representation of the architectural model. The associated tool interprets the annotations, adding or

removing the correct variant elements in the specification. S34 presents the ADL PL-Aspectual ACME specifying the architecture for software product lines. The description is related to a goal model described in a formal visual notation PL-AOV Graph.

CBabel: S42 presents the CBabel language, with features to support software architecture and contract description with a meta-model defined for architectural contracts.

LightPL-ACME: S54 presents an ADL (an extension to ACME (Garlan, Monroe and Wile, 1997)) with the aim of having a simple, lightweight language for SPL architecture description. It enables the association between the architectural specification and the artefacts involved in the SPL development process, including the relationship with the feature model by categorically defining the variability and the representation of both domain and application engineering elements.

Most of the work reported on the use of UML and ADLs for capturing variability at the architectural level was conducted by their original authors. A small proportion of these papers (e.g. S23, S42, S50) reported on work conducted in an industrial setting, but the rest used prototype implementations based in academia. We discuss the context of the research in more detail under RQ3 analysis later in Section 4.3.3.

OVM (Orthogonal Variability Model) and XML (eXtensible Markup Language) approaches represent variability in 7% (4 papers) and 3% (2 papers) of the selected primary studies respectively. Other ways that were identified to capture variability in the software architecture are: CVL (Common Variability Language) in S47; LISA (Language for Integrated Software Architecture) in S45; formal modelling languages/framework

(e.g. S11, S16, S53) and modelling tools (e.g. S21, S28, S55, S56), and; formal/informal textual and visual descriptions such as spreadsheets and process diagrams (e.g. S2, S9, S22, S33, S44, S52).

It is important to state that the number of studies cross-cut multiple variability approaches, and accordingly, appear under more than one category in Table 3 (hence the total of 66 rather than 58). For instance, S17 and S26 covers UML and OVM; S45 covers OVM and LISA; S31 and S41 covers UML and xml variability mechanisms simultaneously. Also, S26 and S31 represent variability in both UML class and component diagrams.

Overall, UML and ADLs seemed to be the most commonly used approaches for capturing variability at an architectural level, making up 69% (40 papers) of the selected primary studies. UML was used in almost half of the studies, where it was extended through various mechanisms to support variability. While ADLs were mostly used in the product line domain.

4.3.2 SLR.RQ2: *What is the quality of the research conducted in the reported approaches?*

Based on the method described in the Chapter 2, each study received a quality score totalling between 0 and 9 (given 9 questions with possible ratings of 0, 0.5 or 1 point each). The list of studies along with their corresponding quality scores (per question) can be found in Appendix A3. Figure 9 below shows the number of studies per quality score. The chart shows a normal (Gaussian) distribution curve with a *mean* of 5.9 and *variance* of 2.4. The most common scores were 6 and 6.5 (29% of the papers). The highest score was 8.5 (two papers) with the lowest being 2 (one paper).

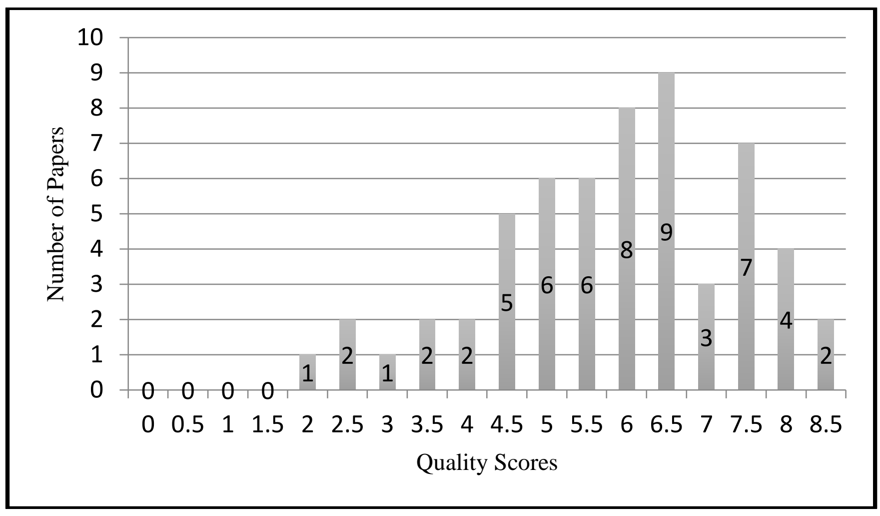


Figure 9: Quality assessment scores of studies (overall)

To further analyse the data, we broke down the quality assessment marks per question as presented in Table 4. The first column of the table shows the quality assessment question as discussed in Section 2.2.5. The remaining three columns show the number of papers assigned to each score per question. The average mark per question is shown in Figure 10.

NUMBER OF PAPERS ASSIGNED TO EACH SCORE PER QUESTION	QUALITY SCORE		
	0	**0.5**	**1**
QA.Q1: Is there a rationale for why the study was undertaken?	0	8	50
QA.Q2: Is there an adequate description of the context (e.g. industry, laboratory setting, products used, etc.) in which the research was carried out?	1	16	43
QA.Q3: Did the paper present enough details about the reference architecture variability approach?	2	14	42
QA.Q4: Is the case study (if exist) using a single or multiple case research design?	10	18	30
QA.Q5: Does the case study consider construct validity, internal validity, external validity, and reliability to the study?	32	19	7
QA.Q6: Is there a description and justification of the research design, including a statement of what the result should be (e.g. a construct, a model, a method, or an instantiation)?	4	16	38
QA.Q7: Is there a clear statement of findings with ‘sufficient' data to support any conclusions?	3	26	29
QA.Q8: Do the authors discuss the credibility of their findings?	6	30	22
QA.Q9: Are limitations of the study discussed explicitly?	49	4	5

Table 4: Quality assessment scores of studies (per question)

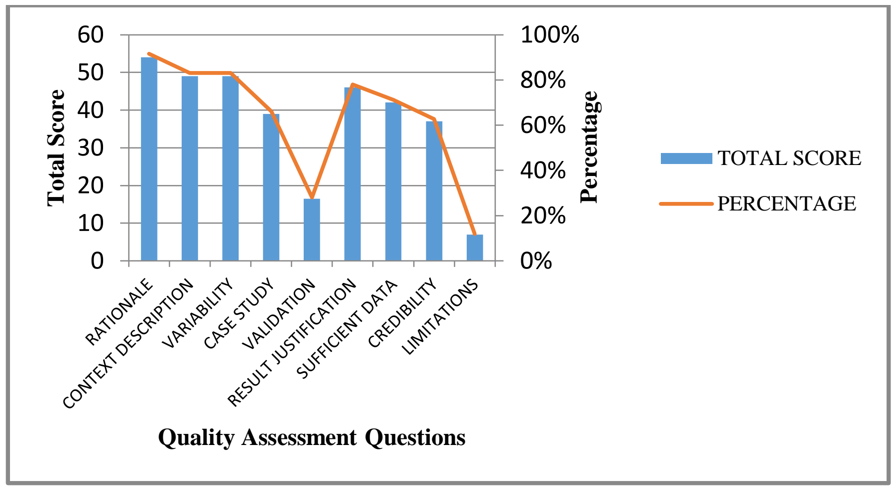


Figure 10: Overall quality assessment scores per question

As can be seen from the above data, while almost all studies presented a rationale and context description for the work conducted, few studies discussed the validity and reliability of their findings, while even fewer studies addressed their limitations (which is discussed under SLR.RQ4 analysis in Section 4.3.4). Thus, we concluded that work in this area can be characterised to generally having a clear rationale and objectives, but lacking proper validation. This might be attributed to the research context where most of this work was conducted, namely academic research with little involvement from industry. The research context is discussed further under the next research question.

4.3.3 SLR.RQ3: *What is the context and areas of research of the studies employing variability in software architecture?*

4.3.3.1 Research Context (Academia vs. Industry)

The research context of each primary study was classified as either: Academia (if the research was conducted in academia and by academics with no reference to industrial usage); Industry (if the research was conducted by industry based researchers or had direct industrial relevance); or both (when the research was a joint undertaking with both academic and industrial relevance). From the selected primary studies, we identified that only a small proportion of research (19%, 11 papers) was conducted in industry. 72% (42 papers) of the research surveyed was academic while 9% (5 papers) was classified as joint context (both industry and academia). A detailed classification of each of the primary studies is provided in Table 5.

Research Context	Total Papers	Study Identifiers
Academia	42	S3 - S6, S9 - S20, S22, S26, S27, S29 - S32, S34- S36, S38 - S41, S43, S45 - S49, S51 - S55, S57
Industry	11	S1, S8, S23, S25, S28, S33, S42, S44, S50, S56, S58
Both	5	S2, S7, S21, S24, S37

Table 5: Research context with study identifier

Figure 11 shows that the majority of studies belong to the academia sector (72%), with 20% in industry and 8% joint. However, it was noticeable that the industry initiated papers doubled between 2011-2015 compared with 2006-2010, while academic papers only gone up by 17%. Yet, joint papers between industry and academia is going down with only 1 primary study published between 2011-2015.

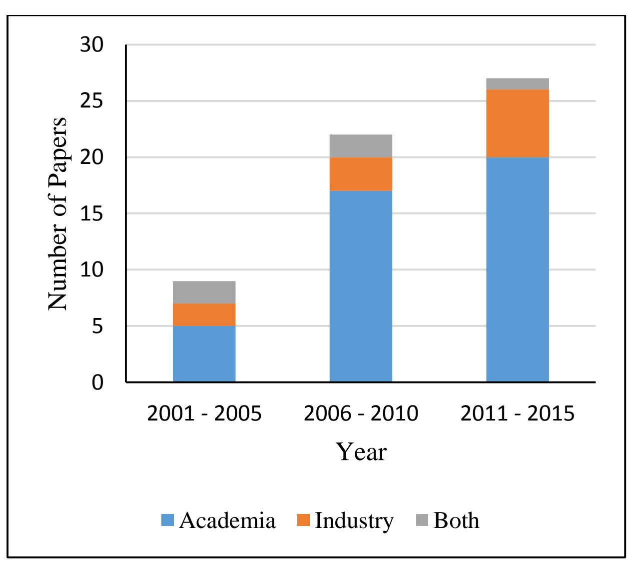


Figure 11: Research context

4.3.3.2 Research Context (Theoretical vs. Practical)

Another way the research context of the primary studies was analysed was by checking whether the reported research had a practical or theoretical focus, or both. The results are reported in Figure 12, which shows the majority of the work conducted is theoretical work with no direct application to practical problems.

Overall, 65% (38 papers) of the primary studies were focused purely on theoretical work with only 14% (8 papers) addressing practical issues and another 21% (12 papers) that can be classified as both. A full breakdown of the classification of the different studies can be found in Table 6.

Research Context	Total Papers	Study Identifiers
Research	38	S1 - S6, S9 -S18, S20, S22, S27, S29 - S32, S34, S35, S39, S40, S43, S45 - S48, S50, S52, S53, S55, S57, S58
Practice	8	S8, S23, S28, S44, S49, S51, S54, S56
Both	12	S7, S19, S21, S24 - S26, S33, S36 - S38, S41, S42

Table 6: Research relevance with study identifier

That said, Figure 12 also shows that the trend is changing with higher percentage of papers with practical relevance appearing in the past 5 years compared to 2006-2010.

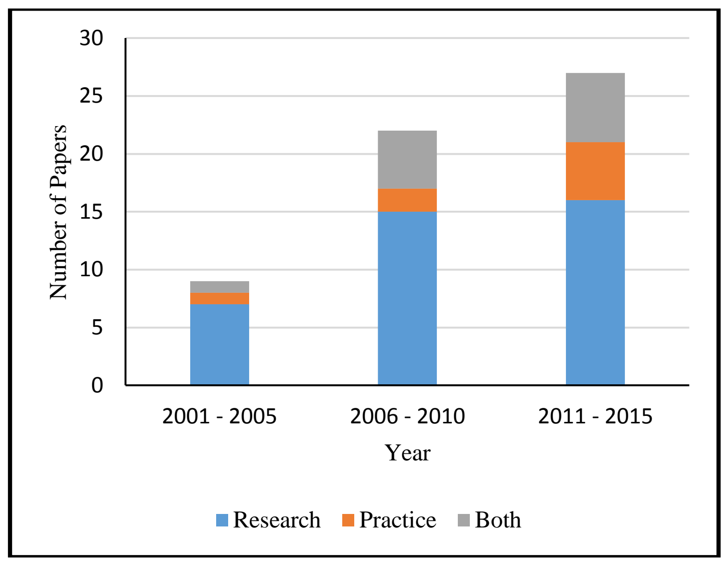


Figure 12: Research relevance

4.3.3.4 Research Area

During the analysis, it became clear that the primary studies can be categorised under four main research areas:

1. Service Oriented Architecture (SOA)
2. Reference Architectures
3. Software Product lines (including Product Line Architectures -PLA and Dynamic SPL -DSPL)
4. Other (general Software Architecture)

The breakdown of primary studies per research area is shown in the Table 7.

Research Area	Total Papers	Study Identifiers
SOA	2	S38, S44
Reference Architecture	4	S7, S8, S19, S44
Other (Software Architecture)	10	S11, S17, S20, S27, S30, S41, S43, S46, S48, S52
SPL/PLA/DSPL	48	S1 - S6, S9 - S18, S21 - S29, S31 - S40, S42, S45, S47 - S51, S53 - S58
	64§	

§ A number of studies cross-cut multiple research areas, and accordingly, appear under more than one research area (hence the total of 64 rather than 58)

Table 7: Breakdown of primary studies over research areas

Figure 13 shows a graphical distribution of the primary studies over the different areas identified. Noticeably, the work on variability in software architecture is dominated by work in the area of Software Product Lines.

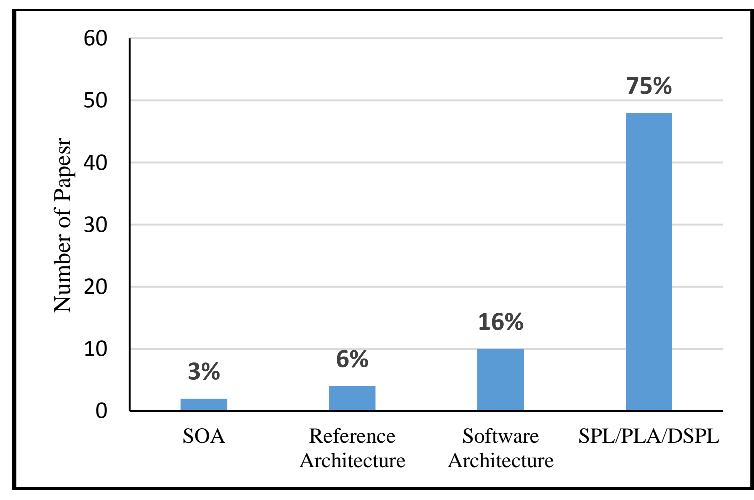


Figure 13: Breakdown of primary studies over research areas

4.3.4 SLR.RQ4: *What are the limitations of the existing approaches to represent variability in software architecture?*

Understanding the limitations of a particular piece of research is an important step towards understanding its applicability and utility. Unfortunately, in the literature reviewed for this study, 78% of the papers surveyed (45 of 58) did not make any attempt to report limitations of the research performed and 2% (1 study) didn't report the limitations of the research explicitly.

This left 12 studies (20%) that fully or partially identified limitations of their work, so helping to understand its maturity and the areas of its likely applicability. The limitations reported can be categorised under the following headers:

- Technical limitations with the research methodology adopted: For example, some papers only used one case study (S33, S49), while others used small unrepresentative study groups (S9).
- Technical limitations with the approach presented: For example, only addressing variability at either design time (S38) or runtime (S3, S27).
- Both of the above (such as S16, S25, S41 and S52).

In reality, almost any piece of research is likely to embody some limitations, so it is surprising not to find all studies reporting limitations of either type.

4.4 Threats to Validity & Limitations

This section discusses the limitations and threats to validity of our study. As with most research methods, there are some inherent limitations to the SLR methodology. The first limitation is the possibility that the search and selection process may not have identified all of the relevant primary studies. This can be due to various reasons such as the use of

different terminology in primary studies to the one we adopted in the search term (particularly given that the work covered by this SLR cuts across multiple domains and research communities). To address this limitation, the search protocol have been extended and introduces a number of mitigating measures. First, automated searches was ran on web sites of prominent publishers (e.g. IEEEXplore) as well as against general indexing search engines (e.g. Google Scholar) which helps to ensure comprehensiveness of results as different search engines use different ranking algorithms. Then, manual searches were conducted on proceedings of known publication outlets and publication lists of known authors in the domain and cross-examined the findings with the results produced from the automated search. Finally, forward and backward reference were checked on the identified primary studies to further ensure that all of the relevant literature was identified.

Another limitation of SLRs is the exclusion of grey literature, such as thesis documents, white papers and technical reports. This could be a problem in some areas such as those where the work is led by industry, as practitioners tend to publish less in peer-reviewed outlets. However, looking at the analysis of SLR.RQ3, and to some extent at the initial results obtained from the automated searches (conducted on general indexing websites such as Google Scholar), it is being noticed that this study area is largely dominated by academic researchers with minimal potential for grey literature. Last but not least, there is the limitation of the language barrier where only primary studies published in English were searched and analysed. This could potentially mean that relevant primary studies published in other languages might have been missed. There is not a strong mitigation to this threat other than noting that the majority of research in these areas appears to be published in English and so we do not believe that there is a high likelihood of significant research in this field remaining unpublished in English for long.

Beyond the inherent SLR methodology limitations, threats to validity can be classified under four main headers: construct, internal, external and conclusion (Matt and Cook, *1994*).

Some of the threats to *construct* and *internal* validity have already been discussed above. These threats arise from weaknesses in the execution of the research method adopted. A popular construct validity problem in SLRs is author bias and we have addressed this by having multiple independent reviewers each primary study and had the overall process reviewed by an independent researcher. As discussed in the Chapter 2 on research methodology.

On the other hand, the threat to *external* validity relates to the applicability of the results of the study beyond the context where it was conducted. Given that this study was not limited to one area, but studied multiple areas where variability in software architecture is used, *inductive generalization* is considerably strengthened. Moreover, we have made all of the raw data used for the study available for readers to better help them understand the reasoning and analysis conducted.

Finally, *conclusion* validity threats relate to the robustness of conclusions made based on the data available. A typical threat is when researchers gear conclusions to agree with their initial hypotheses. In our case, the research did not set any initial hypotheses but rather addressed the research questions with an open view. Additionally, all conclusions are based on grounded theory (Martin and Turner, 1986) and other analysis methods where multiple independent researchers were involved and independently agreed on the conclusions made.

4.5 SLR Update: Work beyond Search Period

This section presents the latest updated information on the research studies that represent variability in software architecture. That is, the research work that has been done after the end of the SLR search period (i.e. August 2015) and before the submission of the final version of this thesis (i.e. April 2016), in this area.

According to the SLR search strategy (defined in Chapter 2), the primary studies published between August 2015 and April 2016 were searched. Moreover, the conferences (listed in Table 1) that held between this time period were manually searched, which are: ECSA 2015, FSE 2015, VaMoS 2016 and WICSA 2016. This led to the identification of the two primary studies that met the inclusion criteria, and were published in a conference (October 2015) and journal (December 2015), respectively.

Those two primary studies are summarized as:

1) A three-peaks process to derive incrementally high variability requirements, behavioural and architecture models were presented (Angelopoulos, Souza and Mylopoulos, 2015). In that, variations of a system's architectural structure is modelled in terms of connectors and components, through UML class diagram. The research context of this study is academia and correspond to the software architecture research area.

2) An Ontology-based product Architecture Derivation (OntoAD) framework that automates the derivation of product-specific architectures from an SPL architecture (Duran-Limon *et al.*, 2015). The framework used UML class and component diagrams notations to represent variability in it. And, the research context of this study is academia while the research area is SPL.

It is important to clarify here that these two new primary studies will not affect the conclusion drawn from the SLR findings. This is because the most commonly used notation to represent variability in software is still the same (i.e. UML) and also the work is conducted within the academia sector only.

4.6 Conclusion

The work in this chapter aimed at cataloguing the state-of-the-art in representing variability in software architecture, making it more accessible to practitioners and researchers alike.

Overall, it can be said that this research area is witnessing an uptrend, especially since 2006 (see Figure 5), and that work in this domain is starting to mature. To conclude, we found that:

- UML (including various extensions) and Architecture Description Languages (ADL) were the most commonly used notations to represent variability in software architecture.
- The work on variability representation at the software architecture level can be largely mapped to three main research areas: Software Product Lines (SPL); Reference Architecture; and Service Oriented Architecture (SOA).
- Most of the work surveyed focused on proposing some form of new or improved design process or traceability technique relating to the development of systems that include variability.
- The majority of the work conducted (72%) was academically led, much of it with a fairly theoretical focus (65%).

- Overall, the research in this domain was found to have clear rationale and objectives, but generally lacking proper validation.

Finally, the top five countries publishing in this area were found to be Germany, Brazil, Korea, USA and Finland.

As analysed in this chapter, ADL is most commonly used formal notation that represents variability at the architectural level (as identified in Table 3), second to UML which is considered as a semi-formal notation. Therefore, in this research work ADL is chosen to represent variability in software architecture.

Considering this, next part of this thesis describes ALI, an ADL with its main focus to support the architectural designing of the large-scale industrial systems along with managing variability and other ADL properties. The initial version of ALI, which was designed before this thesis work began is presented in the following chapter. While the revised version (ALI V2), which overcomes the limitations that exists in its original version has been described in Chapter 6. The revised version takes into consideration the current industrial requirements from the architectural language perspectives.

Part III

ALI

Chapter Five

5

ALI Initial Version

"If I have seen further it is by standing on the shoulders of giants."

-- Isaac Newton

5.1 Introduction

In this chapter, the ***A***rchitecture Description ***L***anguage for ***I***ndustrial Applications, ALI is introduced. The version of ALI (Bashroush *et al.*, 2008) described in this section is the initial version which existed before the research reported in this thesis began. This section is largely based on (Bashroush *et al.*, 2006; Bashroush *et al.*, 2008).

The work on the ALI built on experiences gained with ADLARS (Bashroush *et al.*, 2005) and employed a number of successful concepts which existed in ADLARS to create a more generic and flexible ADL. As the name says, ALI was designed with real-life systems as the main focus.

An ALI model describes a system as a set of linked components and connectors where they are considered as first class citizens. The interfaces that define the possible interactions between components and connectors and their environment are also defined at a meta-level, by their name, their syntax and their binding constraints.

The language meta type is rich enough to capture non-functional properties or annotating the structure with additional information, at the cost of combining structural and quality property information into a single data structure. Reusable architectural structures can be defined using pattern templates that allow a named and parameterised architectural structure to be reused across a number of architectural descriptions.

Variation in the architectural structure is achieved using a feature catalogue that defines the set of features that the architecture can support. These features are then referenced from within the architectural description by using conditional statements that use the "supported" and "unsupported" keywords to vary the architectural description according to the set of features currently enabled.

Architectural configurations are defined using the system construct, which defines a set of components, connectors and a configuration that defines the bindings that define how they are combined together. The top level of an ALI architectural description is a "system" definition representing the system.

As well as describing single instance and single version systems, the design of ALI has allowed the description of variant, evolving and product line systems via first class language concepts.

The following section discusses the rationale behind ALI. Section 5.3 presents ALI's constructs and notations. This is followed by Section 5.4, which highlights a number of limitations that exist within this version in relation to the need of a current architectural language from an industrial context. Finally, Section 5.5 concludes with a summary.

5.2 Rationale

In this section, the rationale behind the ALI language which has been designed on the basis of our previous work on ADLARS ADL has been introduced. It seeks to address a number of limitations that were identified in (Bashroush *et al.*, 2006). Among these limitations are: over constraining syntax, single view presentation of the architecture and lack of tool support. Further, it can be used across multiple application domains unlike ADLARS that only support Software Product Lines (SPLs).

While adopting many of the solution space provided by ADLARS such as the relationship between the feature model and the architectural structure, ALI introduces, among other things, a high level of flexibility for interface description. Major concepts behind the ALI ADL are as follows:

5.2.1 Flexible interface description

Current ADLs allow only for fixed interface types. For example, in ACME (Garlan, Monroe and Wile, 1997) and WRIGHT (Allen and Garlan, 1997), the component interfaces are described in terms of input and output ports, while in Rapide (Luckham *et al.*, 1995) and Koala (Ommering *et al.*, 2000) interfaces are described in terms of provided and required sets of function names. Thus, providing a specific interface type which restricts the usage of an ADL to domains where most components would only have that particular type of interface. In addition to this, it restricts software architect to use a specific style of communication among components (e.g. message-based, method invocation, hardware-like ports, etc.).

The ALI ADL attempts to address this issue by providing no pre-defined interface types. Instead, ALI introduces a sub-language (which is a sub-set of the JavaCC (Java Compiler Compiler ^{tm} (JavaCC tm)) notation) that gives users the flexibility to define their own interface types.

For example, consider a simple web service having a WSDL (Web Services Description Language) interface and containing a number of components which are described with input/output ports as interfaces. Assume also, that each component contains a number of objects/classes that have interfaces defined in terms of functions provided/required as illustrated in Figure 14. Nowadays, this is a fairly standard level of nesting/abstraction particularly within Service Oriented Architectures (SOA).

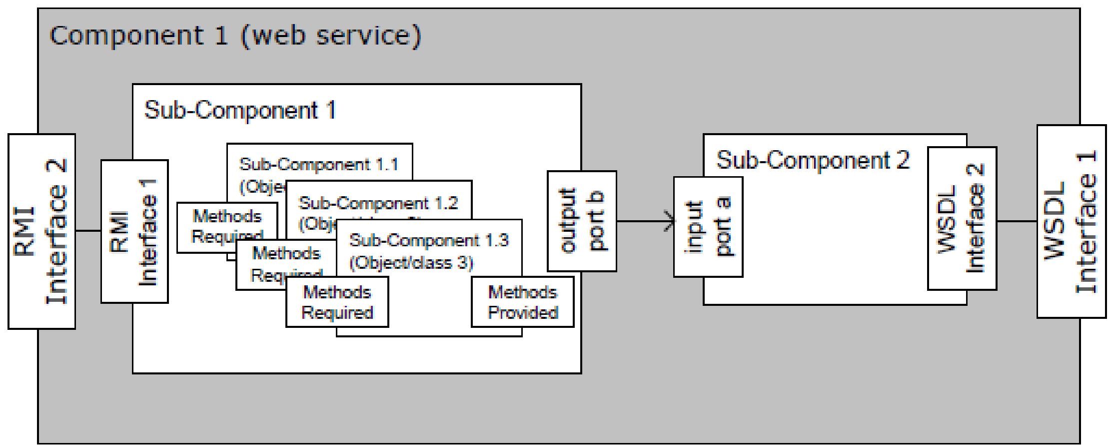


Figure 14: An example architecture of a simple web service (Bashroush *et al.*, 2006)

If we were to model this using any of the existing ADLs, we would have to abstract the different interface types with the single interface type supported by the ADL adopted. By doing so, we would be unnecessarily abstracting away useful and important architectural information - especially in domains such as SOA where interface descriptions/types considered to be of important architectural value.

It would also be difficult to identify a comprehensive set of interface types beforehand to be provided by an ADL due to the large number of interface types that already exist in the literature. Moreover, new interface types emerge with the advancement of different technologies (e.g. GWSDL emerging from the work on grid computing, etc.). So, an ADL may benefit from a flexible mechanism that allows the software architect to define his/her own interface types along with the binding constraints. This is the modelling strategy that is adopted by ALI. Details of interface description are given in Section 5.3.

5.2.2 Architectural pattern description

Architectural patterns (or architectural styles) express a fundamental structural organization or schema for software systems and sub-systems. As these patterns are often

reused within the same system (and sub-systems) or across multiple systems, providing syntax for capturing/describing these patterns to enable better pattern reuse is important.

The importance of capturing and reusing patterns is carried over to the ALI ADL. ALI envisages architectural patterns as the architectural level equivalent of functions (methods) in programming languages.

Within ALI, patterns are defined as functions and can be (re)used throughout the system description. *Pattern templates* are first defined by specifying the way components are connected to form the architectural pattern. Then, these pattern templates are instantiated throughout the architecture definition to connect sets of components (whose interfaces are passed as arguments to the pattern template) according to the pattern template definition (e.g. Figure 15).

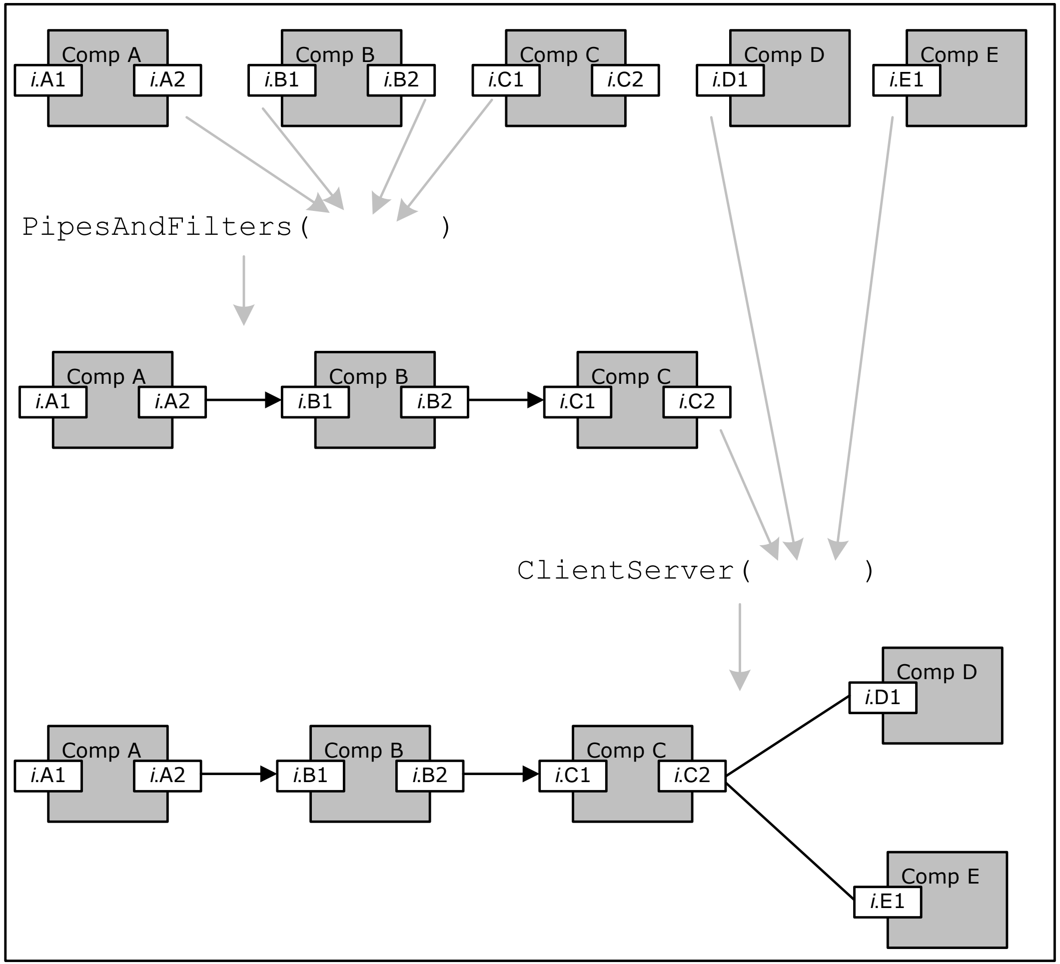


Figure 15: A simple architecture assembled from a number of components using two pattern templates: *PipesAndFilters* and *ClientServer* (Bashroush *et al.*, 2006)

As can be seen from Figure 15, simple architectures can be constructed through the usage of a number of patterns. Detailed description on the ALI notation used for describing pattern templates is given in Section 5.3.

5.2.3 Formal syntax for capturing meta-information

Issues such as component implementation cost/benefit, design decisions, versions, quality attributes, etc. have been overlooked by most of the existing ADLs. They focused more on the structural aspects of the architecture. Although ADLs such as ADLARS and few others allow the addition of free textual comments to the architecture description using standard commenting syntax similar to that used in programming languages (e.g. through the usage of "/*", "//", etc.). But it proves to be problematic if CASE tools are to be used to analyse or produce useful documentation from the free textual comments.

One of the challenges with formalizing the syntax for capturing the meta-information is in deciding what information need to be captured in the architecture description. Although there is some information that is usually captured in most architecture documentations (e.g. design decisions, quality attributes, etc.) while some other information may vary from one domain to the other and from one enterprise to another (depending on the nature of the domain, the structure of the enterprise, etc.).

A special syntax has been introduced in ALI that allows to create *meta types.* Different meta types can be created within a system to act as packages of information (quality attributes, versions, design decisions) which could be attached to different architectural elements throughout the system description. How to create and use *meta types* is demonstrated in Section 5.3.

5.2.4 Linking the feature and architecture spaces

As Feature Models (Kang *et al.*, 1990) are built to capture end-users' and stake-holders' concerns and architectures are designed from technical and business perspectives, a gap exists between the two spaces. This gap introduces a number of challenges including: feature (requirements) traceability into the architecture; the ability to verify variability implementation (in SPL, a product with multiple variants), etc.

ALI attempts at bridging this gap by allowing the software architect to link directly the architectural structures to the feature model. Within ALI, it is possible to relate components, connectors, patterns etc. in an architecture description to features in the feature model using first order logic. This permits the capture of complex relationships that might arise between the two spaces in real-life systems.

ALI has adopted and enhanced this concept from ADLARS which was the first ADL to introduce support for linking the feature space to architectural components.

5.3 ALI Constructs and Notations

In this section, the different parts of the ALI notations are discussed.

ALI is divided into seven parts:

1. ***meta types***: provides a formal notation for capturing meta-information
2. ***interface types***: provides a notation for creating types of interfaces
3. ***connector types***: where architectural connectors are defined
4. ***component types***: where architectural components are defined
5. ***pattern templates***: where design patterns are defined

6. ***features***: where the system features are catalogued
7. ***system***: where the system architecture is described

Each of these notations is discussed in detail in the subsequent sections.

5.3.1 Meta Types

Meta types provide a formal syntactical notation for capturing (meta-)information related to the architecture. A meta type is defined by the information it contains. The information is captured within fields, where each field has a data type (text, number, etc.) and a name (tag). Consider the example below for defining a meta type called `MyMetaType1`:

```
meta type MetaType1 {
   tag creator, description: text;
   tag cost, version: number;
   tag edited*: date; }
```

In this example, the keyword "`meta type`" is used to begin a meta type definition. `MetaType1` is the name of the meta type being specified. Each meta type contains a number of tags which can be either textual, numeral or date (if needed, the tag types could be extended to include: enumeration, character, etc.). In the example above, five tags are defined, two textual, two numeral and one date. The date tag `edited` is marked with an asterisk '*' to indicate an optional tag.

Once meta types are specified, meta objects conforming to these types can then be created throughout in the designing of the system architecture. These meta objects are attached to architectural elements (e.g. components, connectors, etc.) to provide a corner for appending additional information related to these elements. Below is an example meta object that conforms to the meta type given in the example above.

```
meta: MetaType1 {
   creator: "John Smith";
   cost: 5,000;
   version: 1;
   edited: 12-02-2006; // optional
   description: "A GUI component ...";
}
```

A meta object could also conform to more than one meta type. It is also possible to create meta objects that do not conform to any meta type. This enhances the language flexibility. However, little automated analysis can be done over such informally provided information.

The formal specification of meta information would considerably enhance the development of CASE tool support that could harness these meta objects and conduct automated analysis on the data (e.g. cost/benefit analysis, project timing/scheduling, etc. based on what meta information is available). Other meta information might include: design decisions, component compatibility, etc. which, when extracted and formatted using proper CASE tools, allow automated architecture documentation to be achieved on-the-fly.

In general, it is expected that the meta types be created once and used repeatedly within different systems developed by the same enterprise. A standard set of information required (tags) may be first identified by the project management team (or any other stakeholder), and then provided to architects to conform to. This ensures that critical information is always provided within an architecture description. The flexible syntax also allows the architect to augment this information with fields (tags) that they may need temporarily or internally within the architecture team.

Meta information can be introduced anywhere in the architecture description (component type definitions, connector type definitions, etc.). The meta information

would be attached to the placeholder of where it is defined. For example, if a meta object is created within a connector type, then this information belongs to that connector type. If a meta object is defined within an interface description within a component type, the meta object then belongs to the interface, etc.

5.3.2 Interface Types

Interface types have been introduced to ALI to allow for the usage of multiple interfaces within a system description. The idea is to create a set of common interface types needed within an application domain once (e.g. WSDL, Functional, Invocation, etc.), and then use these interfaces in the design of architectural elements (component, connector, etc.) and systems.

The interface type definition is divided into the following two sections:

- *Syntax definition*: where the formal syntax of the interface description is specified using a subset of the JavaCC (Java Compiler Compiler tm (JavaCC tm)) notation.
- *Constraints*: where the constraints for interface binding (connectivity) are specified as follows:
  - *Should match*: means that the terms (identified in the syntax definition section using the JavaCC notation) should match between two interfaces to be considered compatible (allowed to bind) are identified. For example, in a functional interface, for two interfaces to be compatible, the function names and argument types should match.
  - *Protocols supported*: a list of the protocols for communication is provided that is supported by this interface type. E.g.: IIOP, HTTP, method invocation, etc.

- *Allow multiple bindings*: This is a Boolean value that states whether multiple binding is allowed on this interface or not. Example: this property is set to true on a server socket interface to allow for binding multiple client socket interface while it is set to false on the client socket interface.

- *Factory*: This is a Boolean value that states whether the interface is a factory or not. A factory interface means that when a connection request is received on this interface, a new connection dedicated interface is created to handle that particular request while the main interface proceeds to listen to new incoming requests. Example: server socket interfaces in java are factories. On contrary, C++ sockets are not. In C++, the factory functionality is to be implemented by the programmer if required.

- *Persistent*: This is a Boolean value which when set to true indicates a persistent interface (the internal data of the interface component is kept unchanged after the current connection has ended) and when set to false indicates a transient interface (internal data is reset to initial values when the current connection is terminated).

Following is an example for defining an interface type `functional`:

```
interface type functional {
  syntax definition:    {
   "Provided" ":"  "{"
    [ "function" <PROV_FUNCTION_NAME>
         "{"
            "impLanguage"  ":" <PROV_LANGUAGE_NAME> ";"
            "innvocation"  ":" <PROV_INVOCATION> ";"
            "paramterlist" ":" "("[<PROV_PARAMETER_TYPE> [","
                              <PROV_PARAMETER_TYPE:]*]? ")" ";"
            "return type"  ":" <PROV_RETURN_TYPE> ";"
          "}"]*   "}"
     // Required:  etc.
  }
```

```
constraints:  {
     should match: {PROV_INVOCATION_NAME, PROV_PARAMETER_TYPE}
     protocols supported: {RMI-IIOP, JRMP}
     allow multiple bindings: false;
     factory: false;
     persistent: true;
   }
 }
```

For more information about the notation used for specifying the interface syntax, please refer to JavaCC (Java Compiler Compiler tm (JavaCC tm)).

It is important to clarify here that the interface type definition is not meant to be read by humans, but rather created once and then read by CASE tools that would verify the interface descriptions and bindings made throughout the architecture definition.

5.3.3 Connector Types

As in ACME (Allen and Garlan, 1997) and other ADLs, connectors are considered first class citizens in ALI descriptions. For this, a proper syntax was introduced to the language to allow for creating *connector types*.

The following example shows how to create a connector type `ConnectorType1` in ALI:

```
connector type ConnectorType1
   {
        meta:
         {
      description: " something about the connector type ";
         }
        interfaces {
           a, b, c of type functional;
         // etc.
     }
```

```
    layout {
          connect all to all;
     if (supported(FEATURE_A))
               connect a and b;
          else
               connect a to all;
               connect a to c;
   }
}
```

As shown above, it is possible to attach meta objects to connector types.

The connector type definition is divided into two parts:

- *interfaces*: where the connector interfaces are defined. These are the input and output ports of the connector. A connector must have at least two interfaces (for input and output) while theoretically there is no restriction on the maximum number of interfaces. For example, a *bus* connector would need to have a number of bi-directional interfaces to serve all components connected to the bus. On the other hand, a simple connector has only two interfaces. In the example above, we have defined three interfaces `a, b` and `c` to better demonstrate the functionality of the connector type in terms of its configuration as discussed later in this section.
- *layout*: The layout section describes the internal configuration of the connector. It shows how the connector interfaces are connected internally, that is, how the traffic travels among the interfaces. There are two types of connections allowed between connector interfaces:
  - *unidirectional connections (to)*: which specify that the data/requests received on one interface be output on another interface. This is done using the keywords: “`connect`“ and “`to`“. Example: `connect a to b`; outputs the data/requests received on the **a** interface to the **b** interface.

- *bi-directional connections (and)*: which specify that the data/requests received on one interface be output on another interface and vice versa. This is done using the keywords: "`connect`" and "`and`". Example: `connect a and b`; outputs the data/requests received on the **a** interface to the **b** interface and vice versa.

The keyword "`all`" can be used to connect a connector interface to all other interfaces of the connector using a bi-directional or unidirectional communication as described above. For example, `connect a to all` makes the input on interface **a** available as output on all other interfaces of the connector. In contrast, `connect a and all` makes the input on **a** available on all other interfaces and the input on all other interfaces available on **a**. The statement: `connect all to all` can be used to create bi-directional connections among all ports.

Like *interface types* and *meta types*, a set of *connector types* can be created per domain that can then be reused across multiple projects within that domain.

The connector definition can be linked to the system feature model to allow for connector customization based on features selected. This is done using the if/else structure and the keywords "`supported/unsupported`." So, in the example above, if the system supports the `FEATURE_A`, interfaces `a` and `b` are connected as bi-directional (using "`and`"); otherwise, they are connected as unidirectional (using "`to`") as `a` **`to all`** and `a` **`to`** `c`. This syntax introduces a high level of configurability to the connector definition which provides better support for defining configurable and product line architectures.

Meta objects can be attached to connector types by simply defining the meta object (as explained in Section 5.3.1) inside the connector type definition (anywhere between the start and end brackets).

5.3.4 Component Types

This section will provide a formal syntax for the creation of *component type* which is a crucial part of the ALI notation. Once a component type is created, multiple components of that type can be instantiated, each customized based on the feature set it supports.

The component type definition is basically divided into two main sections:

- *interfaces*: which describes the different component type interfaces. These interfaces are described conforming to defined interface types (defined in the interface type section earlier). A component can have one or more interfaces of different types.
- *sub-system*: where the internal structure (sub-system) of the component is described. The sub-system section consists of three sections:
    - *Components*: where the different sub-components included within the component are defined
    - *Connectors*: where the different connectors that will be used in connecting sub-components are defined
    - *Configuration*: where the way sub-components are connected is described. Three methods can be used to connect components:
        - *Using a connector*: where a connector mediates the connection between two or more components.
        - *Direct connection*: without the use of connectors by directly binding the component interfaces together.
        - *Using patterns*: predefined connection patterns can be used to automatically connect components using pre-defined architectural

patterns. More information about on patterns are given in the next section.

An example of the syntax used for defining component types is demonstrated below:

```
component type ComponentType1
  {
       meta: MetaType1, MetaType2
       {
           description: "this is an example component";
           cost: 20,000;
           benefit: "increases system price by 5%!";
           version: 3;
           Author: "John Smith";
           Design_Decision: "this component has been designed…";
       }

       interfaces:
       {
          Interface1 of type functional
             {
                   Provided:
                   {
                         function myAddFunction
                         {
                               impLanguage: Java;
                               invocation: add;
                               parameter list: ( int );
                               return type: void;
                         }

                         function mySubtractFunction
                         {
                               impLanguage: Java;
                               invocation: subtract;
                               parameter list: ( int );
                               return type: void;
                         }
                   }
             }
        if (supported(Provide_WSDL_Interface_Feature))
```

```
        {
          Interface2 of type WSDL
            {
              // WSDL interface description
            }
          }
      }

sub-system:
    {
   components
     {
      component1<customization_feature_set1>: ComponentType1;
      component2<customization_feature_set2>,
      component3<customization_feature_set3>: ComponentType2;
      if (supported(Some_Feature_A))
        component4<customization_feature_set4>: ComponentType3;
      else
        component4<customization_feature_set5>: ComponentType3;
          //etc.
     }
    connectors
      {
       connector1<customization_feature_set1>,
       connector2<customization_feature_set2>: ConnectorType1;
       connector3<customization_feature_set3>: ConnectorType2;
       if (supported(Some_Feature_B))
        connector4<customization_feature_set4>: ConnectorType2;
       // etc.
      }
     configuration
        {
          // 1 - connecting components using connectors
          connect component1.interface1 with connector1.a;
          connect component2.interface1 with connector1.b;

         // 2 - connecting components without connectors
     bind component3.interface1 with component1.interface2;
         // 3 - connecting components using defined patterns
          if (supported(Some_Feature_B)) {
       Client_Server(ServerComponent1.interface1,
                     [ClientComponent1.interface1,
```

```
                    ClientComponent2.interface1,
                    ClientComponent3.interface2]);
            }
          else {
        PipesAndFilters([PipeComponent1.interface2,
                         PipeComponent2.interface1,
                         PipeComponent2.interface2,
                         PipeComponent3.interface1]);
            }

          // connecting the component interface(s) with sub-
    // components' using the keyword "my" (explained later)

     bind component1.interface2 with my.Interface1;
       }
     }
```

In the example above, we begin the component description using the keyword "`component type`" followed by the component type name, `ComponentType1` in this example.

The first section of the component definition contains a meta object which conforms to two meta types: `MetaType1` and `MetaType2`. These meta types are defined in the *`meta type`* section.

The second section is the component *`interfaces`* section where two interfaces are defined:

- *Interface1*: of type `functional` (an interface type that was defined as an example in Section 5.3.2 - interface types)
- *Interface2*: of type `WSDL` that only exists if the feature `Provide_WSDL_Interface_Feature` is supported by the system.

As described in Section 5.3.2, `functional` interfaces are defined in terms of the functions they provide and require. So, in the example above, the interface `Interface1` provides two functions and requires none; and so on.

We could define as many interfaces as we want, where we could link the existence of interfaces to the support/unsupport of system features. We could also attach meta objects to interfaces simply by defining them within the definition of the interface (somewhere between the two curly brackets of the interface definition).

It is recommended that interface definitions conform to defined interface types as per the example above (`functional` and `WSDL` types). However, to allow for maximum flexibility, it is possible to define interfaces that do not conform to any pre-defined interface type, in which case, no analysis or automated tool support can be enabled over that interface definition or any connection made over it (similar to the concept of creating arbitrary meta objects that do not adhere to any meta type definition). This is done by dropping the interface type name that follows the interface name in the interface definition. For example, one could define a port-like interface without having an interface type readily available:

```
myPortInterface3:
   {
    input in1, in2, in3;
    output out1, out2, out3;
   }
```

However, it will not be possible to validate whether the connection between this interface and any other interface within the system is valid or not (as the interface syntax and constraints are not formally defined). This could be practical at early design stages when the exact interface type specification is not clear. When the interface type matures enough throughout the design process, an *interface type* is defined for this type of interface, and then the *interface type* name is appended to the interface definition above

to allow for verification, and perhaps automated analysis with the aid of appropriate CASE tool support.

The third section in the component definition is the description of the sub-system. In the example above, three components are defined in the *`components`* section, one of type `ComponentType1` and two of type `ComponentType2`, where each component is customized with a different feature set. Also, a component of type `ComponentType3` is defined; however, its customisation is dependent on the existence of the feature `Some_Feature_A`.

Similarly, a number of connectors are defined in the *`connectors`* section within the sub-system description.

The configuration section shows how the components and connectors defined in the *sub-system* section are configured (connected). As explained earlier, there are three different ways in which components can be connected:

- *using connectors*: The syntax for connecting two component interfaces via a connector is: **`connect`** *`interface1`* **`with`** *`interface2`*, where *interface1* and *interface2* are the interfaces of the component to be connected and the connector to be used, respectively. This same statement is used again to connect the second component to another interface of the connector. The direction of communication between the components is governed by the connector depending on what connector interface each component is connected to (due to the fact that connector interfaces are specified as input, output, or bi-directional within the connector type definition). A demonstration of this type of connection is found in the example above:

```
...
connect component1.interface1 with connector1.a;
connect component2.interface1 with connector1.b;
```

```
...
```

- *direct connections*: The syntax used to connect two components directly without the use of connectors is: `bind` *`interface1`* `with` *`interface2`*, where *interface1* and *interface2* are the interfaces of the two components to be connected. By default, when two components are connected directly, bi-directional communication between the components is allowed. A demonstration of this type of connectivity is found in the example above:

```
...
bind component3.interface1 with component1.interface2;
...
```

- *using patterns*: To connect components using patterns, the patterns need to be defined as *pattern templates* (explained in the next section). A pattern template definition is similar to a function definition with the arguments being the interfaces to be connected and the definition describing how these interfaces are connected. Once a pattern template is defined, it can then be invoked to connect a number of components using the specified pattern. This is done by providing the component interfaces as arguments to the pattern template as demonstrated in the example above:

```
...
Client_Server(ServerComponent1.interface1,
  [ ClientComponent1.interface1,
    ClientComponent2.interface1,
    ClientComponent3.interface2 ]
);
...
```

Finally, the component interface can be connected to its sub-component interfaces using the keyword "`my`" and the same syntax described above for connecting interfaces

(whether, direct, using connectors, or using patterns). This is demonstrated in the example above in:

```
        ...
    bind component1.interface2 with my.Interface1;
        ...
```

Connectivity among components can be related to system features to allow for sub-system re-configurability based on the feature set it supports (as shown in the example above where the components are either connected in a Client/Server pattern or Pipes & Filters pattern based on the availability of `Some_Feature_B`).

The following section explains more about design patterns and the use of *pattern templates*.

5.3.5 Pattern Templates

Architectural patterns are common solutions to recurring problems at the design level. To allow for better reuse of such patterns, ALI introduces *pattern templates*. Pattern templates are used for defining bundled sets of architectural patterns that can be reused throughout the architecture with a simple call to the pattern template needed. Pattern templates take as an argument the component interfaces to be connected according to the pattern template definition.

Pattern templates are defined in similar way to the definition of functions (methods) in programming languages. A pattern template definition comprises of:

- *Pattern name*: a unique pattern name
- *Arguments*: set of component interfaces to be connected. You can specify as arguments single interfaces and/or arrays of interfaces. In the case of arrays of

interfaces as arguments, you can also specify the minimum and maximum number of interfaces passed (e.g. [MIN < N < MAX] where MIN is the minimum number of interfaces and MAX the maximum number of interfaces). Specifying the maximum number of interfaces is optional.

- *Definition*: the specification of how the interfaces are to be connected. The syntax used for defining patterns is very simple and provides support for:
  - *connecting interfaces*: using the same syntax used in the *connections* section of the *connector type* definition (discussed in Section 5.3.3).
  - *defining loops*: to allow for connecting arrays of interfaces. The syntax used here is the same syntax used in C/C++ for creating *for* loops. Note here that the arrays of interfaces start at index 1 and not at 0 (like in C/C++).

Below is an example that defines two patterns: *Client/Server* and *Pipes & Filters*.

```
pattern templates:
   {
        Client_Server (server : InterfaceType1,
          clients [1 < N] : IntefaceType1 )
        {
           for( i = 1 ; i <= N ; i ++ )
              connect clients[i] and server;
        }

        PipesAndFilters (filters[2 < N]: InterfaceType3 )
        {
           for( i = 1; i < N ; i += 2 )
              connect filters[i] to filters[i+1];
        }
   }
```

Consider the `Client_Server` pattern template definition in the example above. The pattern takes as an argument one interface called `server` of type `InterfaceType1`, and

an array of interfaces called clients (with [1 < N] meaning a minimum of one client interface) of type InterfaceType1. The pattern is defined as: for all N *clients*, create a bi-directional connection with the *server* interface (refer to Section 5.3.3 for more about the usage of the keywords: "connect", "and", and "to" for connecting interfaces).

PipesAndFilters, on the other hand, takes an array of interfaces called filters (with a minimum of 2 filters) of type InterfaceType3. The pattern is defined as: for all N *filter interfaces*, connect the first *filter interface* to the second, and the third to the fourth, the fifth to the sixth, etc. This is due to the fact that the first and second interfaces are the input and output interfaces of the first filter component; the third and fourth interfaces are the input and output interfaces of the second component, and so on.

An example of how to use pattern templates in connecting a number of components have been already illustrated in the previous section (Section 5.3.4) where we demonstrated the use of the Client_Server and PipesAndFilters patterns.

As explained earlier, meta objects can be attached into the formal definition of any architectural element in ALI. This implies to pattern templates as well. A typical meta type that goes with patterns is:

```
meta type MetaPatterns
{
   tag Intent*, Aliases*, Motivation*, Applicability*: text;
}
```

It can be noticed that all tags are made optional by appending an asterisk, "*", to their definition. A sample of meta object that conforms to MetaPatterns is described below with the description of each of their tags:

```
meta: MetaPatterns
{
   Intent: "What the pattern does";
   Aliases: "Other names used for this pattern";
```

```
    Motivation: "An example of a problem and how this pattern
                 solves that problem";
    Applicability: "Scenarios to which this pattern applies";
  }
```

To attach meta objects to pattern template definitions, the meta object can be inserted inside the definition of the pattern template as shown below:

```
pattern templates:
  {
    Client_Server (server : InterfaceType1,
     clients [1 < N] : IntefaceType1 )
       {
  meta: MetaPatterns
  {
  Intent: "What the Client Server pattern does";
  ...
  }
           for( i = 1 ; i <= N ; i++)
                 connect clients[i] and server;
   }
       ...
       ...
}
```

5.3.6 Features

The feature description section provides a catalogue of the features used within the system. The feature definition consists of:

- *Alternative names*: the possible alternative names (if any) that are used to reference the same feature within the different design and development groups involved in the project.
- *Feature parameters*: A feature can carry a number of parameters (textual, numerical, etc.). For example, if the feature is "Manual Gearbox", the parameter might be the "number of gears" available (a numerical value).

In addition to this, meta object is attached to features to provide more meta information about the feature which is optional as discussed in Section 5.3.1.

An example below shows how features are defined in ALI:

```
features
  {
     featureA    // feature name
       {
         meta: featureMeta
          {
               details: "A textual description of the feature";
               development_cost: 10000;
               employees_needed: "3 person/year";
               acceptance_level: "should work on all screen
                                   resolutions";
             }
          alternative names: { "Developer.XY ", "Evaluator.F124"}
          parameters: { (windowTitle: text),
                        (windowWidth, windowHeight: number)}
       }

       featureB
       {
             ...
       }
       // etc.
  }
```

In the example above, we see the definition of `featureA` which contains a meta object that conforms to the `featureMeta` meta type. In the `alternative names` section, we notice that `featureA` is referred to as `XY` by *Developers* and as `F124` by *Evaluators*. In the `parameters` section, the feature encompassses three parameters: `windowTitle` which is a textual value; `windowWidth` and `windowHeight` which are numerical values.

The features defined in this section are usually derived from the feature model of the system. This is usually carried out prior to embarking on the architecture design. In order to read feature models and populate this section, CASE tools could be used. This is an

important part of the notation as it makes ALI independent of any particular feature modelling technique.

5.3.7 System

The last part of the ALI language to be discussed is the *system* section. The *system* section is where the overall product (or product line) architecture is specified.

The syntax used in this section is the same as the syntax used in the *sub-system* section (described earlier in component types, Section 5.3.4) with the major difference that the *system* section is not contained in any component definition but rather provides the description of the overall system architecture. Therefore, the keyword "`my`" used in the *sub-system* section to reference the component interfaces is not supported in this section; however, a new keyword "`external`" is used in place of it to reference interfaces of external systems (if needed) to provide a means to define how the system interacts with its environment (operating system, other systems, etc.).

The example shows a sample *system* description using the same example described in Section 5.3.4 but without the use of the "`my`" keyword and showing how the "`external`" keyword can be used (at the end of the example):

```
system:
      {
 components
    {
     component1<customization_feature_set1>: ComponentType1;
     component2<customization_feature_set2>,
     component3<customization_feature_set3>: ComponentType2;
     if (supported(Some_Feature_A))
        component4<customization_feature_set4>: ComponentType3;
    else
        component4<customization_feature_set5>: ComponentType3;
        //etc.
```

```
        }
    connectors
        {
         connector1<customization_feature_set1>,
         connector2<customization_feature_set2>: ConnectorType1;
         connector3<customization_feature_set3>: ConnectorType2;
         if (supported(Some_Feature_B))
            connector4<customization_feature_set4>: ConnectorType2;
         // etc.
        }
    configuration
        {
          // 1 - connecting components using connectors
          connect component1.interface1 with connector1.a;
          connect component2.interface1 with connector1.b;
          // 2 - connecting components without connectors
         bind component3.interface1 with component1.interface2;
         // 3 - connecting components using pre-defined patterns
          if (supported(Some_Feature_B)) {
             Client_Server(ServerComponent1.interface1,
                           [ClientComponent1.interface1,
                            ClientComponent2.interface1,
                            ClientComponent3.interface2]);
                 }
          else {
             PipesAndFilters([PipeComponent1.interface2,
                             PipeComponent2.interface1,
                             PipeComponent2.interface2,
                             PipeComponent3.interface1]);
             }

        // connecting system component interfaces to
        // external interfaces
        bind component1.interface2 with external.windowHandleAPI;
          }
        }
```

In the above example we also see that connector and component instantiation and connectivity among them can be related to system features to allow for system customisation based on the feature set selected.

5.4 Limitations in original version

ADLs that exists in current research literature were critically analysed in Chapter 3. Accordingly, several limitations were identified in them (explained in Chapter 3) that confines practitioners to adopt those ADLs into their system.

After thoroughly analysing the current ADLs and their limitations with the original version of ALI (discussed earlier in this chapter), the comparative study reveals the following limitations in ALI that might be restricting its adoption into industrial applications:

5.4.1 Architectural artefact reusability

Traditionally, research on software reusability has been primarily focused on the reuse of code-level entities, such as classes, subroutines, and data structures. While there have been tremendous improvements in code reuse technology and methods but code-level artefacts are not the only ones that can be profitably reused. Over the past decade there has been a vast amount of work done in other areas of software engineering such as at architectural level, the concept of reusability along with the support to capture variability were not taken into consideration among them.

At present, the discipline of software architecture has been more focused on original designing but it is now recognised that to achieve better architecture, more quickly and at lower cost, we need to adopt a design process that is based on architecture artefact reuse with its ability to capture variability in its description. For this, architectural languages should be capable enough to design architectural elements in such a way that it can be reused across different projects including in different domains (if needed). Also, it needs to be done by simply plugging the architectural elements into the system designed by

other vendors with relatively little efforts due to their ability to capture variable artefact features.

By focusing on the reuse of architectural elements with the support of capturing variability in an ADL makes practitioners easy to adopt the language. Currently, ALI lacks in designing architectural elements (components, connectors and interfaces) in such a way that it can be reused outside the particular system description. *Component type, connector type* and *interface type* notations as defined earlier in Section 5.3 depends on the features that have been described in features section (Section 5.3.6) in the form of catalogue. These features belong to a particular system which have been populated from the pre-defined feature model that prevents these architectural elements to be reusable elsewhere.

5.4.2 Limited support for behavioural description

At the architectural level, it is important to provide a notation that supports both the structural and the behavioural aspects of design while maintaining a separation of concerns between them. More ahead, these two aspects need to be described in a single formalism, while keeping their syntactical notations separate. By doing this, it would facilitate the understanding of each aspect in isolation while still supporting analysis of the combined interaction between the two.

In that case, an architectural language must have the capability to describe both the structural and the behavioural aspects of a system with a clear separation of concerns in order to have a complete architectural description of it. Particularly, behavioural description need to be designed carefully as it demonstrates the different functionalities of a system within a same structural design. This is very important for ADLs that have

been designed with the intention for its industrial usage because their system behaviour varies more frequently as compared to its static structure.

Unfortunately, the current version of ALI focused more on the structural aspect of the architectural design. It does not consider the behavioural aspect of the system as a first class element in its architectural description. How components will interact with each other under a particular condition, how a component will behave in different scenarios, what are those components and their interactions that performs a particular function of a system and so on, these are some of the basic behavioural perspectives that were not taken into account in this version of ALI.

5.4.3 Lack of support for graphical representation

From the industrial experience, it has been observed that carefully designing the detail of the graphical notation for ADL pays off (Woods and Bashroush, 2015). For this, along with the textual notation, a rich graphical notation for the designing of architecture is required that communicates as much as possible using the shape, line, fill and other visual aspects of the notation.

In addition to this, by using simple graphical notations with different dimensions helps software architect and other system users to remember them, even if they do not guess the link between the shape and the concept themselves. This would be helpful in particular for the new user/architect that takes over the industrial systems, where he/she can easily understand the complexity of the system in lesser period of time in comparison to understanding the textual notation first. Also, it will become much easier for people (especially, non-technical/business personnel's) to understand the system architecture and its functionality without going into any technical training. More specifically from

business perspectives, graphical representation of the architectural language can be useful for knowledge sharing and discussion on about the system.

Considering the importance of graphical representation in an ADL, it has been realised that graphical notation is an essential part for the designing of industrial system architecture which ALI lacks in providing the formal support for graphical representation in its architectural description. Although, an informal graphical notation (boxes and lines only) have been presented in Section 5.2 just to demonstrate the concept of flexible interface and architectural pattern descriptions in ALI. It is not complete and well-defined graphical representation about the ALI architectural concepts as defined textually in Section 5.3. For instance, a clear discrimination about the connections made between the components that either it is via connector or via component interfaces (direct connection) and so on.

5.4.4 Lack of support for architectural views

The complexity of large-scale industrial software systems is increasing rapidly over time. Subsequently, it led to an increase in its architectural information which have to be adequately captured. The need to integrate all the information within and across business boundaries adds a very high level of intricacy. Additionally, balancing between different stakeholders' needs can be an unapproachable business goal without the concept of multiple architectural views in a system description.

Therefore, there is an emerging need for multi-view architectural modelling, where each view delivers a different perspective or point towards a different concern or stakeholder. In this context, multiple views used to describe an architecture must be managed properly, as well as the consistency and completeness across them.

As ALI architectural description focused more on structural aspect of the system (explained in Section 5.4.2) and this aspect is described only through textual notation which limits ALI to a particular architectural view of the system. Although, ALI provides flexibility while maintaining the formality in its textual notation to describe the architectural elements (components, connectors, etc.) individually but it describes overall system description in a single *system* notation as defined in Section 5.3.7. This strategy would become chaotic in case of large-scale and complex systems for different stakeholders (such as management and technical stakeholders), where they want to view the architectural information for a particular set of related concerns not the whole system description.

5.5 Summary

ALI is a flexible architecture description language for industrial applications that was designed within our research group. This chapter introduced the original version of ALI that existed when this research work started.

ALI provides a blend between flexibility and formalism. While flexibility gives freedom for the architect during the design process, formalism allows for architecture analysis and potential automation using proper CASE tool support (e.g. on-the-fly architecture documentation, code generation, etc.). The language notation serves as a central database of the architecture description. In this way, the architectural model will help alleviate the problem of mismatches among multiple views of the system when maintained separately.

The rationale behind the ALI notation were: flexible interface description, architectural pattern description, formal syntax for capturing meta information, and linking the feature and architecture spaces.

ALI notation provided no pre-defined interface types. Instead, it has introduced a sub-language that gives users the flexibility to define their own interface types. Also, the notation focuses on capturing architectural meta-information and introduced formal syntax (meta types and meta objects) for this purpose. Continuing the theme of flexibility, ALI permits the user significant scope for defining architectural patterns. In other words, patterns may be defined and instantiated in similar fashion to function calls in programming languages. It also supports the relationship between components, connectors, patterns etc. in an architecture description and features in the feature model using first order logic.

After taking into account the current industrial needs for architectural language, several limitations were identified in this version of ALI which might be restricting its uptake in industry. Among those are: lacks in providing architectural artefact reusability, limited support for behavioural description, formal graphical representation and multiple architectural views of the system in the language.

To overcome aforementioned limitations, the next chapter presents an enhanced version of ALI (referred to as ALI V2 in this thesis), which is designed with the intention to meet current industrial requirements in terms of architectural description.

Chapter Six
6
ALI V2

"A woman's mind is cleaner than a man's: She changes it more often."

-- Oliver Herford

6.1 Introduction

Recently, ADLs constructed with complex or obscure syntactical notations have been rarely used correctly (Woods and Bashroush, 2015). Generally, practitioners avoid adopting complex languages into their development process, especially in large scale systems, where it becomes tedious to handle and understand.

Therefore, after analysing the existing literature and considering the latest recommended practitioner's guidelines (Garlan, 2014; Malavolta *et al.,* 2013) and characteristics (Bashroush *et al*., 2006) to design an architectural language, a new version of ALI (referred to as ALI V2, in this context) is presented in this chapter. The ALI V2 notation is designed in such a way that it can be easily usable in an industrial setting by overcoming the limitations that exist in its initial version (reported in Chapter 5).

The remainder of this chapter is organised as follows: Section 6.2 presents the design principles behind ALI V2 and the high-level (abstract) description of ALI V2 in the form of a conceptual model in Section 6.3; Section 6.4 covers the details of the language by visiting the different textual constructs in the ALI V2 notation with its graphical representation in Section 6.5. Section 6.6 describes the structural and behavioural semantics of ALI V2. Finally, a summary along with the changes to the ALI initial version (described in the previous chapter) is presented in Section 6.7.

6.2 Design Principles

This section presents the set of design principles, which were used to drive the development of the ALI V2 notation in order to address the limitations pointed out in Chapter 3.

Six general principles guided the creation of the ALI V2 ADL:

P1: *Variability management*

Software architects need adequate support for dealing with variability in designing their system architecture. As stated in (Galster and Avgeriou, 2011), it is essential for the architect to have suitable methods for handling (i.e., representing, managing and reasoning about) variability. From an architectural description perspective, variability is a concern of multiple stakeholders, and in turn affects other concerns. So, variability needs to be treated in a similar way to other essential functionalities of the architectural language.

Our proposed ALI V2 ADL treats variability as a first class citizen and manages it as an integral part of the language. It provides the ability to manage variability not only in the overall system architecture description but also in the design of individual architectural elements – interfaces, connectors and components. This is done with the help of a simple if/else structure concept along with the keywords "**supported**" and "**unsupported**". Additionally, ALI V2 supports variability management in its behavioural description using the same if/else structure (details in Section 6.4.12).

P2: *Requirement traceability*

Tracing requirements from the problem space (specification) into the solution space (implementation) provides a valuable tool for architects to help validate the produced system against its set objectives. However, for such an end-to-end traceability to work, there needs to be continuity in capturing relevant information at all development stages, including architecture.

ALI V2 supports requirements traceability by supporting the linkage of end-user features (see Section 6.4.2) directly to architectural elements (components, connectors, patterns etc.) using first order logic (to allow for complex dependency).

Additionally, ALI V2 supports 'conditioning' the behavioural aspects of the system to external parameters (see Section 6.4.10). Such conditions can also be used to change the behaviour of the system given various external requirements.

P3: *Cross application domain modelling*

With the emergence of new paradigms such as the Internet of Things (IoT) and Cyber-Physical Systems (CPS), architecture descriptions are now faced with the challenge of encompassing multiple application domains. For example, if we consider the Smart Cities scenario, systems in this application domain will entail the applications running sensor platforms (IoT devices), communication gateways, databases (Big Data infrastructure), and Information Systems that deliver end-user services. While the architecture of the sensor systems can be modelled using embedded-system oriented ADLs, these ADLs don't necessarily lend themselves to representing the architecture of Information Systems. Thus, there is a need for new generation ADLs that are capable of modelling system across traditional design boundaries.

ALI V2 supports cross-application domain modelling by introducing flexibility in the notation design at different levels. For example, ALI V2 allows for the creation of custom interface types using a dedicated notation. The case studies discussed show ‘port’ like interfaces, as well as ‘WSDL’ interfaces.

P4: *Balance formality and flexibility to better support the design process*

During the early stages of the design process, architects tend to sketch things at a very high level, using mere lines and boxes. At that stage, for example, it is difficult to start talking about the details of interfaces between components or what meta information to capture about each architectural element.

As the system development process progresses, and as more details are captured about the system, specific details in relation to architectural elements can then be discussed and modelled. Thus, an ADL needs to allow some flexibility at the initial stages of the design process, and at the same time, provide the required formality when details are available.

For example, ALI V2 achieves this balance between formality and flexibility by allowing architects to work with undefined interface types. When such informal interfaces are used, no automated analysis would be possible. Once the design is mature, and interface types are created, ALI V2 could then provide an array of verification checks (as specified in the interface template description).

P5: *Increase architectural artefact reusability*

Architectural artefacts tend to be tailored to particular system requirements. Accordingly, very few ADLs discuss the concept of artefact reusability across multiple systems. Yet, in real-life, it is more often than not we are faced with similar

architectural challenges that could be solved using architectural artefacts we have in existing projects.

ALI V2 supports the concept of large-grain reusability by allowing architectural artefacts to be made configurable based on selected sets of features.

For example, components in ALI V2 can be customised based on which features are selected for a particular component, and the values of these features. Similarly, connectors and interfaces can also be parameterised using feature sets. By mapping the feature set of the source domain (where the component is taken from) and the feature set of the destination domain (where the component is going to be deployed), the component can adapt to the new environment (further details in Section 6.4.6).

P6: *Multiple architectural views*

As systems increase in size and complexity, and as more and more stakeholders take interest in system development (product managers, architects, end-users 'in the loop', etc.), the information captured within an architecture description is expanding. Accordingly, in order to minimise information overload, and sacrifice abstraction for completeness, the need for multiple architectural views catering for different stakeholders is becoming an important feature of an ADL.

ALI V2 is designed with the concept of multiple architectural views, where each view corresponds to a stakeholder (or stakeholder group) and addresses a different set of concerns (see Section 6.4 for textual and Section 6.5 for graphical descriptions).

6.3 Conceptual Model

A *conceptual model* is a high-level description of how a system is organized and operates (Johnson and Henderson, 2002). The aim of a conceptual model is to express the meaning of terms and concepts used by domain experts and to find the correct relationships between different concepts. Several notations (Booch, Rumbaugh and Jacobson, 2005; Halpin, 2010; Halpin and Morgan, 2008; Rumbaugh *et al.*, 1991) exist that are used to describe the conceptual model. Among those, UML (*Unified Modelling Language*) (Booch, Rumbaugh and Jacobson, 2005) is the most commonly used and comprehensive notation (ISO/IEC 19501).

Figure 16 shows the conceptual model of ALI V2. The *reference architecture* describes the overall system description. The reference architecture is made up of *arrangements* and *viewpoints*. Arrangements represent the structural (static) description of the system and are composed of *components* and *connectors*, which communicate through *interfaces*.

Viewpoints are sets of *transaction domains* that pertain to a common concern (e.g. car ignition system). Transaction domains represent the behavioural (dynamic) aspects of the system, and are composed of sets of *transactions* that together serve a particular system feature (e.g. user/key validation). Transactions are expressed in terms of sets of *events* that achieve a system functionality (e.g. key authorisation). And events are the basic communication mechanism between components (e.g. key code update event).

Conditions are parameters that represent external (to the system) environmental aspects that could impact the system behaviour. The architecture description is parameterised using these conditions, which can be either true or false. Different combinations of conditions and their values, called *scenarios*, can be used to test and adapt the behaviour of the system to various contexts.

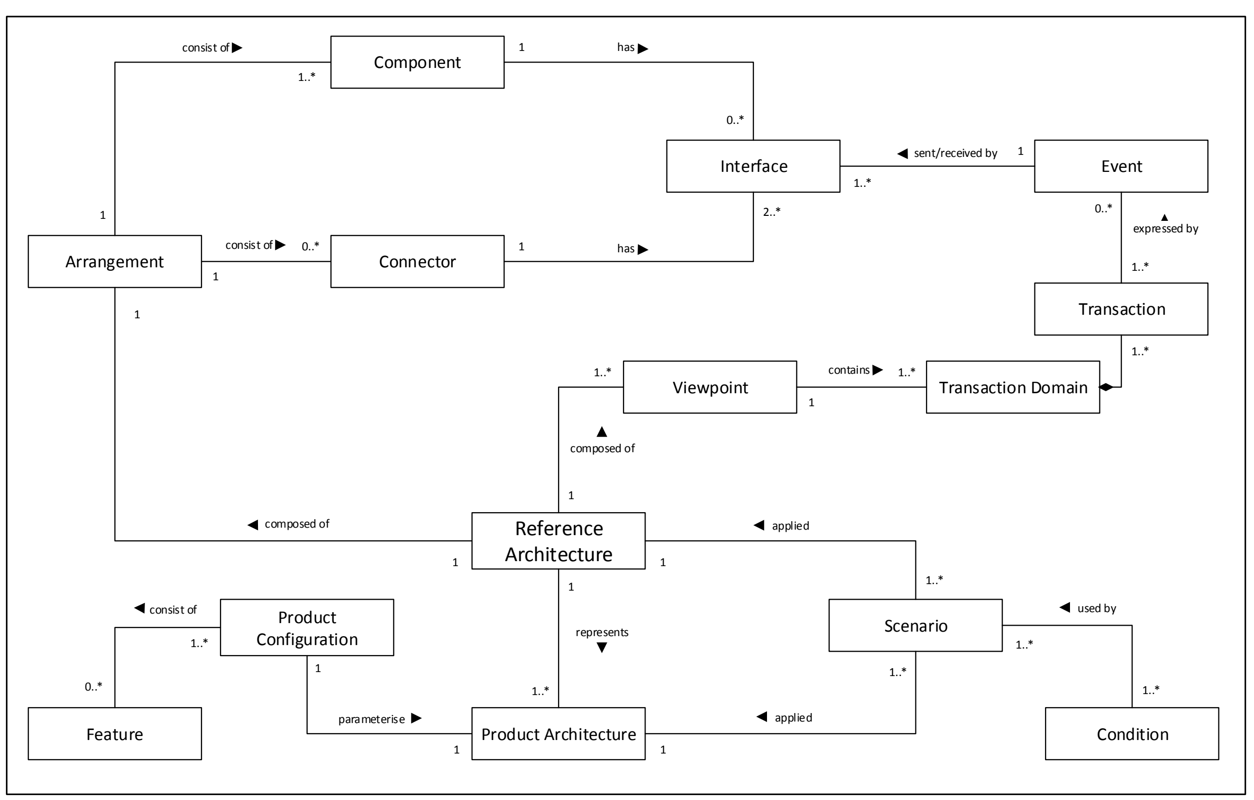


Figure 16: ALI V2 conceptual model

Finally, a *product architecture* can be derived from a reference architecture using a *product configuration*. Product configurations represent desired *features*, and their values, for a specific product in a product line.

In the following section, the details of these constructs, along with the notation used to describe them, are discussed.

6.4 Textual Constructs and Notations

The ALI V2 textual notation is designed based on the principles defined in Section 6.2. The textual notation is made up of 14 main constructs:

1. ***meta types*****:** which provide an extensible mechanism for capturing architectural meta-information
2. ***features*****:** where the system features are catalogued
3. ***interface templates****:* which specifies a dedicated notation for creating categories of interface types
4. ***interface types***: where architectural interfaces are defined
5. ***connector types***: where architectural connectors are defined
6. ***component types***: where architectural components are defined
7. ***pattern templates***: where reusable architectural design patterns are defined
8. ***product configurations***: which provides the feature combinations that characterise individual products
9. ***events****:* where the events that flows within a system are defined
10. ***conditions****:* where the system behavioural conditions (architecture parameterisation) are catalogued

11. ***scenarios:*** where behavioural scenarios are defined (sets of conditions to represent a runtime scenario)
12. ***transaction domain:*** which provides the behavioural interactions within a particular system domain
13. ***viewpoints:*** where different behavioural viewpoints are defined
14. ***system*****:** where the overall system architecture is described

These are discussed in the following sections.

6.4.1 Meta Types

The *meta types* section provides a formal syntax for capturing meta-information of the architectural element (e.g. components, connectors, etc.). A meta type is defined by the information it encompasses. The information is stored in fields, where each field has a name (*tag*) and a data type (text, number, etc.). The following example defines a meta type called `MetaType1`:

```
meta type MetaType1 {
   tag creatorID, description: text;
   tag cost, version: number;
   tag edited*: date;
  }
```

In this example, "meta type" is a keyword which is used to start a meta type definition. `MetaType1` is the name of the meta type being specified. Each meta type contains a set of tags each of which can be either textual, numeric, date, enumeration or character. Five tags are defined in the above example: two textual, two numeric and one date. Asterisk "*" on one of the date tag `edited` indicates that it is an optional tag.

Once meta types are specified, *meta objects* conforming to these types can then be created and attached to architectural elements throughout the architectural description.

These meta objects provide an area for appending additional information related to these elements. Below is an example of a meta object that conforms to the meta type defined in the example above.

```
meta: MetaType1 {
     creatorID: "Martin005";
     cost: 5,000;
     version: 1.3;
     edited: 01-01-2016;
     description: "A GUI component ...";
 }
```

A meta object could also be a combination of more than one meta type. To enhance the language flexibility, it is also possible to create meta objects that do not conform to any meta type. However, little automated analysis can be performed on such informally described data. The reasoning behind this is to allow architects to sketch what they initially think could be relevant meta-information, then, once confirmed, they create the appropriate meta types to ensure conformance.

The formal specification of meta information allows for easier CASE tool development to harness these meta objects and conduct automated analysis (e.g. cost/benefit analysis, project timing/scheduling, etc. depending on what type of meta information is available). Other meta information could include: design decisions, component compatibility, etc. which when extracted and formatted using proper CASE tools, allow automated architecture documentation to be achieved on-the-fly.

In general, it is expected that the meta types will be created once and used repeatedly across the different systems developed by the same enterprise. In order to make sure that critical information is always provided within an architecture description, a project management team (or any other stakeholder) may first identify the standard set of information required (tags), and then provide it to architects to conform to. The flexibility

in the syntax also allows the architects to augment this information with fields (tags) that they may want to use internally within the architecture team.

6.4.2 Features

The feature description notation provides a catalogue of the system *features* (mandatory, optional or alternative) used within the system. The feature definition comprises of:

- *alternative names*: In many cases, different teams within the development process address a feature with different names. This sub-section of the feature definition keeps track of the different names (if any) that are used to address the same feature (within the different design and development teams involved in the project). This property will keep track of the system features and alleviate redundancy.
- *parameters*: A feature can carry different types of parameters -textual, numerical and boolean. Though, not all features would be parameterised.

Below is an example of how features are defined in ALI V2:

```
features {
   FeatureA: {
      alternative names: {
          Designer.AName1, Developer.AName2, Evaluator.AName3;
      }
      parameters: {
           {Parameter1 = text,
            Parameter2 = number};
      }
   }
   FeatureB: {…}
     // etc.
}
```

In the example above, `FeatureA` was defined showing that it is referred to as `AName1` by the design team, `AName2` by the development team, and `AName3` by the evaluation team. The feature encompasses two parameters, one textual and one numeric.

In ALI V2, system features are defined in a stand-alone catalogue as shown above. The catalogue serves as an adapter between any feature modelling technique used and the architecture description, making ALI V2 independent of any particular feature modelling technique.

6.4.3 Interface Templates

The *interface template* notation provides a framework that allows the description of multiple interface type categories within a system description. The idea behind this is to create a set of common interface templates (e.g. WSDL, RMI, etc.) needed within an application domain once, and reuse them in different projects. These interface templates can be used as a specification in defining the *interface types* of the system, either explicitly (as explained in the next section) or in the *component type* definition (Section 6.4.6). This template specification can also be reused outside the defined system depending upon the design requirement as per principle P5 in Section 6.2.

The interface template definition is divided into three main sections:

- *provider syntax definition*: where the syntax of the provider interface is specified using a subset of the JavaCC (Java Compiler Compiler tm (JavaCC tm)) notation. JavaCC (Java Compiler Compiler) is an open source notation that allows the definition of grammars using EBNF style syntax (Scowen, 1993).
- *consumer syntax definition*: where the syntax of the consumer interface is specified using a subset of the JavaCC notation.

- *constraints*: where the interface connectivity constraints are specified. These include:

  - *Should match*: here the terms (identified in the below syntax definition sections using the JavaCC notation) that should match between two interfaces to be considered compatible (allowed to bind) are identified.

  - *Binding*: comprises of three different fields: 1) *multiple* -a Boolean value that states whether multiple binding is allowed on this interface; 2) *data size* -range of the data that can pass through this interface by providing the maximum and minimum values; and 3) *max connections* – maximum number of simultaneous connections allowed on the interface.

  - *Factory*: This is a Boolean value that states whether the interface is a factory or not. A factory interface means that when a connection request is received on this interface, a new instance is created to handle that particular request while the main interface continues to listen to new incoming requests. Example: server socket interfaces in java are factories. On the other hand, C++ sockets do not support factory functionality by default.

  - *Persistent*: This is a Boolean value that indicates a persistent interface (the internal data of the interface component is kept unchanged after the current connection has ended) when set to true and indicates a transient interface (internal data is reset to initial values when the current connection is terminated) when set to false.

An interface template description begins with the keyword *"interface template"* followed by the interface template name such as `MethodInterfac` in the example below:

```
interface template MethodInterface {
  provider syntax definition: {
    "Provider"":"
     "{"
        {"function" <FUNCTION_NAME>
           "{"
                "impLanguage" ":" <LANGUAGE_NAME> ";"
                "invocation" ":" <INVOCATION> ";"
                "parameterlist" ":" "("[ <PARAMETER_TYPE> {","
                                    <PARAMETER_TYPE>}] ")" ";"
                "return_type" ":" <RETURN_TYPE> ";"
            "}" }
      "}"
     }
   consumer syntax definition: {

     "Consumer"":"
        "{"
           "Call" ":" <INVOCATION> "("[<PARAMETER_TYPE> {","
                                        <PARAMETER_TYPE>}] ")" ";"
        "}"
      }
   constraints: {
       should match: {INVOCATION_NAME = .INVOCATION_NAME,
                                          PARAMETER_TYPE}
       binding: {
                 "multiple": true;
                 "data_size": [min, max];
                 "max_connections": 5;
                }
       factory: false;
       persistent: false;

      }
  }
```

For further details about the notation used for specifying the interface template syntax, please refer to JavaCC (Java Compiler Compiler tm (JavaCC tm)).

It is important to clarify here that the interface template definition is not meant to be read by humans, but rather created once and then read by CASE tools that would verify the interface descriptions and connections made throughout the architecture definition.

6.4.4 Interface Types

The *interface type* notation provides a set of pre-defined interface types that are created in conformance to the definition of an *interface template*, described in the previous section. Interface types can be (re)used in the design of architectural elements (components and connectors) throughout the system description (design principle P5).

An interface type definition begins with the keyword *"*interface type*"* as in the example below:

```
interface type {
  InterfaceType1: MethodInterface {
        Provider: {
         function Addition
                {
                 impLanguage: Java;
                 invocation: add;
                 parameterlist: (int);
                 return_type: void;
                }
         function Subtraction {…}
         function Multiplication {…}
        }
        Consumer: {
           Call: getValue (long_int);
        }
    }

 InterfaceType2: MethodInterface {
        Provider: {
         function Average
                {
```

```
                    impLanguage: Java;
                    invocation: average;
                    parameterlist: (int);
                    return_type: void;
                  }
           }
          Consumer: {//nothing consumed}
       }
    // etc.
  }
```

Each interface type is defined by a unique name followed by the *interface template* name, which it conforms to. In the example above, `InterfaceType1` performs basic mathematical operations based on the value it consumed and `InterfaceType2` provides the average calculated value by using the average formula strategy. They all conform to the interface template `MethodInterface` defined in the previous section. We can also define other interface types that conforms to other *interface templates* such as WSDL, RMI, etc.

6.4.5 Connector Types

Like many other ADLs, such as ACME (Garlan, Monroe and Wile, 1997), Aesop (Garlan, Allen and Ockerbloom, 1994), CBabel (Rademaker, Braga and Sztajnberg, 2005), EAST-ADL (Cuenot *et al.*, 2010), UniCon (Shaw *et al.*, 1995), WRIGHT (Allen, Douence and Garlan, 1998) and π-ADL (Oquendo, 2004), to name a few, connectors are considered first class citizens in ALI V2.

A connector type definition begins with the keyword *"*connector type*"* followed by the connector type name and is divided into three sections.

```
connector type ConnectorType1
 {
  features: {
```

```
            Feature1: "textual description",
            Feature2: "textual description",
            Feature3: "textual description";
          }
        interfaces: {
            a1: InterfaceType4;
            a2: InterfaceType1;
            a3: InterfaceType2;
            a4: InterfaceType3;
         }
        layout: {
           connect a4 and a1;
           if (supported(Feature1 || Feature2)){
              {connect a1 to a2;
               connect a2 to a4;}
           else if (supported(Feature3)
              connect valueport3 to valueport4;}
         }
       }
```

▪ *features:* a set of optional/alternative features used to parameterise a connector type. By changing feature values, a connector can be reconfigured to be deployed in different products (based on feature availability and parameter values). The configuration is achieved using `if/else` structure and the keywords "**supported/unsupported**" to link features to the connector definition.

▪ *interfaces*: where the connector interfaces are defined along with their interface types. These resemble the input/output ports of the connector. Basically, interfaces are instances of *interface types* that are defined in accordance to *interface templates*.

▪ *layout*: The layout section describes the internal configuration (structure/arrangement) of the connector. It demonstrates how the connector interfaces are connected internally, that is, how the traffic (information) travels internally from one interface to another. This syntax introduces a high level of configurability to the connector definition which

provides better support for defining configurable product and product line architectures. Two types of configurations are allowed between connector interfaces, namely:

- *uni-directional connections (to)*: which specify that the data from one interface goes to another interface. This is done using the keywords: "**connect**" and "**to**". Example: `connect a1 to a2` in `ConnectorType1` outputs the data on the `a1` interface to the `a2` interface.

- *bi-directional connection (and)*: which specify that the data can travel in both directions between two interfaces. This is done using the keywords: "**connect**" and "**and**". Example: `connect a4 and a1` in `ConnectorType1` outputs the data on the `a4` interface to the `a1` interface and vice versa.

Additionally, the keyword "**all**" can be used to connect a connector interface to all other interfaces of the connector using a bi-directional or unidirectional communication. For example, "`connect all to all`" can be used to create bi-directional connections among all ports. We can also have "`connect a1 to all`" which makes the input on interface `a1` available as output on all other interfaces of the connector.

Lastly, *meta objects* can be attached to connector types by simply defining the meta object (as explained in Section 6.4.1) inside the connector type definition (anywhere between the start and end brackets).

6.4.6 Component Types

The *component type* definition is divided into three main sections:

- *features:* a set of optional/alternative features that make up a component type. The purpose and definition of this section is exactly similar to the concept of features defined in the *connector type* (see Section 6.4.5). That is, it provides the capability to

reuse components in multiple products and systems by varying feature values (*product configurations*).

- *interfaces*: which specify the different interfaces used by the component. The interfaces section is divided into two sections, *definition* where new interfaces can be created from scratch; and *implements* where already defined interfaces can be reused (interfaces implemented here are instances of *interface types*).
- *sub-system*: where the internal structure (sub-system) of the component is described. The sub-system section is divided into three sections:
  - *components*: where the different sub-components included within the component are defined.
  - *connectors*: where the different connectors used in connecting sub-components are defined.
  - *arrangement*: where the way in which sub-components are connected is described. To allow flexibility, ALI V2 provides three different methods that can be used to connect components:
    a. *Using connectors*: where a connector mediates the connection between two or more components. This is done by using the keyword "**connect**" (e.g. `connect Component.Interface1 with Connector.Interface1`).
    b. *Direct binding*: where component interfaces are bound directly without the use of a connector. This is done by using the keyword "**bind**" (e.g. `bind Component1.Interface1 with Component2.Interface1`).
    c. *Using patterns*: where predefined connection patterns can be used to connect a set of components according to a selected architectural pattern (see Section 6.4.7).

Component type description begins with the keyword *"component type"* followed by the component type name `ComponentType1` as shown in the example below.

```
component type ComponentType1
  {
   meta: MetaType1 {…}
   features: {
      FeatureA: "textual description",
      FeatureB: "textual description",
      FeatureC: "textual description";
    }
   interfaces:   {
    definition:  {
      interfaceA: InterfaceType1 {
       Provider: {
            function myAddFunction
                     {
             impLanguage: JAVA;
             invocation: add;
             parameterlist: ( int );
             return_type: void;
                     }// etc.
        }
       Consumer: { }
        //no consumed functions specified
       }
     if(supported(FeatureC))
     {
      interface: InterfaceType2 {…}
     }
    }
    implements:{
     interfaceA1: InterfaceType1;
     interfaceB1: InterfaceType2;
    }
 } //end of interfaces
 sub-system: {
  components {
   comp1<FeatureA, false, true>: ComponentType1;
   if( supported(FeatureC))
     comp4<true, true>: ComponentType2;
```

```
 else
   comp4<false, true>: ComponentType2;
   //etc.
 }
 connectors {
  connA<FeatureA, false, false>: ConnectorType1;
  connB<FeatureB, true>, connC<false, true>: ConnectorType2;
  if( supported(FeatureB) )
    connD<true, FeatureC>: ConnectorType2;
  // etc.
}
 arrangement {
      // 1 - connecting components using connectors
      connect comp1.interfaceA with connA.a2;
      connect comp4.interfaceA1 with connA.a4;
      // 2 - connecting components without connectors
      bind comp4.interfaceB1 with comp1.interfaceA;

         // 3 - connecting components using defined patterns
         if( supported(Feature_C) ) {
            Client_Server(ServerComp1.interfaceB1,
                          [ClientComp1.interfaceB1,
                           ClientComp2.interfaceB1,
                           ClientComp3.interfaceB]);
            }
          else {
               PipesAndFilters(…);
            }
        /* connecting the component interface(s) with sub-
           components' */
           bind comp1.interfaceA with my.InterfaceA1;
     }
}
```

The example above shows how the component configuration can change depending on what features are supported. The keyword "**my**" is used to reference the component's own interfaces as opposed to sub-component interfaces (similar to the use of "**this**" in some programming languages).

6.4.7 Pattern Templates

The *pattern template* notation in ALI V2 allows the definition and use of architectural patterns. They are first defined and then (re)used throughout the architecture by calling the pattern template needed. The pattern template definition takes the interfaces to be connected as an argument and is defined in a similar way to the definition of functions (methods) in programming languages. A pattern template definition comprises of:

- *pattern name*: a unique pattern name.
- *arguments*: a set of interfaces to be connected. Single interface and/or arrays of interfaces can be passed as arguments. The minimum and maximum number of interfaces passed can be specified as arguments for arrays of interfaces.
- *definition*: the description of how the interfaces are to be connected (the pattern). The syntactical notation used for defining patterns is very simple and provides support for:
    - *connecting interfaces*: uses syntax similar to that used in the connections section of the connector type definition (discussed in Section 6.4.5).
    - *defining loops*: to allow for connecting arrays of interfaces. The syntax used here is similar to the syntax used in most programming languages for creating *for* loops. The point to be noted is that the arrays of interfaces start at index 1 and not at 0 (like in most programming languages).

Below is an example that defines `Client_Server` pattern:

```
pattern templates:
{
 Client_Server (server : MethodInterface,
                clients [1…N] : MethodInterface)
  {
   for( i = 1 ; i <= N ; i++ )
      connect clients[i] and server;}
  }
```

In this example, the `Client_Server` pattern takes as an argument one interface `server` of template `MethodInterface`, and an array of interfaces called `clients` (with `[1..N]` meaning at least one `client` interface) of template `MethodInterface`. The pattern is defined as: for all N `clients` interfaces, create a bi-directional connection with the `server` interface (see Section 6.1.5 on the use of the keywords: "**connect**", "**and**", and "**to**" for connecting interfaces).

An example of how to invoke the `Client_Server` pattern template to connect a number of component interfaces can be seen in the example in Section 6.4.6.

6.4.8 Product Configurations

A *product configuration* is a set of features, along with their values, representing a particular product configuration (this is also called *product feature set* in Software Product Line Engineering). Product configurations can be used to generate specific products from the parameterised reference architecture. Below is an example how to define each product:

```
product configurations {
   Product1: {
               FeatureA = false;
               FeatureB = true;
               FeatureC {x = 3, y = t};
              }
   Product2: {…}
    // etc.
 }
```

6.4.9 Events

Events are abstractions of actions performed during the execution of the system, such as a message transmission from one component to another. In ALI V2, events are defined using a unique name, along with the interface templates they travel to and from. Below is an example how to define events:

```
events {
   EventName1: <sourceInterfaceTemplate,
                destinationInterfaceTemplate>;
  EventName2: …
 }
```

It is also possible for events to travel from, and to, more than one interface template. In this case, interface templates are listed within parentheses and separated by commas as shown in the example below.

```
     …
EventName3: <(sourceInterfaceTemplate1, sourceInterfaceTemplate2),
 (destinationInterfaceTemplate1, destinationationInterfaceTemplate2)>;
     …
```

6.4.10 Conditions

Conditions are used to parameterise the system description to make it adapt to certain environmental conditions. Every set of conditions (a scenario) can then be used to simulate a certain environmental situation (e.g. failure, market changes, etc.). These can be used to test the way the architecture definition can adapt to different operational changes (design principle P2). Conditions are defined with unique name along with a simple textual description. Below is an example definition of three different conditions.

```
conditions {
     Condition1: "definition";
     Condition2: "definition";
     Condition3: "definition";}
```

6.4.11 Scenarios

Scenarios are basically collections of different conditions, along with their values, which together can simulate a certain operational scenario. A scenario description includes a textual description (what the scenario is) along with a list of conditions affected and their values. Below is an example scenario description.

```
scenarios {
    Scenario1: {
                Description: "textual description";
                Parameterisation: {
                                Condition1 = false;
                                Condition2 = false;
                                Condition3 = true;
                                }
    Scenario2: {…}
     // etc.
  }
```

In the above example, scenario `Scenario1` encapsulates three *conditions* (defined in the previous section) in which two are false and one is false. Scenarios can be very useful when evaluating different architectural configurations.

6.4.12 Transaction Domains

Transaction domains represent the behavioural aspects of the system. Each transaction domain comprises a set of components and connectors within a system that work together to achieve some system functionalities (e.g. portfolio evaluation). Within a transaction domain, various transactions are defined, each describing a particular system transaction (e.g. valuation processing, MTM valuation, etc., as demonstrated in Chapter 7). Transactions are defined in terms of event flows.

The *transaction domain* definition is divided into two main sections, *contents* which lists the components and connectors included in a transaction domain; and *transactions* which describes the transactions encompassed in the transaction domain. Each transaction is defined in terms of the events that flow to achieve the transaction, and the description of the event flow (*interactions*). Table 8 below summarises the textual notation used in defining *interactions* within a transaction.

Notation	Meaning
Component.Interface	Component name with interface name
* Component	External component (or system)
Event	Event name
Event/Connector	Event traveling on Connector
TRANSACTION	Transaction name
sends receives from to	Keywords describing the path of an event
if/else	Alternation (OR Fork)
|	Alternation (OR Join)
,	Concurrency (AND)
[…]	Multiple simultaneous interactions (concurrency)
(…)	Grouping of events
;	Interaction termination

Table 8: ALI V2 transaction domain textual notation

Below is an example of a transaction domain that represents the practical implementation of all the notations listed in Table 8:

```
transaction domain TransactionDomain1
  {
    meta: MetaType2 {}
    contents:
     {
         /*components and connectors involved in this transaction
           domain*/
         components: {Comp1, Comp2, Comp3, Comp4, Comp5, Comp6,
                      *Comp7, Comp8}
         connectors: {ConnA, ConnB, ConnC, ConnD, ConnE, ConnF}
     }
    transactions:
     {
         TRANSACTIONNAME1:
          {
            events: {E1, E2, E3, E8}
            interactions:
            {
              Comp1.A sends E1/ConnA to Comp2.C;
              Comp2.C sends E8/ConnA to Comp1.A;
              [Comp1.C sends E2, Comp1.D sends E3/ConnC];
            }
          }
         TRANSACTIONNAME2:
         {
           events: {E3, E5, E6, E9, E10, E11, E12}
           interactions:
            {
              Comp4.B receives E3/ConnC;
              if (supported(featureA)&& (Condition1))
                 Comp4.C sends E6 to *Comp7;
              else
                 {Comp4.A sends E5/ConnB to Comp5.C;
                   Comp5.B sends E11 to Comp8.D;}
              [Comp6.C receives E9/ConnE from *Comp7 |
               Comp6.A receives E10 from Comp8.B];
              Comp6.D sends E12/ConnD;
            }
```

```
 }
TRANSACTIONNAME3:
{
  events: {E2, E4, E5, E7, E8, E10, E11}
  interactions:
   {
    Comp3.A receives E2;
    [Comp3.D sends E4/ConnD to Comp5.B,
     //via same interface of components
     Comp3.B sends [E5, E11] to Comp6.C,
     Comp3.D sends E3/ConnC to TRANSACTIONNAME2,
     Comp3.A sends E7 to Comp4.B];
    Comp4.A sends E10/ConnD to Comp2.D;
     [Comp8.C sends E7/ConnF,
     TRANSACTIONNAME2 sends E12/ConnD,
     Comp6.C sends E8];
   }
}
TRANSACTIONNAME:
{
  events: {E2, E3, E7, E8, E12}
  interactions:
   {
     [TRANSACTIONNAME1 sends E2 to TRANSACTIONAME3,
      TRANSACTIONNAME1 sends E3/ConnC to
      TRANSACTIONAME2];
     [(Comp1.A receives E7/ConnF from TRANSACTIONNAME3,
        Comp1.C receives E8 from TRANSACTIONNAME3,
        Comp1.B receives E12/ConnD from
        TRANSACTIONNAME3),
       Comp1.B receives E12/ConnD from TRANSACTIONNAME2];
   }// end of interaction
} // end of transaction
} // end of transactions section
} //end of transaction domain
```

Given the way interaction domains represent event flows, graphical representations (discussed in Section 6.5) tend to work much better in expressing complex flows.

6.4.13 Viewpoints

Viewpoints in ALI V2 represent collections of interaction domains that relate to a particular stakeholder. A viewpoint definition includes: a unique name, description and a list of related transaction domains. Below is an example how viewpoint can be defined:

```
Viewpoints {
    Viewpoint1: {
                  Description: "textual description";
                  Transaction Domain: {TransactionDomain1,
                                       TransactionDomain3;}
                }
    Viewpoint2: {…}
     // etc.
  }
```

6.4.14 System

Finally, the *system* notation describes the overall product (or product line) architecture. It uses very similar notation to the component description section with some minor changes, such as the usage of asterisk "*" to link to external (to the system) components or systems. Additionally, a system description includes a listing of viewpoints. Below is an example system description.

```
system {
 components {
  comp1<SomeFeature, true, false>,
  comp2<SomeFeature, true, true>: ComponentType1;
  comp3<SomeFeature, SomeFeature, true, true>: ComponentType2;
  if (supported(Feature_D))
     comp4<true, true>: ComponentType3;
   else
      comp4<false, true>: ComponentType3;
 //etc.
 }
 connectors {
    connA<false, SomeFeature, true>: ConnectorType1;
```

```
      // etc.
    }
    arrangement {
        … //similar to component type arrangement
                    bind comp1.interface with *externalsystem;
     }
    viewpoints {
      Viewpoint1, Viewpoint2;
     }
} // end of system
```

6.5 Graphical Constructs and Notations

Many of the existing ADLs such as AADL (Feiler, Gluch and Hudak, 2006), ACME (Garlan, Monroe and Wile, 1997), Aesop (Garlan, Allen and Ockerbloom, 1994), MontiArc^{HV} (Haber *et al.*, 2011), Darwin (Magee and Kramer, 1996), Koala (Ommering *et al.*, 2000), UniCon (Shaw *et al.*, 1995) and π-ADL (Oquendo, 2004), provide both textual and graphical notations, though none provide a behavioural graphical notation. Yet, in some cases, the need for such graphical behavioural representation was argued, e.g. using Use Case Maps (UCMs) with ADLARS (Bashroush *et al.*, 2005; Brown *et al.*, 2006). ALI V2 provides graphical notations for structural and behavioural aspects of the systems. The following sections discuss ALI V2's graphical notation.

6.5.1 Structural Notation

To maintain the theme of flexibility, ALI V2 provides a flexible visual notation for its structural description. Table 9 illustrates the meaning of the symbols used to specify architectural structures in ALI V2. There is also the flexibility to extend the notation used to represent components and introduce other graphical objects (e.g. a cylinder to represent

a database component) that architects identify with, and already use, in certain application domains.

Symbol	Name (Meaning)
	Component
	External Component (or System)
	Interfaces (different shapes represent different interface templates)
	Connectors representing different interface templates
	Direct Binding (no connector)
	Transaction
	Transaction Domain

Table 9: ALI V2 graphical structural notation

Figure 17 represents the structural description of the whole system that clearly demonstrates the transaction domain `TransactionDomain1` (defined earlier in Section 6.4.12 in textual format).

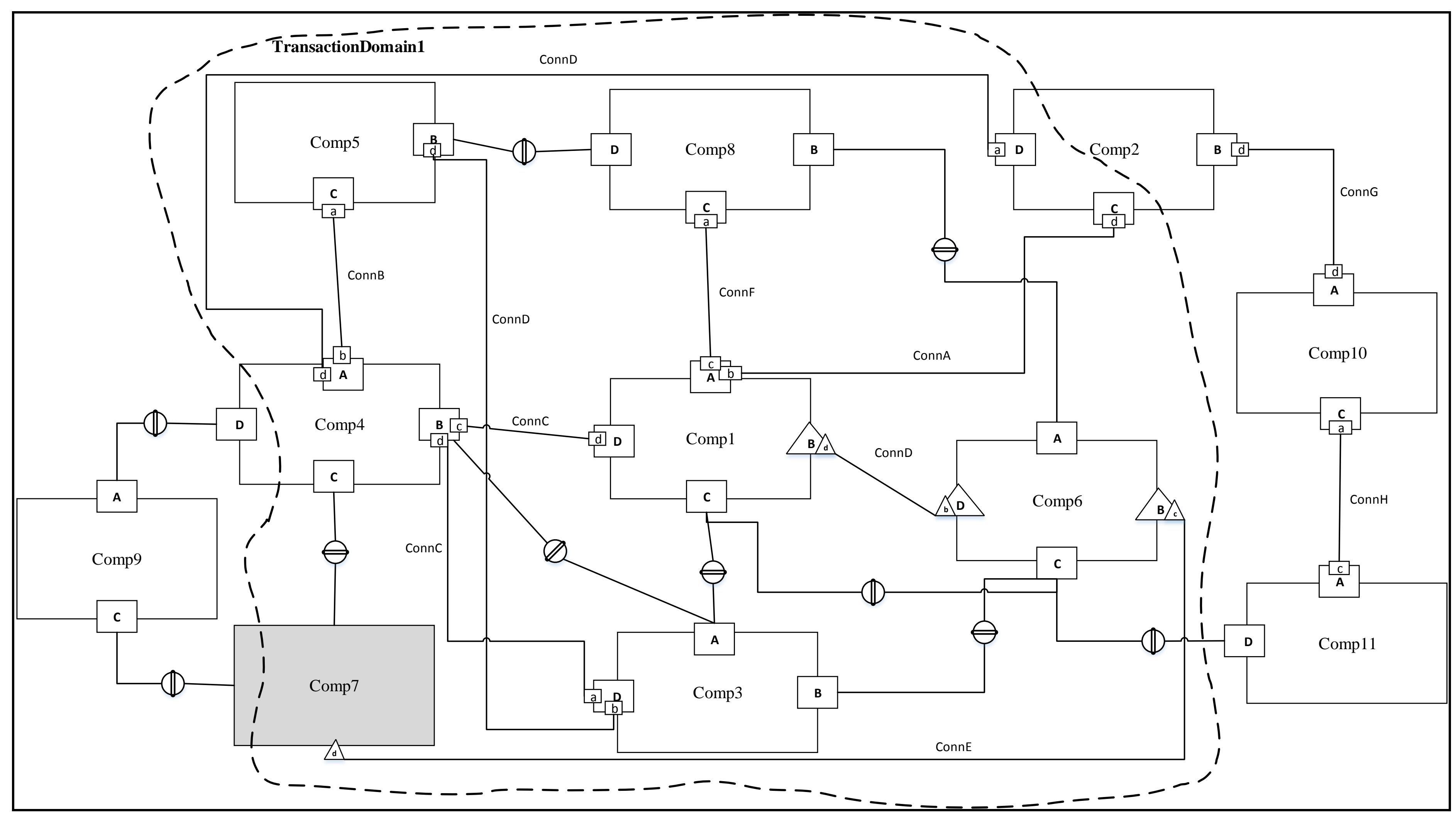


Figure 17: Graphical structural representation for a system

6.5.2 Behavioural Notation

6.5.2.1 Event Traces

In ALI V2, *event traces* constitute the graphical representation of *transactions*, described textually in Section 6.4.12. Table 10 below provides the detailed description of the symbols used to design event traces. Some of the symbols used are adopted from the UML Activity diagram (Booch, Rumbaugh and Jacobson, 2005), with added notation to represent concurrency (based on some extended concepts from Petri Nets (Murata, 1989)).

Symbol	Name	Meaning
●	START	A node that starts the interaction in an event trace by a component that invokes an event.
◉	END	A node that stops the interaction of all the transactions in an event trace.
⊗	FINAL	A node that terminates the interaction of the transaction.
*EventName/Cr** →	Event Flow	The direction of an event flow from one component to another component, specifying the event name and the connector* being traversed.
	AND Fork	A source component sending two or more concurrent events to destination components.
	AND Join	A destination component receives two or more concurrent events from source components. This blocks until all events are received before progressing.

Symbol	Name	Meaning
	OR Fork	A source component sends one or more events to destination components. Selection of the destination components can be linked to system *conditions* and *features.*
	OR Join	A destination component receives any of the events from any one of the source components (non-blocking) as soon as it arrives (without waiting for all expected events).
	Component	A component within the system that sends/receives events.
	External Component/ System	A system (or component) outside the system that communicates with our system.
	Transaction	Transaction is a package containing a set of interactions. It can be nested wherever required in another transaction.

* means optional i.e. if connection is made using connectors (see Section 6.4.6)

Table 10: ALI V2 event traces notation

The notation comparison between ALI V2 event traces, UML Activity Diagram, UCM and Petri Nets can be found in Appendix B.

Figure 18 shows an example of the graphical behavioural representation of the transaction domain `TransactionDomain1` (this maps to the textual representation provided in Section 6.4.12). The example demonstrates the *transactions* that occur in `TransactionDomain1` along with the interactions that take place in the `TRANSACTIONNAME1`, `TRANSACTIONNAME2` and `TRANSACTIONNAME3` transactions.

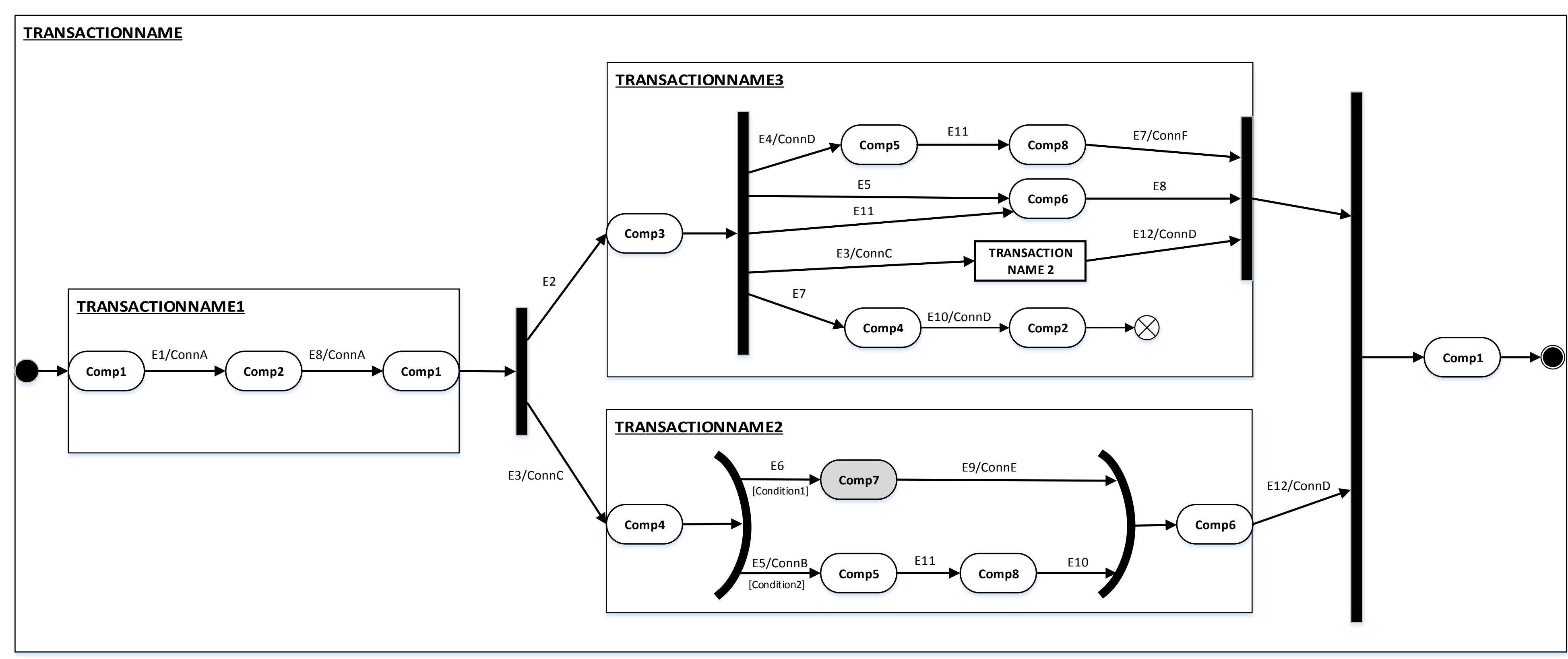


Figure 18: Graphical behavioural representation of transaction domain `TransactionDomain1`

6.5.2.2 Component Interaction

This section provides the graphical notation used to describe the interactions of an individual component. While event traces model the complete event flow path, component interactions focus on modelling the interactions of a particular component (focus on components rather than events). For this, UML Sequence diagrams (Booch, Rumbaugh and Jacobson, 2005) are used to model component interactions. Sequence diagrams are known to many architects and are comprehensive to model handshakes, timing, etc.

Figure 19 shows the component interaction diagram for the component `Comp1` in the transaction domain `TransactionDomain1` (defined textually in Section 6.4.12, with interactions described in Figure 18).

The squares at the top of the sequence diagram represent component interfaces. White squares represent the interfaces of the component being model (in this case `Comp1`), and greyed squares represent external interfaces (of other components `Comp1` is communicating with).

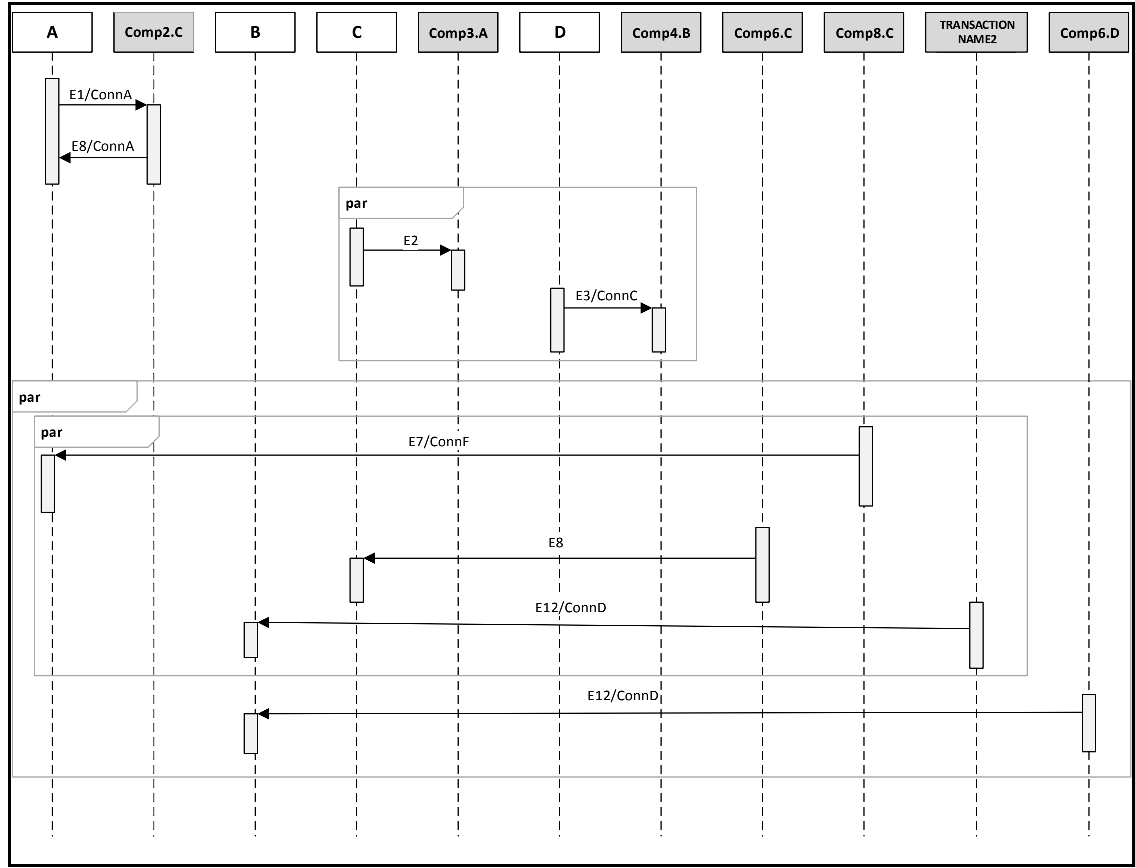


Figure 19: Component `Comp1` interactions in transaction domain `TransactionDomain1`

6.6 Semantics

In this section, the ALI V2 notation semantics (Harel and Rumpe, 2004) are discussed. It starts by discussing the semantics of the structural notation, then the behavioural notation. It is worth noting that proofs of correctness and completeness of the semantics are beyond the scope of this research work.

The following notation convention is used in the subsequent two sections: $\boldsymbol{i}$ for interface, **Ct** for component, **Cr** for connector and **e** for event. The name of each element (where applicable) is indicated in the subscript. For example, $\boldsymbol{i}_{\mathbf{A}}$ denotes interface **A**.

6.6.1 Structural Semantics

In this section, the semantics of the structural notation, namely covering: components, connectors and interfaces are discussed.

For simplicity, a component is a finite set of n interfaces:

$$i \in \mathrm{Ct} = \{1i_{A,} 2i_{B}, \ldots, \mathrm{n}i_{Z}\} \quad (6.1.1)$$

Different combinations of interfaces in a component can occur depending on the *feature(s)* supported in its specification. All possible occurrences can be defined as:

$$\mathrm{Ct} = \mathcal{P}(i) \quad (6.1.2)$$

The notation $\mathcal{P}(i)$ refers to the power set of the set interfaces of a component. It also includes the null/empty set (∅) which relates 0…* relationship between the interface and the component as explained in the conceptual model section (see Section 6.3 and Figure 16).

Similarly, a connector is a finite set of n interfaces:

$$i \in \mathrm{Ct} = \{2i_{B}, \ldots, \mathrm{n}i_{Z} \mid i \geq 2\} \quad (6.1.3)$$

The number of interfaces in a connector must be at least two to form a connection between two components.

The following are the naming rules:

Rule 1: The names of all the components must be unique within a system

$$\mathrm{Ct_A} \cap \mathrm{Ct_B} = \emptyset,\ \mathrm{A} \neq \mathrm{B} \quad (6.1.4)$$

Rule 2: The names of all the connectors must be unique within a system

$$Cr_A \cap Cr_B = \emptyset,\ A \neq B \quad (6.1.5)$$

Rule 3: The names of all the interfaces of a component must be unique

$$\forall i : Ct = \{\forall x \in Ct, \forall y \in Ct \mid x \neq y\} \quad (6.1.6)$$

Rule 4: The names of all the interfaces of a connector must be unique

$$\forall i \in Cr = \{\forall x \in Cr, \forall y \in Cr \mid x \neq y, |x| \geq 2\} \quad (6.1.7)$$

6.6.2 Behavioural Semantics

In this section, the semantics of the behavioural notation of ALI V2 by translating the notation into formal specification theory, namely CSP (Hoare, 1985) are discussed. The main construct of the behaviour notation in ALI V2 is the process that defines the interactions of a transaction in a transaction domain.

Using the CSP process notation, where $(x{:}A \rightarrow P(x))$ [pronounced "*x* from *A* then *P* of *x*"], it transform interaction (Itn) into:

$$Itn = (e_1.Cr_A^{\dagger} : Ct_S.i_A \rightarrow Ct_r.i_B) \quad (6.2.1)$$

To recall, an interaction in ALI V2 is an event flowing via connector or via direct binding from one component to another component (see Section 6.4.6 for in-depth description). Equation 6.2.1 describes an interaction in terms of CSP as: event (e_1) via connector (Cr_A) from sender component (Ct_S) of its interface (i_A) goes to the receiver component (Ct_r) on its interface (i_B). Symbol '†' represents the optionality of the connector, that is, it will be defined if event flows via a connector. Similarly, an asterisk

‘*’ before the sender/receiver component (Ct_S/Ct_r) can be inserted without specifying the interface name if it is an external component (or system), as explained in Section 6.4.12 and Section 6.5.2.1.

Interactions can occur in different combinations with other interactions. Those combinations are: AND fork, AND join, OR fork and OR join, each combination is explained categorically in the transaction domain (see Section 6.5.2.1).

In the rest of this section, the formal semantics for each combination using CSP are elucidated.

AND Fork: Two or more interactions that occur concurrently. Considering different Ct_r and different interface of Ct_S, it is defined as:

$$\text{AND_Fork} = (e_1.Cr_A : Ct_S.i_A \rightarrow Ct_{r1}.i_C) \parallel (e_2 : Ct_S.i_B \rightarrow Ct_{r2}.i_D) \parallel \ldots \quad (6.2.2a)$$

Where ‘||’ is the CSP parallel operator which represents concurrent activity. Hence, it is not necessary that AND Fork always has different receiver components (like Ct_{r1} and Ct_{r2} as above), we could have a situation where two or more events flow to one Ct_r via the same or different interfaces as discussed in Section 6.4.12.

Considering the same Ct_r, using the same interface, an AND fork can be defined as:

$$\text{AND_Fork} = ((e_1.Cr_A : Ct_S.i_A \parallel e_2 : Ct_S.i_B \parallel \ldots) \rightarrow Ct_r.i_D) \quad (6.2.2b)$$

In addition to the above expression conditions, it can be define by considering Ct_S, using the same interface, as:

$$\text{AND_Fork} = ((e_1.Cr_A \,||\, e_2 \,||\, \ldots) : Ct_S.i_B \rightarrow Ct_r.i_A) \quad (6.2.2c)$$

AND Join: Two or more interactions that go to the Ct_r concurrently. Considering different Ct_S and different interfaces of Ct_r, it is defined as:

$$\text{AND_Join} = ((e_1.Cr_A: Ct_{S1}.i_A \rightarrow Ct_r.i_B) \wedge (e_2 : Ct_{S2}.i_B \rightarrow Ct_r.i_D) \wedge \ldots) \rightarrow (\text{WAIT} \sum ; Ct_r) \quad (6.2.3a)$$

Where '$\wedge$' is the logical AND operator, **WAIT** is a time-based CSP operator (Armstrong *et al.*, 2012), '$\sum$' is submission (union) of all the events and '**;**' means successfully followed by. Thus, '**WAIT $\sum$; Ct_r**' designates: Ct_r will not proceed with other interaction(s) until it receives all the events.

Considering the same interface of Ct_r, an AND join can be define as:

$$\text{AND_Join} = ((e_1 : Ct_{S1}.i_B \wedge e_2. Cr_C : Ct_{S2}.i_A \wedge \ldots) \rightarrow Ct_r.i_C) \rightarrow (\text{WAIT} \sum ; Ct_r) \quad (6.2.3b)$$

Moreover, the definition for the same Ct_S with its different interfaces can be defined in a similar way as above. But if we have the same Ct_S with its same interface and the same interface of Ct_r then it can define as:

$$\text{AND_Join} = ((e_1.Cr_A \wedge e_2 \wedge \ldots) : Ct_S.i_C \rightarrow Ct_r.i_D) \rightarrow (\text{WAIT} \sum ; Ct_r)$$

(6.2.3c)

OR Fork: Two or more interactions that occur alternatively in accordance to the *condition(s)* and *feature(s)* supported. Considering different Ct_r and different interface of Ct_S, it is defined as:

$$\text{OR_Fork} = (e_1.Cr_A : Ct_S.i_A \rightarrow Ct_{r1}.i_D) \square (e_2 : Ct_S.i_B \rightarrow Ct_{r2}.i_C) \square \ldots \quad (6.2.4a)$$

Where '□' is the CSP deterministic choice operator.

If the same event flows to different Ct_r depending on the *condition(s)* and *feature(s)* supported from the same interface of Ct_S then it can be define as:

$$\text{OR_Fork} = (e_1.Cr_A : Ct_S.i_A \rightarrow (Ct_{r1}.i_C \square Ct_{r2}.i_B \square \ldots)) \quad (6.2.4b)$$

Another case, when different events flow to same Ct_r to its same interface depending on the *condition(s)* and *feature(s)* supported from the same interface of Ct_S then it can be define as:

$$\text{OR_Fork} = ((e_1.Cr_A \square e_2 \square \ldots) : Ct_S.i_A \rightarrow Ct_r.i_D) \quad (6.2.4c)$$

OR Join: Two or more interactions that go to the Ct_r alternatively. Unlike AND join, Ct_r will proceed with other interaction(s) after receiving the first event from any Ct_S without waiting for all the events to occur. Considering different Ct_S and different interface of Ct_r, it is defined as:

$$OR_Join = (e_1.Cr_A : Ct_{s1}.i_A \rightarrow Ct_r.i_A) \,\square\, (e_2 : Ct_{s2}.i_A \rightarrow Ct_r.i_C) \,\square\, \ldots$$

(6.2.5a)

Considering the same interface of Ct_r, we can define it as:

$$OR_Join = ((e_1 : Ct_{s1}.i_A \,\square\, e_2.Cr_C : Ct_{s2}.i_A \,\square\, \ldots) \rightarrow Ct_r.i_B) \quad (6.2.5b)$$

Also, we can define the same Ct_S with its same interface along with the same interface of Ct_r as:

$$OR_Join = ((e_1.Cr_A \,\square\, e_2 \,\square\, \ldots) : Ct_S.i_A \rightarrow Ct_r.i_D) \quad (6.2.5c)$$

6.7 Summary and Changes to ALI Initial Version

In this chapter, the updated version of ALI, referred to as ALI V2 in this context, was discussed. The changes made were the result of experience gained through the SLR (conducted in Chapter 4), and from the detailed analysis of the existing ADLs (discussed in Chapter 3), as well as the discussions and feedback from colleagues in both industry and academia.

The design principles guiding the creation of ALI V2 architectural description were defined to address the limitations that exist in ADLs (stated in Chapter 3). The design principles demonstrate the capability to manage variability by providing flexibility while maintaining the formality to reuse the architectural elements (components, connectors, and interfaces) along with the multiple architectural views. A high-level (abstract) description of ALI V2 is presented as a conceptual model that states structural and behavioural relationships of the ALI V2 concepts.

ALI V2 supports formal specification (and corresponding verification) of structural and behavioural aspects of software architectures. This is a key activity in the architectural design phase. The constructs and notations (textual and graphical both) show how the design principles are realized in the language. More specifically, as compared to other ADLs, ALI V2 provides behavioural graphical notations with different views in parallel with its structural notation in the form of event traces for transaction domain and sequence diagrams for component interaction.

Finally, the ALI V2 notation semantics by defining the structural and behavioural aspects explicitly were presented. In structural semantics, rules set for the structural designing of ALI V2 using mathematical set theory were defined. For behavioural semantics, CSP notation is used to describe the behavioural system of ALI V2.

Table 11 summarised ALI V2 in comparison with the ALI initial version (described in the previous chapter) by considering the limitations that exist in the architectural languages (stated in Chapter 3). It clearly demonstrates the changes made in ALI V2 and how it overcomes the limitations that restrict ADL's uptake into industrial applications.

Limitation (Keyword)	**ALI Initial Version**	**ALI V2**
L1 (Variability)	Manage variable features using "if/else" statement and keyword "supported/unsupported"	Manage variable features using "if/else" statement and keyword "supported/unsupported" Manage variable behavioural conditions using "if/else" statement
L2 (Traceability)	Via features	Via features and conditions
L3 (Dependency)	Support for flexible interface type and component type representation Not architecture style specific	Same with minor refinement
L4 (Restrictive Syntax)	Flexible syntax to design structural architectural elements	Flexible syntax to design structural (with enhancement) and behavioural architectural elements both
L5 (Reusability)	Limited to interface type definition (called as interface template in ALI V2)	Extended to the connector type and component type definitions via features description in them
L6 (Information Overload)	Textual architectural view-structural description only	Textual and graphical architectural views that supports different abstraction level (depends on user requirement and/or system complexity)

Limitation (Keyword)	**ALI Initial Version**	**ALI V2**
L7 (Behavioural)	None	Explicit constructs (textual): events, conditions, scenarios, transaction domain, and viewpoint Explicit constructs (graphical): event traces and component interactions via sequence diagram

Table 11: ALI initial version Vs ALI V2

A complete BNF for the textual architectural description of ALI V2 can be found in Appendix C.

To gain experience with ALI V2 and fine tune the language, the different case studies was needed to demonstrate the broader scope of the language and to identify the limitations in it. The next part of this thesis presents the two case studies (belonging to the different application domains) that use ALI V2 to design an Asset Management System (AMS), and the Wheel Brake System (WBS).

Part IV

CASE STUDIES

Chapter Seven

7

Case Study: Asset Management System

"Knowing is not enough; we must apply. Being willing is not enough; we must do."

-- Leonardo da Vinci

7.1 Introduction

In the previous chapter, the new version of ALI (referred to as ALI V2) was defined; this version was designed by taking into account the current limitations which restrict the uptake of ADLs (particularly those developed within academia) in practical industrial systems. Although the framework of the language has been defined, no experience has been gained regarding its application to real problems. In light of this, there is a need to assess the scope of the language using case studies. This approach will further clarify any misconceptions that may be created while learning the concepts of ALI V2 notation.

Therefore, in this chapter, a case study is presented where ALI V2 is applied to an Asset Management System (AMS). The AMS is a generic information system that manages financial assets in an investment bank. Essentially, the AMS is used by a fund management team to support making, and executing, investment decisions for a large-scale investment portfolio. This case study is chosen to demonstrate the suitability of ALI V2 ADL to this problem, and to highlight its importance in the Information System (IS) domain.

The next section details the AMS case study. Section 7.3 presents an architectural description of the AMS using concepts from ALI V2. In Section 7.4 the AMS architecture is evaluated in relation with how it overcomes the limitations in current ADLs (stated in Chapter 2) and how the design principles of ALI V2 ADL have been applied (described

in Chapter 6). Finally, the results obtained from the AMS architecture and are discussed and evaluated in Section 7.5.

7.2 Description of the AMS Case Study

An Asset Management System (AMS) is a financial asset management system used by a fund manager, or fund management team, to support making, and executing, investment decisions for a multi-scale investment portfolio. Essentially, portfolio investments are designed for investors who are looking for the potential to earn returns greater than cash deposits, either by taking a regular income or leaving their money to accumulate over the medium-to-long term. Of course, the investments made by investors can either rise or fall in value, over time.

From a financial perspective, a portfolio is defined as a collection of investments held by an investment company, financial institution or individual. It can also be referred to as mutual funding of financial assets. This case study demonstrates the portfolio managed by an investment bank.

Investment decisions made, and executed, to manage portfolios vary between banks and even on the basis of which country (or continent) a particular bank is situated in. In this case study, essential portfolio operations are demonstrated that regularly take place in all types of AMS and are also commonly performed by leading investment banks in the world. Further, the AMS description provided in this section is concluded with a detailed discussion with finance employees of investment banks: Barclays (UK), UBS (UK) and HSBC (Middle East).

The primary aim of the system is to allow a fund manager (or fund management team) to manage a portfolio of holdings in financial instruments (tradeable assets). There are

four different types of financial instruments: 1) *Equities* that correspond to shares, 2) *Commodities* that primarily correspond to metals (such as copper, gold, and so on), agriculture, oil, gas and energy, 3) *Interest Rate Products* that correspond to saving bonds, and 4) *Currency* that corresponds to Foreign Exchange (FX) rates. Along with these instruments, a derivative (security) value is determined by analysing the fluctuations and potential risks underlying in these financial instruments.

The AMS architectural description for equities is described in the following section, where the discussion focussed primarily on shares. A similar approach can be adopted for the design of other financial instruments.

The system allows the user (e.g. the fund manager or management team) to view the content of their equity portfolios, trading and market data (in our case, share trades and prices) in order to make investment decisions. It facilitates the automatic calculation of suggested changes to portfolios on-demand or on a regular schedule. This functionality is performed on a daily basis to calculate the portfolio value at the end of each working day, after the closure of stock market. In an investment bank, portfolio valuation can be performed using two methods, depending on the user's request. The first method is mark to market (MTM), where share prices are matched with the current stock market price and individual company share price (companies which are not listed in the stock market). The second method is applied on monthly/quarterly/bi-annual basis by checking the company's financial statement, depending on the company's fiscal period.

Another key function of an AMS is to rebalance the portfolio. Equity portfolio rebalancing can be performed in two ways: 1) Further investment in the form of cash. For example: if a portfolio is assigned £100, 80% of which is allocated to shares, and the fund manager decides, instead, to allocate around 50% for shares but does not want to sell shares. To achieve this, a new investor can be involved, who brings £50 cash into the

portfolio, which now contains £150 of which around 50% (i.e. £80 out of £150) is allocated to shares, 2) Making amendments to the existing financial equity instruments. For example: if a portfolio is 50% BP and 50% EDF in its equity allocation, and BP and EDF consists of 2 shares each, and the value for each is £50, the total value of the shares will be £200. Now suppose the BP share price doubles and EDFs drops by half, now the BP share value is £200 and EDF's is £50, so the total share value becomes £250. As a result, the distribution of shares is now 80% BP and 20% EDF. Now it depends on the fund manager's strategy to change the equity balance back to 50:50 by buying and/or selling shares, or perhaps maintain the 80:20 split.

The decision for rebalancing depends on the fund manager's pre-defined strategy and also by considering the current market conditions and any other environmental conditions, such as political or geographical factors.

7.3 AMS Architecture Representation Using ALI V2

This section presents the architectural description of the AMS case study explained in the previous section, using the ALI V2 notation that has been defined in Chapter 6.

The AMS architectural description is composed of the following architectural elements:

7.3.1 AMS Meta Types

The AMS architectural description is comprised of nine different *meta types*, which provide more information about its architectural elements. In particular, these elements require meta information, which have a complex structural (such as AMS *component types*, discussed in Section 7.3.6) and behavioural design (such as AMS *transaction domain* discussed in Section 7.3.11).

```
meta type Meta_AMSFeature {
   tag creation_date: date;
   tag standardized: boolean;
 }

meta type Meta_EquityServer {
   tag creatorID, intention*: text;
   tag cost, version: number;
   tag last_updated: date;
 }
```

`Meta_AMSFeature` described above can be attached to any AMS feature description, if required. `Meta_EquityServer` can be attached to server component types, or to any other architectural element which requires similar information to be described in its definition. Similarly, seven other meta types are defined in Appendix D1.

7.3.2 AMS Features

In the AMS, the equity portfolio requirements are determined by nine *features*, of which, some are parameterised and some are non-parameterised. Two of the features defined below take part in valuing and rebalancing the equity portfolio.

```
features {
        Equity: {
            alternative names: {
                Designer.FI1, Developer.Ey, Evaluator.F11;
            }
            parameters: {
               {Equity_Type = text;}
            }
          }

        Equity_Share: {
            alternative names: {
                Designer.IE1, Developer.ES, Evaluator.F12;
            }
```

```
                parameters: {
                    //no parameters
                }
            }
    …
} // end of features
```

The remaining seven features are defined in Appendix D2 some of which are specific to equity portfolio valuation (such as `MarkToMarket_Method`) and rebalancing (such as `Cash_Investment`).

7.3.3 AMS Interface Templates

The following extract is the syntax definition of the AMS interface template `MethodInterface` in accordance to which AMS *interface types* are created (as described in the next section):

```
interface template MethodInterface {
   provider syntax definition: {
    "Provider"":"
     "{"
        {"function" <FUNCTION_NAME>
           "{"
                "impLanguage" ":" <LANGUAGE_NAME> ";"
                "invocation" ":" <INVOCATION> ";"
                "paramterlist" ":" "("[ <PARAMETER_TYPE> {","
                                     <PARAMETER_TYPE}] ")" ";"
                "return_type" ":" <RETURN_TYPE> ";"
            "}" }
     "}"
     }
```

```
consumer syntax definition: {

 "Consumer"":"
  "{"
      "Call" ":" <INVOCATION> "("[<PARAMETER_TYPE> {","
                                 <PARAMETER_TYPE}]")" ";"
  "}"

 }

constraints: {

    should match: {INVOCATION_NAME = .INVOCATION_NAME,
                                         PARAMETER_TYPE}
    binding: {
               "multiple": true;
               "data_size": [50KB, 500MB];
               "max_connections": 20;
             }

    factory: true;
    persistent: false;

   }
}
```

7.3.4 AMS Interface Types

In AMS architecture, several *interface types* are involved that perform different functions required to value and rebalance the equity portfolio.

Two interface types (`ArithmeticOperation` and `ValueOperation`) are described below that conform to the interface template `MethodInterface` (defined in the previous section) and are used to calculate and rebalance the equity portfolio.

```
interface type {
     ArithmeticOperation: MethodInterface {
        Provider: {
         function Addition
                {
                 impLanguage: Java;
                 invocation: add;
                 parameterlist: (int);
                 return_type: void;
                }
```

```
          function Subtraction
                 {
                  impLanguage: Java;
                  invocation: subtract;
                  parameterlist: (int);
                  return_type: void;
                 }
          function Multiplication
            {
              impLanguage: Java;
                  invocation: multiply;
                  parameterlist: (int);
                  return_type: void;
            }
        }
          Consumer: {
            Call: getValue (long_int);
         }
      }

      ValueOperation: MethodInterface {
          Provider: {
           function GetValue
                 {
                  impLanguage: Java;
                  invocation: getValue;
                  parameterlist: (void);
                  return_type: long_int;
                 }
           }
          Consumer: {//nothing consumed}
      }
    …
  } // end of interface types
```

The other seven interface types of the AMS architecture that conform to the interface template `MethodInterface` are defined in Appendix D3.

7.3.5 AMS Connector Types

AMS architecture is composed of ten different *connector types* that are used to create connections between components, in order to value and rebalance the equity portfolio. The instances of these connector types were used for designing the AMS *component types* (defined in the next section) internal configuration (*sub-system* section). They have also been used to design the overall AMS architecture, as demonstrated in Section 7.3.13.

For example, the connector type `Calculator_Equity`, described below, is used as an instance in the component type `Portfolio_EquityValuator` (see Appendix D5) and in the AMS overall architectural description (see Appendix D10). It is used in the calculation of the equity Portfolio.

```
connector type Calculator_Equity
 {
  features: {
     MTM_Price_Method: "Share prices matched with market price",
     Company_Price_Method: "Unlisted share price of an individual
                            company",
     Weighted_Average_Method: "Portfolio Valuation is done on the
                               basis of average share price";
    }
  interfaces: {
      valueport1: ValueOperation;
      valueport2: ArithmeticOperation;
      valueport3: AverageOperation;
      valueport4: NumericOperation;
   }
  layout: {
     connect valueport4 and valueport1;
     if (supported(MTM_Price_Method || Company_Price_Method))
       {connect valueport1 to valueport2;
        connect valueport2 to valueport4;}
     else if (supported(Weighted_Average_Method)
        connect valueport3 to valueport4;
   }
 }
```

The other nine connector types that are used in the designing of the AMS architecture are detailed in Appendix D4.

7.3.6 AMS Component Types

The AMS architecture is composed of the following eleven *component types*:

- `PortfolioAMS_GUI` provides the asset managers using the system with the ability to view, analyse and value portfolios, to request (and monitor the progress of) long running system operations (such as order generation) and to check, enter, dispatch and monitor orders that go for execution in trading systems.

- `Portfolio_EquityUIServer` provides data access facilities that the UI requires (accessing data from the internal database) and dispatches requests for orders or for long running work (such as analysis processing) to be carried out by other parts of the system.

- `Portfolio_EquityValuator` calculates a portfolio value based on the valuation method requested by the asset manager. It supports three methods of valuation: 1) By checking the current price of the shares from the stock exchange via internal market data, 2) By checking the current price of the shares of those companies which are not listed in stock exchange via their financial statements, 3) By calculating the average value of the existing shares. Method 1 and Method 3 are mutually exclusive while Method 2 occurs monthly/quarterly/ bi-annually, depending on criteria set by the asset manager.

- `EquityCalculator` performs the mathematical operation based on the value and the method or message it received and then outputs the calculated portfolio value. It also calculates the derivative value of equities, if requested.

- `Portfolio_Processor` executes long running processing items ("jobs") and generates an order list. The processor can be configured to run particular jobs on temporal schedules and can also be requested to execute particular jobs on demand.

- `AMS_EquityDb` stores the portfolio, analytical, market and (system) operational data that the system requires to operate.

- `PortfolioDb` stores the different sets of equity portfolio.

- `Order_Generator` accepts incoming orders to buy and sell shares, forwards these requests to a trading system (both internal and external) for execution and then receives the execution reports which indicate order execution and broadcasts these to other relevant parts of the system.

- `DerivativeValuator` performs the derivative operations (if requested) on the existing shares for its security, based on the derivation strategy (options, futures or swaps), as requested.

- `Internal_EquityData` retrieves the various forms of market data from different external stock market systems and provides updated data for the portfolio valuation and, finally, loads the data into the database `AMS_EquityDb`.

- `Internal_EquityTrade` provides information about a request for buying and/or selling of shares that has been made internally by fund management teams within the organisation. Further, it allows internal trading as well.

In order to illustrate the design notations (both textual and graphical) of the AMS component types, a component type `Internal_EquityData` is described in this section. The other ten AMS component types are described in Appendix D5.

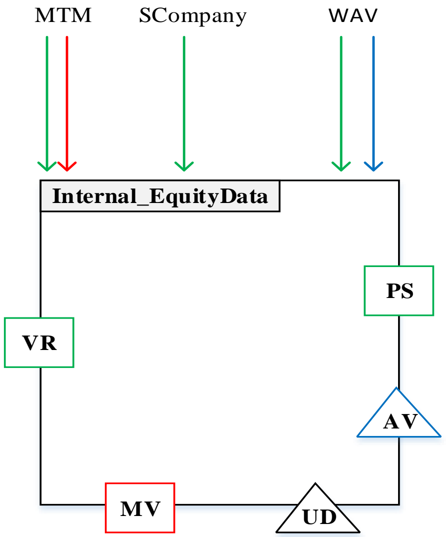


Figure 20: AMS component type `Internal_EquityData`

Figure 20 demonstrates the graphical structural notation of the component type `Internal_EquityData`. It consists of five interfaces that conform to two different *interface templates*. The colour conventions represent the dependencies of the interfaces with the features. For example, blue corresponds to the *WAV* feature and its dependent interface is `AverageValue (AV)` that conforms to the interface template `MethodInterface`. Black represents a mandatory interface.

The same colour conventions are used for the components and connectors that are described in the component type definition. For example, the component type `AMS_EquityDb` is defined in Appendix D5. It is important to clarify here that the colour convention has not been applied to the interfaces of the components and connectors that are used as an instance in the component type definition because their specification has already been defined in their own type definition.

Table 12 provides the list of all the acronyms that have been used to define the interfaces of the AMS component types graphically.

Acronym	Term	Acronym	Term
AM	AverageMessage	OA	OrderAccess
AR	AverageRequest	OD	OrderData
AV	AverageValue	OM	OrderMessage
CD	CurrentData	OV	OperationalValue
CM	CalculationMessage	PS	PriceStatus
CR	CalculationRequest	SR	ServiceRequest
CV	CalculationValue	TD	TradeData
DA	DataAccess	TM	TradeMessage
DR	DerivativeRequest	UD	UpdatedData
DV	DerivativeValue	UM	UpdationMessage
IV	InvestmentValue	UR	UserRequest
MV	MarketValue	US	UpdationStatus
NM	NotificationMessage	VR	ValuationRequest
NV	NumericalValue		

Table 12: List of acronyms for AMS component types interfaces

Table 13 provides the graphical notation of the AMS *interface templates* that the interfaces conform to.

Symbol	Name
△	MethodInterface
□	WSDL

Table 13: AMS interface templates notations

Below is the textual notation for the component type `Internal_EquityData`:

```
component type Internal_EquityData
  {
   meta: Meta_ShareValueData {
      stock_market: "LSE, NYSE";
      intention: "To have updated share price in accordance to
                    market rate";
      price_synchronised: 29-02-2016;
    }
   features: {
      MTM: "Share prices matched with market price",
      SCompany: "Unlisted share price of an individual company",
      WAV: "Portfolio Valuation is done on the basis of average
```

```
            share price;
        }
      interfaces: {
       definition: {
          //no need to define any interface/s
       }
       implements:{
          // MethodInterface interface template
          UpdatedData: DatabaseUpdation;
          if (supported(MTM || SCompany || WAV)) {
            //WSDL interface template
            ValuationRequest: PortfolioMessenger;
            PriceStatus: ValueData;}
          if (supported(MTM))
            MarketValue: ValueOperation;
          if (supported(WAV))
            AverageValue: ValueOperation;
        }
      } //end of interfaces
    sub-system: {
     components {}
     connectors {}
     arrangement {}
    } // end of sub-system
  } // end of component type
```

Furthermore, in terms of the graphical representation, Table 14 provides the list of all the acronyms that have been used to define the interfaces of the connectors. These connectors are used to connect the components, if they exist, in the component type definition.

Acronym	Term
m1	messageport1
m2	messageport2
m3	messageport3
m4	messageport4
r1	requestport1
r2	requestport2
r3	requestport3
r4	requestport4
v1	valueport1
v2	valueport2
v3	valueport3
v4	valueport4

Table 14: List of acronyms for AMS connector interfaces

For other AMS component types with different architectural configurations, such as those without variable features/interfaces (e.g. `PortfolioAMS_GUI`) and those with a detailed sub-system description (e.g. `Portfolio_EquityValuator`), please refer to Appendix D5.

7.3.7 AMS Product Configurations

As discussed in Section 7.2, AMS manages portfolios of different financial instruments, where each instrument may have different specifications (depending on user requirements) that together form a *product*. In other words, the AMS is a product-line architecture.

In particular, a financial instrument (equity, in this case), is comprises of four products where two correspond to portfolio valuation requirements (one described below), one corresponds to a portfolio rebalancing requirement, and one for its derivative requirement.

```
product configurations {
     Equity_Share_ExchangeTraded: {
                Equity {Equity_Type = (long, short)};
                Equity_Share = true;
                MarkToMarket_Method = true;
               Share_Company_Method = false;
          }
        …
  } // end of product configurations
```

For the other three *products* related to equities, please refer to Appendix D6.

7.3.8 AMS Events

From the behavioural architectural description perspectives, the AMS is comprised of the twenty-six *events* that occur in order to value and rebalance the equity portfolio.

Below is a snippet of the events that occur while calculating the equity portfolio value:

```
events {
    ValuationRequest: <WSDL, WSDL>;
    RequestValuationDetails: <MethodInterface, MethodInterface>;
    SendValuationDetails: <MethodInterface, MethodInterface>;
    RequestPrice: <WSDL, WSDL>;
    CurrentStatus: <WSDL, WSDL>;
    RequestPriceList: <WSDL, WSDL>;
    …
  } // end of events
```

An event in the AMS architecture also conforms to more than one *interface template* (as discussed in Chapter 6) such as `Inform` and `PlaceOrder` (see Appendix D7). All the events that occur during the valuation and the rebalancing of the equity portfolio can be found in Appendix D7.

7.3.9 AMS Conditions

Conditions demonstrate the various behavioural descriptions of an AMS under which an equity portfolio value is calculated and its rebalancing is done, as follows:

```
conditions {
    PriceUnchanged: "No change in share price";
    PriceChanged: "Change in share price";
    ShareTrade: "Buying/Selling of shares";
    Exchange_Traded: "Shares listed in stock exchange";
    Illiquid: "Shares not listed in stock exchange";
    Further_Investment: "Investing more amount in Portfolio";
    Financial_Instr_Equity: "Dealing with equity financial
                             instrument";
    OrderForwarded: "Order request has forwarded to external trading
                     system";
    OrderFilled: "Order request has been filled by internal trading
                  system";
  } // end of conditions
```

In the above definition, the first five are the possible conditions on which the equity portfolio is valuated daily. While the remaining four are the possible conditions relevant to rebalance the equity portfolio as demonstrated in Appendix D9.2.

7.3.10 AMS Scenarios

The AMS architecture behavioural description encapsulates eight different *scenarios* to revalue and rebalance the equity portfolio.

Two scenarios related to the equity portfolio revaluation are described below:

```
scenarios {
   P.RevaluatingPC: {
      Description: "Revaluating portfolio due to change in share
                    price with no trading";
      Parameterisation: {
                          PriceChanged = true;
                          PriceUnchanged = false;
                          ShareTrade = false;
                        }
        }
   P.RevaluatingPC.ST_ET: {
      Description: "Revaluating portfolio due to change in share
                    price and exchange trading both";
      Parameterisation: {
                          PriceChanged = true;
                          PriceUnchanged = false;
                          ShareTrade = true;
                          Exchange_Traded = true;
                          Illiquid = false;
                        }
       }
      …
   } // end of scenarios
```

The remaining three scenarios related to the equity portfolio revaluation and the other three related to equity portfolio rebalancing are described in Appendix D8.

7.3.11 AMS Transaction Domains

The AMS architecture is comprised of two *transaction domains* that reflect the functionalities of the AMS case study explained in Section 7.2. The transaction domains are:

- `PortfolioValuation` values an equity portfolio on daily basis, which is done by MTM and/or retrieving an individual company's share price when it is not stock exchange listed.
- `PortfolioRebalance` rebalances the equity portfolio (if needed) via further investment or trading of shares.

The textual architectural description of the transaction domain `PortfolioValuation` is:

```
transaction domain PortfolioValuation
 {
   meta: Meta_PortfolioDomain
     {
      purpose: "To calculate portfolio value";
      compatibility: "financial instrument -equity";
      occurrence: "Once at the end of every working day";
     }
   contents:
     {
       /*provides the list of components and connectors involved in
         this transaction domain*/
       components: {Portfolio_GUI, UI_Server, EquityDb,
                    Job_Processor, Value_Processor,
                    Market_Share_Data, Equity_Market_Data,
                    *Stock_Market,*Company_Financial_Account,
                    UI_Price_Server, Portfolio_Value_Calculator,
                    *P/L_System}
       connectors: {HTTP_GUI, HTTP_Status, HTTP_Processor,
                    HTTP_ExMRate, HTTP_ExCRate, HTTP_CRate,
                    HTTP_Price, HTTP_External, Cal_Processor,
                    DB_VProcessor}
```

```
    }

transactions:
 {
  VALUATIONREQUEST: {
   events: {ValuationRequest, RequestValuationDetails,
            SendValuationDetails, Inform, RequestPriceList,
            RequestPrice}
   interactions: {
     Portfolio_GUI.ServiceRequest sends ValuationRequest/HTTP_GUI
     to UI_Server.ServiceRequest;
     UI_Server.NotificationMessage sends
     ValuationRequest/HTTP_Processor to
     Job_Processor.NotificationMessage;
     Job_Processor.DataAccess sends
     RequestValuationDetails/DB_VProcessor to
     EquityDb.DataAccess;
     EquityDb.DataAccess sends SendValuationDetails/DB_VProcessor
     to Value_Processor.DataAccess;
     if ( supported(Equity_Share)){
      if (PriceUnchanged){
        Value_Processor.NotificationMessage sends
        Inform/HTTP_Processor;
      else {
        [Value_Processor.CalculationMessage sends
         RequestPriceList/HTTP_Processor |
         Value_Processor.PriceStatus sends
         RequestPrice/HTTP_ExCRate];}
     }
    }
  }
 VALUATIONUPDATE: {
  events: {CurrentStatus, Inform}
  interactions: {
    UI_Server.NotificationMessage receives Inform/HTTP_Processor;
    UI_Server.UpdationStatus sends CurrentStatus/HTTP_Status to
    Portfolio_GUI.UpdationStatus;
   }
  }
```

```
MTMVALUATION: {
 events: {RequestPriceList, RequestPrice, CurrentPrice}
 interactions: {
   Market_Share_Data.CalculationRequest receives
   RequestPriceList/HTTP_Processor;
   Market_Share_Data.MarketValue sends RequestPrice/HTTP_ExMRate
   to *Stock_Market;
   *Stock_Market sends CurrentPrice/HTTP_ExMRate;
  }
 }
UNLISTEDVALUATION: {
 events: {RequestPrice, CurrentPrice}
 interactions: {
   *Company_Financial_Account receives
   RequestPrice/HTTP_ExCRate;
   *Company_Financial_Account sends CurrentPrice/HTTP_ExCRate to
   UI_Price_Server.PriceStatus;
   UI_Price_Server.PriceStatus sends CurrentPrice/HTTP_CRate;
  }
 }
REVALUATION: {
 events: {CurrentPrice, UpdatedPriceList, SendValuation,
          UpdateValue, Notify, Inform, CurrentStatus}
 interactions: {
   [Equity_Market_Data.PriceStatus receives
    CurrentPrice/HTTP_ExMRate |
    Equity_Market_Data.PriceStatus receives
    CurrentPrice/HTTP_CRate];
   Equity_Market_Data.PriceStatus sends
   UpdatedPriceList/HTTP_Price to
   Portfolio_Value_Calculator.PriceStatus;
   Portfolio_Value_Calculator.NumericalValue sends
   SendValuation/Cal_Processor to Value_Processor.OperatedValue;
   [Value_Processor.DataAccess sends UpdateValue/DB_VProcessor
    to EquityDb.DataAccess,
    Value_Processor.NotificationMessage sends
    Notify/HTTP_External to *P/L_System,
    Value_Processor.NotificationMessage sends
    Inform/HTTP_Processor to UI_Server.NotificationMessage];
   EquityDb.DataAccess sends Notify/DB_VProcessor to
   Job_Processor.DataAccess;
   UI_Server.UpdationStatus sends CurrentStatus/HTTP_Status to
```

```
      Portfolio_GUI.UpdationStatus;
     }
    }
   VALUATIONPROCESS: {
     events: {Inform, RequestPriceList, RequestPrice, CurrentPrice}
     interactions: {
       if (supported(Equity_Share)) {
        if (PriceUnchanged) {
          VALUATIONUPDATE receives Inform/ODBC_Processor from
           VALUATIONREQUEST;
        else {
         if (supported(MarkToMarket_Method)&& (Exchange_Traded))
           MTMVALUATION receives RequestPriceList /HTTP_Processor
           from VALUATIONREQUEST;
         else
           UNLISTEDVALUATION receives RequestPrice/HTTP_ExCRate
           From VALUATIONREQUEST;}
        }
       }
       [REVALUATION receives CurrentPrice/HTTP_ExMRate from
        MTMVALUATION | REVALUATION receives CurrentPrice/HTTP_CRate
         from UNLISTEDVALUATION];
      } //end of interaction
    } //end of transaction
  } //end of transactions section
} //end of transaction domain
```

Figure 21 and Figure 22 demonstrate the graphical behavioural (in the form of *event traces*) and structural notations of the transaction domain `PortfolioValuation`, respectively, which is described textually above.

The acronyms that have been used to demonstrate the component and connector interfaces in Figure 22 are defined in Table 12 and Table 14. Structural notation for the interfaces to which *interface template* conforms to are defined in Table 13.

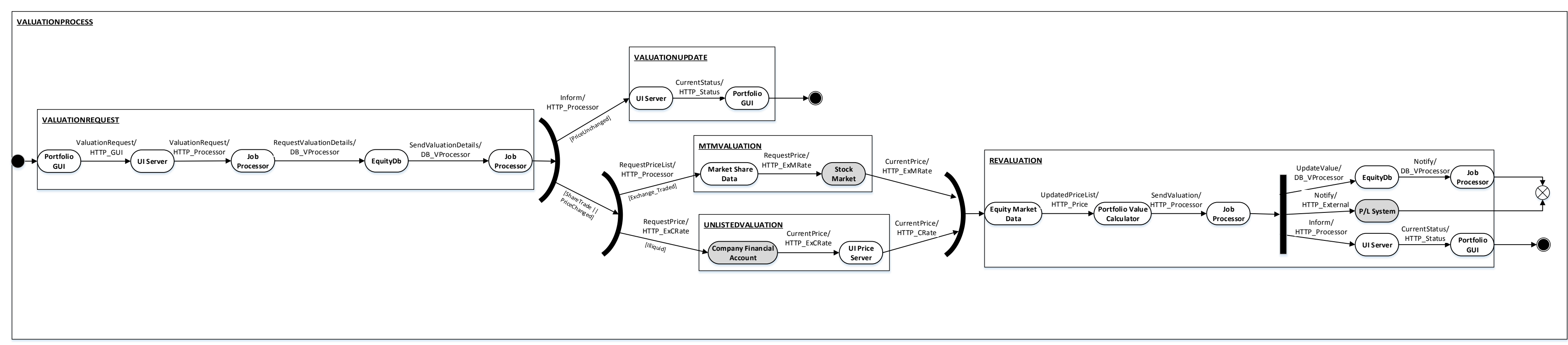


Figure 21: Graphical behavioural representation of transaction domain `PortfolioValuation`

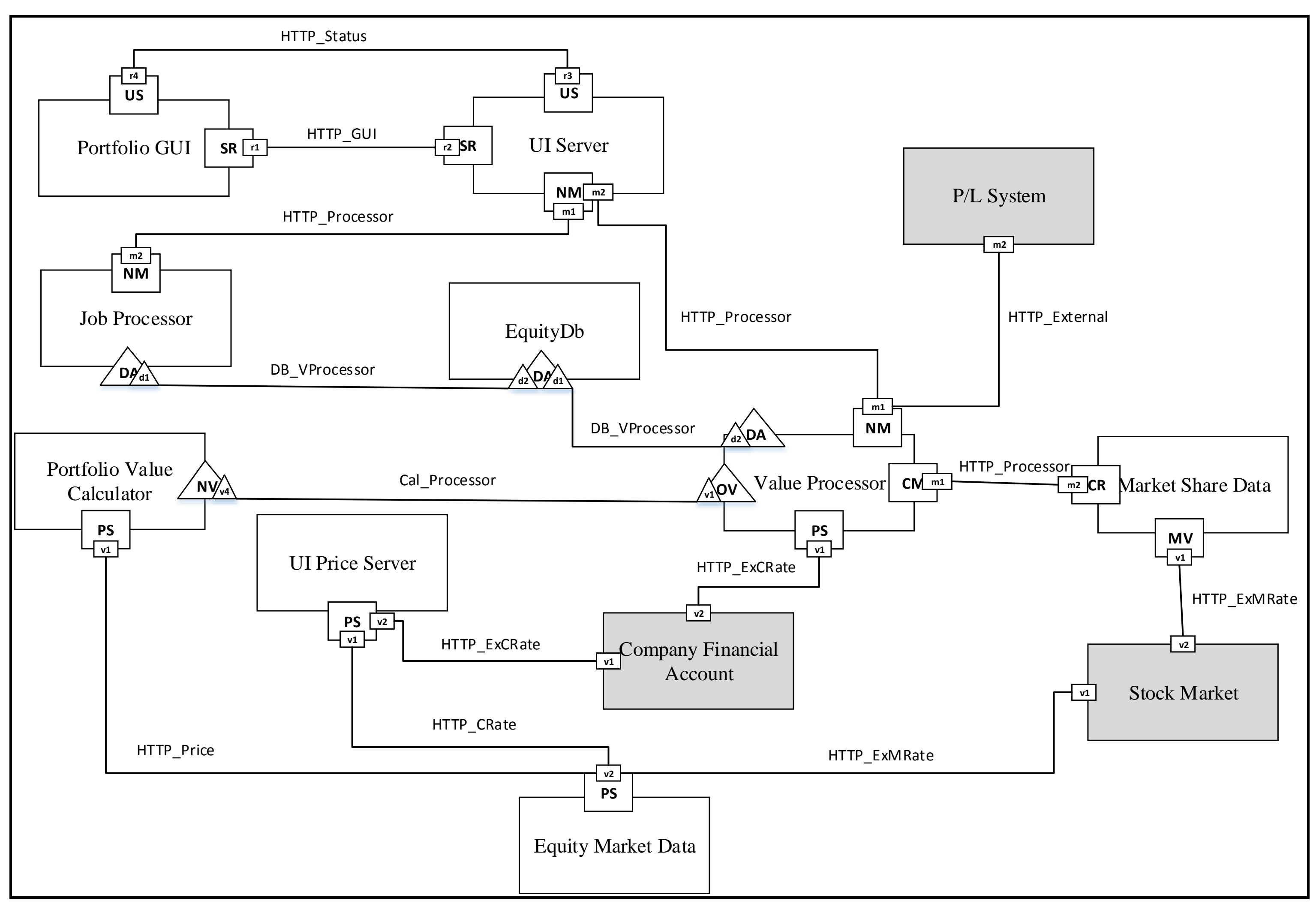


Figure 22: Graphical structural representation of transaction domain `PortfolioValuation`

The interactions of component `Portfolio_GUI` within the transaction domain `PortfolioValuation` is demonstrated in the Figure 23.

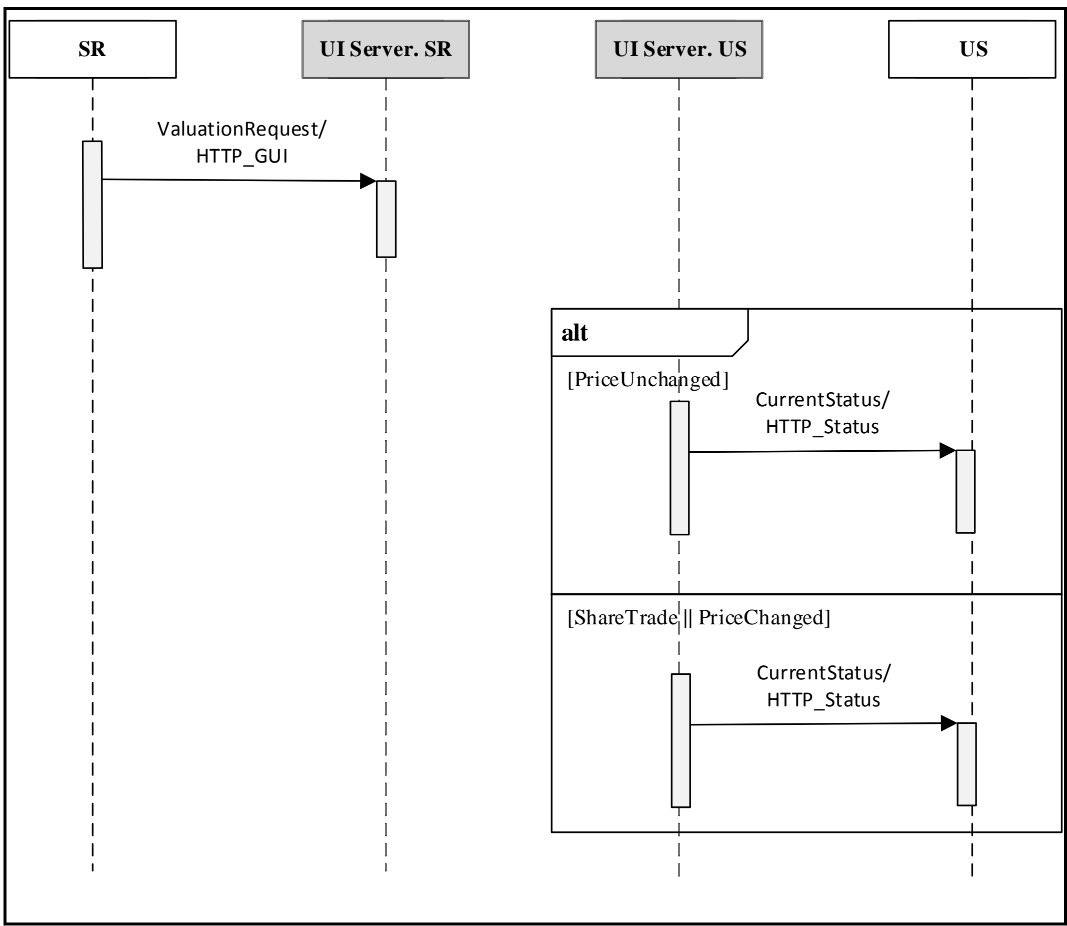


Figure 23: AMS component `Portfolio_GUI` interactions in transaction domain `PortfolioValuation`

The remaining component interactions that are involved in the transaction domain `PortfolioValuation` can be found in Appendix D9.1. Also, please refer to Appendix D9.2 for the transaction domain `PortfolioRebalance` architectural description.

The interface acronyms that have been used in Figure 23 and in the rest of the component interactions (see Appendix D9) are defined in Table 12.

7.3.12 AMS Viewpoint

In accordance with the AMS description discussed in Section 7.2, its architecture consist of the following *viewpoint*:

```
Viewpoints {
   PortfolioInvestment: {
       Description: "Investment made into the Portfolio";
       Transaction Domain: {PortfolioValuation,
                            PortfolioRebalance,
                            PortfolioStrategy};
    }
}
```

The viewpoint `PortfolioInvestment` demonstrates that three transaction domains need to be viewed or accessed when there is an investment in the equity portfolio. Transaction domains `PortfolioValuation` and `PortfolioRebalance` are described in the previous section while `PortfolioStrategy` is beyond the scope of this chapter.

7.3.13 Asset Management System (AMS)

This section presents the overall system architecture of the AMS case study explained in Section 7.2. Below is an extract of the AMS architecture:

```
system {
  components {
    Portfolio_GUI<>: PortfolioAMS_GUI;
    UI_Server<false, false, false>: Portfolio_EquityUIServer;
    …
    if(supported(Equity_Share)){
     // portfolio valuation
      Value_Processor<false, false, false, true, true, false>:
                                                Portfolio_Processor;
     …
  }// end of components
```

```
connectors {
  HTTP_GUI<true, false, false, false>: HTTP_AMSUserInterface;
  HTTP_Processor<true, false>: HTTP_Equity;
  …
   if( supported(Cash_Investment))
    DB_CRebalance<true, false>: ODBC_EquityPortfolio;
  …
} // end of connectors

 arrangement {
   //similar to component type arrangement
  connect Portfolio_GUI.ServiceRequest with HTTP_GUI.requestport1;
  connect UI_Server.ServiceRequest with HTTP_GUI.requestport2;
  …
  if (supported(Equity_Share)){
   // portfolio valuation
   connect EquityDb.DataAccess with DB_VProcessor.dataport1;
   connect Value_Processor.DataAccess with
   DB_VProcessor.dataport2;
  …
 } // end of arrangement

 viewpoints {
   PortfolioInvestment;
 } // end of viewpoints
} // end of system
```

For a complete textual description of the AMS architecture, please refer to Appendix D10.

It is important to state that the graphical structural notation of the whole system is not provided in this section due to the size and complexity of the AMS architecture. An alternative to this, as discussed in Chapter 6, is to divide it into segments by demonstrating it as AMS *transaction domains* in Section 7.3.12.

7.4 AMS Evaluation

In this section, the AMS architecture model designed in the previous section is evaluated on how it overcomes the limitations that exist in architectural languages and how it addresses the principles, on which ALI V2 is based. Table 15 presents the evaluation of the AMS case study.

Limitations Addressed *(Keywords)*	CASE STUDY: Asset Management System (AMS)	ALI V2 Principles Used *(Keywords)*
L1 (Variability)	According to the AMS case study described in Section 7.2, variability occurs when portfolio valuation is calculated due to change in the share price and/or share trading. Similarly, it occurs when portfolio rebalancing is performed via investing cash or buying/selling shares. From this perspective, the following variabilities were identified: -*Variable features*: `Equity_Share`, `MarkToMarket_Method`, `Share_Company_Method`, `Cash_Investment` and `Share_Investment`. -*Variable conditions*: All the conditions defined in Section 7.3.9.	P1 (Variability)
L2 (Traceability)	From the structural aspect of the AMS, the features `Equity_Share`, `MarkToMarket_Method` and `Share_Company_Method` represent the requirement and its traceability to calculate the equity portfolio value in the `system` description (Section 7.3.13). When rebalancing the equity portfolio, the features `Cash_Investment` and `Share_Investment` represent the requirement and its traceability. From the behavioural aspect of the AMS, the conditions `PriceUnchanged`, `PriceChanged`, `ShareTrade`, `ExchangeTraded` and `Illiquid` can occur during the equity portfolio valuation depending on the external requirement. `Further_Investment`, `Financial_Instr_Equity`, `OrderForwarded` and `OrderFilled` can	P2 (Traceability)

Limitations Addressed ***(Keywords)***	**CASE STUDY: Asset Management System (AMS)**	**ALI V2 Principles Used** ***(Keywords)***
	occur during the equity portfolio rebalancing, depending upon the external requirement. These conditions are represented in the transaction domains `PortfolioValuation` and `PortfolioRebalance` (Section 7.3.11).	
L3 (Dependency)	Not applicable.	P3 (Cross Domain)
L4 (Restrictive Syntax)	All the architectural elements defined in the AMS architecture are formal and flexible enough to design, and better support, the `system` description. For example, component type `Internal_EquityData` (defined in Section 7.3.6) used pre-defined interfaces (`UpdatedData` of type `DatabaseUpdation`, `MarketValue` and `AverageValue` of type `ValueOperation`, `ValuationRequest` of type `PortfolioMessenger` and `PriceStatus` of type `ValueData`) instead of defining them within its *definition* section using the interface templates `MethodInterface` and `WSDL`. The component type `Internal_EquityData` has the flexibility to define an interface similarly to the method used in the *interface types* section in its *definition* section (as explained in Chapter 6). Similarly, the event `Inform` supports the interface templates `MethodInterface` and `WSDL` in the transaction domain `PortfolioValuation`, but it has the flexibility to support only the interface template `MethodInterface` as in the transaction domain `PortfolioRebalance`.	P4 (Flexibility & Formality)

Limitations Addressed *(Keywords)*	**CASE STUDY: Asset Management System (AMS)**	**ALI V2 Principles Used *(Keywords)***
L5 (Reusability)	Interface template `MethodInterface` and the interfaces of type `MethodInterface` (defined in Section 7.3.3 and 7.3.4, respectively) can be used in any type of system architecture wherever an interface of this type is required. *Connector types* and *component types* (defined in Section 7.3.5 and 7.3.6, respectively) can be easily reused in any investment bank (or by any fund management company) as part of their asset management system to calculate and rebalance their equity portfolio. As explained in Chapter 6, the system can be adopted by simply mapping their feature set to the required system where it will be deployed. For example, connector type `Calculator_Equity` and component type `Portfolio_EquityValuator` (see Appendix D6) have features `Weight_Average_Method` and `Weighted_Average_Value_Method`, respectively. This feature is one of the methods used to calculate the equity portfolio and is not adopted by an investment bank nowadays where they must manage large-scale equity portfolios. Therefore, it is not considered in the `system` description (Section 7.3.13). But the artefact description of the connector type `Calculator_Equity` and component type `Portfolio_EquityValuator` are defined in such a way that it may be used in another system where they support the weighted average value method to calculate their equity portfolio value due to the support of its relevant features.	P5 (Reusability)

Limitations Addressed *(Keywords)*	**CASE STUDY: Asset Management System (AMS)**	**ALI V2 Principles Used *(Keywords)***
	As components and connectors are dependent on interfaces, the connector type `Calculator_Equity` and component type `Portfolio_EquityValuator` should reuse the interfaces from their definition, using pre-defined interface templates `MethodInterface` and `WSDL`; and their corresponding *interface types*. Similarly, instances of other component types (such as `Portfolio_EquityCalculator`, Appendix D6) and connector types (such as `HTTP_EquityValulator`, Appendix D5) that have been defined internally to design the component type `Portfolio_EquityValuator` will also be reused.	
L6 (Information Overload)	In the AMS architecture, in order to calculate the equity portfolio value, the transaction domain `PortfolioValuation` is defined textually in Section 7.3.11 and presented graphically using event traces in Figure 21 to illustrate its behavioural description. It is defined textually in the `system` description (Section 7.3.13) and presented graphically in Figure 22 as its structural description. In addition, the sequential interaction of all the components involved in the transaction domain `PortfolioValuation` (such as component `Portfolio_GUI` in Figure 23) are presented. In a similar way, to rebalance the equity portfolio, the transaction domain `PortfolioRebalance` is designed and presented in Appendix D9.2. Thus, the AMS architecture provides multiple architectural views of a particular function of the AMS (as a *transaction domain*) with a clear separation between structural and behavioural descriptions while maintaining	P6 (Multiview)

Limitations Addressed *(Keywords)*	CASE STUDY: Asset Management System (AMS)	ALI V2 Principles Used *(Keywords)*
	consistency between them. These different views capture the massive complexity of the AMS that can cater to different stakeholders, depending upon their concern.	
L7 (Behavioural)	Considering the behavioural aspect of the AMS architecture, events such as `ValuationRequest`, `RequestPrice`, etc., have been defined clearly, with their source and destination *interface templates* specified in Section 7.3.8. From the behavioural visualisation perspective, the transaction domains `PortfolioValuation` and `PortfolioRebalance` are presented in the form of event traces that demonstrate the ways an event can occur to calculate and rebalance the equity portfolio, as demonstrated in Figure 21 and Appendix D9.2, respectively. For example, in transaction domain `PortfolioValuation`, `RequestPriceList/HTTP_Processor` and `RequestPrice/HTTP_ExCRate` are the events that depends on the conditions `Exchange_Traded` and `Illiquid`, respectively. This is represented using OR Fork notation (as defined in Chapter 6) which means that it can occur one at a time to do equity portfolio valuation. Similarly, other event trace notations described in Chapter 6 are used while designing these transaction domains. Moreover, the interactions of all the components that are involved in both transaction domains are explicitly presented using a UML sequence diagram. These aspects demonstrate the detailed behavioural description of the AMS architecture.	P6 (Multiview)

Table 15: AMS evaluation

7.5 Discussion

This section will further discuss how ALI V2 attempts to reconcile the competing principles (as discussed on Chapter 6) for the language, in the context of the AMS case study.

AMS is a product line comprising various back-office portfolio management applications for financial instruments such as equity, commodity and currency, and corresponds to the Information System (IS) application domain.

AMS architecture is highly customisable, having a number of variable features and conditions, as identified in Table 15. This signifies the presence of significant inherent variability and the ability of ALI V2 to manage its variability (design principle P1).

The structural design of the AMS architecture uses connectors to join components. This is visualised using the AMS graphical structural notation in Figure 22. In that design, all system information is captured through a single transaction domain view (as shown in Figure 22 and Appendix D9.2), which shows all the transactions. However, it is not possible to present the overall AMS system architecture through graphical structural design due to the complexity of the system and the amount of information that needs to be captured. Accordingly, two structural views are produced, each capturing the information pertaining to one transaction domain. One view calculates the equity portfolio value (`PortfolioValuation`) and the other view rebalances the equity portfolio (`PortfolioRebalance`). Thus, this demonstrates that the notation can scale seamlessly (design principle P6), while still capturing all required information at an appropriate level of abstraction (L6).

Additionally, the AMS architecture contains several components and connectors leading to a relatively more complex structural description, compared to the behavioural description which has simpler interactions. Subsequently, architectural elements (such as

components, connectors and interfaces) defined in the AMS architecture can be reused with minimal, or no, changes to their internal description in other systems, as mentioned in Table 15. This is due to the granularity and reusability with the support of variability in ALI V2 as per design principle P5. Finally, AMS architecture is also linked with the external system (such a trading system) and sub-system (such as P/L system).

In the next chapter, another case study is presented that demonstrates the further applicability of ALI V2 constructs and notations.

Chapter Eight

Case Study: Wheel Brake System

8

"The more that you read, the more things you will know. The more that you learn, the more places you'll go."

-- Dr. Seuss

8.1 Introduction

In order to gain more experience on the applicability of ALI V2, another case study – the Wheel Brake System (WBS) – is presented in this chapter to demonstrate its application. This case study will also touch upon the other concepts of ALI V2 notations that were not practically applied while designing the Asset Management System (AMS) architecture in the previous chapter.

WBS is a standardised case study taken from the SAE ARP4761 standard (ARP4761, 1996), and it is being introduced to demonstrate the safety of the airborne system. The main aim of the WBS is to provide the necessary support for stopping/decelerating the commercial (civil) aircraft during landing or parking.

Furthermore, WBS corresponds to the Embedded System domain as compared with the AMS case study (described in the previous chapter) that corresponds to the Information System (IS) domain. From this aspect, the demonstration of the ALI V2 on the WBS case study supports its cross-application domain modelling capabilities (i.e. design principle P3, defined in Chapter 6).

The next section elucidates the WBS case study. Section 8.3 presents an architectural description of WBS using the concepts of ALI V2. In Section 8.4, the WBS architecture is evaluated in relation to how it overcomes the limitations that exist in current ADLs

(stated in Chapter 2) and how the design principles established for ALI V2 ADL have been applied (described in Chapter 6). Finally, results obtained from the WBS architecture and its evaluation are discussed in Section 8.5.

8.2 Description of the WBS Case Study

The Wheel Brake System (WBS) described in this section has been adopted from the SAE Standard Aerospace Recommended Practice (ARP) 4761, *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* (ARP4761, 1996).

According to ARP4761, the primary purpose of the WBS is to decelerate the commercial aircraft wheel on the ground during landing or parking. The WBS consists of a digital controller - Brake System Control Unit (*BSCU*) and the hydraulic pipe assembly that carries the braking pressure to the wheels. Different valves are embedded that receive commands and control the flow of brake pressure. While the brake system annunciation correspond to a non-functional requirement. Figure 24 presents the visual representation of the WBS.

ARP4761 states that the loss of all wheel braking is less probable than 5×10^{-7} per flight. Considering this, *BSCU* contains two independent systems (System1 and System2) to meet the availability and integrity requirements. Each system has the following subcomponents:

1. A *monitor* function that indicates if the values are valid or not.
2. A *command* function that produces data from the pedal values.

In contrast, the main *BSCU* (as shown in Figure 24) receives data (considered as electrical pedal positions) and power and forwards it to each subsystem. The reason for

having two systems is that if System1 generates an invalid value/command then System2 will operate. But if both fail to generate a valid value/command or did not receive any of the inputs (data or power), then that leads to *BSCU* failure. The *BSCU* sends the generated value/command to the other required parts of the system.

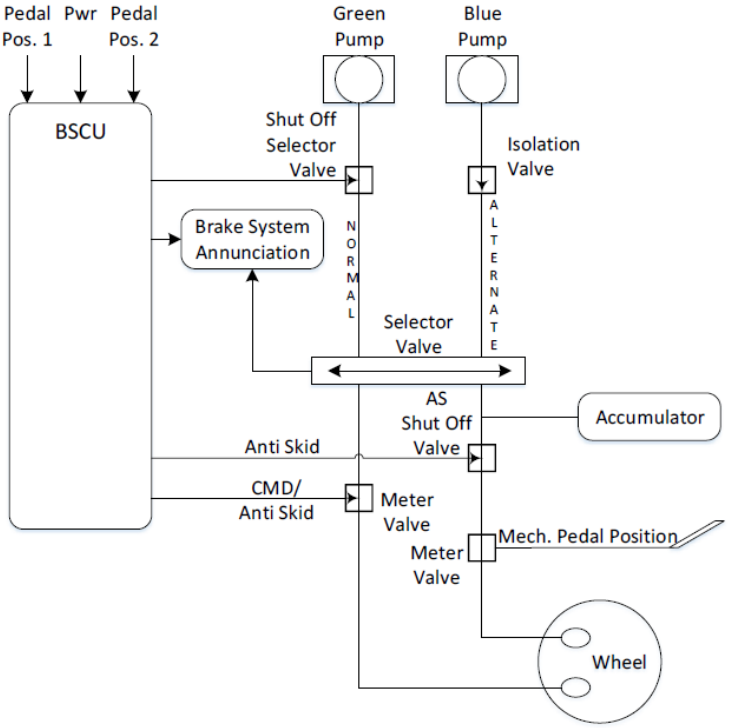


Figure 24: Wheel Brake System (ARP4761, 1996)

Moreover, a design decision was also made that each wheel has a brake assembly operated by two independent sets of hydraulic pistons. One set of pistons is operated from the *Green Pump* and is used in the *NORMAL* braking mode. In *NORMAL* mode, the *Green Pump* will receive an electrical brake command (*CMD*) and an *Anti Skid* command from the *BSCU* and then supplies the required pressure to the wheel.

The *ALTERNATE* braking system is on standby and is selected automatically in case of *NORMAL* system failure (due to *BSCU* and/or *Green Pump* failure). In *ALTERNATE* mode, the *Blue Pump* provides the hydraulic pressure to the system, and if the failure is due to the *Green Pump*, then it will also receive the *Anti Skid* command; otherwise, this

will not occur in case of *BSCU* failure. The failure of both of the pumps can be due to the absence of the hydraulic pressure supply or pressure being below the threshold value.

Subsequently, if the *ALTERNATE* system fails, then the system will be in *EMERGENCY* braking mode where the wheel will receive the reserve pressure from the *Accumulator*. It acts as a parking brake, as well. A mechanical pedal is used to apply the brake in both *ALTERNATE* and *EMERGENCY* modes. Switch-over between the hydraulic pistons and the different pumps is automatic under various failure conditions, or can be manually selected.

8.3 WBS Architecture Representation Using ALI V2

This section presents the architectural description of the WBS case study explained in the previous section using the ALI V2 notation that has been defined in Chapter 6.

The WBS architectural description is composed of the following architectural elements:

8.3.1 WBS Meta Types

The WBS architectural description is comprised of six different *meta types* that provide more information about its architectural elements. In particular, these elements require meta information, which has a complex structural (such as WBS *component types*, discussed in Section 8.3.5) and behavioural design (such as WBS *transaction domain* discussed in Section 8.3.10).

```
meta type Meta_WheelPedal {
   tag intention, consequences: text;
   tag cost*: number;
   tag last_checked: date;
 }
```

`Meta_WheelPedal` described above can be attached to the WBS brake pedal description. Similarly, five other meta types are defined in Appendix E1.

8.3.2 WBS Features

In the WBS, aircraft wheel braking requirements are determined by six *features*, of which, some are parameterised and some are non-parameterised. One of the features defined below is a mandatory feature to stop/decelerate the commercial aircraft wheels.

```
features {
        Wheel_Brake: {
              alternative names: {
                 Designer.F1, Developer.WB, Evaluator.F11;
             }
             parameters: {
                  // no parameters
             }
           }
        …
    } // end of features
```

The remaining five features are defined in Appendix E2, which some are related to the brake pedal (such as `Electrical_Brake`) and to the hydraulic pressure (such as `Piston_Pressure`) in the WBS.

8.3.3 WBS Interface Template

The following extract is the syntax definition of the WBS interface template `MethodInterface`, which all the WBS *interface types* are created with the same template (as described in the next section):

```
interface template MethodInterface {
   provider syntax definition: {
    "Provider"":"
     "{"
        {"function" <FUNCTION_NAME>
           "{"
                "impLanguage" ":" <LANGUAGE_NAME> ";"
                "invocation" ":" <INVOCATION> ";"
                "paramterlist" ":" "("[ <PARAMETER_TYPE> {","
                                    <PARAMETER_TYPE}] ")" ";"
                "return_type" ":" <RETURN_TYPE> ";"
            "}" }
     "}"
     }
   consumer syntax definition: {
    "Consumer"":"
     "{"
         "Call" ":" <INVOCATION> "("[<PARAMETER_TYPE> {","
                                    <PARAMETER_TYPE}]")" ";"
     "}"
    }
   constraints: {
       should match: { INVOCATION_NAME = .INVOCATION_NAME,
                                         PARAMETER_TYPE}
       binding: {
                 "multiple": true;
                 "data_size": [5KB, 500KB];
                 "max_connections": 5;
               }
       factory: false;
       persistent: false;
     }
  }
```

8.3.4 WBS Interface Types

In WBS architecture, six *interface types* are involved that perform different functions that are required to stop/decelerate the commercial aircraft wheels.

One interface type `DataOperation` that conforms to the interface template `MethodInterface` (defined in the previous section) is the following:

```
interface type {
     DataOperation: MethodInterface {
        Provider: {
         function InsertBrakeData
              {
               impLanguage: Java;
               invocation: insert;
               parameterlist: (string);
               return_type: void;
              }
         }
       }
       Consumer: {
           Call: insert (string);
        }
     }
    …
  } // end of interface types
```

The other five interface types of the WBS architecture that conform to the interface template `MethodInterface` are defined in Appendix E3.

8.3.5 WBS Component Types

The WBS architecture is composed of the following ten *component types*:

- `Aircraft_BrakePedal` - provides electrical braking data to the braking system as an input to the control unit. In the case of mechanical braking, it provides the pedal force and its position values to the metering valve.

- `Aircraft_ElectricPower` - provides an electrical voltage to the control unit to activate the electrical braking system, and generate commands and messages.

- `Brake_ControlUnit` – a processing unit that receives data from the electrical pedal and power and forwards the notification message, validated brake, and antiskid command values to the other required parts of the system.

- `Aircraft_WheelControlUnit` – demonstrates the main BSCU which consists of the two component type `Brake_ControlUnit` instances. This reduces the brake failure rate as discussed in Section 8.2, along with the ability to provide the valid command.

- `Aircraft_PressurePump` – provides hydraulic pressure from the piston to the metering valves in order to apply brakes to the commercial aircraft wheel.

- `Aircraft_BrakeValve` – valves that communicate with the hydraulic pistons to supply pressure to the system.

- `Aircraft_PressureValve` – valves that process the received commands and pressure and then forwards the demanded pressure to the corresponding element of the system in order to stop/decelerate the commercial aircraft wheel.

- `Command_Generator` – generates brake and/or antiskid commands on the basis of the electrical pedal value/s and the mechanical pedal position/s, including the pressure supplied by the pumps.

- `Value_Monitor` – validates the generated brake command from the electrical braking system.

- `Aircraft_Wheel` – provides the friction force to the aircraft wheel to stop/decelerate it.

In order to illustrate the design notations (both textual and graphical) of the WBS component types, a component type `Aircraft_BrakePedal` is described in this section. The other nine WBS component types are described in Appendix E4.

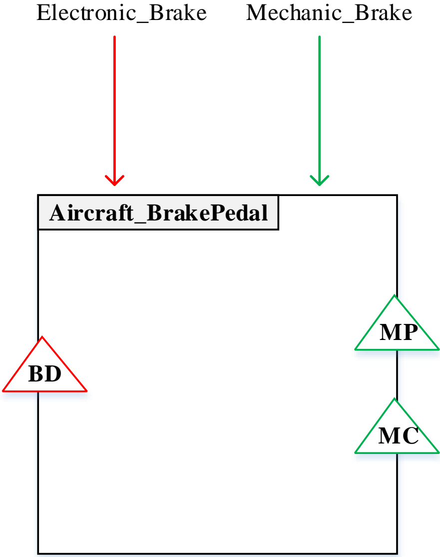


Figure 25: WBS component type `Aircraft_BrakePedal`

Figure 25 demonstrates the graphical structural notation of the component type `Aircraft_BrakePedal`. It consists of three interfaces that conform to interface template `MethodInterface`. Similar to the AMS case study (defined in Chapter 7) visualisation strategy, the colour conventions represent the dependencies of the interfaces with the features. For example, red corresponds to the *Electronic_Brake* feature and its dependent

interface is BrakeData (BD) that conforms to the interface template MethodInterface (represented as a triangle). Black represents a mandatory interface such as defined in the component type Aircraft_ElecticPower (see Appendix E4) which does not depend on any variable feature.

The same colour conventions are used for the components and connectors that are described in the component type definition. For example, the component type Aircraft_PressureValve is defined in Appendix E4. To recall, just like in the AMS case study, the colour convention has not been applied to the interfaces of the components and connectors that are used as an instance in the component type definition because their specification has already been defined in their own type definition.

Table 16 provides the list of all the acronyms that have been used to define the interfaces of the WBS component types graphically.

Acronym	Term	Acronym	Term
AC	AntiskidCommand	IP	InputPressure
AP	AlternatePressure	MC	MechanicalCommand
AV	AntiskidValue	MP	MechanicalPosition
BC	BrakeCommand	NP	NormalPressure
BD	BrakeData	PM	PressureMessage
BP	BrakePressure	RP	ReservePressure
CN	CommandNotification	VP	ValidatedPressure
CV	CommandValue	VV	ValidatedValue
EV	ElectricVoltage		

Table 16: List of acronyms for WBS component types interfaces

Below is the textual notation for the component type Aircraft_BrakePedal:

```
component type Aircraft_BrakePedal
  {
   meta: Meta_WheelPedal {
      intention: "To apply the brake";
      consequences: "Aircraft will not stop";
      cost: 5000;
```

```
      last_checked: 14-03-2016;
     }
    features: {
      Electronic_Brake: "Electrical pedal used to stop the
                         aircraft wheel",
      Mechanic_Brake: "Mechanical pedal applied to stop the
                       aircraft wheel";
     }
    interfaces: {
     definition: {
        // no need to define any interface/s
     }
     implements:{
        if (supported(Electronic_Brake))
          BrakeData: DataOperation;
        if (supported(Mechanic_Brake)){
          MechanicalPosition: ValueOperation;
          MechanicalCommand: CommandOperation;}
      }
     } //end of interfaces
   sub-system: {
    components { }
    connectors { }
    arrangement { }
   } // end of sub-system
 } // end of component type
```

WBS component types with different architectural configurations, such as those without variable features/interfaces (e.g. `Aircraft_ElectricPower`) and those with a detailed sub-system description (e.g. `Aircraft_WheelControlUnit`) are explained in Appendix E4.

8.3.6 WBS Product Configuration

As described in Section 8.2, WBS provides different braking modes (such as *NORMAL*) under which the commercial aircraft wheels can be stopped/decelerated. Thus, from the structural architectural configuration aspect, the WBS is a single product system with multiple variable features which is described as follows:

```
product configurations {
     CommercialAircraftBrake: {
              Electrical_Brake {Pedal_Value = 850KN};
              Electrical_Power {Voltage = 240V AC};
              Mechanical_Brake {Max_Pedal_Force = 980KN};
              Piston_Pressure {Maximum = 10.75 Pa,
                               Minimum = 5.25 Pa};
              Accumulator_Pressure {Pressure_Supplied = 9.5 Pa};
        }
} // end of product configurations
```

8.3.7 WBS Events

From the behavioural architectural description perspective, the WBS is comprised of the fifteen *events* that occur in order to stop/decelerate the commercial aircraft under different braking modes.

Below is the textual notation of all the events that occur while applying brake to the commercial aircraft wheels:

```
events {
   Send_EPedal_Position1: <MethodInterface, MethodInterface>;
   Send_EPedal_Position1: <MethodInterface, MethodInterface>;
   Send_Power_Signal1: <MethodInterface, MethodInterface>;
   Send_Power_Signal2: <MethodInterface, MethodInterface>;
   Inform: <MethodInterface, MethodInterface>;
   Notify: <MethodInterface, MethodInterface>;
   CMD: <MethodInterface, MethodInterface>;
   AntiSkid: <MethodInterface, MethodInterface>;
```

```
  Hydraulic_Pressure_Request: <MethodInterface, MethodInterface>;
  Send_Hydraulic_Pressure: <MethodInterface, MethodInterface>;
  No_Hydraulic_Pressure_Supply: <MethodInterface, MethodInterface>;
  MPedal_Position_Request: <MethodInterface, MethodInterface>;
  Send_MPedal_Position: <MethodInterface, MethodInterface>;
  Reserve_Pressure_Request: <MethodInterface, MethodInterface>;
  Decelerate: <MethodInterface, MethodInterface>;
} // end of events
```

8.3.8 WBS Conditions

Conditions described below demonstrate the various behavioural descriptions of the WBS under which brake can be applied to the commercial aircraft wheels.

```
conditions {
  BSCU_Active: "BSCU working properly";
  BSCU_Failed: "Unable to provide brake command";
  GreenPressure: "Provide hydraulic pressure in a normal mode";
  GreenPressure_Failed: "No hydraulic pressure supply or below
                         threshold value in a normal mode";
  BluePressure: "Provide hydraulic pressure in an alternate mode";
  BluePressure_Failed: "No hydraulic pressure supply or below
                    threshold value in an alternate mode";
  AccumulatorPump: "Provide hydraulic pressure in an emergency
                    mode";
} // end of conditions
```

In the above definition, seven of the possible conditions to apply the brake in different braking modes as demonstrated in Section 8.3.10.

8.3.9 WBS Scenarios

The WBS architecture behavioural description encapsulates four different *scenarios* that correspond to different braking modes of the commercial aircraft.

One scenario related to the *NORMAL* system braking mode is described below:

```
scenarios {
   NormalOperation {
      Description: "WBS in a normal mode";
      Parameterisation {
                        BSCU_Active = true;
                        GreenPressure = true;
                        BluePressure = false;
                        AccumulatorPump = false;
                       }
 …
} // end of scenarios
```

The remaining three scenarios related to the wheel braking are described in Appendix E5.

8.3.10 WBS Transaction Domain

The WBS architecture is comprised of the transaction domain `WheelDecelerationOnGround` that reflect the functionalities of the WBS case study explained in Section 8.2. It demonstrates how the wheels of the commercial aircraft can be stopped/decelerated on the ground during landing or take off.

The textual architectural description of the transaction domain `WheelDecelerationOnGround` is as follows:

```
transaction domain WheelDecelerationOnGround
  {
   meta: Meta_DecelerationDomain
      {
```

```
   purpose: "To stop the commercial aircraft on ground";
   minimum_wheels_active: 4;
  }
contents:
  {
    /*provides the list of components involved in this
      transaction domain*/
    Components: {Electrical_Pedal, Mechanical_Pedal, Power,
                 BSCU, ShutOff_Valve, Selector_Valve,
                 Green_Pump, Blue_Pump, Accumulator,
                 Meter_Valve, Wheel}
    //No connectors -direct binding
   }
transactions:
 {
 NORMALMODE: {
  events: {SendEPedalPosition1, SendEPedalPosition2,
           SendPowerSignal1, SendPowerSigna2, Inform, Notify,
           CMD, AntiSkid, HydraulicPressureRequest,
           SendHydraulicPressure, Decelerate}
  interactions: {
    [(BSCU.BrakeData receives SendEPedalPosition1 from
      Electrical_Pedal.Brakedata,
      BSCU.BrakeData receives SendEPedalPosition2 from
      Electrical_Pedal.Brakedata),
     (BSCU.ElectricVoltage receives SendPowerSignal1 from
      Power.ElectricVoltage,
      BSCU.ElectricVoltage receives SendPowerSignal2 from
      Power.ElectricVoltage)];
    [BSCU.CommandNotification sends Inform to
     ShutOff_Valve.CommandNotification,
     BSCU.BrakeCommand sends CMD to Meter_Valve.BrakeCommand,
     BSCU.AntiskidCommand sends AntiSkid to
     Meter_Valve.AntiskidCommand];
    ShutOff_Valve.PressureMessage sends Notify to
    Selector_Valve.PressureMessage;
    Selector_Valve.PressureMessage sends
    HydraulicPressureRequest to Green_Pump.PressureMessage;
    [Meter_Valve.NormalPressure receives SendHydraulicPressure
     from Green_Pump.NormalPressure,
     Meter_Valve.BrakeCommand receives CMD from
     BSCU.BrakeCommand,
```

```
    Meter_Valve.AntiskidCommand receives AntiSkid from
    BSCU.AntiskidCommand];
   Meter_Valve.BrakePressure sends Decelerate to
   Wheel.InputPressure;
  }
 }
EMERGENCYMODE: {
 events: {MPedalPositionRequest, ReservePressureRequest,
          SendHydraulicPressure, SendMPedalPosition,
          Decelerate}
 interactions: {
  [Selector_Valve.MechanicalPosition sends
   MPedalPositionRequest to
   Mechanical_Pedal.MechanicalPosition,
   Selector_Valve.PressureMessage sends ReservePressureRequest
   to Accumulator.PressureMessage];
  [Meter_Valve.MechanicalCommand receives SendMPedalPosition
   from Mechanical_Pedal.MechanicalCommand,
   Meter_Valve.ReservePressure receives SendHydraulicPressure
   from Accumulator.ReservePressure];
  Meter_Valve.BrakePressure sends Decelerate to
  Wheel.InputPressure;
 }
}
ALTERNATEMODE1: {
 events: {SendEPedalPosition1, SendEPedalPosition2,
          SendPowerSignal1, SendPowerSignal2, Inform, Notify,
          AntiSkid, HydraulicPressureRequest,
          NoHydraulicPressure, MPedalPositionRequest,
          SendHydraulicPressure, SendMPedalPosition,
          Decelerate}
interactions: {
 [(BSCU.BrakeData receives SendEPedalPosition1 from
  Electrical_Pedal.Brakedata,
  BSCU.BrakeData receives SendEPedalPosition2 from
  Electrical_Pedal.Brakedata),
 (BSCU.ElectricVoltage receives SendPowerSignal1 from
  Power.ElectricVoltage,
  BSCU.ElectricVoltage receives SendPowerSignal2 from
  Power.ElectricVoltage)];
 [BSCU.CommandNotification sends Inform to
  ShutOff_Valve.CommandNotification,
```

```
      BSCU.AntiskidCommand sends AntiSkid to
      Meter_Valve.AntiskidCommand];
     ShutOff_Valve.PressureMessage sends Notify to
     Selector_Valve.PressureMessage;
     Selector_Valve.PressureMessage sends HydraulicPressureRequest
     to Green_Pump.PressureMessage;
     Green_Pump.PressureMessage sends NoHydraulicPressureSupply to
     Selector_Valve.PressureMessage;
    [Selector_Valve.MechanicalPosition sends MPedalPositionRequest
     to Mechanical_Pedal.MechanicalPosition,
     Selector_Valve.PressureMessage sends HydraulicPressureRequest
     to Blue_Pump.PressureMessage];
    [Meter_Valve.MechanicalCommand receives SendMPedalPosition from
     Mechanical_Pedal.MechanicalCommand,
     Meter_Valve.AlternatePressure receives SendHydraulicPressure
     from Blue_Pump.AlternatePressure,
     Meter_Valve.AntiskidCommand receives AntiSkid from
     BSCU.AntiskidCommand];
    Meter_Valve.BrakePressure sends Decelerate to
    Wheel.InputPressure;
   }
  }
ALTERNATEMODE2: {
  events: {MPedalPositionRequest, HydraulicPressureRequest,
           SendHydraulicPressure, SendMPedalPosition, Decelerate}
  interactions: {
   [Selector_Valve.MechanicalPosition sends MPedalPositionRequest
    to Mechanical_Pedal.MechanicalPosition,
    Selector_Valve.PressureMessage sends HydraulicPressureRequest
    to Blue_Pump.PressureMessage];
   [Meter_Valve.MechanicalCommand receives SendMPedalPosition from
    Mechanical_Pedal.MechanicalCommand,
    Meter_Valve.AlternatePressure receives SendHydraulicPressure
    from Blue_Pump.AlternatePressure];
   Meter_Valve.BrakePressure sends Decelerate to
   Wheel.InputPressure;
  }
 }
```

```
DECELERATINGWHEEL: {
  /* No events in this transaction therefore, there is no event
    section */
  interactions: {
    if (supported(Electircal_Brake && Electrical_Power) &&
         (BSCU_Active && GreenPressure))
      {NORMALMODE;}
    else if (unsupported(Electrical_Power && Piston_Pressure) &&
             (BluePressure_Failed))
      {EMERGENCYMODE;}
    else if (supported(Electircal_Power) &&
             unsupported (Accumulator_Pressue) &&
             (BSCU_Active && GreenPressure_Failed))
      {ALTERNATEMODE1;}
    else
      {ALTERNATEMODE2;}
     } // end of interaction
    } // end of transaction
   } // end of transactions section
  } // end of transaction domain
```

Figure 26 demonstrates the graphical behavioural (in the form of *event traces*) of the transaction domain `WheelDecelerationOnGround`, which is described textually above, while the structural notation is presented in Section 8.3.12 (Figure 28).

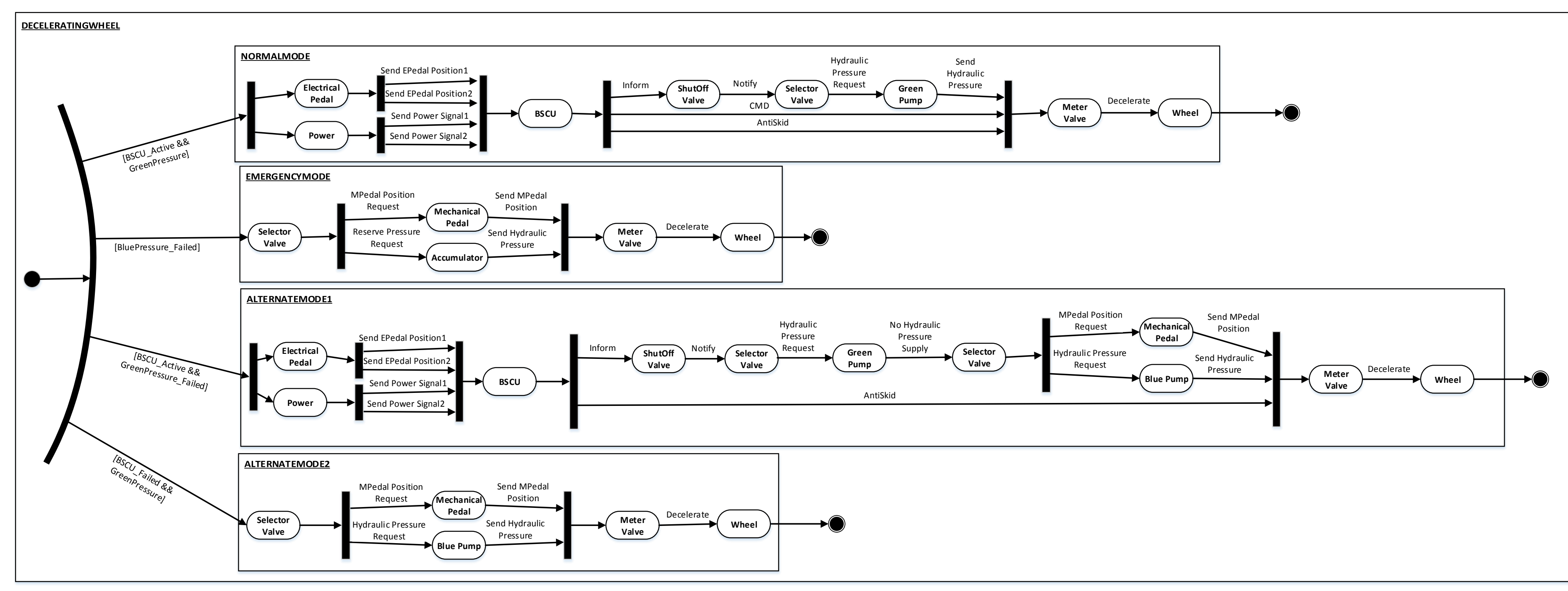


Figure 26: Graphical behavioural representation of transaction domain `WheelDecelerationOnGround`

The interactions of component Electrical_Pedal within the transaction domain WheelDecelerationOnGround is demonstrated in Figure 27.

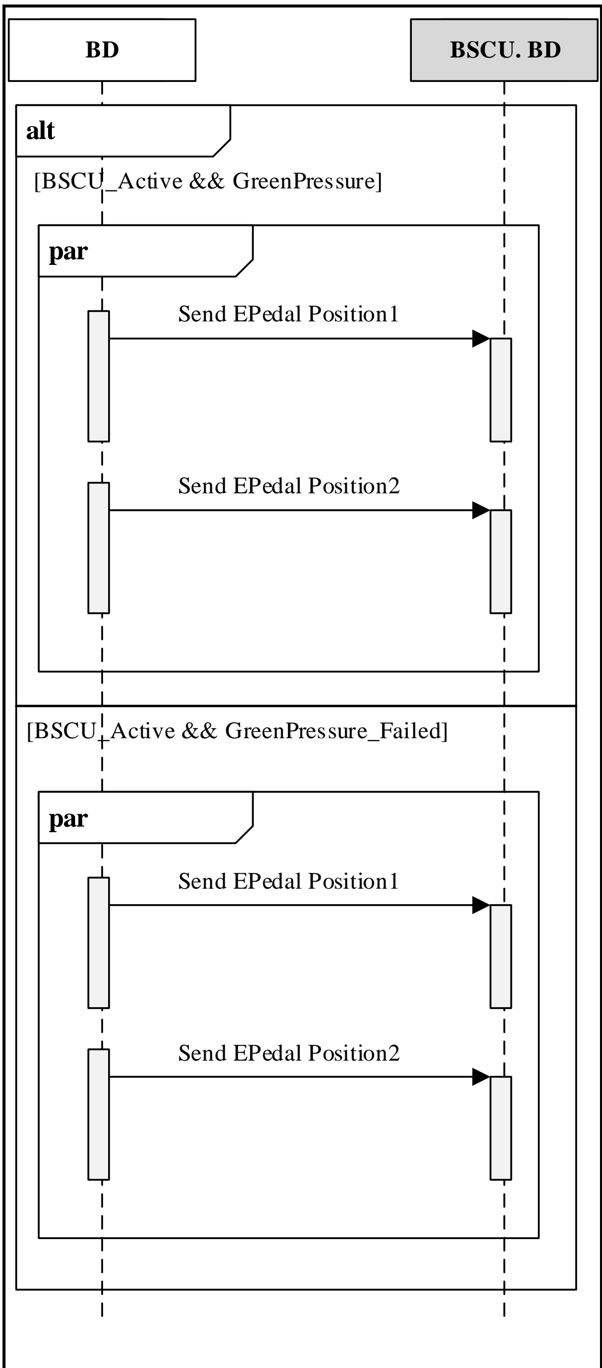


Figure 27: WBS component Electrical_Pedal interactions in transaction domain WheelDecelerationOnGround

The remaining component interactions that are involved in the transaction domain WheelDecelerationOnGround can be found in Appendix E6.

The interface acronyms that have been used in Figure 27 and in the rest of the component interactions (see Appendix E6) are defined in Table 16.

8.3.11 WBS Viewpoint

In accordance with the WBS description discussed in Section 8.2, its architecture consists of the following *viewpoint*:

```
viewpoints {
    WheelDeceleration: {
        Description: "Decelerating the aircraft wheel";
        Transaction Domain: {WheelDecelerationOnGround,
                             WheelDecelerationOnGear};
    }
} // end of viewpoints
```

The viewpoint `WheelDeceleration` demonstrates that two transaction domains need to be viewed or accessed when there is a need to stop/decelerate the commercial aircraft. Transaction domains `WheelDecelerationOnGround` is described in the previous section while `WheelDecelerationOnGear` is beyond the scope of this chapter.

8.3.12 Wheel Brake System (WBS)

This section presents the overall system architecture of the WBS case study explained in Section 8.2. Below is an extract of the WBS architecture:

```
system
 {
  components {
    Selector_Valve<Electrical_Power>: Aircraft_BrakeValve;
    Wheel<>: Aircraft_Wheel;
    Meter_Valve<Electrical_Brake, Mechanical_Brake, Piston_Pressure,
            Accumulator_Pressure, Electrical_Power>:
                                         Aircraft_PressureValve;
```

```
      …
    } // end of components

  connectors { }

  arrangement {
    bind Meter_Valve.BrakePressure with Wheel.InputPressure;
      if (supported(Electrical_Power)) {
        {bind Power.ElectircVoltage with BSCU.ElectircVoltage;
         bind BSCU.CommandNotification with
         hutoff_Valve.CommandNotification;
         bind BSCU.AntiskidCommand with
         Meter_Valve.AntiskidCommand;
        }
       …
    } // end of arrangement

   viewpoints {
     WheelDeceleration;
   } // end of viewpoints
  } // end of WBS
```

For a complete textual description of the WBS architecture, please refer to Appendix E7. Figure 28 demonstrates the graphical structural notation of the whole system.

The acronyms that have been used to demonstrate the component interfaces in Figure 28 are defined in Table 16.

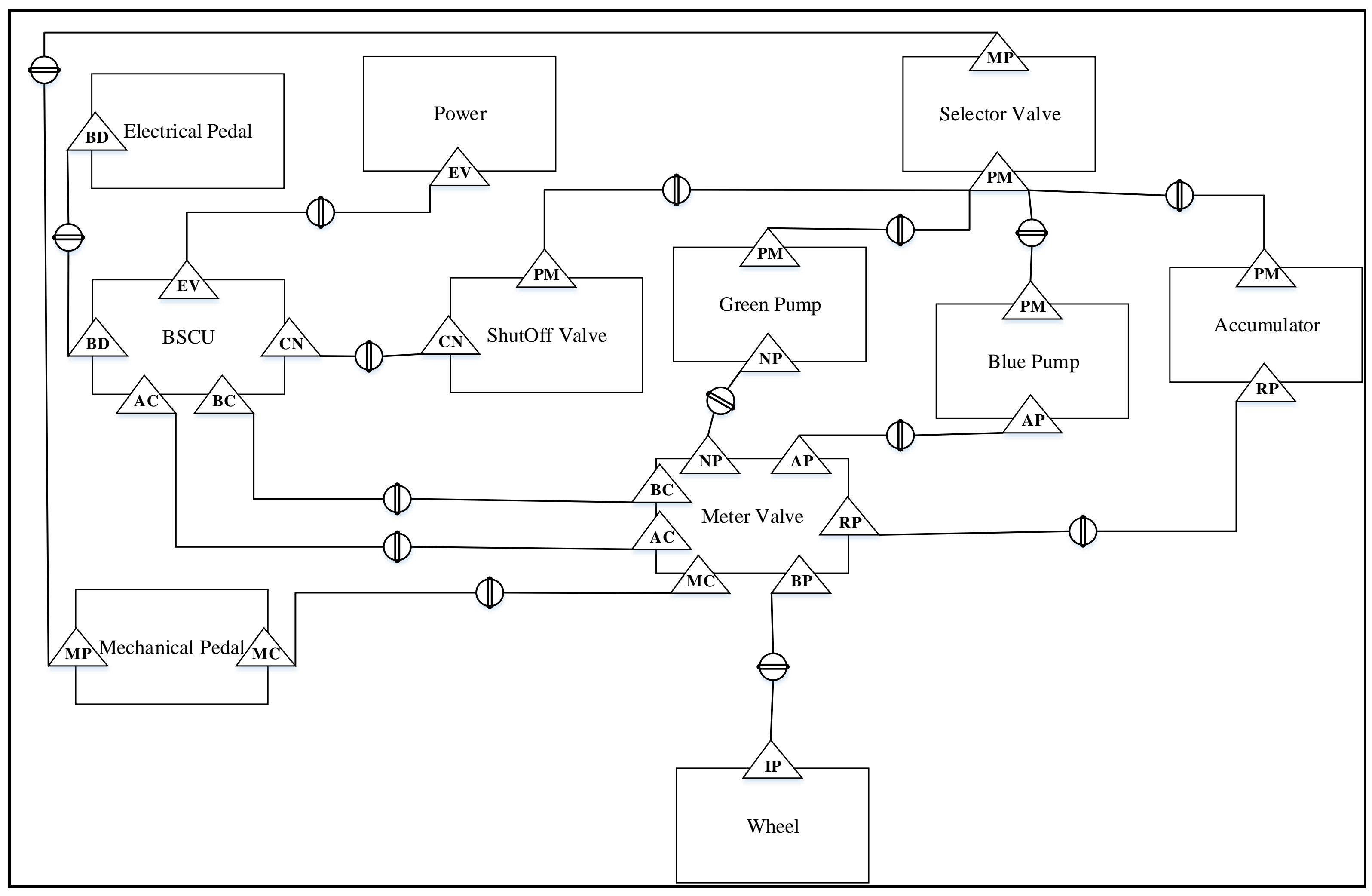


Figure 28: WBS graphical structural notation

8.4 WBS Evaluation

In this section, the WBS architecture model designed in the previous section is evaluated on how it overcomes the limitations that exist in architectural languages and how it addresses the principles, on which ALI V2 is based. Table 17 presents the evaluation of the WBS case study.

Limitations Addressed *(Keywords)*	**CASE STUDY: Wheel Brake System (WBS)**	**ALI V2 Principles Used *(Keywords)***
L1 (Variability)	According to the WBS case study described in Section 8.2, variability occurs when the wheels of the commercial aircraft are stopped/decelerated in different braking modes. From this perspective, the following variabilities were identified: -*<u>Variable features</u>*: `Electrical_Brake`, `Mechanical_Brake`, `Electrical_Power`, `Piston_Pressure` and `Accumulator_Pressure`. -*<u>Variable conditions</u>*: All the conditions defined in Section 8.3.8.	P1 (Variability)
L2 (Traceability)	From the structural aspect of the WBS, the features `Electrical_Brake`, `Electrical_Power` and `Piston_Pressure` represent the requirement and its traceability when the brake is applied in the *NORMAL* mode in the `system` description (Section 8.3.12). When brake is applied in the *ALTERNATE* modes, the features `Mechanical_Brake Piston_Pressure` and/or `Electrical_Power` represent the requirement and its traceability. And, when it is applied in an *EMERGENCY* mode, the features `Mechanical_Brake` and `Accumulator_Pressure` represent the requirement and its traceability. From the behavioural aspect of the WBS, the conditions `BSCU_Active` and `GreenPressure` demonstrates the *NORMAL* braking mode while `BSCU_Failed` and/or `GreenPressure_Failed` demonstrates the *ALETRNATE*	P2 (Traceability)

Limitations Addressed *(Keywords)*	CASE STUDY: Wheel Brake System (WBS)	ALI V2 Principles Used *(Keywords)*
	braking modes. An *EMERGENCY* braking mode is demonstrated by the condition `BluePressure_Failed`. These conditions are represented in the transaction domain `WheelDecelerationOnGround` (Section 8.3.10).	
L3 (Dependency)	Not applicable.	P3 (Cross Domain)
L4 (Restrictive Syntax)	All the architectural elements defined in the WBS architecture are formal and flexible enough to design, and better support, the `system` description. For example, component type `Aircraft_BrakePedal` (defined in Section 8.3.5) used pre-defined interfaces (`BrakeData, MechanicalPosition` and `MechanicalCommand` of type `DataOperation, ValueOperation` and `CommandOperation,` respectively) instead of defining them within its *definition* section using the interface template `MethodInterface`. The component type `Aircraft_BrakePedal` has the flexibility to define an interface similarly to the method used in the *interface types* section in its *definition* section (as explained in Chapter 6). Similarly, the events described in Section 8.3.7 supports the interface template `MethodInterface` in the transaction domain `WheelDecelerationOnGround`, but it has the flexibility to support other the interface template/s in its notation (as explained in Chapter 6).	P4 (Flexibility & Formality)

Limitations Addressed *(Keywords)*	**CASE STUDY: Wheel Brake System (WBS)**	**ALI V2 Principles Used *(Keywords)***
L5 (Reusability)	Interface template `MethodInterface` and the interfaces of type `MethodInterface` (defined in Section 8.3.3 and 8.3.4, respectively) can be used in any type of system architecture wherever an interface of this type is required. *Component types* (defined in Section 8.3.5) can be easily reused in any type of civil airborne system as part of their wheel brake system to apply the brake on the wheel. As explained in Chapter 6, the system can be adopted by simply mapping their feature set to the required system where it will be deployed. For example, component type `Aircraft_BrakePedal` have features `Electronic_Brake` and `Mechanic_Brake`. These features are one of the methods used to apply the brake on the wheels of the commercial aircraft. The artefact description of the component type `Aircraft_BrakePedal` are defined in such a way that it allows to use it in another system where electrical braking is not supported by simply adopting the feature `Mechanic_Brake` of it. As components are dependent on interfaces, the component type `Aircraft_BrakePedal` should reuse the interfaces from their definition, using pre-defined interface template `MethodInterface` and their corresponding *interface types*. Similarly, instances of other component types that have been defined internally to design the *component type* will also be reused. For example, component type `Aircraft_PressureValve` (defined in	P5 (Reusability)

Limitations Addressed *(Keywords)*	CASE STUDY: Wheel Brake System (WBS)	ALI V2 Principles Used *(Keywords)*
	Appendix E4) used instances of component types `Command_Generator` and `Value_Monitor` in its internal configuration.	
L6 (Information Overload)	In the WBS architecture, in order to stop/decelerate the commercial aircraft wheels, the transaction domain `WheelDecelerationOnGround` is defined textually in Section 8.3.10 and presented graphically using event traces in Figure 26 to illustrate its behavioural description. It is defined textually in the `system` description (Section 8.3.12) and presented graphically in Figure 28 as its structural description. In addition, the sequential interaction of all the components involved in the transaction domain `WheelDecelerationOnGround` (such as component `Electrical_Pedal` in Figure 27) are presented. Thus, the WBS architecture provides multiple architectural views of a particular function of the WBS (as a *transaction domain*) with a clear separation between structural and behavioural descriptions while maintaining consistency between them. These different views capture the complexity of the WBS that can cater to different stakeholders, depending upon their concern.	P6 (Multiview)

Limitations Addressed *(Keywords)*	CASE STUDY: Wheel Brake System (WBS)	ALI V2 Principles Used *(Keywords)*
L7 (Behavioural)	Considering the behavioural aspect of the WBS architecture, events such as `HydraulicPressureRequest`, `AntiSkid`, etc., have been defined clearly, with their source and destination *interface templates* specified in Section 8.3.7. From the behavioural visualisation perspective, the transaction domains `WheelDecelerationOnGround` is presented in the form of event traces that demonstrate the ways an event can occur to stop/decelerate the commercial aircraft wheels, as demonstrated in Figure 26. For example, in transaction domain `WheelDecelerationOnGround`, `Reserve_Pressure_Request` and `MPedal_Position_Request` are the events that depends on the conditions `BluePressure_Failed`. This is represented using AND Fork notation (as defined in Chapter 6) which means that it can occur concurrently in the *EMERGENCY* braking mode. Similarly, other event trace notations described in Chapter 6 are used while designing this transaction domain. Moreover, the interactions of all the components that are involved in the transaction domain `WheelDecelerationOnGround` are explicitly presented using a UML sequence diagram. These aspects demonstrate the detailed behavioural description of the WBS architecture.	P6 (Multiview)

Table 17: WBS evaluation

8.5 Discussion

This section further discusses how ALI V2 attempts to reconcile the competing principles (as discussed on Chapter 6) for the language in the context of the WBS case study.

WBS is a single product system with multiple variants as defined in Section 8.3.6 that specifies different modes of how brakes can be applied to wheels of commercial aircrafts in order to stop/decelerate them on the ground. It corresponds to the Embedded Systems application domain.

From the structural design perspective of the WBS, the connections made between components are done via direct binding without the use of connectors because of its embedded nature. This is visualised using the WBS graphical structural notation in Figure 28. In that design, it captures the complete structural information of the WBS architecture in a single view (i.e. an overall system architecture). This is due to the simpler structural architecture having fewer components with the simpler internal configuration.

Moreover, majority of the component interfaces are variable depending upon the mode in which brake is applied to the wheels of the commercial aircraft. Variability to manage these interfaces has been easily achieved using the ALI V2 architectural description (design principle P1).

In spite of the simpler structural elements, WBS architecture has a sophisticated behavioural architecture leading to complex interactions within the transactions. This is visualised using the WBS graphical behavioural notation in the form of event traces in Figure 26. Such interactions between components take place often with multiple events flowing concurrently within them. Thus, it demonstrates that ALI V2 has the ability to

provide the right mechanisms to capture the behavioural complexity as needed by overcoming the limitation (L7) with the principle (design principle P6).

Additionally, architectural elements (such as components and interfaces) defined in the WBS architecture can be reused with minimal, or no, changes to their internal description in other civil airborne systems, as mentioned in Table 17. This is due to the granularity and reusability with the support of capturing variability in ALI V2 as per design principle P5.

Part V

CONCLUSION

Chapter Nine

Conclusion and Future Perspectives

9

"A conclusion is the place where you got tired thinking."

--Martin H. Fischer

"The empires of the future are the empires of the mind."

--Winston Churchill

9.1 Summary and Conclusion

The contribution of the research work described in this thesis is threefold: First, it identified the available approaches that represent variability in software architecture by analysing the current state-of-the-art through a well-defined and methodical way known as a Systematic Literature Review (SLR). Second, an existing ALI ADL was redesigned to capture the variability in a comprehensive way (covering both structural and behavioural aspects of the system) along with other essential functionalities (such as reusability and multiple architectural views). Furthermore, it addressed the challenges that were confining industrial practitioners from adopting the existing ADLs into their practice. Finally, the evaluation of the new version of ALI (referred to as ALI V2) was done using the two case studies: Asset Management System (AMS) and Wheel Brake System (WBS).

The findings of the SLR were presented in a form that makes it accessible to practitioners working in the area who are looking to choose the best variability approach that fits their design needs. In addition, it will benefit researchers trying to identify areas that require further exploration.

As a formalised notation, ALI ADL was chosen because it had already been developed in an initial form within our research group. Moreover, an effort to develop ALI emerged from the belief that a formal notation that can provide high level of flexibility in the architectural designing can make an important contribution to a streamlined. Thus, by tapping on its strength, an appropriate restructuring and refinement of the language was made and a new version of the notation (referred to as ALI V2) was developed. Then, the two real-life case studies were carried out using ALI V2 concepts to demonstrate the better understanding of its constructs and notations.

To summarise this thesis, Part I presented the motivating factors and research questions that led to this research along with the original contributions made in this work. Subsequently, the adopted research methodology was described to carry out this research.

In Part II, ADLs that exists in the research literature were analysed in detail. The main limitations were identified from those ADLs that have emerged from academic research, but have failed to achieve any notable industrial adoption. This was followed by the approaches that represent variability in software architecture being identified through the SLR.

Part III described the original version of ALI before this research started along with the rationale behind it. This part also introduced a new enhanced version of ALI (i.e. ALI V2), which addressed the limitations identified in its initial version and also considered the current industrial requirements. Like behavioural description notations, architectural artefact reusability concepts and multiple architectural views (both textually and graphically) were introduced. Finally, in order to gain hands-on experience with the ALI V2 notations, the two case studies were presented in Part IV.

The results achieved in this research work in the context of the corresponding research questions (described in Chapter 1) are defined as the following:

RQ1: *What approaches have been proposed to represent the variability in software architecture and what are the limitations of these approaches?*

A number of different approaches that represent variability in software architecture were identified through the SLR. Among those, UML (a semi-formal notation) and ADLs (a formal notation) seemed to be the most commonly used approaches for capturing variability at an architectural level. UML was used in almost half of the selected primary studies (via SLR review protocol), and it was extended through various mechanisms to support variability. In addition, ADLs were mostly used in the product line domain.

Furthermore, the work on variability representation at the software architecture level has been largely mapped into three main research areas: Software Product Lines (SPL); Reference Architecture; and Service Oriented Architecture (SOA). The majority of the work conducted in representing variability in software architecture was academically led, and much of it had a fairly theoretical focus.

The limitations that exist in these approaches were technical limitations with the research methodology adopted (for example, some papers only used one case study), technical limitations with the approach presented (for example, only addressing variability at either design time or runtime), and the combination of both limitations.

RQ2: *How can variability be represented formally throughout the architectural description? Furthermore, how will this representation assist in addressing the system's stakeholder concerns, particularly in large-scale industrial systems?*

To manage the size and complexity of the system, ALI V2 considered variability in its architectural description as a first class citizen and as an integral part of the language. The architectural design notations of ALI V2 (defined in Chapter 6) have the capability

to manage variability in the designing of the overall system architecture while designing the individual architectural elements.

ALI V2 captures variability in its structural architectural elements (interfaces, connectors and components) and in the overall system designing through the if/else structure concept with the keywords "**supported**" and "**unsupported**". The if/else structure (similar to the concept of programming language) is used in its behavioural description (transaction domain).

Within the if/else structure, ALI V2 supports the linkage of end-user *features* and environmental *conditions* from the structural and behavioural aspect of the system, respectively. This approach addresses the system's stakeholder concerns by tracing the requirements and increasing the architectural artefact reusability with the support of managing variability.

RQ3: *Which architectural description constructs (textual and graphical) are required to best capture system behaviour, while maintaining support for variability?*

ALI V2 architectural description provides explicit constructs to describe the behavioural aspect of the system. The ALI V2 notation for the constructs -*transaction domain* (textually) and *event traces* (graphically) describes an architectural behavioural description of a particular system function. Variability is captured while using the *conditions* construct within the if/else structure as explained in RQ2.

Also, individual component interactions within a transaction domain are presented through the UML sequence diagram. The *events* construct is also explicitly defined which creates the interaction between the components and *scenarios* that describes system behaviour in various contexts.

RQ4: *How can ADLs be extended to support system modelling that spans multiple application domains?*

In order to illustrate how the proposed ADL (ALI V2) can be applied, two case studies were used to demonstrate the concepts of ALI V2 (described in Chapter 7 and Chapter 8, respectively). The two case studies were selected from two distinct application domains, namely Information Systems (Asset Management System -AMS) and embedded systems (Wheel Brake System -WBS) to illustrate ALI V2's cross application domain modelling capabilities. Moreover, a number of other criteria were considered to select the case studies, including existence of inherent variability in the application domain, different types of connectivity between the components, complexity (information overload), varied emphasis on behavioural versus structural descriptions, potential for artefact reusability within the case study, and access to full technical details.

Criteria	**Case Studies**	
	AMS	**WBS**
Existence of inherent variability	High	Low
Types of connectivity	With connectors	Direct binding
Level of complexity (overall)	High	Low
Level of complexity (structural)	High	Low
Level of complexity (behavioural)	Low	High
Artefact reusability	Medium	Medium

Table 18: Case studies criteria

Table 18 demonstrates the comparison between the two case studies against the selection criteria for the case studies.

9.2 Future Perspectives

ALI V2 represents a significant benefit for software architects, but while addressing the research questions for this work, a number of new tasks and challenges were identified. These are summarized in this section and can be explored further as future perspectives for research.

The future perspectives are categorised into two parts, short term and long term, which are as follows:

9.2.1 Short Term

The short-term future perspective aims to address issues directly related to the work conducted throughout this research. They are:

- ***Architecture evaluation in a broader scope:*** Architecture evaluation is one of the important fields in which we plan to place top priority for further research. The evaluation of ALI V2 will be done in a broader scope by applying it to several additional types of case studies. The aim will be to demonstrate the clear applicability of all the ALI V2 constructs and notations that were not covered in Part IV of this thesis, such as application of *pattern templates* notation and simultaneous use of all types of architectural structuring (i.e., using connectors, patterns and direct connection via component interfaces) in one system.

  To accomplish this, along with its application into other real-life case studies, ALI V2 will be particularly evaluated within Cyber Physical System (CPS) and the Internet of Things (IoT), where the system crosscuts different domains. Information systems (IS) and embedded systems are the domains involved in IoT that exchange data between smart devices, and seeking access to such systems

from potential industrial partners in order to evaluate ALI V2 on real-life industrial applications is under way.

- ***Increased level of reusability architectural artefacts:*** To rapidly use already designed components and connectors (along with their interface descriptions) in system designing, our aim is to provide a formal syntactical definition of features, defined in *component type* and *connector type* notations in Chapter 6, by replacing the feature descriptions with simple textual descriptions. This could be done in the form of defining some attributes/parameters related to features along with some matching constraints so that software architects can easily recognise component/connector functionality they want to use and their level of compatibility into a system. This design strategy can be accomplished by taking some concepts from *interface template* notation (defined in Chapter 6).

- ***Tool support:*** Tool support is an important factor for successful industrial adoption of a language or process, so it needs to be developed to support the creation of ALI V2 descriptions and their transformations to design-level descriptions. The aim is to develop the ALI V2 tool in collaboration with industrial partners by considering their requirements from tool support perspectives. Particularly, those industrial partners will be approached where ALI V2 architectural descriptions have already been evaluated, as demonstrated in Part IV of this thesis.

  To achieve this, first a parser will be developed to provide a complete textual editor for ALI V2. Then, an open source tool platform will provide full support for the ALI V2 conceptual model. The tool will be intended for both end users and tool developers (somewhat similar to the concept of the AADL tool, OSATE).

9.2.2 Long Term

The longer-term future perspective goals focus on the broader aims, future research policies and vision are as follows:

- ***Architecture abstraction levels:*** To enhance the capability of multiple architectural views in ALI V2 and to handle the complexity of the system (particularly, large-scale software system), the concept of setting different levels in order to abstract the system architecture will be introduced. These levels will be set in the ALI V2 architectural description (from both structural and behavioural aspects) and in its tool as well.

  For example, when a component *A* interacts with component *B* (let's say a database component). At the first level, the event `access_database` to be visible and flow from *A* to *B* will be set. Then, in the second level, the events `search_database`, `update_database`, etc., to be visible and their flow between those components will be set. In parallel, the levels of components and the connectors associated with them to be displayed from the structural aspect will also be set. All the corresponding elements visible in a particular level will be hidden in other levels (both before and after abstraction levels) if they are not playing any role in that level description.

  This approach will be similar to the concept of using Google Maps. Let's say to locate a particular city in the world, we can find it by moving from continent (level one) to country (level two) and then to city (level three) by simply tapping it. This will be the visual representation in terms of application in the ALI V2 tool, but our intention is one step ahead of this: to present the textual notation in ALI V2 ADL.

- ***Energy-aware ADL:*** Another potential route with this research, once the textual and graphical notations of ALI V2 reach a stable version (after the successful application of the prior future perspectives), is to make ALI V2 an energy-aware ADL.

  As architectural design decisions decisively impact the energy aware software systems, an energy awareness–related constraints will be designed in the form of ALI V2 constructs and notations. This will be done by adopting concepts from existing energy consumption approaches or models that have been successfully implemented in other phases of software development.

  In addition to this, our plan is to investigate whether an energy aware design approach at the architectural level (particularly, in an ADL) should be considered, either implicitly or explicitly, by practitioners and experts. Moreover, to identify their requirements in relation to this aspect will be considered. Currently, the process to identify the potential industrial partners for that reason are under consideration, because their feedback will be an important factor in making such decisions.

- ***Architecture-focused testing:*** In order to gain confidence in the quality of the software system, including its architecture, the best approach is to perform a thorough analysis, such as via software testing. A well-designed ADLs represent significant opportunities for testing because they formally describe how a software system is expected to behave in a high-level view that allows test engineers to focus on system structure.

  Currently, we are working on a technique to develop test cases at the architectural level based on existing state-based testing algorithms using a well-

known ADL, Architecture Analysis and Design Language (AADL) (Feiler, Gluch and Hudak, 2006). Along with this, defining testability profiles (developed by the Software Engineering Institute [SEI]) on existing AADL designs is in progress. This work is on-going in collaboration with the Strategic Software Engineering Research Group at Clemson University, USA.

Once experience is gained with AADL, a similar approach will be applied by providing a testing mechanism specifically design for ALI V2 architectural descriptions. This perspective would make ALI V2 the first complete and powerful language that practitioners can apply into their systems without any doubt of their system failure.

It is also important to clarify here that the order in which perspectives were defined will be worked out in the future accordingly to that order, excluding architecture evaluation, which is a recurring prospective throughout the research.

References

Abu-Matar, M. and Gomaa, H. (2011) 'Variability Modeling for Service Oriented Product Line Architectures', *Proceedings of the 15th International Software Product Line Conference (SPLC),* pp. 110-119.

Adjoyan, S. and Seriai, A. (2015) 'An Architecture Description Language for Dynamic Service-Oriented Product Lines', *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Pittsburgh, USA.

Ahn, H. and Kang, S. (2011) 'Analysis of Software Product Line Architecture Representation Mechanisms', *Proceedings of the Ninth International Conference on Software Engineering Research, Management and Applications*. Baltimore, MD. pp. 219-226.

Albassam, E. and Gomaa, H. (2013) 'Applying software product lines to multiplatform video games', *Proceedings of the 3rd International Workshop on Games and Software Engineering (GAS)*, pp. 1-7.

Allen, R., Douence, R. and Garlan, D. (1998) 'Specifying and Analyzing Dynamic Software Architectures', Astesiano, E. (ed.) *Proceedings of the First International Conference on Fundamental Approaches to Software Engineering, FASE*. Springer Berlin Heidelberg, pp. 21-37.

Allen, R. and Garlan, D. (1997) 'A formal basis for architectural connection', *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3), pp. 213-249. doi: 10.1145/258077.258078.

Alloui, I. and Oquendo, F. (2002) 'Supporting Decentralised Software-Intensive Processes Using ZETA Component-Based Architecture Description Language.'. in J. Filipe, B. Sharp and Miranda, P. (eds.) *Enterprise Information Systems III.* pp 97-106.

Andersson, J. and Bosch, J. (2005) 'Development and use of dynamic product-line architectures', *IEE Proceedings -Software*, 152(1), pp. 15-28. doi: 10.1049/ip-sen:20041007.

Asikainen, T., Männistö, T. and Soininen, T. (2007) 'Kumbang: A domain ontology for modelling variability in software product families', *Adv. Eng. Inform.*, 21(1), pp. 23-40. doi: 10.1016/j.aei.2006.11.007.

Angelopoulos, K., Souza, V. E. S. and Mylopoulos, J. (2015) 'Capturing Variability in Adaptation Spaces: A Three-Peaks Approach', Johannesson, P., Lee, L.M., Liddle, W.S., Opdahl, L.A. and Pastor López, Ó. (eds.). *Proceedings of the 34th International Conference on Conceptual Modeling (ER 2015)*. Stockholm, Sweden. Springer International Publishing, pp. 384-398.

Auguston, M. (2009) 'Monterey Phoenix, or how to make software architecture executable', *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. Orlando, Florida, USA. ACM, pp. 1031-1040.

Bachmann, F. and Bass, L. (2001) 'Managing variability in software architectures', *SIGSOFT Softw. Eng. Notes*, 26(3), pp. 126-132. doi: 10.1145/379377.375274.

Barbosa, E. A., Batista, T., Garcia, A. and Silva, E. (2011a) 'PL-AspectualACME: an aspect-oriented architectural description language for software product lines'. *Proceedings of the 5th European conference on Software architecture*. Essen, Germany. 2041808: Springer-Verlag, pp. 139-146.

Barbosa, E. A., Batista, T., Garcia, A. and Silva, E. (2011b) 'PL-AspectualACME: an aspect-oriented architectural description language for software product lines'. *Proceedings of the 5th European conference on Software architecture (ECSA)*. Essen, Germany. 2041808: Springer-Verlag, pp. 139-146.

Bashroush, R. (2010) 'A NUI Based Multiple Perspective Variability Modeling CASE Tool'. in Babar, M.A. and Gorton, I. (eds.) *Software Architecture.* Springer Berlin Heidelberg, 55 55 pp 523-526.

Bashroush, R., Brown, T. J., Spence, I. and Kilpatrick, P. (2005) 'ADLARS: An Architecture Description Language for Software Product Lines', *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop*, pp. 163-173.

Bashroush, R., Spence, I., Kilpatrick, P. and Brown, J. (2006) 'Towards more flexible architecture description languages for industrial applications', *Proceedings of the Third European conference on Software Architecture*. Nantes, France. 2081985: Springer-Verlag, pp. 212-219.

Bashroush, R., Spence, I., Kilpatrick, P., Brown, T. J., Gilani, W. and Fritzsche, M. (2008) 'ALI: An Extensible Architecture Description Language for Industrial Applications', *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, ECBS*. 1396050: IEEE Computer Society, pp. 297-304.

Bass, L., Clements, P. and Kazman, R. (2012) *Software Architecture in Practice.* 3rd edn. Addison-Wesley Professional.

Bastarrica, M. C., Rivas, S. and Rossel, P. O. (2007) 'From a Single Product Architecture to a Product Line Architecture', *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science, SCCC*. pp. 115-122.

Booch, G., Rumbaugh, J. and Jacobson, I. (2005) *The Unified Modeling Language User Guide, 2nd Edition (Addison-Wesley Object Technology Series).* Addison-Wesley Professional.

Brito, P. H. S., Rubira, C. M. F. and de Lemos, R. (2009) 'Verifying architectural variabilities in software fault tolerance techniques', *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture*. pp. 231-240.

Brown, T. J., Gawley, R., Bashroush, R., Spence, I., Kilpatrick, P. and Gillan, C. (2006) 'Weaving behavior into feature models for embedded system families', *Proceedings of the 10th International Software Product Line Conference (SPLC)*. pp. 52-61.

Canal, C., Pimentel, E. and Troya, J. M. (1999) 'Specification and Refinement of Dynamic Software Architectures', *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA)*. San Antonio, Texas USA. pp. 107-126.

Carvalho, S. T., Murta, L. and Loques, O. (2012) 'Variabilities as first-class elements in product line architectures of homecare systems', *Proceedings of the 4th International Workshop on Software Engineering in Health Care (SEHC)*. pp. 33-39.

Cassou, D., Bertran, B., Loriant, N. and Consel, C. (2009) 'A generative programming approach to developing pervasive computing systems', *ACM SIGPLAN Notices*, 45(2), pp. 137-146. doi: 10.1145/1837852.1621629.

Cavalcante, E., Medeiros, A. L. and Batista, T. (2013) 'Describing Cloud Applications Architectures', *Proceedings of the 7th European conference on Software Architecture, ECSA*. Montpellier, France. Springer-Verlag, pp. 320-323.

Chang, C. K. and Seongwoon, K. (1999) 'I³: a Petri-net based specification method for architectural components', *Proceedings of the Twenty-Third Annual International Computer Software and Applications Conference, COMPSAC*. pp. 396-402.

Chaudet, C. and Oquendo, F. (2000) 'pi-SPACE: a formal architecture description language based on process algebra for evolving software systems', *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, ASE*. pp. 245-248.

Chen, L., Babar, M. A. and Ali, N. (2009) 'Variability management in software product lines: a systematic review', *Proceedings of the 13th International Software Product Line Conference*. San Francisco, California. 1753247: Carnegie Mellon University, pp. 81-90.

Clements, P. C. (1996) 'A Survey of Architecture Description Languages', *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE Computer Society, pp. 16-25.

Coelho, K. and Batista, T. (2011) 'From Requirements to Architecture for Software Product Lines', *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. pp. 282-289.

Cuenot, P., Frey, P., Johansson, R., Lönn, H., Papadopoulos, Y., Reiser, M.-O., . . . Weber, M. (2010) 'The EAST-ADL Architecture Description Language for Automotive Embedded Software', Giese, H., Karsai, G., Lee, E., Rumpe, B. and Schätz, B. (eds.). *Proceedings of the International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*. Springer Berlin Heidelberg, pp. 297-307.

Dai, L. (2009) 'Security Variability Design and Analysis in an Aspect Oriented Software Architecture', *Proceedings of the Third IEEE International Conference on Secure Software Integration and Reliability Improvement*. 1685781: IEEE Computer Society, pp. 275-280.

Dashofy, E. M., van der Hoek, A. and Taylor, R. N. (2005) 'A comprehensive approach for the development of modular software architecture description languages', *ACM Transactions on Software Engineering Methodology*, 14, pp. 199-245.

Davy Su, Bruno De Fraine and Vanderperren, W. (2005) 'FuseJ: An architectural description language for unifying aspects and components', *Proceedings of the AOSD 2005 Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT).* pp. 1-8.

de Moraes, A. L. S., de C Brito, R., Contieri, A. C., Ramos, M. C., Colanzi, T. E., de S Gimenes, I. M. and Masiero, P. C. (2010) 'Using Aspects and the Spring Framework to Implement Variabilities in a Software Product Line', *Proceedings of the XXIX International Conference of the Chilean Computer Science Society (SCCC)*. pp. 71-80.

Dhungana, D., Neumayer, T., Grünbacher, P. and Rabiser, R. (2008) 'Supporting the Evolution of Product Line Architectures with Variability Model Fragments', *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture, WICSA*. pp. 327-330.

Diaz, J., Perez, J., Fernandez-Sanchez, C. and Garbajosa, J. (2013) 'Model-to-Code Transformation from Product-Line Architecture Models to AspectJ', *Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. pp. 98-105.

Dobrica, L. and Niemelä, E. (2008) 'An Approach to Reference Architecture Design for Different Domains of Embedded Systems', *Proceedings of the Software Engineering Research and Practice*. pp. 287-293.

Duran-Limon, H. A., Garcia-Rios, C. A., Castillo-Barrera, F. E. and Capilla, R. (2015) 'An Ontology-Based Product Architecture Derivation Approach', *IEEE Transactions on Software Engineering*, 41(12), pp. 1153-1168. doi: 10.1109/TSE.2015.2449854.

Eklund, U., Askerdal, Ö., Granholm, J., Alminger, A. and Axelsson, J. (2005) 'Experience of introducing reference architectures in the development of automotive electronic systems', *Proceedings of the second international workshop on Software engineering for automotive systems*. St. Louis, Missouri. 1083195: ACM, pp. 1-6.

Feiler, P. H., Gluch, D. P. and Hudak, J. J. (2006) *The Architecture Analysis & Design Language AADL: An Introduction.* Pittsburgh, USA.: *Software Engineering Institute, Carnegie Mellon University*.

Galster, M. and Avgeriou, P. (2011) 'Handling Variability in Software Architecture: Problems and Implications', *Proceedings of the Ninth Working IEEE/IFIP Conference on Software Architecture*. 2015583: IEEE Computer Society, pp. 171-180.

Galster, M. and Avgeriou, P. (2011) 'The notion of variability in software architecture: results from a preliminary exploratory study', *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. Namur, Belgium. 1944899: ACM, pp. 59-67.

Galster, M., Avgeriou, P. and Tofan, D. (2013) 'Constraints for the design of variability-intensive service-oriented reference architectures – An industrial case study', *Information and Software Technology*, 55(2), pp. 428-441.

Galster, M., Männistö, T., Weyns, D. and Avgeriou, P. (2014a) 'Variability in software architecture: the road ahead', *ACM SIGSOFT Software Engineering Notes*, 39(4), pp. 33-34.

Galster, M., Weyns, D., Tofan, D., Michalik, B. and Avgeriou, P. (2014b) 'Variability in Software Systems - A Systematic Literature Review.', *IEEE Transaction Software Engineering*, 40(3), pp. 282-306.

Garcia, A., Chavez, C., Batista, T., Sant'anna, C., Kulesza, U., Rashid, A. and Lucena, C. (2006) 'On the Modular Representation of Architectural Aspects', Gruhn, V. and Oquendo, F. (eds.). *Proceedings of the Third European Workshop on Software Architecture, EWSA*. Springer Berlin Heidelberg, pp. 82-97.

Garlan, D. (2014) 'Software architecture: a travelogue', *Proceedings of the on Future of Software Engineering*. Hyderabad, India. ACM, pp. 29-39.

Garlan, D., Allen, R. and Ockerbloom, J. (1994) 'Exploiting style in architectural design environments', *SIGSOFT Softw. Eng. Notes*, 19(5), pp. 175-188. doi: 10.1145/195274.195404.

Garlan, D., Monroe, R. and Wile, D. (1997) 'Acme: an architecture description interchange language', *Proceedings of the Centre for Advanced Studies on Collaborative research, CASCON'*. pp. 169-183.

Gomaa, H. (2013) 'Evolving software requirements and architectures using software product line concepts', *Proceedings of the 2nd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*. pp. 24-28.

Gorlick, M. M. and Razouk, R. R. (1991) 'Using weaves for software construction and analysis', *Proceedings of the 13th International Conference on Software Engineering, ICSE*. pp. 23-34.

Groher, I. and Weinreich, R. (2013) 'Strategies for Aligning Variability Model and Architecture', *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC)*. pp. 511-516.

Groher, I. and Weinreich, R. (2013) 'Supporting Variability Management in Architecture Design and Implementation', *Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS)*. pp. 4995-5004.

Haber, A., Hölldobler, K., Kolassa, C., Look, M., Müller, K., Rumpe, B. and Schaefer, I. (2013) 'Engineering Delta Modelling Languages'. *Proceedings of the 17th International Software Product Line Conference (SPLC)*. Tokyo, Japan. pp. 22-31.DOI: 10.1145/2491627.2491632.

Haber, A., Kutz, T., Rendel, H., Rumpe, B. and Schaefer, I. (2011a) 'Delta-oriented architectural variability using MontiCore', *Proceedings of the 5th European Conference on Software Architecture ECSA*. Essen, Germany. ACM New York, NY, USA, pp. 1-10.

Haber, A., Rendel, H., Rumpe, B. and Schaefer, I. (2011b) 'Delta Modeling for Software Architectures', *Proceedings of the Dagstuhl Workshop on Model-Based Development of Embedded Systems MBEES*. Germany.

Haber, A., Rendel, H., Rumpe, B., Schaefer, I. and van der Linden, F. (2011c) 'Hierarchical Variability Modeling for Software Architectures', *Proceedings of the 15th International Software Product Line Conference (SPLC)*. pp. 150-159.

Haber, A., Ringert, J. O. and Rumpe, B. (2012) *MoniArc - Architectural Modeling of Interactive Distributed ad Cyber-Physical Systems.* Germany: RWTH Aachen University.

Halpin, T. (2010) 'Object-Role Modeling: Principles and Benefits', *International Journal of Information System Modeling and Design*, 1(1), pp. 33-57. doi: 10.4018/jismd.2010092302.

Halpin, T. and Morgan, T. (2008) *Information Modeling and Relational Databases.* Morgan Kaufmann Publishers Inc.

Harel, D. and Rumpe, B. (2004) 'Meaningful modeling: what's the semantics of "semantics"?', *Computer*, 37(10), pp. 64-72. doi: 10.1109/MC.2004.172.

Helleboogh, A., Weyns, D., Schmid, K., Holvoet, T., Schelfthout, K. and Van Betsbrugge, W. (2009) 'Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines',

Proceedings of the Third International Workshop on Dynamic Software Product Lines (DSPL {@} SPLC 2009). Pittsburgh, PA, USA. Carnegie Mellon University, pp. 18-27.

Hoare, C. A. R. (1985) *Communicating sequential processes.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Hoek, A. v. d. (2004) 'Design-time product line architectures for any-time variability', *Sci. Comput. Program*, 53(3), pp. 285-304. doi: 10.1016/j.scico.2003.04.003.

Hwi, A., Sungwon, K. and Jihyun, L. (2013) 'A Case Study Comparison of Variability Representation Mechanisms with the HeRA Product Line', *Proceedings of the IEEE 16th International Conference on Computational Science and Engineering (CSE)*. pp. 416-423.

ARP 4761 (1996) 'ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment'.

ISO/IEC/IEEE (2011), *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1-46. doi: 10.1109/IEEESTD.2011.6129467.

Java Compiler Compiler tm (JavaCC tm). Available at: https://javacc.java.net/.

Johnson, J. and Henderson, A. (2002) 'Conceptual models: begin by designing what to design', *Interactions*, 9(1), pp. 25-32. doi: 10.1145/503355.503366.

Kakarontzas, G., Stamelos, I. and Katsaros, P. (2008) 'Product Line Variability with Elastic Components and Test-Driven Development', *Proceedings of the International Conference on Computational Intelligence for Modelling Control & Automation*. pp. 146-151.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. and Patterson, A. S. (1990) *Feature Oriented Domain Analysis (FODA) feasibility study.* Software Engineering Institute, Carnegie Mellon University CMU/SEI-90-TR-21.

Kim, Y.-G., Lee, S. K. and Jang, S.-B. (2011) 'Variability Management for Software Product-line Architecture Development', *International Journal of Software Engineering and Knowledge Engineering*, 21(07), pp. 931-956. doi:10.1142/S0218194011005542.

Kitchenham, B. and Charters, S. (2007) 'Guidelines for Performing Systematic Literature Reviews in Software Engineering (version 2.3)'.in Technical report, Keele University and University of Durham.

Klien, P. (2010) 'The Architecture Description Language MoDeL'. in Engels, G., Lewerentz, C., Schäfer, W., Schürr, A. and Westfechtel, B. (eds.) *Graph Transformations and Model-Driven Engineering.* Springer Berlin Heidelberg, pp 249-273.

Kruchten, P., Obbink, H. and Stafford, J. (2006) 'The Past, Present, and Future for Software Architecture', *IEEE Software*, 23(2), pp. 22-30. doi: 10.1109/ms.2006.59.

Lago, P., Malavolta, I., Muccini, H., Pelliccione, P. and Tang, A. (2015) 'The Road Ahead for Architectural Languages', *IEEE Software*, 32(1), pp. 98-105. doi: 10.1109/MS.2014.28.

Laser, M. S., Rodrigues, E. M., Domingues, A., Oliveira, F. and Zorzo, A. F. (2015) 'Architectural Evolution of a Software Product Line: an experience report', *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Pittsburgh, USA.

Losavio, F., Ordaz, O., Levy, N. and Baiotto, A. (2013) 'Graph modelling of a refactoring process for Product Line Architecture design', *Proceedings of the XXXIX Latin American Computing Conference (CLEI)*. pp. 1-12.

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D. and Mann, W. (1995) 'Specification and analysis of system architecture using Rapide', *IEEE Transactions on Software Engineering*, 21, pp. 336-355.

Lytra, I., Eichelberger, H., Tran, H., Leyh, G., Schmid, K. and Zdun, U. (2014) 'On the Interdependence and Integration of Variability and Architectural Decisions', *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Sophia Antipolis, France. pp. 1-8.

López, N., Casallas, R. and Hoek, A. v. d. (2009) 'Issues in mapping change-based product line architectures to configuration management systems'. *Proceedings of the 13th International Software Product Line Conference*. San Francisco, California. Carnegie Mellon University, pp. 21-30.

Magableh, B. and Barrett, S. (2010) 'Primitive component architecture description language', *Proceedings of the 7th International Conference on Informatics and Systems (INFOS)*. pp. 1-7.

Magee, J. and Kramer, J. (1996) 'Dynamic structure in software architectures', *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*. San Francisco, California, USA. ACM, pp. 3-14.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P. and Tang, A. (2013) 'What Industry Needs from Architectural Languages: A Survey', *IEEE Transactions on Software Engineering*, 39(6), pp. 869-891. doi: 10.1109/TSE.2012.74.

Mann, S. and Rock, G. (2009) 'Dealing with Variability in ArchitectureDescriptions to Support Automotive Product Lines: Specification and Analysis Methods', *Proceedings of the Embedded World Conference*. Nurnberg, Deutschland. WEKA Fachmedien.

Martin, P. Y. and Turner, B. A. (1986) 'Grounded Theory and Organizational Research', *The Journal of Applied Behavioral Science*, 22(2), pp. 141-157. doi: 10.1177/002188638602200207.

Matt, G. E. and D.Cook, T. (*1994*) 'Threats to the validity of research synthesis'. in Cooper, H. and Hedges, L.V. (eds.) *In the Handbook of Research Synthesis.* New York: Russell Sage Foundation, pp 503-520.

Medvidovic, N. and Taylor, R. N. (2000a) 'A Classification and Comparison Framework for Software Architecture Description Languages', *IEEE Transactions on Software Engineering*, 26(1), pp. 70-93.

Medvidovic, N. and Taylor, R. N. (2000b) 'A Classification and Comparison Framework for Software Architecture Description Languages', *IEEE Trans. Softw. Eng.*, 26(1), pp. 70-93.

Medvidovic, N., Taylor, R. N. and Whithead, E. J. J. (1996) 'Formal modeling of software architectures at multiple levels of abstraction', *Proceedings of the California Software Symposium*. California, USA. pp. 28-40.

Mei, H., Chen, F., Wang, A. Q. and Feng, A. Y.-D. (2002) 'ABC/ADL: An ADL Supporting Component Composition'. *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*. London, UK. Springer-Verlag, pp. 38-47.

Mikyeong, M., Heung Seok, C., Taewoo, N. and Keunhyuk, Y. (2007) 'A Metamodeling Approach to Tracing Variability between Requirements and Architecture in Software Product Lines', *Proceedings of the 7th IEEE International Conference on Computer and Information Technology*. pp. 927-933.

Moon, M., Chae, H. S. and Yeom, K. (2006) 'A metamodel approach to architecture variability in a product line', *Proceedings of the 9th international conference on Reuse of Off-the-Shelf Components*. Turin, Italy. 2172044: Springer-Verlag, pp. 115-126.

Murata, T. (1989) 'Petri nets: Properties, analysis and applications', *Proceedings of the IEEE*, 77(4), pp. 541-580. doi: 10.1109/5.24143.

Myllärniemi, V., Raatikainen, M. and Männistö, T. (2015) 'Representing and Configuring Security Variability in Software Product Lines', *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. Montréal, QC, Canada. pp. 1-10.

Myllärniemi, V., Ylikangas, M., Raatikainen, M., Pääkkö, J., Männistö, T. and Aaltonen, T. (2012) 'Configurator-as-a-service: tool support for deriving software architectures at runtime', *Proceedings of the WICSA/ECSA,* Helsinki, Finland. 2362031: ACM, pp. 151-158.

Navasa, A., Pérez-Toledano, M. A. and Murillo, J. M. (2009) 'An ADL dealing with aspects at software architecture stage', *Information and Software Technology*, 51(2), pp. 306-324. doi: http://dx.doi.org/10.1016/j.infsof.2008.03.009.

Ommering, R. V., van der Linden, F., Kramer, J. and Magee, J. (2000) 'The Koala Component Model for Consumer Electronics Software', *IEEE Computer*, 33, pp. 78-85. doi: 10.1109/2.825699.

Oquendo, F. (2004) 'π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures', *ACM SIGSOFT Softw. Eng. Notes*, 29(3), pp. 1-14.

Ortiz, F. J., Pastor, J. A., Alonso, D., Losilla, F. and de Jódar, E. (2005) 'A reference architecture for managing variability among teleoperated service robots', *Proceedings of the 2nd International Conference on Informatics in Control Automation and Robotics (ICINCO).* pp. 322-328.

P. Armstrong, G. Lowe, J. Ouaknine and Roscoe, A. W. (2012) 'Model checking Timed CSP', Korovina, A.V.a.M. (ed. *Proceedings of the HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*. EasyChair, pp. 13-33.

Binns, P., Englehart, M., Jackson, M. and Vestal, S. (1996) 'Domain-Specific Software Architectures for Guidance, Navigation and Control.', *International Journal of Software Engineering and Knowledge Engineering*, 6(2), pp. 201-227.

Poizat, P., and Royer, J.-C. (2006) 'A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic', *Journal of Universal Computer Science*, 12(12), pp. 1741-1782.

Pascual, G. G., Pinto, M. and Fuentes, L. (2013) 'Run-Time support to manage architectural variability specified with CVL', *Proceedings of the 7th European conference on Software Architecture (ECSA)*. Montpellier, France. pp. 282-298.

Peng, X., Shen, L. and Zhao, W. (2009) 'An Architecture-based Evolution Management Method for Software Product Line', *Proceedings of the* International *Conference on Software Engineering and Knowledge Engineering (SEKE)*. pp. 135-140.

Perez, J., Ramos, I., Jaen, J., Letelier, P. and Navarro, E. (2003) 'PRISMA: towards quality, aspect oriented and dynamic software architectures', *Proceedings of the Third International Conference on Quality Software*. pp. 59-66.

Pinto, M., Fuentes, L. and Troya, J. M. (2003) 'DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development', Pfenning, F. and Smaragdakis, Y. (eds.). *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, GPCE*. Springer Berlin Heidelberg, pp. 118-137.

Pérez, J., Díaz, J., Costa-Soria, C. and Garbajosa, J. (2009) 'Plastic Partial Components: A solution to support variability in architectural components', *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture*. pp. 221-230.

Qureshi, T. N., Chen, D., Lönn, H. and Törngren, M. (2011) 'From EAST-ADL to AUTOSAR software architecture: a mapping scheme'. *Proceedings of the 5th European conference on Software architecture*. Essen, Germany. pp. 328-335.

Rademaker, A., Braga, C. and Sztajnberg, A. (2005) 'A Rewriting Semantics for a Software Architecture Description Language', *Electronic Notes in Theoretical Computer Science*, 130(0), pp. 345-377. doi: http://dx.doi.org/10.1016/j.entcs.2005.03.018.

Razavian, M. and Khosravi, R. (2008) 'Modeling variability in the component and connector view of architecture using UML', *Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications*. 1544448: IEEE Computer Society, pp. 801-809.

Ringert, J. O., Rumpe, B. and Wortmann, A. (2013) 'MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems', *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Karlsruhe, Germany.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and William, L. (1991) *Object-Oriented Modeling and Design.* Prentice-Hall, Inc.

Ruscio, D., Malavolta, I., Muccini, H., Pelliccione, P. and Pierantonio, A. (2010) 'ByADL: An MDE Framework for Building Extensible Architecture Description Languages', Babar, M.A. and Gorton, I. (eds.). *Proceedings of the 4th European Conference on Software Architecture, ECSA*. Springer Berlin Heidelberg, pp. 527-531.

Satyananda, T. K., Danhyung, L. and Sungwon, K. (2007a) 'Formal Verification of Consistency between Feature Model and Software Architecture in Software Product Line', *Proceedings of the International Conference on Software Engineering Advances, ICSEA*.

Satyananda, T. K., Danhyung, L., Sungwon, K. and Hashmi, S. I. (2007b) 'Identifying Traceability between Feature Model and Software Architecture in Software Product Line using Formal Concept Analysis', *Proceedings of the International Conference on Computational Science and its Applications, ICCSA 2007.*, 26-29 Aug. 2007. pp. 380-388.

Savolainen, J., Oliver, I., Mannion, M. and Hailang, Z. (2005) 'Transitioning from product line requirements to product line architecture', *Proceedings of the 29th Annual International Computer Software and Applications Conference, COMPSAC*. pp. 186-195 Vol. 182.

Scowen, R. S. (1993) 'Extended BNF - A Generic Base Standard'. *Proceedings of the Software Engineering Standards Symposium (SESS)*. Brighton, United Kingdom.

Shaw, M. (2001) 'The coming-of-age of software architecture research', *Proceedings of the 23rd International Conference on Software Engineering, ICSE*. pp. 657-664.

Shaw, M. and Clements, P. (2006) 'The golden age of software architecture', *IEEE Software*, 23(2), pp. 31-39. doi: 10.1109/MS.2006.58.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M. and Zelesnik, G. (1995) 'Abstractions for software architecture and tools to support them', *IEEE Transactions on Software Engineering,*, 21(4), pp. 314-335. doi: 10.1109/32.385970.

Silva, E., Medeiros, A. L., Cavalcante, E. and Batista, T. V. (2013) 'A Lightweight Language for Software Product Lines Architecture Description', *Proceedings of the 7th European Conference on Software Architecture, ECSA*. Montpellier, France. Springer, pp. 114-121.

Sinnema, M., Ven, J. S. v. d. and Deelstra, S. (2006) 'Using variability modeling principles to capture architectural knowledge', *SIGSOFT Softw. Eng. Notes*, 31(5), p. 5. doi: 10.1145/1163514.1178645.

Smiley, K., Mahate, S. and Wood, P. (2014) 'A Dynamic Software Product Line Architecture for Prepackaged Expert Analytics: Enabling Efficient Capture, Reuse and Adaptation of Operational Knowledge', *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*. pp. 205-214.

'Software Engineering Institute (SEI) / Carnegie Mellon University (CMU)'. Available at: https://www.sei.cmu.edu/architecture/start/glossary/.

Stéphane Faulkner and Kolp, M. (2003) 'Towards An Agent Architectural Description Language For Information Systems', *Proceedings of the 5th International Conference on Enterprise Information Systems ICEIS*. France. Press, pp. 59-66.

Svahnberg, M. and Bosch, J. (2000) 'Issues Concerning Variability in Software Product Lines', *Proceedings of the International Workshop on Software Architectures for Product Families*. pp. 146-157.

Sánchez, P., Loughran, N., Fuentes, L. and Garcia, A. (2009) 'Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines', Dragan, G., evi, Ralf, L., mmel and Eric, W. (eds.). *Proceedings of the International Conference on Software Language Engineering*. 1532466: Springer-Verlag, pp. 188-207.

Tekinerdogan, B. and Sözer, H. (2012) 'Variability viewpoint for introducing variability in software architecture viewpoints', *Proceedings of the WICSA/ECSA,* Helsinki, Finland. pp. 163-166.

Thiel, S. and Hein, A. (2002a) 'Modeling and Using Product Line Variability in Automotive Systems', *IEEE Softw.*, 19(4), pp. 66-72. doi: 10.1109/ms.2002.1020289.

Thiel, S. and Hein, A. (2002b) 'Systematic Integration of Variability into Product Line Architecture Design', *Proceedings of the Second International Conference on Software Product Lines*. 672257: Springer-Verlag, pp. 130-153.

Ubayashi, N., Nomura, J. and Tamai, T. (2010) 'Archface: a contract place where architectural design and code meet together', *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering, ICSE*. pp. 75-84.

Vestal, S. (1993) *A Cursory Overview and Comparison of Four Architecture Description Languages.* Honeywell Technology Center.

Wang, Z., Peng, H., Guo, J., Zhang, Y., Wu, K., Xu, H. and Wang, X. (2012) 'An Architecture Description Language Based on Dynamic Description Logics', Shi, Z., Leake, D. and Vadera, S. (eds.). *Proceedings of the 7th TC 12 International Conference on Intelligent Information Processing VI*. Springer Berlin Heidelberg, pp. 157-166.

Woods, E. and Bashroush, R. (2015) 'Modelling large-scale information systems using ADLs – An industrial experience report', *Journal of Systems and Software*, 99, pp. 97-108. doi: http://dx.doi.org/10.1016/j.jss.2014.09.018.

Woods, E. and Hilliard, R. (2005) 'Architecture Description Languages in Practice Session Report', *Proceedings of the 5th Working IEEE/IFIP Conference onSoftware Architecture, WICSA*. pp. 243-246.

Xiangyang, J., Shi, Y., Honghua, C. and Xie, D. (2007) 'A New Architecture Description Language for Service-Oriented Architec', *Proceedings of the Sixth International Conference on Grid and Cooperative Computing, GCC*. pp. 96-103.

Yao, C., Xiaoqing, L., Lingyun, Y., Dayong, L., Liu, T. and Hongli, Y. (2010) 'A ten-year survey of software architecture', *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences (ICSESS)*. pp. 729-733.

Oh, Y., Lee, D. H., Kang, S., and Lee, J. H. (2007) 'Extended Architecture Analysis Description Language for Software Product Line Approach in Embedded Systems', *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE*. Nice, France. IEEE, pp. 87-88.

Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J. and Leite, J. C. S. P. (2008) 'From goals to high-variability software design', *Proceedings of the 17th international conference on Foundations of intelligent systems*. Toronto, Canada. 1786476: Springer-Verlag, pp. 1-16.

Zhang, G.-q., Shi, H.-j. and Rong, M. (2011) 'Mismatch Detection of Asynchronous Web Services with Timed Constraints', *Proceedings of the IEEE Asia-Pacific Services Computing Conference (APSCC)*. pp. 251-258.

Zhang, T., Xiang, D. and Wang, H. (2005) 'vADL: A Variability-Supported Architecture Description Language for Specifying Product Line Architectures', *Proceedings of the Second International Software Product Lines Young Researchers Workshop (SPLYR) in conjunction with the 9th International Software Product Line conference (SPLC)*. Rennes, France. pp. 31-37.

Zhu, J., Peng, X., Jarzabek, S., Xing, Z., Xue, Y. and Zhao, W. (2011) 'Improving product line architecture design and customization by raising the level of variability modeling', *Proceedings of the 12th international conference on Top productivity through software reuse*. Pohang, South Korea. Springer-Verlag, pp. 151-166.

Appendices

Appendix A: Systematic Literature Review (SLR)

A1: List of Included Primary Studies in SLR

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S1	Systematic Integration of Variability into Product Line Architecture Design	2002	Thiel, S., Hein, A.	(Thiel and Hein, 2002b)
S2	Modeling and Using Product Line Variability in Automotive Systems	2002	Thiel, S., Hein, A	(Thiel and Hein, 2002a)
S3	Design-time product line architectures for any-time variability	2004	Hoek, A.	(Hoek, 2004)
S4	vADL: A Variability-Supported Architecture Description Language for Specifying Product Line Architectures	2005	Zhang, T., Xiang, D., Wang, H.	(Zhang, Xiang and Wang, 2005)
S5	ADLARS: An Architecture Description Language for Software Product Lines	2005	Bashroush, R., Brown, T. J., Spence, I., Kilpatrick, P.	(Bashroush *et al.*, 2005)
S6	Transitioning from Product Line Requirements to Product Line Architecture.	2005	Savolainen, J., Oliver, I., Mannion, M., Hailang, Z.	(Savolainen *et al.*, 2005)

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S7	A Reference Architecture for Managing Variability among Teleoperated Service Robots	2005	Ortiz, F. J., Pastor, J. A., Alonso, D., Losilla, F., de Jódar, E.	(Ortiz *et al.*, 2005)
S8	Experience of Introducing Reference Architectures in the Development of Automotive Electronic Systems	2005	Eklund, U., Askerdal, O., Granholm, J., Alminger, A., Axelsson, J.	(Eklund *et al.*, 2005)
S9	Development and use of dynamic product-line architectures	2005	Andersson, J., Bosch, J.	(Andersson and Bosch, 2005)
S10	A Metamodel Approach to Architecture Variability in a Product Line	2006	Moon, M., Chae, H. S., Yeom, K.	(Moon, Chae and Yeom, 2006)
S11	Using variability modeling principles to capture architectural knowledge	2006	Sinnema, M., Salvador van der Ven, J., Deelstra, S.	(Sinnema, Ven and Deelstra, 2006)
S12	From a Single Product Architecture to a Product Line Architecture	2007	Bastarrica, M. C., Rivas, S., Rossel, P. O.	(Bastarrica, Rivas and Rossel, 2007)
S13	Kumbang: A domain ontology for modelling variability in software product families	2007	Asikainen, T., Männistö, T., Soininen, T.	(Asikainen, Männistö and Soininen, 2007)

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S14	Identifying Traceability between Feature Model and Software Architecture in Software Product Line using Formal Concept Analysis	2007	Satyananda, T. K., Danhyung, L., Sungwon, K., Hashmi, S. I.	(Satyananda *et al.*, 2007)
S15	A Metamodeling Approach to Tracing Variability between Requirements and Architecture in Software Product Lines	2007	Moon, M., Chae, H. S., Nam, T., Yeom, K.	(Mikyeong *et al.*, 2007)
S16	Formal Verification of Consistency between Feature Model and Software Architecture in Software Product Line	2007	Satyananda, T. K., Danhyung, L., Sungwon, K.	(Satyananda, Danhyung and Sungwon, 2007)
S17	Modeling Variability in the Component and Connector View of Architecture Using UML	2008	Razavian, M., Khosravi, R.	(Razavian and Khosravi, 2008)
S18	ALI: An Extensible Architecture Description Language for Industrial Applications	2008	Bashroush, R., Spence, I., Kilpatrick, P., Brown, T. J., Gilani, W., Fritzsche, M.	(Bashroush *et al.*, 2008)
S19	An Approach to Reference Architecture Design for Different Domains of Embedded Systems	2008	Dobrica, L., Niemelä, E.	(Dobrica and Niemelä, 2008)

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S20	From goals to high-variability software design	2008	Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J., Leite, J. C. S. P.	(Yu *et al.*, 2008)
S21	Supporting the Evolution of Product Line Architectures with Variability Model Fragments	2008	Dhungana, D., Neumayer, T., Grünbacher, P., Rabiser, R.	(Dhungana *et al.*, 2008)
S22	Product Line Variability with Elastic Components and Test-Driven Development	2008	Kakarontzas, G., Stamelos, I., Katsaros, P.	(Kakarontzas, Stamelos and Katsaros, 2008)
S23	Engineering languages for specifying product-derivation processes in Software Product Lines	2009	Sánchez, P., Loughran, N., Fuentes, L., Garcia, A.	(Sánchez *et al.*, 2009)
S24	An Architecture-based Evolution Management Method for Software Product Line	2009	Peng, X., Shen, L., Zhao, W.	(Peng, Shen and Zhao, 2009)
S25	Issues in mapping change-based product line architectures to configuration management systems	2009	López, N., Casallas, R., Hoek, A.	(López, Casallas and Hoek, 2009)

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S26	Adding Variants on-the-fly: Modeling Meta-Variability in Dynamic Software Product Lines	2009	Helleboogh, A., Weyns, D., Schmid, K., Holvoet, T., Schelfthout, K., Van Betsbrugge, W.	(Helleboogh *et al.*, 2009)
S27	Plastic Partial Components: A solution to support variability in architectural components	2009	Pérez, J., Díaz, J., Costa-Soria, C., Garbajosa, J.	(Pérez *et al.*, 2009)
S28	Dealing with variability in architecture descriptions to support automotive product lines: Specification and analysis methods	2009	Mann, S., Rock, G.	(Mann and Rock, 2009)
S29	Verifying Architectural Variabilities in Software Fault Tolerance Techniques	2009	Brito, P. H. S., Rubira, C. M. F., de Lemos, R.	(Brito, Rubira and de Lemos, 2009)
S30	Security Variability Design and Analysis in an Aspect Oriented Software Architecture	2009	Dai, L.	(Dai, 2009)
S31	Using aspects and the Spring framework to implement variabilities in a software product line	2010	de Moraes, A. L. S., de C Brito, R., Contieri, A. C., Ramos, M. C., Colanzi, T. E., de S Gimenes, I. M., Masiero, P. C.	(de Moraes *et al.*, 2010)

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S32	Variability Management for Software Product-line Architecture Development	2011	Kim, Y., Lee, S., Jang, S.	(Kim, Lee and Jang, 2011)
S33	Improving Product Line Architecture Design and Customization by Raising the Level of Variability Modeling	2011	Zhu, J., Peng, X., Jarzabek, S., Xing, Z., Xue, Y., Zhao, W.	(Zhu *et al.*, 2011)
S34	From Requirements to Architecture for Software Product Lines	2011	Coelho, K., Batista, T.	(Coelho and Batista, 2011)
S35	Analysis of Software Product Line Architecture Representation Mechanisms	2011	Ahn, H., Kang, S.	(Ahn and Kang, 2011)
S36	Delta Modeling for Software Architectures	2011	Haber, A., Rendel, H., Rumpe, B., Schaefer, I.	(Haber *et al.*, 2011b)
S37	PL-AspectualACME: an aspect-oriented architectural description language for software product lines	2011	Barbosa, E.A., Batista, T., Garcia, A., Silva, E.	(Barbosa *et al.*, 2011)
S38	Variability Modeling for Service Oriented Product Line Architectures	2011	Abu-Matar, M., Gomaa, H.	(Abu-Matar and Gomaa, 2011)
S39	Hierarchical Variability Modeling for Software Architectures	2011	Haber, A., Rendel, H., Rumpe, B., Schaefer, I., van der Linden, F.	(Haber *et al.*, 2011c)

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S40	Delta-oriented Architectural Variability Using MontiCore	2011	Haber, A., Kutz, T., Rendel, H., Rumpe, B., Schaefer, I.	(Haber *et al.*, 2011a)
S41	Configurator-as-a-service: tool support for deriving software architectures at runtime	2012	Myllärniemi, V., Ylikangas, M., Raatikainen, M., Pääkkö, J., Männistö, T., Aaltonen, T.	(Myllärniemi *et al.*, 2012)
S42	Variabilities as First-Class Elements in Product Line Architectures of Homecare Systems	2012	Carvalho, S. T., Murta, L., Loques, O.	(Carvalho, Murta and Loques, 2012)
S43	Variability viewpoint for introducing variability in software architecture viewpoints	2012	Tekinerdogan, B., Sözer, H.	(Tekinerdogan and Sözer, 2012)
S44	Constraints for the design of variability-intensive service-oriented reference architectures – An industrial case study	2013	Galster, M., Avgeriou, P., Tofan, D.	(Galster, Avgeriou and Tofan, 2013)
S45	Supporting Variability Management in Architecture Design and Implementation	2013	Groher, I., Weinreich, R.	(Groher and Weinreich, 2013b)
S46	Engineering Delta Modeling Languages	2013	Haber, A., Hölldobler, K., Kolassa, C., Look, M., Müller, K., Rumpe, B., Schaefer, I.	(Haber *et al.*, 2013)

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S47	Run-Time support to manage architectural variability specified with CVL	2013	Pascual, G.G., Pinto, M., Fuentes, L.	(Pascual, Pinto and Fuentes, 2013)
S48	Evolving Software Requirements and Architectures using Software Product Line Concepts	2013	Gomaa, H.	(Gomaa, 2013)
S49	Model-to-Code transformation from Product-Line Architecture Models to AspectJ	2013	Díaz, J., Pérez, J., Fernández-Sánchez, C., Garbajosa, J.	(Diaz *et al.*, 2013)
S50	Applying Software Product Lines to Multiplatform Video Games	2013	Albassam, E., Gomaa, H.	(Albassam and Gomaa, 2013)
S51	Graph Modelling of a Refactoring Process for Product Line Architecture Design	2013	Losavio, F., Ordaz, O., Levy, N., Baïotto, A.	(Losavio *et al.*, 2013)
S52	Strategies for Aligning Variability Model and Architecture	2013	Groher, I., Weinreich, R.	(Groher and Weinreich, 2013a)
S53	A Case Study Comparison of Variability Representation Mechanisms with the HeRA Product Line	2013	Ahn, H., Kang, S., Lee, J.	(Hwi, Sungwon and Jihyun, 2013)
S54	A Lightweight Language for Software Product Lines Architecture Description	2013	Silva, E., Medeiros, A.L., Cavalcante, E., Batista, T.	(Silva *et al.*, 2013)

Study Identifier	Paper Title	Publication Year	Author(s)	Source
S55	On the Interdependence and Integration of Variability and Architectural Decisions	2014	Lytra, I., Eichelberger, H., Tran, H., Leyh, G., Schmid, K., Zdun, U.	(Lytra *et al.*, 2014)
S56	A Dynamic Software Product Line Architecture for Prepackaged Expert Analytics: Enabling Efficient Capture, Reuse and Adaptation of Operational Knowledge	2014	Smiley, K., Mahate, S., Wood, P.	(Smiley, Mahate and Wood, 2014)
S57	Representing and Configuring Security Variability in Software Product Lines	2015	Myllärniemi, V., Raatikainen, M., Männistö, T.	(Myllärniemi, Raatikainen and Männistö, 2015)
S58	Architectural Evolution of a Software Product Line: an experience report	2015	Laser, M.S., Rodrigues, E.M., Domingues, A., Oliveira, F., Zorzo, A.F.	(Laser *et al.*, 2015)

A2: List of publication outlet per primary study in the SLR

Study Identifier	Publication Outlet	Abbreviation
S1	9th International Software Product Line Conference	SPLC
S2	IEEE Software	IEEE/Software
S3	Journal of Computer Programming - Special issue: Software variability management	JCP
S4	2nd International Software Product Lines Young Researchers Workshop	SPLYR
S5	29th Annual IEEE/NASA Software Engineering Workshop	IEEE/NASA SEW
S6	29th Annual International Computer Software and Applications Conference	COMPSAC
S7	2nd International Conference on Informatics in Control Automation and Robotics	ICINCO
S8	2nd International Workshop on Software Engineering for Automotive Systems	SEAS
S9	IEE Proceedings – Software	IEEProceedings/Software
S10	9th International Conference on Reuse of Off-the-Shelf Components	OTS
S11	ACM SIGSOFT Software Engineering Notes	SEN
S12	XXVI International Conference of the Chilean Society of Computer Science	SCCC
S13	Journal of Advanced Engineering Informatics	JAEI
S14	15th International Conference Computational Science and its Applications	ICCSA
S15	7th IEEE International Conference on Computer and Information Technology	ICCIT

Study Identifier	Publication Outlet	Abbreviation
S16	2nd International Conference on Software Engineering Advances	ICSEA
S17	6th ACS/IEEE International Conference on Computer Systems and Applications	AICCSA
S18	15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems	ECBS
S19	6th International Conference on Software Engineering Research and Practice	SERP
S20	17th International Conference on Foundations of Intelligent Systems	FOIS
S21	7th Working IEEE/IFIP Conference on Software Architecture	WICSA
S22	1st International Conference on Computational Intelligence for Modelling Control & Automation	CIMCA
S23	2nd International Conference on Software Language Engineering	SLE
S24	21st International Conference on Software Engineering and Knowledge Engineering	SEKE
S25	13th International Software Product Line Conference	SPLC
S26	3rd International Workshop on Dynamic Software Product Lines	DSPL
S27	Joint 8th Working IEEE/IFIP Conference on Software Architecture & 3rd European Conference on Software Architecture	WICSA ECSA
S28	Embedded World 2009 Exhibition & Conference	EWEC
S29	Joint 8th Working IEEE/IFIP Conference on Software Architecture & 3rd European Conference on Software Architecture	WICSA ECSA
S30	3rd IEEE International Conference on Secure Software Integration and Reliability Improvement	SSIRI

Study Identifier	Publication Outlet	Abbreviation
S31	XXIX International Conference of the Chilean Computer Science Society	SCCC
S32	International Journal of Software Engineering and Knowledge Engineering	JSEKE
S33	12th International Conference on Top Productivity through Software Reuse	ICSR
S34	9th Working IEEE/IFIP Conference on Software Architecture	WICSA
S35	9th International Conference on Software Engineering Research, Management and Applications	SERA
S36	Dagstuhl Workshop on Model-Based Development of Embedded Systems.	MBEES
S37	5th European conference on Software architecture	ECSA
S38	15th International Software Product Line Conference	SPLC
S39	15th International Software Product Line Conference	SPLC
S40	1st International Workshop on Software Architecture Variability	VARSA
S41	Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture	WICSA ECSA
S42	4th International Workshop on Software Engineering in Health Care	SEHC
S43	Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture	WICSA ECSA
S44	Information and Software Technology	IST
S45	46th Hawaii International Conference on System Sciences	HICSS
S46	17th International Software Product Line Conference	SPLC

Study Identifier	Publication Outlet	Abbreviation
S47	7th European conference on Software Architecture	ECSA
S48	2nd International Workshop on the Twin Peaks of Requirements and Architecture	TwinPeaks
S49	39th Euromicro Conference Series on Software Engineering and Advanced Applications	SEAA
S50	3rd International Workshop on Games and Software Engineering	GAS
S51	XXXIX Latin American Computing Conference	CLEI
S52	20th Asia-Pacific Software Engineering Conference	APSEC
S53	16th IEEE International Conference on Computational Science and Engineering	CSE
S54	7th European conference on Software Architecture	ECSA
S55	8th Workshop on Variability Modeling of Software-Intensive Systems	VaMoS
S56	11th Working IEEE/IFIP Conference on Software Architecture	WICSA
S57	11th International ACM SIGSOFT Conference on Quality of Software Architectures	QoSA
S58	27th International Conference on Software Engineering and Knowledge Engineering	SEKE

A3: List of quality assessment scores per primary study in the SLR

Study Identifier	QA.Q1	QA.Q2	QA.Q3	QA.Q4	QA.Q5	QA.Q6	QA.Q7	QA.Q8	QA.Q9	TOTAL
S1	0.5	1	1	0.5	0	0	0.5	0.5	0	**4**
S2	1	1	1	1	0.5	1	1	1	0	**7.5**
S3	1	1	1	1	1	1	1	1	0.5	**8.5**
S4	0.5	1	1	0.5	0	0.5	0.5	0.5	0	**4.5**
S5	1	0.5	1	0	0	0.5	0	0.5	0	**3.5**
S6	1	1	1	1	0	0.5	0.5	1	0	**6**
S7	0.5	1	0.5	1	0	1	0.5	0	0.5	**5**
S8	1	1	0	1	1	1	0.5	0.5	0	**6**
S9	1	1	1	1	0.5	1	1	1	0.5	**8**
S10	1	1	1	1	0.5	1	1	1	0	**7.5**
S11	1	1	0.5	0	0	0.5	0.5	1	0	**4.5**
S12	1	1	0.5	0.5	1	1	0.5	0.5	0	**6**
S13	1	1	1	0.5	0.5	1	0.5	0.5	0	**6**
S14	1	1	1	1	1	1	1	1	0	**8**
S15	1	0.5	0.5	0	0	1	0.5	0.5	0	**4**
S16	1	1	1	1	0.5	1	1	1	1	**8.5**
S17	1	1	1	0.5	0	1	1	1	0	**6.5**

Study Identifier	QA.Q1	QA.Q2	QA.Q3	QA.Q4	QA.Q5	QA.Q6	QA.Q7	QA.Q8	QA.Q9	TOTAL
S18	1	0.5	1	0	0	0.5	1	0.5	0	**4.5**
S19	0.5	0.5	0	0	0	0	0.5	0.5	0	**2**
S20	1	1	0.5	0.5	0.5	1	0.5	0.5	0	**5.5**
S21	1	1	1	1	1	1	1	1	0	**8**
S22	1	1	1	0.5	0	1	1	1	0	**6.5**
S23	1	1	0.5	1	0	1	1	0	0	**5.5**
S24	1	0.5	1	1	0.5	1	0.5	0.5	0	**6**
S25	1	0.5	1	1	0.5	1	0.5	0.5	1	**7**
S26	1	1	1	1	0.5	1	0.5	0.5	0	**6.5**
S27	1	0.5	1	1	0	1	0.5	1	0	**6**
S28	0.5	1	0.5	0.5	0	0.5	0.5	0	0	**3.5**
S29	1	0.5	0.5	0	0	0.5	0	0.5	0	**3**
S30	1	1	1	1	0.5	0	1	1	0	**6.5**
S31	1	1	1	1	0.5	1	1	1	0	**7.5**
S32	1	1	1	1	0.5	1	1	1	0	**7.5**
S33	1	1	1	1	0.5	0.5	1	1	1	**8**
S34	1	0.5	1	0.5	0.5	1	1	0.5	0	**6**
S35	1	0.5	1	0	0	1	1	0.5	0	**5**
S36	0.5	0	1	0	0	0.5	0.5	0	0	**2.5**

Study Identifier	QA.Q1	QA.Q2	QA.Q3	QA.Q4	QA.Q5	QA.Q6	QA.Q7	QA.Q8	QA.Q9	TOTAL
S37	1	1	1	1	0	1	1	0.5	0	**6.5**
S38	1	1	1	1	0.5	1	1	1	0	**7.5**
S39	0.5	0.5	0.5	1	0	0	0	0	0	**2.5**
S40	0.5	0.5	0.5	1	0	0.5	1	0.5	0	**4.5**
S41	1	0.5	0.5	1	0.5	0.5	0.5	1	1	**6.5**
S42	1	1	1	0.5	0	0.5	0.5	0	0	**4.5**
S43	1	1	1	0	0	1	0.5	1	0	**5.5**
S44	1	1	1	1	1	1	0.5	1	0	**7.5**
S45	1	0.5	1	0.5	0.5	0.5	1	0.5	0	**5.5**
S46	1	1	1	0.5	0	0.5	1	0.5	0	**5.5**
S47	1	1	1	1	0	1	1	0.5	0	**6.5**
S48	1	0.5	1	0.5	0	1	0.5	0.5	0	**5**
S49	1	1	0.5	0.5	0	0.5	1	1	1	**6.5**
S50	1	1	1	1	1	1	1	0.5	0	**7.5**
S51	1	1	0.5	0.5	0	1	0.5	0.5	0	**5**
S52	1	1	1	0	0	0.5	0.5	0.5	0.5	**5**
S53	1	1	1	1	0	1	1	0.5	0	**6.5**
S54	1	1	1	1	0.5	1	1	0.5	0	**7**
S55	1	1	1	0.5	0	1	1	0.5	0	**6**

Study Identifier	QA.Q1	QA.Q2	QA.Q3	QA.Q4	QA.Q5	QA.Q6	QA.Q7	QA.Q8	QA.Q9	TOTAL
S56	1	1	1	1	0.5	1	0.5	1	0	**7**
S57	1	1	0.5	0.5	0	1	1	0.5	0	**5.5**
S58	1	0.5	1	0.5	0	1	0.5	0.5	0	**5**
TOTAL	**54**	**49**	**49**	**39**	**16.5**	**46**	**42**	**37**	**7**	
AVERAGE	**0.9**	**0.8**	**0.8**	**0.7**	**0.3**	**0.8**	**0.7**	**0.6**	**0.1**	

Appendix B: ALI V2 Event Traces Notation Comparison

	ALI V2		UML 2.5 (Activity Diagram)		UCM		Petri Nets	
	Symbol	Name	Symbol	Name	Symbol	Name	Symbol	Name
Start		START		Initial Node		Start Point		
End		END		Activity Final Node		End Point		
Termination		FINAL		Flow Final Node				
Concurrency		AND Fork		Fork Node		AND Fork		And-split

	ALI V2		UML 2.5 (Activity Diagram)		UCM		Petri Nets	
	Symbol	**Name**	**Symbol**	**Name**	**Symbol**	**Name**	**Symbol**	**Name**
Choice		OR Fork		Decision Node		OR Fork	1) 2) P $P_{condition}$	Selection/ Choice/ Decision
Synchronisation		AND Join		Join Node		AND Join		And-join

	ALI V2		UML 2.5 (Activity Diagram)		UCM		Petri Nets	
	Symbol	**Name**	**Symbol**	**Name**	**Symbol**	**Name**	**Symbol**	**Name**
Mutual		OR Join		Merge Node		OR Join		Meriana
Component	Name	Component			Name	Component	Name	Element*
Connection		Connector		Object Flow Edge		Path		Arc Inhibitor Arc Test Arc

	ALI V2		UML 2.5 (Activity Diagram)		UCM		Petri Nets	
	Symbol	**Name**	**Symbol**	**Name**	**Symbol**	**Name**	**Symbol**	**Name**
Event	Name/Cr** →	Event Flow	Name	Action Node	Name X {2} X With repetition count	Responsibility Point	Name (timed) Name Name Name	Discrete Transition Continuous Transition Immediate Transition
Condition	[ConditionName]	Condition	[ConditionName]	Guard	[ConditionName]	Condition	$P_{ConditionName}$	Condition

Not Supported

Known as 'Places' in petri net which denotes States.

* As used in I[3] ADL (Chang and Seongwoon, 1999)

** Optional i.e. if connection is made using connectors (see Chapter 6)

Appendix C: ALI V2 BNF

BNF for Meta Type

```
<input>            ::= <meta_type> <EOF>

<meta_type>        ::= "meta" "type" <identifier> "{"
                        {tag_definition} "}"

<tag_definition>   ::= "tag" <tag_name_list> ":" <tag_type> ";"

<tag_name_list>    ::=  <identifier> ["*"]{"," <identifier> ["*"]}

<tag_type>         ::=  "text"
                     |  "number"
                     |  "date"
                     |  "boolean"
```

BNF for Features

```
Input                          ::= <features> <EOF>

<features>                     ::= "features" "{" {<feature_description>} "}"

<feature_description>          ::= <feature_name> ":" "{"
                                   {<meta_object>}
                                   <alternative_names_section>
                                   <parameters_section> "}"

<alternative_names_section> ::= "alternative" "names" ":" "{" [
                                   <feature_alternative_name> {","
                                   <feature_alternative_name> } ] "}"

<parameters_section>           ::= "parameters" ":" "{" [ <feature_parameter> {","
                                   <feature_parameter>} ] "}"

<feature_parameter>            ::= "{" <feature_instance_name> {","
                                   <feature_instance_name>} "="
                                   <feature_parameter_type> "}"

<feature_parameter_type>       ::= "text"
                                 |  "number"

<feature_alternative_name>     ::= <string_literal>
```

BNF for Interface Template

```
<input>                          ::= <interface_template> <EOF>

<interface_template>             ::= "interface" "template" <Identifier> "{"
                                     <syntax_definition> <constraint_section> "}"

<syntax_definition>              ::= ("provider" | "consumer")"syntax" "definition" ":"
                                     "{" <provider_interface_section> |
                                     <consumer_interface_section> "}"

<provider_interface_section>     ::= "provider" ":" "{"
                                     {<provider_function_defintion>} "}"
<function_definition>            ::= "function" <identifier> "{"
                                     "impLanguage" ":" <identifier> ";"
                                     "invocation" ":" <identifier> ";"
                                     "paramterlist" ":" "("[ <identifier> {","
                                     <identifier>}] ")" ";" "return_type" ":" <identifier>
                                     ";" "}"

<consumer_interface_section>     ::= "consumer" ":" "{"
                                     {<consumer_function_defintion>} "}"

<consumer_function_defintion>    ::= "call" ":" <identifier> "("[ <identifier> {","
                                     <identifier>}] ")" ";"

<constraint_section>             ::= "constraints" ":" "{" "should" "match" ":" "{"
                                     <identifier> "=" "."<identifier> "," <identifier> "}"
                                     "binding" ":" "{" <binding_section> "}" ";"
                                     "factory" ":" ( "true" | "false" ) ";" "persistent" ":" (
                                     "true" | "false" ) ";" "}"

<binding_section>                ::= "multiple" ":" ( "true" | "false" ) ";" "data_size" ":"
                                     "[" <natural_literal> "," <natural_literal> "]" ";"
                                     "max_connections" ":" <natural_literal> ";"
```

BNF for Interface Type

```
<input>                  ::= <interface type> <EOF>

<interface_type>         ::= "interface" "type" "{"
                              {<meta_object>}
                              {<interface_defintion>} "}"

<interface_definition> ::= <interface_instance_name> ":" <interface_template> "{"
                              <interface_instance_definition> "}"
```

BNF for Connector Type

```
<input>                          ::= <connector_type> <EOF>

<connector_type>                 ::= "connector" "type" <identifier> "{"
                                     {<meta_object>}
                                     {<features_section>}
                                      <interfaces_section>
                                      <layout_section> "}"

<features_section>               ::= "features"":" "{" {
                                      <feature_section_body> } "}"

<feature_section_body>           ::= <feature_name> ":" <string_literal> {","
                                     <feature_name> ":" <string_literal>} ";"

<interfaces_section>             ::= "interfaces"":" "{" {
                                      <interface_section_body> } "}"

<interface_section_body>         ::= <connector_interface_definition>
                                  |  <connector_conditional_interface>

<connector_interface_definition> ::= {<interface_instance_name> { ","
                                     <interface_instance_name> } ":"
                                     <interface_type> ";" }

<connector_conditional_interface> ::= "if" "(" <conditional_inclusion_expression> ")"
                                     (<connector_interface_definition> | "{"
                                     {<connector_interface_definition>} "}" )
                                     [ "else"
                                     ( <interface_section_body> | "{"
                                     {<connector_interface_definition>} "}" ) ]

<layout_section>                 ::= "layout"":" "{" {
                                      <layout_section_body>} "}"

<layout_section_body>            ::= <layout_definition_statement>
                                  |  <conditional_layout_definition>

<layout_definition_statement>    ::= "connect" ( <interface_instance_name> | "all" )
                                     ( "and" | "to" ) ( <interface_instance_name> |
                                     "all" ) ";"

<conditional_layout_definition>  ::= "if" "(" <conditional_inclusion_expression> ")"
                                     ( <layout_definition_statement>
                                     |  "{" {<layout_definition_statement>} "}"  )
                                     [ "else" ( <layout_section_body> | "{" {
                                     <layout_definition_statement> } "}" ) ]
```

BNF for Component Type

```
<input>                                ::= <component_type> <EOF>

<component_type>                       ::= "component" "type" <identifier> "{"
                                           {<meta_object>}
                                           {<features_section>}
                                           <interfaces_section> <sub_system_section>
                                           "}"

<features_section>                     ::= "features" "{" {
                                            <feature_section_body> } "}"

<feature_section_body>                 ::= <feature_name> ":" <string_literal> {","
                                           <feature_name> ":" <string_literal>} ";"

<interfaces_section>                   ::= "interfaces" ":" "{"
                                           {<interface_section_body>} "}"

<interface_section_body>               ::= <sub_interface_definition_section>
                                         |  <sub_interface_implements_section>

<sub_interface_definition_section>     ::= "definition" ":" "{"
                                           {<sub_interface_definition_body>}
                                           “}”

<sub_interface_definition_body>        ::= <interface_definition>
                                         |  <component_conditional_interface>

<component_conditional_interface>      ::= "if" "(" <conditional_inclusion_expression>
                                           ")" ( {<interface_definition>} |
                                           "{" {<interface_definition>} "}" )
                                           [ "else"  (<sub_interface_definition_body> |
                                           "{" {<interface_definition>})
                                            "}" ) ]

<sub_interface_implements_section>     ::= "implements" ":" "{"
                                           {<sub_interface_implements_body>}
                                           “}”

<sub_interface_implements_body>        ::= <interface_implement_defintion>
                                          | <conditional_implements_interface>

<interface_implements_definition>      ::= {<interface_instance_name> {","
                                           <interface_instance_name> ":"
                                           <interface_type>}}"}"
```

```
<component_conditional_interface>    ::= "if" "(" <conditional_inclusion_expression>
                                          ")" ( {<interface_implements_definition>} |
                                          "{" {<interface_implements_definition>}
                                          "}" )
                                          [ "else"
                                          (<sub_interface_implements_body> | "{"
                                          {<interface_implements_definition>})
                                           "}" ) ]

<sub_system_section>                 ::= "sub-system" ":" "{" <components_section>
                                          <connectors_section>
                                          <arrangement_section> "}"

<components_section>                 ::= "components" "{"
                                          {<components_section_body>} "}"

<components_section_body>            ::= <components_definition_statement>
                                        |  <conditional_components_definition>

<components_definition_statement>    ::= <component_instance_name> [ "<"
                                          <feature_name> {"," <feature_name>} ">" ]
                                          {"," <component_instance_name> [ "<"
                                          <feature_name> {"," <feature_name>} ">" ]
                                          } ":" <component_type_name> ";"

<conditional_components_definition>  ::= "if" "(" <conditional_inclusion_expression>
                                          ")" ( <components_definition_statement> |
                                          "{" {<components_definition_statement>}
                                          "}" )
                                          [ "else" (<components_section_body> | "{"
                                          {<components_definition_statement>} "}" )
                                          ]

<connectors_section>                 ::= "connectors" "{"
                                          {<connectors_section_body>} "}"

<connectors_section_body>            ::= <connectors_definition_statement>
                                        |  <conditional_connectors_definition>

<connectors_definition_statement>    ::= <connector_instance_name> [ "<"
                                          <feature_name> {"," <feature_name>} ">" ]
                                          {"," <connector_instance_name> [ "<"
                                          <feature_name> {"," <feature_name>} ">" ]
                                          } ":" <connector_type_name> ";"

<conditional_connectors_definition>  ::= "if" "(" <conditional_inclusion_expression>
                                          ")" ( <connectors_definition_statement> |
                                          "{" {<connectors_definition_statement>}
                                          "}" )
```

["else" (<connectors_section_body> | "{" {<connectors_definition_statement>} "}")]

<arrangement_section> ::= " arrangement " "{" {< arrangement _section_body>} "}"

< arrangement _section_body> ::= < arrangement _definition_ statement>
| <conditional_ arrangement _definition>

< arrangement _definition_statement> ::= < arrangement _defintion_manually>
| < arrangement _defintion_using_ patterns>

< arrangement_defintion_manually> ::= ("connect" | "bind") <connection_argument> ["." <interface_instance_name>] "with" <connection_argument> ["." <interface_instance_name>] ";"

<connection_argument> ::= <component_instance_name>
| <connector_instance_name>
| "my"
| "*"

< arrangement_defintion_ using_patterns> ::= <pattern_name> "(" [<interface_argument> {"," <interface_argument>}] ")" ";"

<interface_argument> ::= <interface_instance_name>
| <interface_name_array>

<conditional_arrangement_definition> ::= "if" "(" <conditional_inclusion_expression> ")" (< arrangement_definition_statement> | "{" {< arrangement_definition_statement>} "}")
["else" (<arrangement_section_body> | "{" {<arrangement_definition_statement>} "}")]

<interface_name_array> ::= "[" <interface_instance_name> {"," <interface_instance_name>} "]"

BNF for Pattern Template

<input> ::= <pattern_template> <EOF>

<pattern_template> ::= "pattern" "templates" ":" "{" {<pattern_defintion>} "}"

<pattern_defintion> ::= <pattern_name> "(" <pattern_parameter_list> {"," <pattern_parameter_list>} ")" "{" {<meta_object>} {<simple_connection_statement> | <compound_connection_statement>} "}"

<pattern_parameter_list> ::= <interface_instance_name> [<array_specification>] ":" <interface_template>

<array_specification> ::= "[" <minimum_array_count> ".." <maximum_array_count> "]"

<simple_connection_statement> ::= <connection_definition_manually>
| <connection_definition_using_patterns>

<connection_definition_manually> ::= "connect" (<generic_interface_instance_name> | "all") ("and" | "to") (<generic_interface_instance_name> | "all") ";"

<connection_definition_using_patterns> ::= <pattern_name> "(" <generic_interface_instance_name> {"," <generic_interface_instance_name>} ")" ";"

<compound_connection_statement> ::= <for_loop> (<simple_connection_statement> | ("{" (<simple_connection_statement> | <compound_connection_statement>)+ "}"))

<for_loop> ::= "for" "(" <for_loop_initialization> ";" <for_loop_condition> ";" <for_loop_counter_modify> ")"

<for_loop_initialization> ::= <identifier> "=" (<natural_literal> | <identifier>)

<for_loop_condition> ::= <identifier> ("<" | ">" | "<=" | ">=" | "==" | "!=") (<identifier> | <natural_literal>)

<for_loop_counter_modify> ::= <uni_counter> | <binary_counter>

<binary_counter> ::= <identifier> ("+=" | "-=") <natural_literal>

<uni_counter> ::= (("++" | "--") <identifier>) | (<identifier> ("++" | "--"))

<generic_interface_instance_name> ::= <interface_instance_name> ["[" (<natural_Literal> | (<identifier> [("+" | "-" | "*" | "/") <natural_Literal>])) "]"]

<minimum_array_count> ::= <natural_literal>

<maximum_array_count> ::= <natural_literal>

BNF for Product Configuration

```
<input>                       ::= <product_configuration> <EOF>

<product_configuration>       ::= "product" "configurations" "{"
                                   {<product_instances_section>} "}"

<product_instances_section>   ::= <product_instance_name> ":" "{"
                                  {<meta_object>}
                                  {product_defintion>}
                                  "}"

<product_instance_name>       ::= <identifier>

<product_defintion>           ::= <simple_feature>
                                |  <parameterise_feature>

<simple_feature>              ::= <feature_name> "=" <boolean_literal> ";"

<parameterise_feature>        ::= <feature_name> "{" <feature_instance_name> "="
                                  <natural_literal> {"," <feature_instance_name> "="
                                  <natural_literal>}"}" ";"
```

BNF for Event

```
<input>                        ::=  <event> <EOF>

<event>                        ::=  "events" "{"
                                     {<meta_object>}
                                     {<event_instances>} "}"

<event_instances>              ::=  <event_instance_name> ":"
                                    <single_interface_template> |
                                    <multiple_interface_template> ";"

<single_interface_template>    ::=  "<" <interface_template> ","
                                       <interface_template> ">" ";"

<multiple_interface_template>  ::=  "<" "(" <interface_template> { ","
                                       <interface_template>} ")" "," "("
                                       <interface_template> { ","
                                       <interface_template>} ")"">" ";"
```

BNF for Condition

<input>	::=	<condition> <EOF>
<condition>	::=	"conditions" "{" {<meta_object>} {<condition_instances>} "}"
<condition_instances>	::=	<condition_instance_name> ":" <condition_description> ";"
<condition_description>	::=	<string_literal>

BNF for Scenario

<input>	::=	<scenario> <EOF>
<scenario>	::=	"scenarios" "{" {<scenario_instances_section>} "}"
<scenario_instances_section>	::=	<scenario_instance_name> ":" "{" {<meta_object>} <scenario_description_section> <parameterisation_section> "}"
<scenario_instance_name>	::=	<identifier>
<scenario_description_section>	::=	"description" ":" <string_literal> ";"
<parameterisation_section>	::=	"parameterisation" ":" "{" {<condition_instance_name> "=" <boolean_literal>} ";" "}"

BNF for Transaction Domain

```
<input>                              ::= <transaction_domain> <EOF>

<transaction_domain>                 ::= "transaction" "domain" <identifier> "{"
                                         {<meta_object>}
                                         <contents_section>
                                         <transactions_section> "}"

<contents_section>                   ::= "contents" ":" "{"
                                          <component_instances_section>
                                          [<connector_instances_section>]
                                         "}"

<component_instances_section>        ::= "components" ":" "{"
                                         ["*"]<component_instance_name> {","
                                         ["*"]<component_instance_name>}
                                         "}"

<connector_instances_section>        ::= "connectors" ":" "{"
                                         <connector_instance_name> {","
                                         <connector_instance_name>}
                                         "}"

<transactions_section>               ::= "transactions" ":" "{"
                                         {<transaction_instances_section>}
                                         "}"

<transaction_instances_section>      ::= <transaction_instance_name> ":" "{"
                                         {<meta_object>}
                                         [<event_instances_section>]
                                         <interaction_section>
                                         "}"

<transaction_instance_name>          ::= <identifier>

<event_instances_section>            ::= "events" ":" "{"
                                         <event_instance_name> {","
                                         <event_instance_name>}
                                         "}"

<interaction_section>                ::= "interactions" ":" "{"
                                          {<interaction_section_body>}
                                         "}"

<interaction_section_body>           ::= <interaction_definition_statement>
                                       | <conditional_interaction_definition>
```

```
<interaction_definition_statement>   ::= <simple_interaction_statement>
                                       |  <fork_interaction_statement>
                                       |  <join_interaction_statement>
                                       |  <fork_join_interaction_statement>

<simple_interaction_statement>       ::= <send_interaction_statement>
                                       |  <receive_interaction_statement>

<send_interaction_statement>         ::= ((<component_instance_name> "."
                                          <interface_instance_name>) |
                                          <transaction_instance_name>)
                                         [ "sends" <event_instance_name> "/"
                                         <connector_instance_name> "to"
                                         ((<component_instance_name> "."
                                          <interface_instance_name>) |
                                          <transaction_instance_name>)] ";"

<receive_interaction_statement>      ::= ((<component_instance_name> "."
                                          <interface_instance_name>) |
                                          <transaction_instance_name>) [ "receives"
                                         <event_instance_name> "/"
                                         <connector_instance_name>
                                         "from" ((<component_instance_name> "."
                                          <interface_instance_name>) |
                                          <transaction_instance_name>)] ";"

<fork_interaction_statement>         ::= "[" <fork_interaction_body> "]" ";"

<fork_interaction_body>              ::= <send_interaction_statement>
                                           {("," | "|")
                                            <send_interaction_statement>}

<join_interaction_statement>         ::= "[" <join_interaction_body> "]" ";"

<join_interaction_body>              ::= <receive_interaction_statement>
                                           {("," | "|")
                                            <receive_interaction_statement>}

<fork_join_interaction_statement>    ::= "[" "(" (join_interaction_body |
                                                 <fork_interaction_body>)
                                             ")" {("," | "|")
                                             "(" (join_interaction_body |
                                                 <fork_interaction_body>) ")"} "]" ";"

<conditional_interaction_definition> ::= "if" "(" <conditional_inclusion_expression>
                                         ")" (<interaction_definition_statement> |
                                         "{" {<interaction_definition_statement>}
                                         "}") [ "else"
                                         (<interaction_section_body> |
                                         {<interaction_definition_statement>})]
```

BNF for Viewpoint

```
<input>                            ::= <viewpoint> <EOF>

<viewpoint>                        ::= "viewpoints" "{"
                                        {<viewpoint_instances_section>} "}"

<viewpoint_instances_section>      ::= <viewpoint_instance_name> ":" "{"
                                       {<meta_object>}
                                       <viewpoint_description_section>
                                       <transaction_domain_section>
                                       "}"

<viewpoint_instance_name>          ::= <identifier>

<viewpoint_description_section>    ::= "description" ":"
                                        <string_literal> ";"

<transaction_domain_section>       ::= "transaction" "domain" ":" "{"
                                        <transaction_domain_instance_name> {","
                                       < transaction_domain_instance_name >} ";"
                                       "}"

<transaction_domain_instance_name> ::= <identifier>
```

BNF for System Description

```
<input>                              ::= <system_description> <EOF>

<system_description>                 ::= "system" ":" "{"
                                          {<meta_object>}
                                          <components_section>
                                          <connectors_section>
                                          <arrangement_section>
                                          <viewpoints_section> "}"

<components_section>                 ::= "components" "{"
                                         {<components_section_body>} "}"

<components_section_body>            ::= <components_definition_statement>
                                       |  <conditional_components_definition>

<components_definition_statement>    ::= <component_instance_name> [ "<"
                                         <feature_name> {"," <feature_name>} ">" ]
                                         {"," <component_instance_name> [ "<"
                                         <feature_name> {"," <feature_name>} ">" ] }
                                         ":" <component_type_name> ";"

<conditional_components_definition>  ::= "if" "(" <conditional_inclusion_expression>
                                         ")" ( <components_definition_statement> |
                                         "{" {<components_definition_statement>}
                                         "}" )
                                         [ "else" (<components_section_body> | "{"
                                         {<components_definition_statement>} "}" ) ]

<connectors_section>                 ::= "connectors" "{"
                                         {<connectors_section_body>} "}"

<connectors_section_body>            ::= <connectors_definition_statement>
                                       |  <conditional_connectors_definition>

<connectors_definition_statement>    ::= <connector_instance_name> [ "<"
                                         <feature_name> {"," <feature_name>} ">" ]
                                         {"," <connector_instance_name> [ "<"
                                         <feature_name> {"," <feature_name>} ">" ] }
                                         ":" <connector_type_name> ";"

<conditional_connectors_definition>  ::= "if" "(" <conditional_inclusion_expression>
                                         ")" ( <connectors_definition_statement> | "{"
                                         {<connectors_definition_statement>} "}" )
                                         [ "else" (<connectors_section_body> | "{"
                                         {<connectors_definition_statement>} "}" ) ]
```

```
< arrangement_section>                  ::= " arrangement " "{"  {<
                                            arrangement_section_body>} "}"

< arrangement_section_body>             ::= < arrangement_definition_
                                             statement>
                                          |  <conditional_ arrangement_definition>

< arrangement_definition_statement>     ::= < arrangement_defintion_manually>
                                          |  < arrangement_defintion_using_
                                             patterns>

< arrangement_defintion_manually>       ::= ( "connect" | "bind" ) <connection_argument>
                                            ["." <interface_instance_name>] "with"
                                            <connection_argument>  [ "."
                                            <interface_instance_name> ] ";"

<connection_argument>                   ::= <component_instance_name>
                                          |  <connector_instance_name>
                                          |  "my"
                                          |  "*"

<structure_defintion_using_patterns>    ::= <pattern_name> "(" [ <interface_argument>
                                            {"," <interface_argument>} ] ")" ";"

<interface_argument>                    ::= <interface_instance_name>
                                          |  <interface_name_array>

<conditional_                           ::= "if" "(" <conditional_inclusion_expression>
arrangement_definition>                     ")" ( < arrangement_definition_statement> |
                                            "{" {< arrangement_definition_statement>}
                                            "}" )
                                            [ "else"
                                            (<arrangement_section_body> | "{" {<
                                            arrangement_definition_statement>} "}" ) ]

<interface_name_array>                  ::= "[" <interface_instance_name> {","
                                            <interface_instance_name>} "]"

<viewpoints_section>                    ::= "viewpoints" "{" <viewpoint_instance_name>
                                            {"," <viewpoint_instance_name>} "}"

<viewpoint_instance_name>               ::= <identifier>
```

Miscellaneous

ALI Structural Literals

<meta_object>	::= "meta" ":" <identifier> { "," <identifier> } "{" { <identifier> ":" (<string_literal> | <natural_literal>) ";" } "}"
<interface_definition>	::= <interface_instance_name> ":" <interface_template> "{" <interface_instance_definition> "}"
<interface_instance_name>	::= <identifier>
<interface_instance_definition>	::= <string_literal>
<interface_template>	::= <identifier>
<component_instance_name>	::= <identifier>
<component_type_name>	::= <identifier>
<connector_instance_name>	::= <identifier>
<connector_type_name>	::= <identifier>
<pattern_name>	::= <identifier>
<feature_name>	::= <identifier> | <boolean_literal>
<feature_instance_name>	::= <identifier>

ALI Behavioural Literals

<event_instance_name>	::= <identifier>
<condition_instance_name>	::= <identifier>

ALI If Condition Expression

<conditional_inclusion_expression> ::= <or_expression>

<or_expression> ::= <and_expression> {"||" <and_expression>}

<and_expression> ::= <equality_expression> {"&&" <equality_expression>}

<equality_expression> ::= <unary_expression> {("==" | "!=" | "<" | ">" | "<=" | ">=") <unary_expression>}

<unary_expression> ::= "!" <unary_expression> | <boolean_literal> | <feature_value_literal> | <feature_name> | <condition_instance_name> | <predicate_expression> | "(" <conditional_inclusion_expression> ")"

<feature_value_literal> ::= <string_literal> | <natural_literal>

<predicate_expression> ::= <predicate> "(" <feature_name> ")"

<predicate> ::= "supported" | "unsupported"

Generic Literals

<identifier> ::= <id_character> | <identifier> <id_character> | <identifier> <digit>

<id_character> ::= <letter> | <break_character>

<letter> ::= A|B|C ... |Z|a|b|c ... |z

<digit> ::= 0|1| ... |9

<break_character> ::= _ | @ | # | $

<string_literal> ::= " <character> { <character> } "

<character> ::= <letter> | <digit> | <break_character>

<natural_literal> ::= <letter> | <digit>

<boolean literal> ::= "true" | "false"

Appendix D: AMS Case Study

This section contains the remaining architectural description of the AMS case study presented in Chapter 7 using ALI V2 notations (discussed in Chapter 6).

D1: AMS Meta Types

```
meta type Meta_Processor {
   tag queuing_method, priority_process: text;
   tag max_jobs*: number;
  }

meta type Meta_DbEquity {
   tag last_updated: date;
   tag DBA*, description*: text;
  }

meta type Meta_ShareValueData {
   tag stock_market*, risks*, intention: text;
   tag price_synchronised*: date;
  }

meta type Meta_Valuator {
   tag acceptance_value, value_approximation,
       currency_acceptance*: text;
   tag last_request: date;
  }

meta type Meta_Derivative {
   tag risk_mitigation: text;
   tag renewal_deadline: date;
  }

meta type Meta_PortfolioDomain {
     tag purpose, compatibility, occurrence: text;
    }

meta type Meta_Trade {
     tag updation_frequency, trade_condition*: text;
     tag max_request_per_order*, max_amount_per_order*: number;}
```

D2: AMS Features

```
features {
        … // defined in Section 7.3.2
  Share_Sector: {
     alternative names: {
          Designer.IS2, Developer.SS, Evaluator.F13;
         }
     parameters: {
          {Holdings = number,
           Total_Share_Value = number,
           Share_Sector_Category = text};
         }
   }

  Equity_Derivative: {
     alternative names: {
          Designer.ID1, Developer.SD, Evaluator.F14;
         }
     parameters: {
          {Derivative_Type = text,
           Premium_Period = text,
           OTC = boolean};
         }
   }

  MarkToMarket_Method: {
     meta: Meta_AMSFeature {
          creation_date: 29-02-2016;
          standardized: true;
         }
     alternative names: {
          Designer.VF1, Developer.MTM, Evaluator.F15;
         }
     parameters: {
           // no parameters
         }
   }
```

```
  Share_Company_Method: {
    alternative names: {
         Designer.VF2, Developer.SC, Evaluator.F16;
        }
    parameters: {
          // no parameters
        }
   }

  Cash_Investment: {
    alternative names: {
         Designer.RF1, Developer.CI, Evaluator.F17;
        }
    parameters: {
         {InvestmentCurrency = text};
         }
    }

   Share_Investment: {
     alternative names: {
          Designer.RF2, Developer.SI, Evaluator.F18;
         }
     parameters: {
          {Max_Offer_Quantity = number,
           Max_Bid_Quantity = number};
         }
    }

   Financial_Asset: {
     alternative names: {
          Designer.GF1, Developer.FA, Evaluator.F19;
         }
     parameters: {
           // no parameters
         }
    }
} // end of features
```

D3: AMS Interface Types

```
interface type {
    … // defined in Section 7.3.4
    AverageOperation: MethodInterface {
       Provider: {
        function Average
               {
                impLanguage: Java;
                invocation: average;
                parameterlist: (int);
                return_type: void;
               }
        }
        Consumer: {//nothing consumed}
     }

     NumericOperation: MethodInterface {
        Provider: {
         function GetValue
               {
                impLanguage: Java;
                invocation: getValue;
                parameterlist: (void);
                return_type: long_int;
               }
        }
        Consumer: {
         Call: add (long_int);
         Call: subtract (long_int);
         Call: multiply (long_int);
         Call: average (long_int);
        }

     InvestmentOperation: MethodInterface {
        Provider: {
         function Addition
               {
                impLanguage: Java;
                invocation: add;
                parameterlist: (int);
                return_type: void;
```

```
            }
     function GetValue
            {
             impLanguage: Java;
             invocation: getValue;
             parameterlist: (void);
             return_type: long_int;
            }
     }

   }
   Consumer: {
       Call: getValue (long_int);
       Call: add (long_int);
    }
}

DatabaseOperation: MethodInterface {
    Provider: {
     function InsertSQLData
            {
             impLanguage: Java;
             invocation: insert;
             parameterlist: (string);
             return_type: void;
            }
     function DeleteSQLData
            {
             impLanguage: Java;
             invocation: delete;
             parameterlist: (string);
             return_type: void;
            }
     }
   }
    Consumer: {
       Call: insert (string);
       Call: delete (string);
    }
}
```

```
DatabaseUpdation: MethodInterface {
   Provider: {
    function SearchSQLData
           {
            impLanguage: Java;
            invocation: search;
            parameterlist: (void);
            return_type: string;
           }
    function UpdateSQLData
           {
            impLanguage: Java;
            invocation: update;
            parameterlist: (string);
            return_type: void;
           }
    }
  }
   Consumer: {
      Call: search (string);
      Call: update (string);
   }
}

DatabaseOrder: MethodInterface {
   Provider: {
    function GetSQLMessage
           {
            impLanguage: Java;
            invocation: getMessage;
            parameterlist: (void);
            return_type: string;
           }
    }
  }
   Consumer: {
      Call: insert (string);
      Call: update (string);
   }
}
```

```
    DerivativeOperation: MethodInterface {
      Provider: {
       function ValuePercentage
              {
                impLanguage: Java;
                invocation: percentage;
                parameterlist: (int);
                return_type: void;
              }
       function GetValue
              {
                impLanguage: Java;
                invocation: getValue;
                parameterlist: (void);
                return_type: long_int;
              }
       }
      Consumer: {
         Call: percentage (long_int);
         Call: getValue (long_int);
      }
   }
} // end of interface types
```

D4: AMS Connector Types

HTTP_AMSUserInterface

```
connector type HTTP_AMSUserInterface
 {
  features: {
      Equity: "Type of financial instrument that deals with
               shares",
      Commodity: "Type of financial instrument that deals with
                  metal, agriculture, Oil & Gas and energy",
      Foreign_Exchange: "Type of financial instrument that deals
                         with currency",
      Interest_Rate: "Type of financial instrument that deals
                      with bonds";
   }
  interfaces: {
      requestport1, requestport2: PortfolioService;
      requestport3, requestport4: PortfolioStatus;
   }
  layout: {
      connect requestport3 and requestport4;
      if (supported(Equity || Commodity || Foreign_Exchange ||
                    Interest_Rate))
        connect requestport1 and requestport2;
   }
 }
```

HTTP_Equity

```
connector type HTTP_Equity
 {
  features: {
      Share: "Type of equity in financial instruments",
      Derivative: "Used as a security for the equity asset";
    }
  interfaces: {
     messageport1, messageport2, messageport3:
                                          PortfolioMessenger;
   }
   layout: {
     if (supported(Share)|| supported(Derivative))
```

```
        connect messageport1 and messageport2;
     else
        connect messageport1 and messageport3;
   }
 }
```

ODBC_EquityPortfolio

```
connector type ODBC_EquityPortfolio
 {
  features: {
      Currency_Investment_Method: "Managing portfolio with cash",
      Share_Investment_Method: "Managing portfolio with share
                                trading";
   }
  interfaces: {
      dataport1, dataport2: DatabaseUpdation;
      dataport3, dataport4: DatabaseOperation;
    }
  layout: {
      connect dataport1 and dataport2;
      if (supported(Currency_Investment_Method ||
                   Share_Investment_Method))
         connect dataport3 and dataport4;
   }
 }
```

HTTP_EquityValuator

```
connector type HTTP_EquityValuator
 {
  features: {
      MTM_Price_Method: "Share prices matched with market price",
      Company_Price_Method: "Unlisted share price of an
                             individual company",
      Weighted_Average_Method: "Portfolio Valuation is done on
                                the basis of average price";
    }
  interfaces: {
      messageport1, messageport2: PortfolioMessenger;
      valueport1, valueport2, valueport3, valueport4:
                                               ValueData;
```

```
    }
   layout: {
      connect valueport3 and valueport4;
      if (supported(MTM_Price_Method || Company_Price_Method))
         connect valueport1 and valueport2;
      if (supported(Weighted_Average_Method))
         connect messageport1 to messageport2;
    }
 }
```

HTTP_ExternalSystem

```
connector type HTTP_ExternalSystem
 {
   features: {
       Financial_Asset: "Tradeable assets to manage portfolio",
       MTM_Method: "Share prices matched with market price",
       SCompany_Method: "Unlisted share price of an individual
                         company";
    }
   interfaces: {
       messaageport1, messageport2: PortfolioMessenger;
       valueport1, valueport2: ValueData;
    }
   layout: {
      if (supported(Financial_Asset))
         connect messageport1 and messageport2;
      if (supported(MTM_Method || SCompany_Method))
         connect valueport1 and valueport2;
    }
 }
```

ODBC_EquityTrade

```
connector type ODBC_EquityTrade
 {
  features: {
      Financial_Asset: "Tradeable assets to manage portfolio",
      Share_Investment_Method: "Managing portfolio with share
                                trading",
      Share: "Type of equity in financial instruments";
   }
  interfaces: {
      dataport1, dataport2: PortfolioData;
      dataport3, dataport4: DatabaseOrder;
    }
  layout: {
     if (supported(Share_Investment_Method || Share))
        connect dataport1 and dataport2;
     if (supported(Financial_Asset || Share_Investment_Method))
        connect dataport3 and dataport4;
   }
 }
```

HTTP_EquityTrade

```
connector type HTTP_EquityTrade
 {
  features: {
      Currency_Investment_Method: "Managing portfolio with cash",
      Share_Investment_Method: "Managing portfolio with share
                                trading";
   }
  interfaces: {
      msgport1, msgport2, msgport3: OrderMessenger;
   }
  layout: {
     connect msgport1 and msgport3;
     if (supported(Share_Investment_Method))
        connect msgport1 and msgport2;
     if (supported(Currency_Investment_Method))
        connect msgport1 to msgport3;
   }
  }
```

HTTP_EquityRate

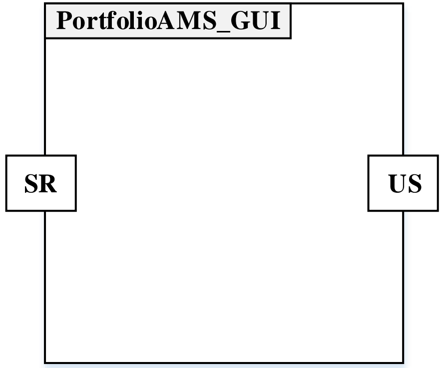
```
connector type HTTP_EquityRate
 {
  features: {
      // no optional/alternative and parameterized features
   }
  interfaces: {
      valueport1, valueport2: ValueData;
   }
  layout: {
       connect valueport1 and valueport2;
   }
 }
```

Calculator_Derivative

```
connector type Calculator_Derivative
 {
  features: {
      // no optional/alternative and parameterized features
   }
  interfaces: {
      valueport1, valueportt2: DerivativeOperation;
   }
  layout: {
      connect valueport1 and valueport2;
   }
  }
```

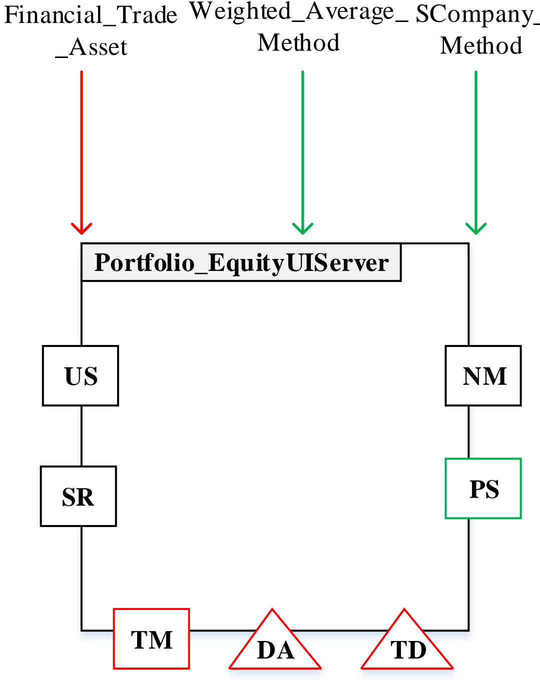
D5: AMS Component Types

PortfolioAMS_GUI




```
component type PortfolioAMS_GUI
  {
   meta: { }
   features: {
       // no optional/alternative and parameterized features
    }
   interfaces: {
    definition: {
       // no need to define any interface/s
     }
    implements: {
       ServiceRequest, UpdationStatus: PortfolioMessenger;
     }
    }
   sub-system: {
    components { }
    connectors { }
    arrangement { }
   } // end of sub-system
  } // end of component type
```

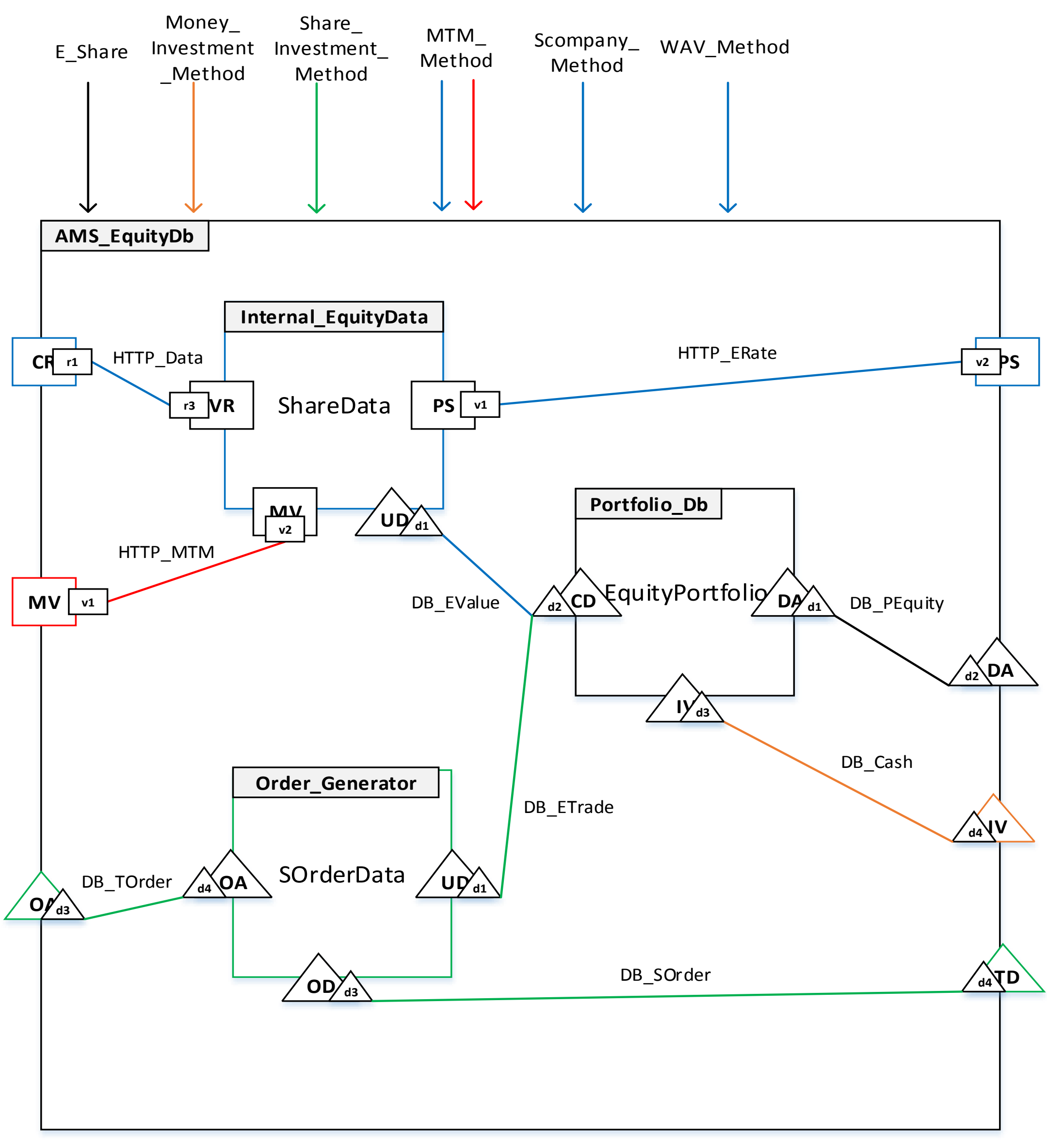
Portfolio_EquityUIServer




```
component type Portfolio_EquityUIServer
  {
   meta: Meta_EquityServer {
     creatorID: "david045";
     cost: 2000;
     version: 1.3;
     last_updated: 29-02-2016;
    }
   features: {
     Financial_Trade_Asset: "Tradeable assets to manage
                             portfolio",
     SCompany_Method: "Unlisted share price of an individual
                       company",
     Weighted_Average_Method: "Portfolio Valuation is done on the
                               basis of average share price";
    }
   interfaces: {
    definition: {
      // no need to define any interface/s
     }
```

```
  implements: {
     ServiceRequest, UpdationStatus, NotificationMessage:
                                                        PortfolioMessenger;
     if (supported (Financial_Trade_Asset) &&
         unsupported (SCompany_Method))
        {TradeMessage: PortfolioMessenger;
        TradeData, DataAccess: DataUpdation;}
     if (supported (SCompany_Method || Weighted_Average_Method))
        PriceStatus: ValueData;
   }
  }
 sub-system: {
  components { }
  connectors { }
  arrangement { }
 } // end of sub-system
} // end of component type
```

AMS_EquityDb

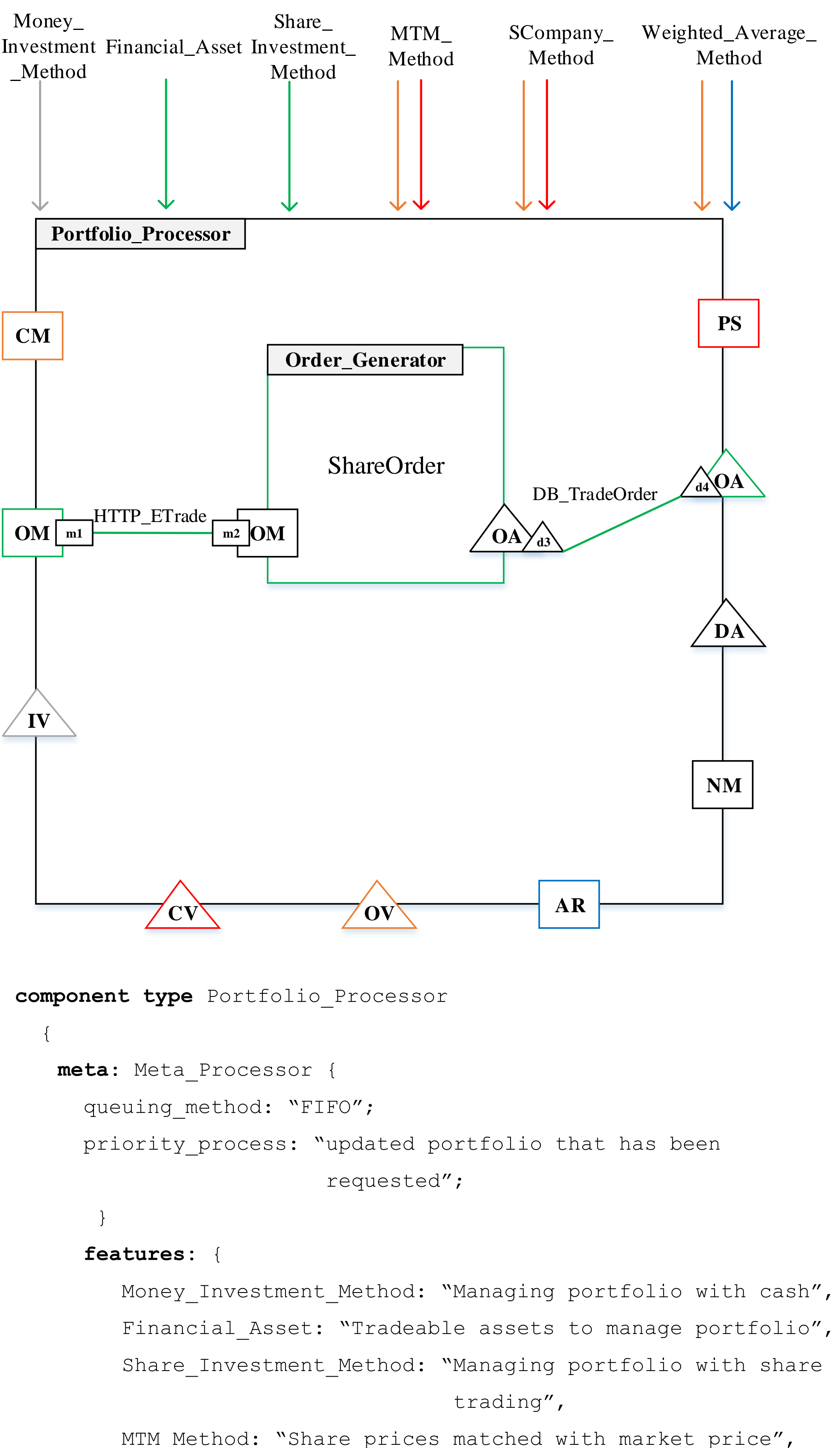



```
component type AMS_EquityDb
  {
   meta: Meta_DbEquity {
     last_updated: 29-02-2016;
     DBA: "David";
     description: "stores all data related to equity-shares";
   }
   features: {
      E_Share: "Type of equity in financial instruments",
      Money_Investment_Method: "Managing portfolio with cash",
      Share_Investment_Method: "Managing portfolio with share
                                trading",
      MTM_Method: "Share prices matched with market price",
```

```
    SCompany_Method: "Unlisted share price of an individual
                      company",
    WAV_Method: "Portfolio Valuation is done on the basis of
                 Average share price";
  }
 interfaces: {
  definition: { //no need to define any interface/s }
  implements:{
      DataAccess: DatabaseUpdation;
      if (supported(Financial_Asset ||
                    Share_Investment_Method)){
         OrderAccess: DatabaseOperation;
          TradeData: DatabaseOrder;}
      if (supported(Money_Investment_Method))
         InvestmentValue: InvestmentOperation;
      if (supported(MTM_Method || SCompany_Method ||
                    WAV_Method)){
         CalculationRequest: PortfolioMessenger;
         PriceStatus: ValueData;}
      if (supported(MTM_Method))
         MarketValue: ValueOperation;
   }
 } //end of interfaces
sub-system: {
 components {
   EquityPortfolio<Money_Investment_Method,
   Share_Investment_Method, MTM_Method, SCompany_Method,
   WAV_Method>: Portfolio_Db;
   if (supported(Share_Investment_Method))
      SOrderData<true, false>: Order_Generator;
   if (supported(MTM_Method || SCompany_Method ||
                 Weighted_Average_Method))
      ShareData <MTM_Method, SCompany_Method, WAV_Method>:
                                        Internal_EquityData;
  }
 connectors {
   DB_PEquity<false, false>: ODBC_EquityPortfolio;
   if (supported(MTM_Method || SCompany_Method ||
                 Weighted_Average_Method)){
      HTTP_Data<true, false, false, false>:
                                     HTTP_AMSUserInterface;
      HTTP_ERate< >: HTTP_EquityRate;
```

```
      DB_EValue<false, false, E_Share>: ODBC_EquityTrade;}
    if (supported(MTM_Method))
       HTTP_MTM<true, false, false>: HTTP_EquityValuator;
    if (supported(Share_Investment_Method)){
       DB_SOrder<true, true, E_Share>: ODBC_EquityTrade;
       DB_TOrder<false, true>: ODBC_EquityPortfolio;
       DB_EValue<false, false, E_Share>: ODBC_EquityTrade;}
    if (supported(Money_Investment_Method))
       DB_Cash<true, false>: ODBC_EquityPortfolio;
 }
 arrangement {
   connect EquityPortfolio.DataAccess with DB_PEquity.dataport1;
   connect my.DataAccess with DB_PEquity.dataport2;
   if (supported(MTM_Method || SCompany_Method ||
                Weighted_Average_Method)){
      connect ShareData.ValuationRequest with
      HTTP_Data.requestport3;
      connect my.CalculationRequest with HTTP_Data.requestport1;
      connect ShareData.PriceStatus with HTTP_ERate.valueport1;
      connect my.PriceStatus with HTTP_ERate.valueport2;
      connect ShareData.UpdatedData with DB_EValue.dataport1;
      connect EquityPortfolio.CurrentData with
      DB_EValue.dataport2;}
   if (supported(MTM_Method)){
      connect ShareData.MarketValue with HTTP_MTM.valueport2;
      connect my.MarketValue with HTTP_MTM.valueport1;}
   if (supported(Share_Investment_Method)){
      connect SOrderData.OrderAccess with DB_TOrder.dataport4;
      connect my.OrderAccess with DB_TOrder.dataport3;
      connect SOrderData.OrderData with DB_SOrder.dataport3;
      connect my.TradeData with DB_SOrder.dataport4;
      connect SOrderData.UpdatedData with DB_ETrade.dataport1;
      connect EquityPortfolio.CurrentData with
      DB_Trade.dataport2;}
   if (supported(Money_Investment_Method)){
      connect EquityPortfolio.InvestmentValue with
      DB_Cash.dataport3;
      connect my.InvestmentValue with DB_Cash.dataport4;}
  } // end of arrangement
 } // end of sub-system
} // end of component type
```

Portfolio_Processor

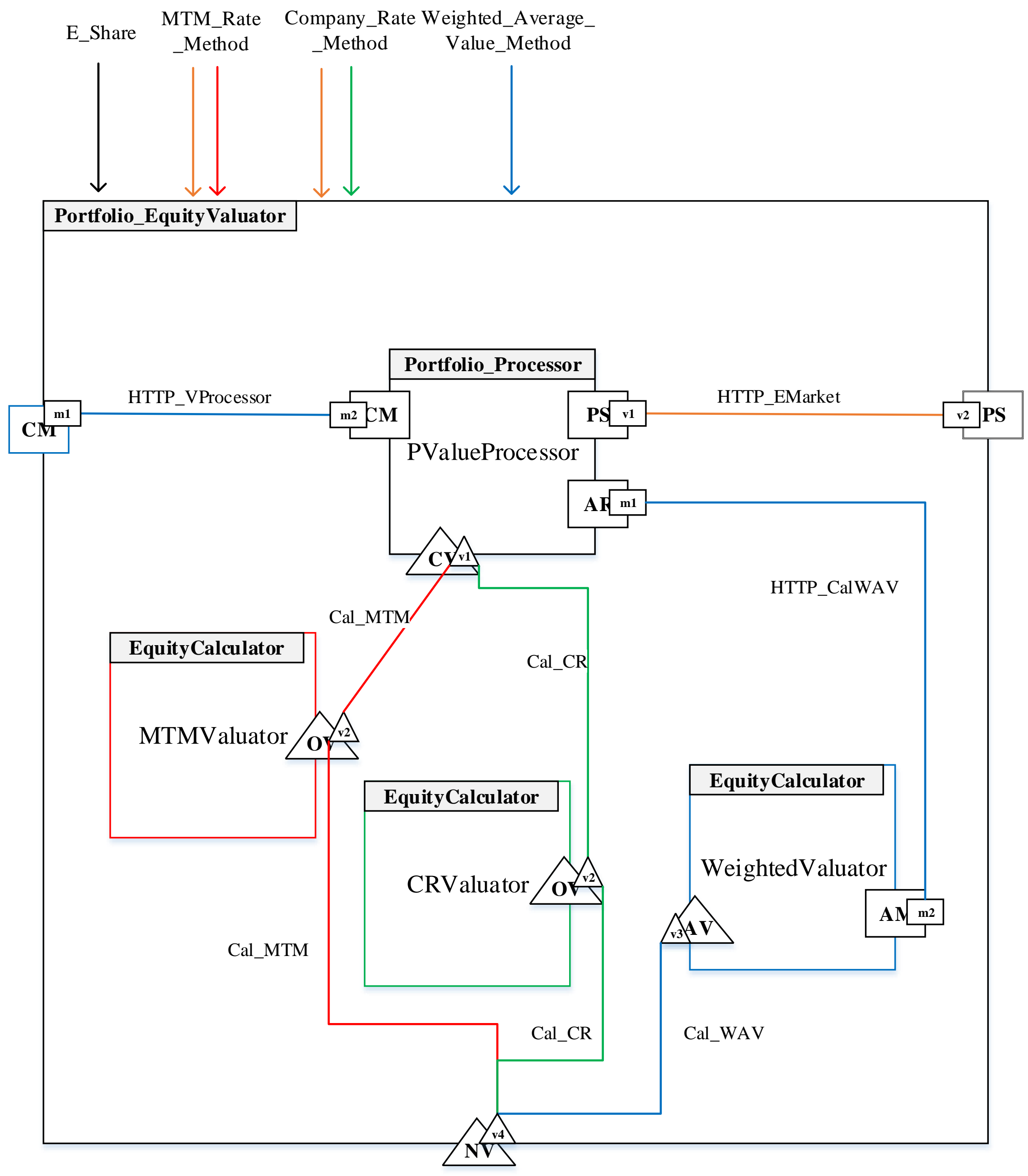



```
component type Portfolio_Processor
  {
   meta: Meta_Processor {
     queuing_method: "FIFO";
     priority_process: "updated portfolio that has been
                        requested";
      }
     features: {
        Money_Investment_Method: "Managing portfolio with cash",
        Financial_Asset: "Tradeable assets to manage portfolio",
        Share_Investment_Method: "Managing portfolio with share
                                  trading",
        MTM_Method: "Share prices matched with market price",
```

```
      SCompany_Method: "Unlisted share price of an individual
                        company",
      Weighted_Average_Method: "Portfolio Valuation is done on
                                the basis of average share
                                price";
    }
  interfaces: {
    definition: {
      //no need to define any interface/s
    }
    implements:{
        DataAccess: DatabaseUpdation;
        NotificationMessage: PortfolioMessenger;
        if (supported(Financial_Asset ||
                      Share_Investment_Method)){
           OrderMessage: PortfolioMessenger;
           OrderAccess: DatabaseOperation;}
        if (supported(Money_Investment_Method))
           InvestmentValue: InvestmentOperation;
        if (supported(MTM_Method || SCompany_Method ||
                      Weighted_Average_Method)){
           CalculationMessage: PortfolioMessenger;
           OperationalValue: ArithmeticOperation;}
         if (supported(MTM_Method || SCompany_Method)){
            CalculationValue: ValueOperation;
            PriceStatus: ValueData;}
       if (supported(Weighted_Average_Method))
          AverageRequest: PortfolioMessenger;
   }
 } //end of interfaces
sub-system: {
 components {
  if (supported(Financial_Asset || Share_Investment_Method))
     ShareOrder<true, true>: Order_Generator;
 }
 connectors {
  if (supported(Financial_Asset || Share_Investment_Method)){
     HTTP_ETrade<false, true>:HTTP_EquityTrade;
     DB_TradeOrder<false, true, true>: ODBC_EquityTrade;}
}
```

```
  arrangement {
   if (supported(Financial_Asset || Share_Investment_Method)){
      connect ShareOrder.OrderMessage with
      HTTP_ETrade.messageport2;
      connect my.OrderMessage with HTTP_ETrade.messageport1;
      connect ShareOrder.OrderAccess with
      DB_TradeOrder.dataport3;
      connect my.OrderAccess with DB_TradeOrder.dataport4;}
  } // end of arrangement
 } // end of sub-system
} // end of component type
```

Portfolio_EquityValuator

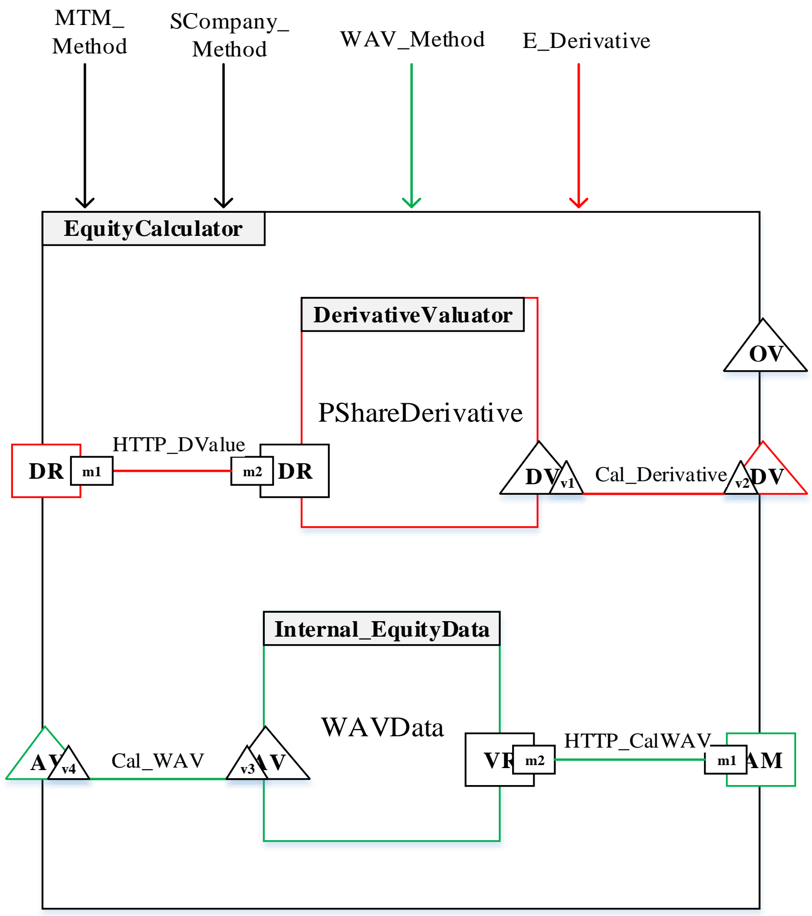



```
component type Portfolio_EquityValuator
  {
   meta: Meta_Valuator, Meta_ShareTradeData {
    // demonstrates meta object comprises of two meta types
    acceptance_value: "any numerical value";
    value_approximation: "2 significant figures";
    curreny_acceptance: "all top international trading currencies
                         that exists in stock exchange";
    last_request: 18-01-2016;
    intention: "to calculate the portfolio value on the basis of
                current business day trading";
```

```
    }
   features: {
     E_Share: "Type of equity in financial instruments",
     MTM_Rate_Method: "Share prices matched with market price"'
     Company_Rate_Method: "Unlisted share price of an individual
                           company",
     Weighted_Average_Value_Method: "Portfolio Valuation is done on
                                 the basis of average share price";
    }
   interfaces: {
    definition: { //No need to define any interface/s }
    implements:{
      NumercialValue: NumericOperation;
      if (supported (MTM_Rate_Method || Company_Rate_Method))
        PriceStatus: ValueData;
      if (supported (Weighted_Average_Value_Method))
         CalculationMessage: PortfolioMessenger;
     }
   } //end of interfaces
 sub-system: {
  components {
   PValueProcessor<false, false, false, true, true, true>:
                                             Portfolio_Processor;
   if (supported(E_Share)) {
    if (supported(MTM_Rate_Method)&&
         unsupported(Weighted_Average_Value_Method))
       MTMValuator<true, false, false, false>: EquityCalculator;
    else if (supported(Company_Rate_Method))
       CRValuator<false, true, false, false>: EquityCalculator;
    else
       WeightedValuator<false, false, true, false>:
                                                EquityCalculator;
    }
   }
  connectors {
    HTTP_EMarket<MTM_Rate_Method, Company_Rate_Method, false>:
                                             HTTP_EquityValuator;
    if (supported(MTM_Rate_Method) &&
        unsupported(Weighted_Average_Value_Method))
       Cal_MTM<true, false, false>: Calculator_Equity;
    else if (supported(Company_Rate_Method))
       Cal_CR<false, true, false>: Calculator_Equity;
```

```
  else {
     HTTP_VProcessor<true, false>: HTTP_Equity;
     HTTP_CalWAV<false, false, true>: HTTP_EquityCalculator;
     Cal_WAV<false, false, true>: Calculator_Equity;}
 }
 arrangement {
   connect PValueProcessor.CalculationMessage with
   HTTP_VProcessor.msgport2;
   connect my.CalculationMessage with HTTP_VProcessor.msgport1;
   if (supported (MTM_Rate_Method || Company_Rate_Method)){
     connect PValueProcessor.PriceStatus with
     HTTP_EMarket.valueport1;
     connect my.PriceStatus with HTTP_EMarket.valueport2;}
   if (supported(MTM_Rate_Method) &&
       unsupported(Weighted_Average_Value_Method)){
     connect PValueProcessor.CalculationValue with
     Cal_MTM.valueport1;
     connect MTMValuator.OperationalValue with
     Cal_MTM.valueport2;
     connect MTMValuator.OperationalValue with
     Cal_MTM.valueport2;}
     connect my.NumericalValue with Cal_MTM.valueport4;
   else if (supported(Company_Rate_Method)) {
     connect PValueProcessor.CalculationValue with
     Cal_CR.valueport1;
     connect CRValuator.OperationalValue with Cal_CR.valueport2;
     connect CRValuator.OperationalValue with Cal_CR.valueport2;
     connect my.NumericalValue with Cal_CR.valueport4;}
  else {
     connect PValueProcessor.AverageRequest with
     HTTP_CalWAV.messageport1;
     connect WeightedValuator.AverageMessage with
     HTTP_CalWAV.messageport2;
     connect WeightedValuator.AverageValue with
     Cal_WAV.valueport3;
     connect my.NumericalValue with Cal_WAV.valueport4;}
  } // end of arrangement
 } // end of sub-system
} // end of component type
```

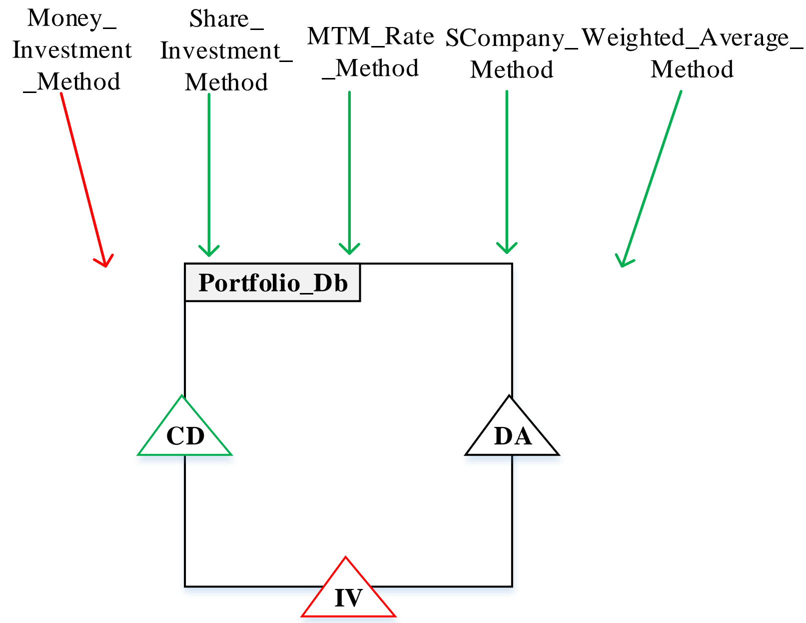
Portfolio_EquityCalculator




```
component type EquityCalculator
  {
   meta: Meta_Valuator {
     acceptance_value: "any numerical value";
     value_approximation: "2 significant figures";
     last_request: 10-02-2016;
    }
   features: {
      MTM_Method: "Share prices matched with market price"'
      SCompany_Method: "Unlisted share price of an individual
                        company",
      WAV_Method: "Portfolio Valuation is done on the basis of
                   average share price",
      E_Derivative: "Used as a security for the equity asset";
    }
   interfaces: {
    definition: { //no need to define any interface/s }
    implements:{
       if (supported(MTM_Method || SCompany_Method))
```

```
            OperationalValue: ArithmeticOperation;
         if (supported(WAV_Method)){
            AverageMessage: PortfolioMessenger;
            AverageValue: AverageOperation;}
         if (supported(E_Derivative)){
            DerivativeRequest: PortfolioMessenger;
            DerivativeValue; DerivativeOperation;}
       }
     } //end of interfaces
   sub-system: {
    components {
      if (supported(E_Derivative))
         PShareDerivative< >: DerivativeValuator;
      if (supported(WAV_Method))
         WAVData <false, false, true>: Internal_EquityData;
     }
    connectors {
      if (supported(E_Derivative)){
         Cal_Derivative< >: Calculator_Derivative;
         HTTP_DValue<true, true>: HTTP_Equity;}
      if (supported(WAV_Method)){
         HTTP_CalWAV<false, false, true>: HTTP_EquityCalculator;
         Cal_WAV<false, false, true>: Calculator_Equity;}
    }
    arrangement {
      if (supported(E_Derivative)){
         connect PShareDerivative.DerivativeRequest with
         HTTP_DValue.messageport2;
         connect my.DerivativeRequest with HTTP_DValue.messageport1;
         connect PShareDerivative.DerivativeValue with
         Cal_Derivative.valueport1;
         connect my.DerivativeValue with Cal_Derivative.valueport2;}
      if (supported(WAV_Method)){
         connect WAVData.ValuationRequest with
         HTTP_CalWAV.messageport2;
         connect my.AverageMessage with HTTP_CalWAV.messageport1;
         connect WAVData.AverageValue with Cal_WAV.valueport3;
         connect my.AverageValue with Cal_WAV.valueport4;}
    } // end of arrangement
   } // end of sub-system
 } // end of component type
```

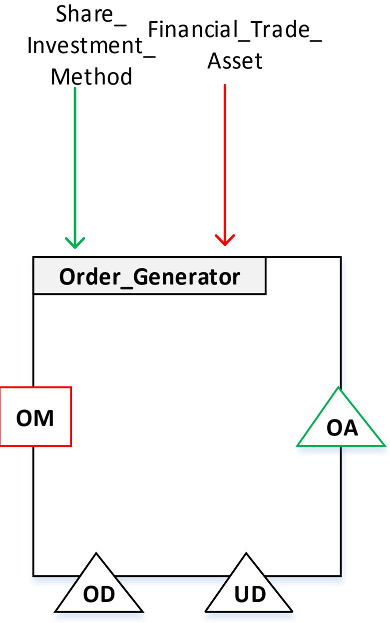
PortfolioDb




```
component type PortfolioDb
  {
   meta: Meta_DbEquity {
     last_updated: 29-02-2016;
     DBA: "Mark";
     description: "stores portfolios of all the financial
                   instruments";
    }
   features: {
      Money_Investment_Method: "Managing portfolio with cash",
      Share_Investment_Method: "Managing portfolio with share
                                trading",
      MTM_Rate_Method: "Share prices matched with market price",
      SCompany_Method: "Unlisted share price of an individual
                        company",
      Weighted_Average_Method: "Portfolio Valuation is done on the
                                basis of average share price";
    }
   interfaces: {
    definition: {
       //no need to define any interface/s
    }
```

```
    implements:{
       DataAccess: DatabaseUpdation;
       if (supported(Share_Investment_Method || MTM_Rate_Method ||
           SCompany_Method || Weighted_Average_Method))
          CurrentData: DatabaseUpdation;
       if (supported(Money_Investment_Method))
          InvestmentValue: InvestmentOperation;
     }
   } //end of interfaces
 sub-system: {
  components {}
  connectors {}
  arrangement {}
 } // end of sub-system
} // end of component type
```

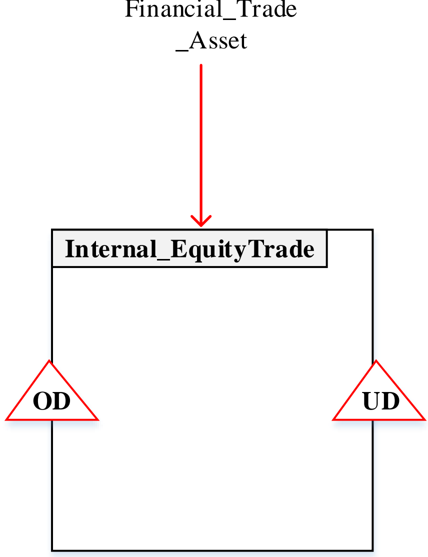
Order_Generator




```
component type Order_Generator
  {
   meta: Meta_Trade {
       updation_frequency: "whenever trade request is made";
       max_request_per_order: 50;
       max_amount_per_order: 10,050;
    }
   features: {
       Financial_Trade_Asset: "Tradeable assets to manage
                               portfolio",
       Share_Investment_Method: "Managing portfolio with share
                                trading;
    }
   interfaces: {
    definition: {
        //no need to define any interface/s
    }
    implements:{
        OrderData: DatabaseOrder;
        UpdatedData: DatabaseUpdation;
        if (supported(Share_Investment_Method))
           OrderAccess: DatabaseOperation;
        if (supported(Financial_Trade_Asset))
           OrderMessage: PortfolioMessenger;
     }
```

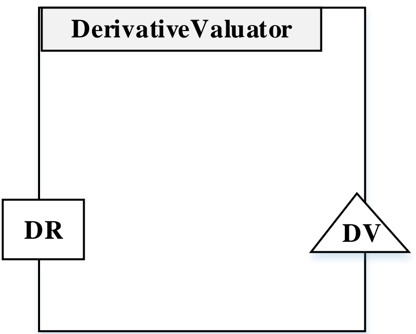
```
    } //end of interfaces
  sub-system: {
   components {}
   connectors {}
   arrangement {}
 } // end of sub-system
} // end of component type
```

Internal_EquityTrade




```
component type Internal_EquityTrade
  {
   meta: Meta_Trade {
     updation_frequency: "whenever trade request is made";
     trade_condition: "internal share trade price should not
                       exceed external share trade price";
    }
   features: {
      Financial_Trade_Asset: "Tradeable assets to manage
                              portfolio";
    }
   interfaces: {
    definition: { //no need to define any interface/s }
    implements:{
       if (supported(Financial_Trade_Asset))
         OrderData: DatabaseOrder;
         UpdatedData: DatabaseUpdation;
     }
   } //end of interfaces
  sub-system: {
   components {}
   connectors {}
   arrangement {}
  } // end of sub-system
} // end of component type
```

Derivative_Valuator




```
component type DerivativeValuator
  {
   meta: Meta_Derivative {
       risk_mitigation: "alpha and beta";
       renewal_deadline: 28-02-2018;
     }
   features: { }
   interfaces: {
    definition: {
        //no need to define any interface/s
     }
    implements:{
        DerivativeRequest: PortfolioMessenger;
        DerivativeValue: DerivativeOperation;
      }
   } //end of interfaces
 sub-system: {
  components {}
  connectors {}
  arrangement {}
 } // end of sub-system
} // end of component type
```

D6: AMS Product Configurations

```
product configurations {
            … // defined in Section 7.3.7
    Equity_Share_Traded: {
           Equity {Equity_Type = long};
           Equity_Share = true;
           MarkToMarket_Method = false;
            Share_Company_Method = true;
          }

    Equity_Share_Investment: {
           Equity {Equity_Type = (long, short)};
           Equity_Share = true;
           Financial_Asset = true;
           Cash_Investment {InvestmentCurrency = GBP};
            Share_Investment {Max_Offer_Quantity = 5,
                              Max_Bid_Quantity = 10};
          }

    Equity_Share_Derivative: {
           Equity {Equity_Type = long};
           Equity_Share = true;
           Equity_Derivative {Derivative_Type = Options,
                              Premium_Period = 1year,
                              OTC = false};
          Share_Sector {Holdings = 100,
                        Total_Share_Value = 1,550,
                        Share_Sector_Category = (Banking,
                                   Pharmaceutical, Automotive)};
         }
   } // end of product configuration
```

D7: AMS Events

```
events {
  ValuationRequest: <WSDL, WSDL>;
  RequestValuationDetails: <MethodInterface, MethodInterface>;
  SendValuationDetails: <MethodInterface, MethodInterface>;
  RequestPrice: <WSDL, WSDL>;
  CurrentStatus: <WSDL, WSDL>;
  RequestPriceList: <WSDL, WSDL>;
  CurrentPrice: <WSDL, WSDL>;
  UpdatedPriceList: <WSDL, WSDL>;
  SendValuation: <MethodInterface, MethodInterface>;
  UpdateValue: <MethodInterface, MethodInterface>;
  Update: <MethodInterface, MethodInterface>;
  Notify: <MethodInterface, MethodInterface>;
  Inform: <(MethodInterface, WSDL), (MethodInterface, WSDL)>;
  Access: <MethodInterface, MethodInterface>;
  RebalanceRequest: <WSDL, WSDL>;
  PortfolioRequest: <MethodInterface, MethodInterface>;
  SendCurentPortfolio: <MethodInterface, MethodInterface>;
  UpdatedPortfolio: <MethodInterface, MethodInterface>;
  CurrentPortfolio: <WSDL, WSDL>;
  WriteOrderList: <MethodInterface, MethodInterface>;
  SendOrderList: <WSDL, WSDL>;
  CurrentStatus: <WSDL, WSDL>;
  TradingRequest: <WSDL, WSDL>;
  OrderRequest: <WSDL, WSDL>;
  PlaceOrder: <(MethodInterface, WSDL), (MethodInterface, WSDL)>;
  OrderUpdate: <(MethodInterface, WSDL), (MethodInterface, WSDL)>;
} // end of events
```

D8: AMS Scenarios

```
scenarios {
    … // defined in Section 7.3.10
   P.RevaluatingPC.ST_IL: {
        Description: “Revaluating portfolio due to change in share
                      price and illiquid shares trading both”;
        Parameterisation: {
                      PriceChanged = true;
                      PriceUnchanged = false;
                      ShareTrade = true;
                      Exchange_Traded = true;
                      Illiquid = true;
                     }
      }

   P.RevaluatingST_ET: {
        Description: “Revaluating portfolio due to exchange
                      trading”;
        Parameterisation: {
                     PriceChanged = false;
                     PriceUnchanged = true;
                     ShareTrade = true;
                     Exchange_Traded = true;
                     Illiquid = false;
                    }
      }

    P.RevaluatingST_IL: {
         Description: “Revaluating portfolio due to illiquid
                       shares trading”;
         Parameterisation: {
                      PriceChanged = false;
                      PriceUnchanged = true;
                      ShareTrade = true;
                      Exchange_Traded = false;
                      Illiquid = true;
                     }
      }
```

```
 P.RebalancingCash: {
      Description: “Portfolio rebalancing is done via cash
                    investment”;
      Parameterisation: {
                   Further_Investment = true;
                   Financial_Instr_Equity = false;
                  }
   }

 P.Rebalancing_EquityInternally: {
      Description: “Portfolio rebalancing is done via financial
                    instrument – equity as internal trading”;
      Parameterisation: {
                   Further_Investment = false;
                   Financial_Instr_Equity = true;
                   OrderFilled = true;
                   OrderForwarded = false;
                  }
   }

P.Rebalancing_EquityExternally: {
     Description: “Portfolio rebalancing is done via financial
                   instrument – equity as external trading”;
     Parameterisation: {
                  Further_Investment = false;
                  Financial_Instr_Equity = true;
                  OrderFilled = false;
                  OrderForwarded = true;
                 }
   }
} // end of scenarios
```

D9: AMS Transaction Domain

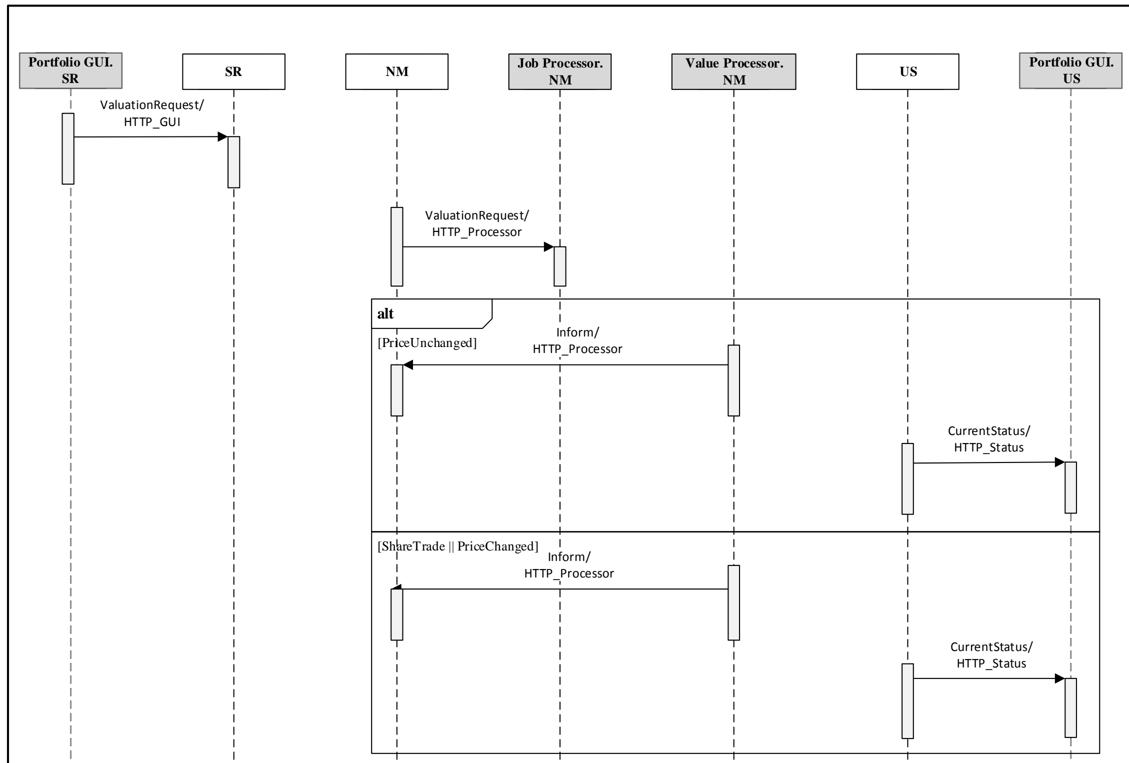
D9.1: Interactions of the components in the transaction domain

PortfolioValuation

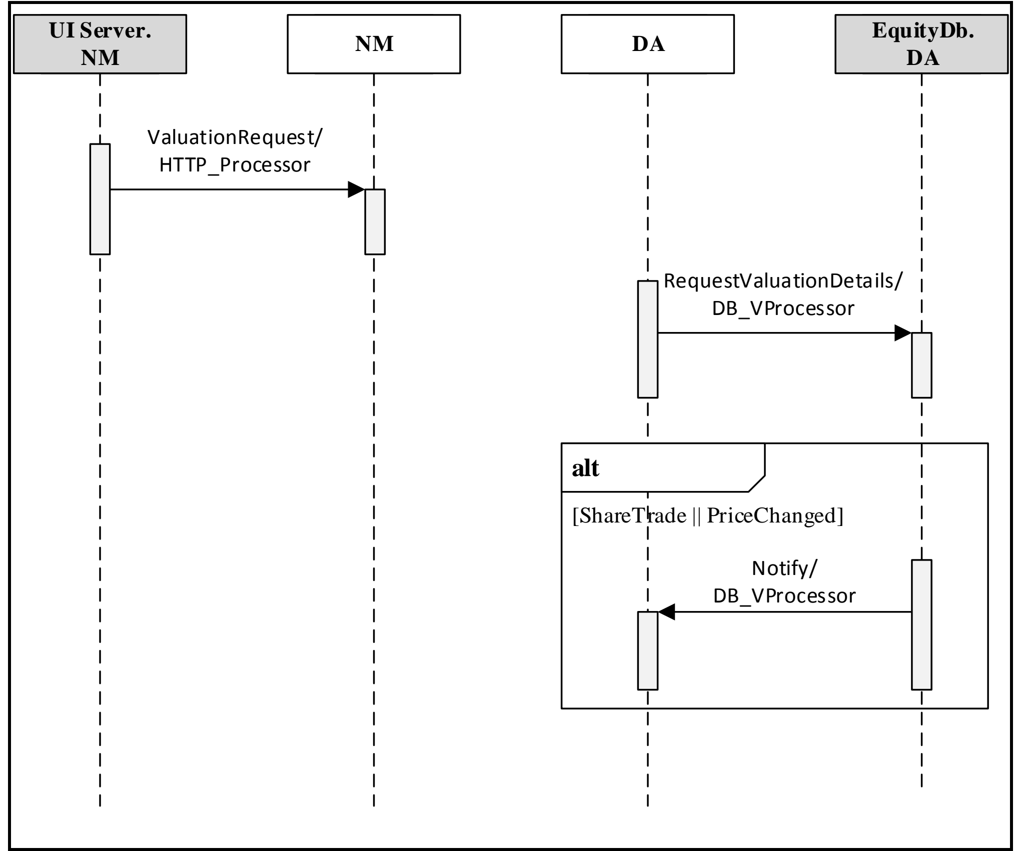
Portfolio GUI

Provided in Section 7.3.11

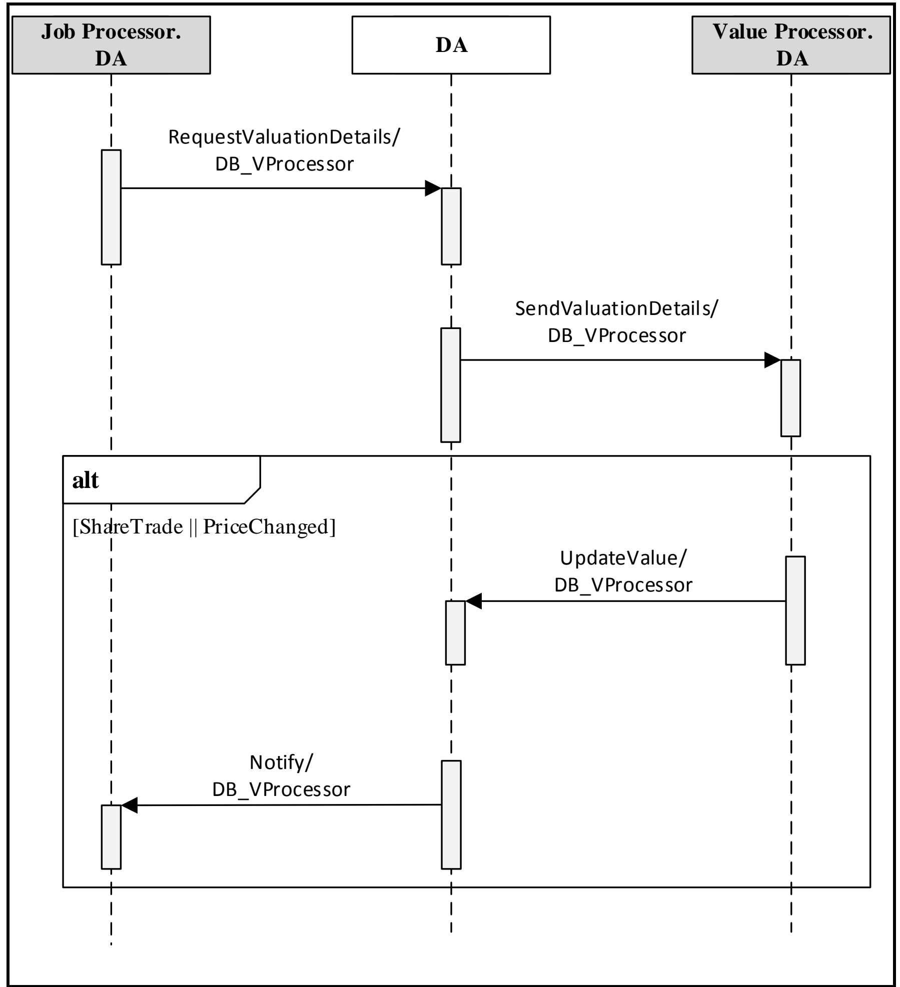
UI Server



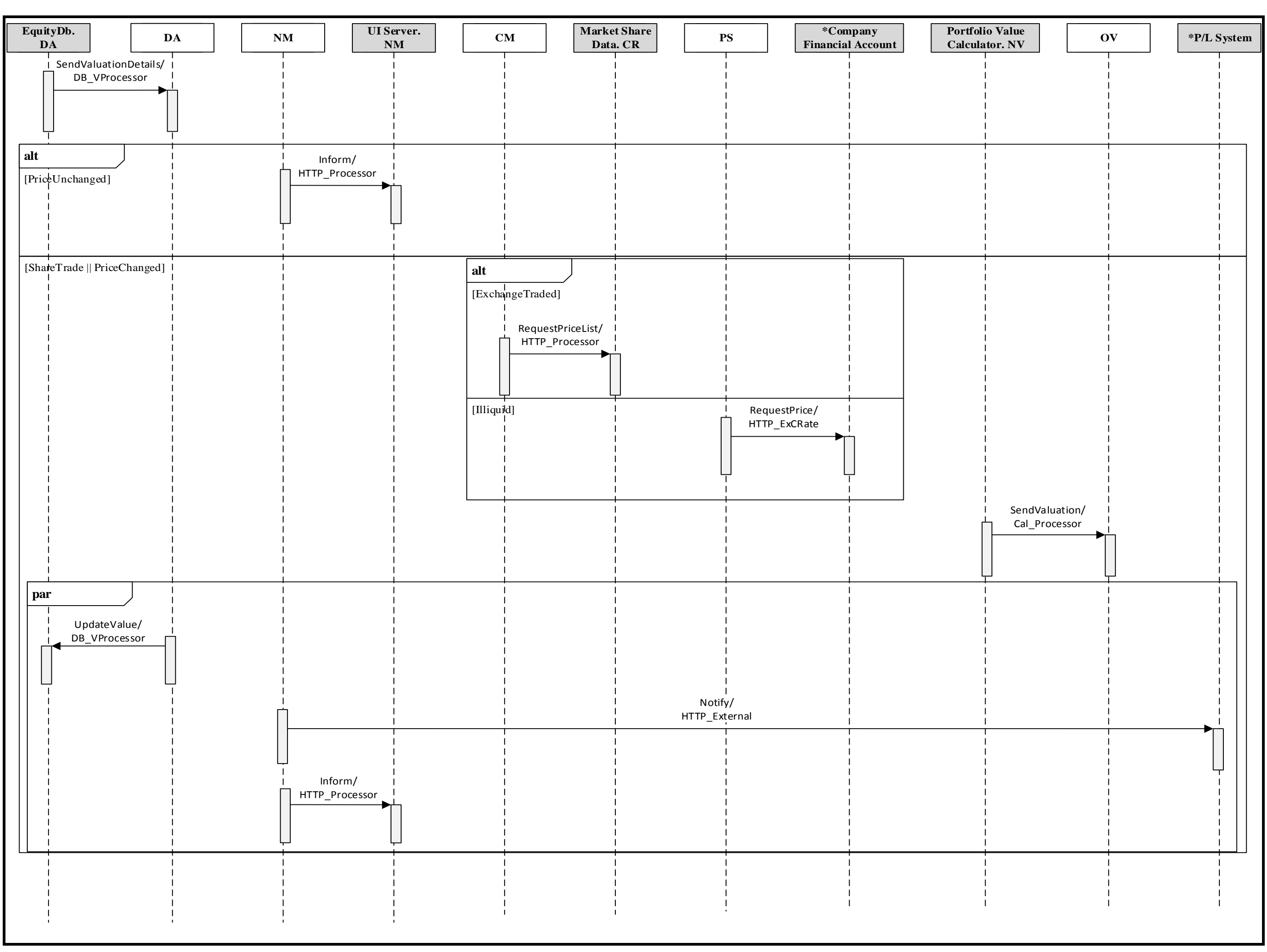
Job Processor



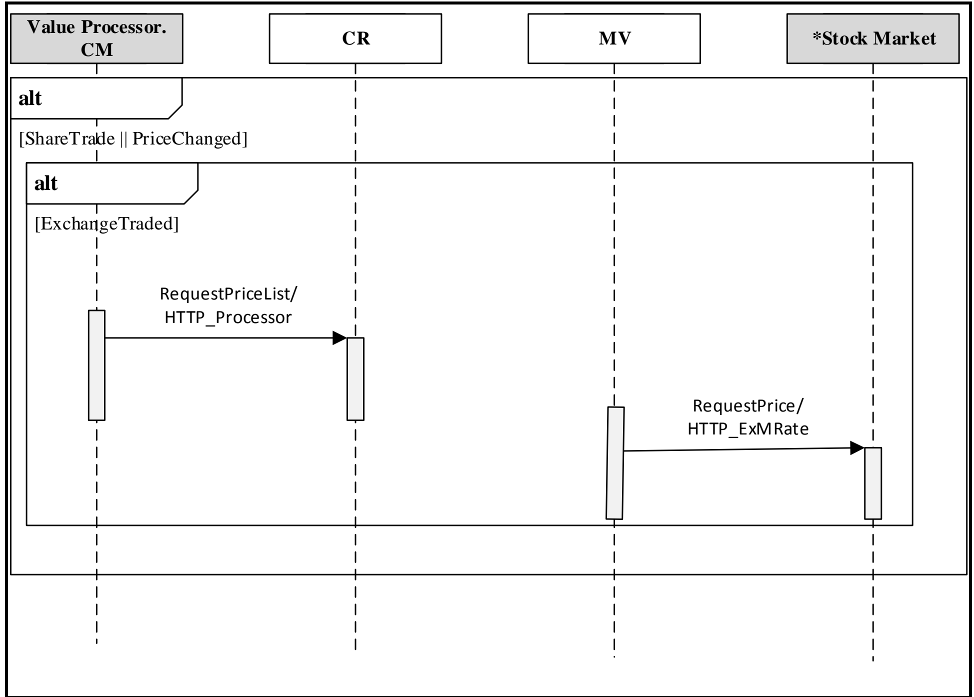
EquityDb



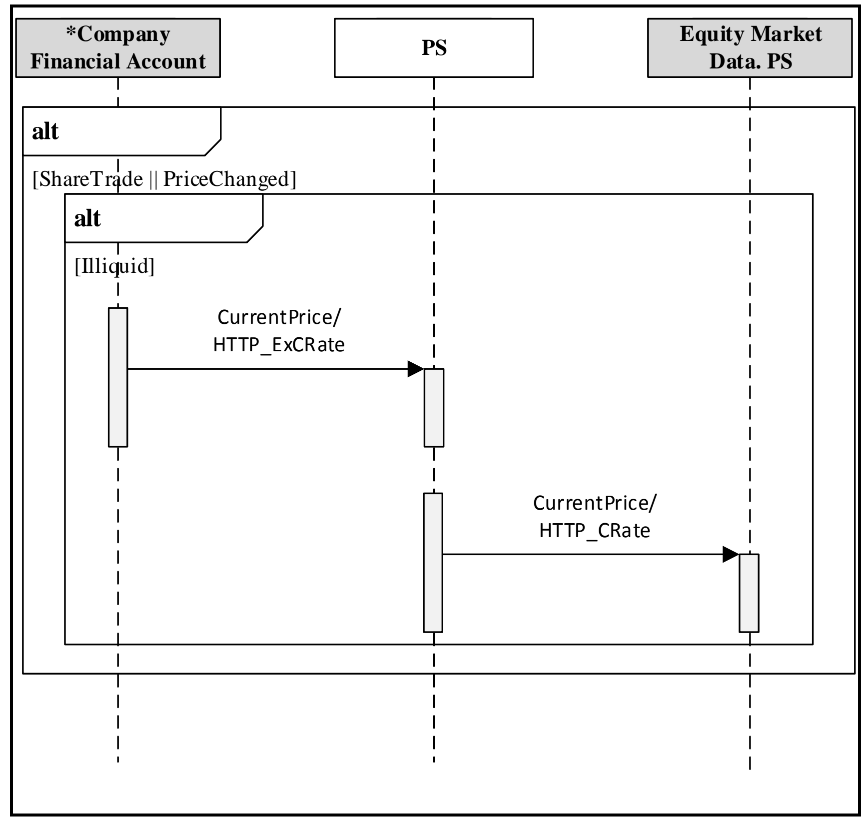
Value Processor



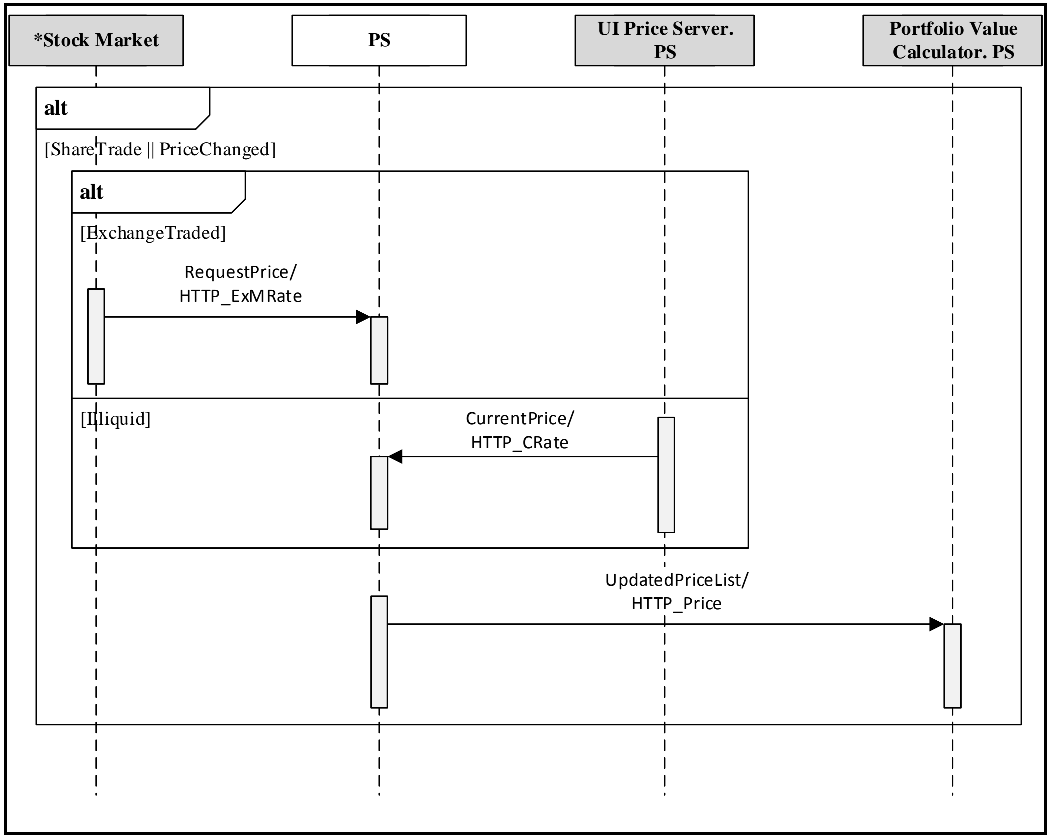
Market Share Data



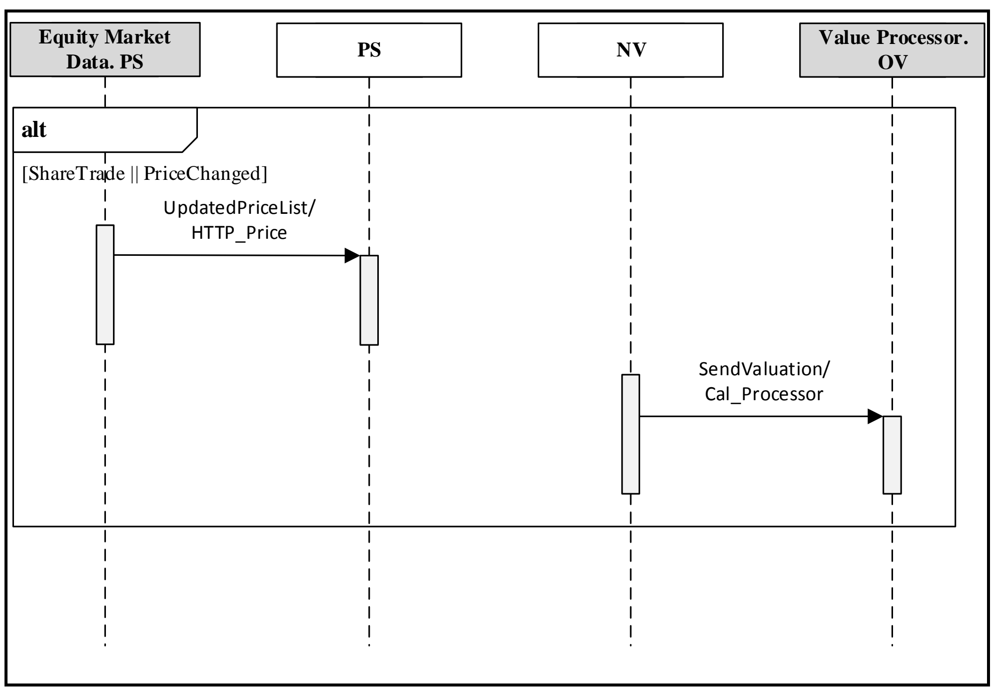
UI Price Server



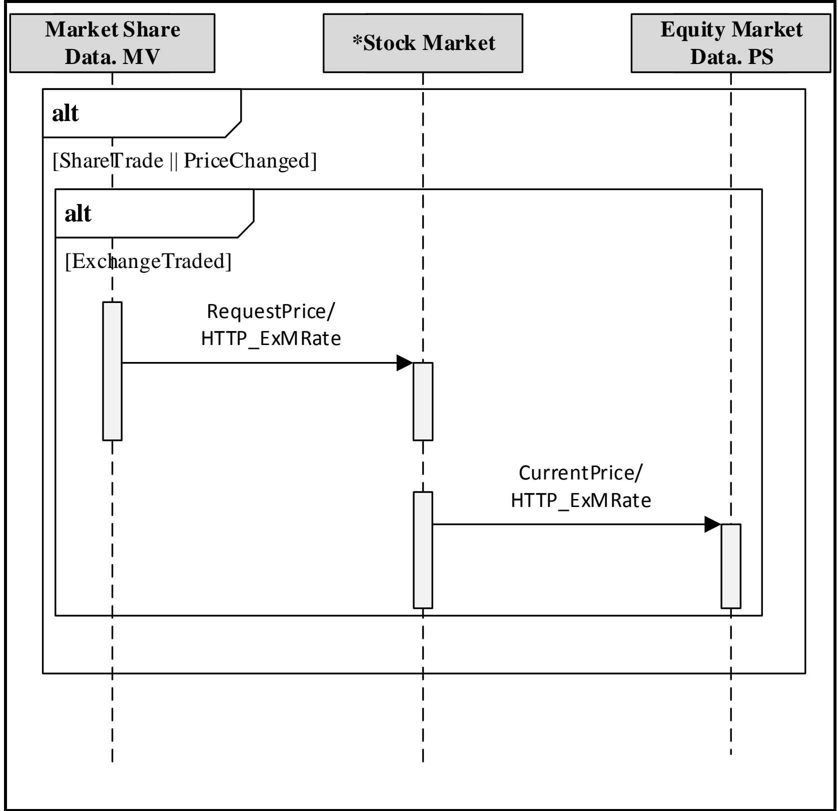
Equity Market Data



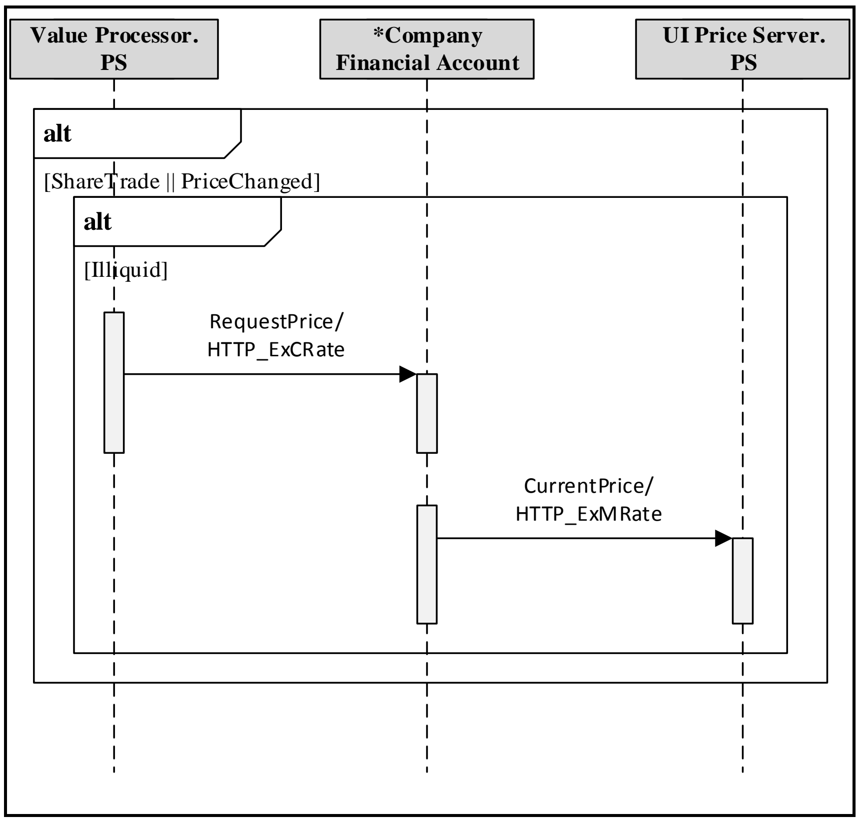
Portfolio Value Calculator



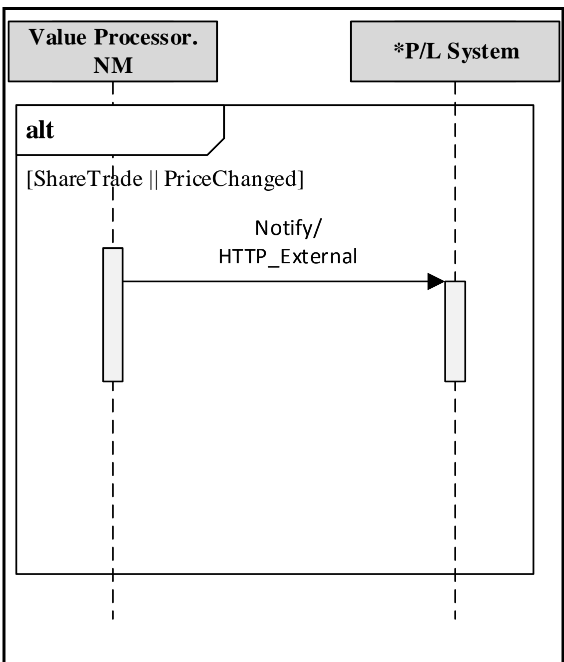
Stock Market



Company Financial Account



P/L System



D9.2: Transaction Domain *PortfolioRebalance*

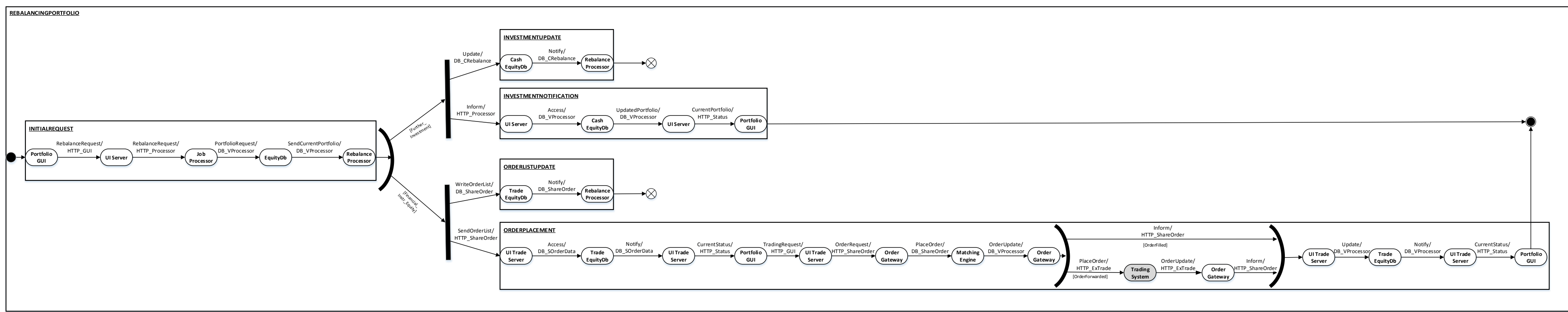
```
transaction domain PortfolioRebalance
  {
   meta: Meta_PortfolioDomain {
     purpose: "To rebalance portfolio";
     compatibility: "financial instrument –equity";
     occurrence: "Depends on the portfolio strategy set by the
                  fund manager";
    }
   contents: {
     /*provides the list of components and connectors involved in
       this transaction domain*/
     Components: {Portfolio_GUI, UI_Server, EquityDb,
                  Job_Processor, Rebalance_Processor,
                  Cash_EquityDb, Trade_EquityDb, UI_Trade_Server,
                  Order_Gateway, Matching_Engine, *Trading_System}
     Connectors: {HTTP_GUI, HTTP_Status, HTTP_Processor,
                  HTTP_ShareOrder, HTTP_ExTrade, DB_VProcessor,
                  DB_CRebalance, DB_ShareOrder}
   }
   transactions:
       {
       INITIALREQUEST:
         {
          events: {RebalanceRequest, PortfolioRequest,
                   SendCurrentPortfolio, Update, Inform,
                   WriteOrderList, SendOrderList}
          interactions: {
            Portfolio_GUI.ServiceRequest sends
            RebalanceRequest/HTTP_GUI to UI_Server.ServiceRequest;
            UI_Server.NotificationMessage sends
            RebalanceRequest/HTTP_Processor to
            Job_Processor.NotificationMessage;
            Job_Processor.DataAccess sends
            PortfolioRequest/DB_VProcessor to EquityDb.DataAccess;
            EquityDb.DataAccess sends
            SendCurrentPortfolio/DB_VProcessor to
            Rebalance_Processor.DataAccess;
            if (supported (Cash_Investment)&&
                          (Further_Investment)){
              [Rebalance_Processor.InvestmentValue sends
```

```
         Update/DB_CRebalance,
        Rebalance_Processor.NotificationMessage sends
        Inform/HTTP_Processor];}
      else {
        [Rebalance_Processor.OrderAccess sends
         WriteOrderList/DB_ShareOrder,
         Rebalance_Processor.OrderMessage sends
         SendOrderList/HTTP_ShareOrder];}
    }
   }
 INVESTMENTUPDATE:
   {
    events: {Update, Notify}
    interactions: {
       EquityDb.DataAccess receives Update/DB_CRebalance;
       EquityDb.DataAccess sends Notify/DB_CRebalance to
       Rebalance_Processor.DataAccess;
    }
   }
 INVESTMENTNOTIFICATION:
   {
    events: {Inform, Access, UpdatedPortfolio,
             CurrentPortfolio}
    interactions: {
       UI_Server.NotificationMessage receives
       Inform/HTTP_Processor;
       UI_Server.DataAccess sends Access/DB_VProcessor to
       Cash_EquityDb.DataAccess;
       EquityDb.DataAccess sends
       UpdatedPortfolio/DB_VProcessor to
       UI_Server.DataAccess;
       UI_Server.UpdationStatus sends
       CurrentPortfolio/HTTP_Status to
       Portfolio_GUI.UpdationStatus;
   }
  }
ORDERLISTUPDATE:
  {
   events: {WriteOrderList, Notify}
   interactions: {
      Trade_EquityDb.OrderAccess receives
      WriteOrderList/DB_ShareOrder;
```

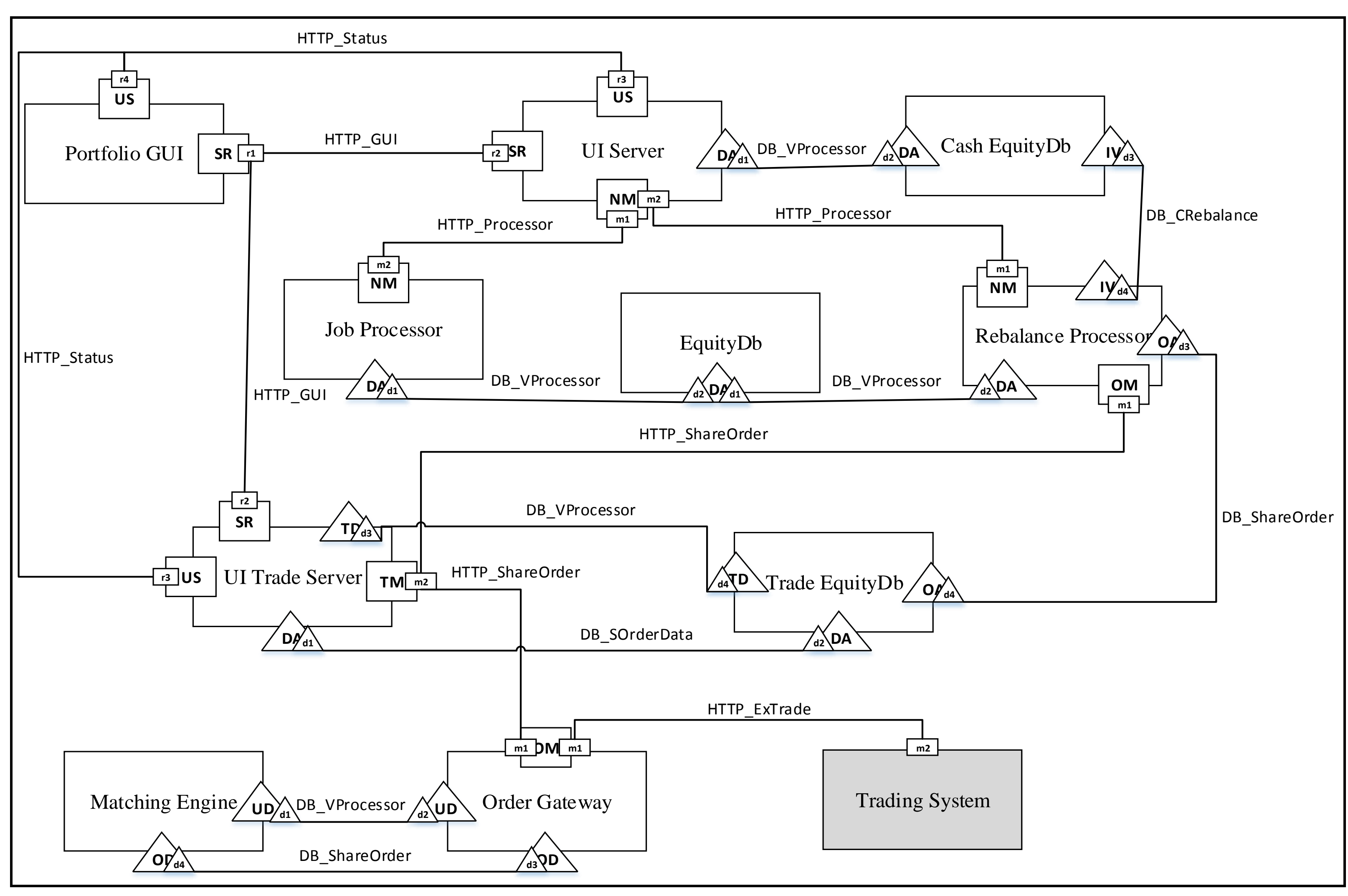
```
        Trade_EquityDb.OrderAccess sends Notify/DB_ShareOrder
        to Rebalance_Processor.OrderAccess;
    }
  }
ORDERPLACEMENT:
    {
     events: {SendOrderList, Access, Notify, CurrentStatus,
              TradingRequest, OrderRequest, PlaceOrder,
              OrderUpdate, Inform, Update}
     interactions:{
        UI_Trade_Server.TradeMessage receives
        SendOrderList/HTTP_ShareOrder;
        UI_Trade_Server.DataAccess sends Access/DB_SOrderData
        to Trade_EquityDb.DataAccess;
        Trade_EquityDb.DataAccess sends Notify/DB_SOrderData to
        UI_Trade_Server.DataAccess;
        UI_Trade_Server.UpdationStatus sends
        CurrentStatus/HTTP_Status to
        Portfolio_GUI.UpdationStatus;
        Portfolio_GUI.ServiceRequest sends
        TradingRequest/HTTP_GUI to
        UI_Trade_Server.ServiceRequest;
        UI_Trade_Server.TradeMessage sends
        OrderRequest/HTTP_ShareOrder to
        Order_Gateway.OrderMessage;
        Order_Gateway.OrderData sends PlaceOrder/DB_ShareOrder
        to Matching_Engine.OrderData;
        Matching_Engine.UpdatedData sends
        OrderUpdate/DB_VProcessor to Order_Gateway.UpdatedData;
        if (supported(Share_Investment)&& (OrderForwarded)) {
           Order_Gateway.OrderMessage sends
           PlaceOrder/HTTP_ExTrade to *Trading_System;
           *Trading_System sends OrderUpdate/HTTP_ExTrade to
           OrderGateway.OrderMessage;}
        UI_Trade_Server.TradeMessage receives
        Inform/HTTP_ShareOrder from Order_Gateway.OrderMessage;
        UI_Trade_Server.TradeData sends Update/DB_VProcessor to
        Trade_EquityDb.TradeData;
        Trade_EquityDb.TradeData sends Notify/DB_VProcessor
        to UI_Trade_Server.TradeData;
        UI_Trade_Server.UpdationStatus sends
        CurrentStatus/HTTP_Status to
```

```
            Portfolio_GUI.UpdationStatus;
         }
        }
  REBALANCINGPORTFOLIO:
      {
        events: {Update, Inform, WriteOrderList, SendOrderList}
        interactions: {
           if (supported (Cash_Investment)&& (Further_Investment)){
              [INVESTMENTUPDATE receives Update/DB_CRebalance from
               INITIALREQUEST,
               INVESTMENTNOTIFICATION receives
               Inform/HTTP_Processor from INITIALREQUEST];}
           else {
             [ORDERLISTUPDATE receives WriteOrderList/DB_ShareOrder
              from INITIALREQUEST,
              ORDERPLACEMENT receives SendOrderList/HTTP_ShareOrder
              from INITIALREQUEST];}
         } //end of interactions
        } //end of transaction
     } //end of transactions
} //end of transaction domain
```

AMS Graphical Behavioural Representation of Transaction Domain *PortfolioRebalance*

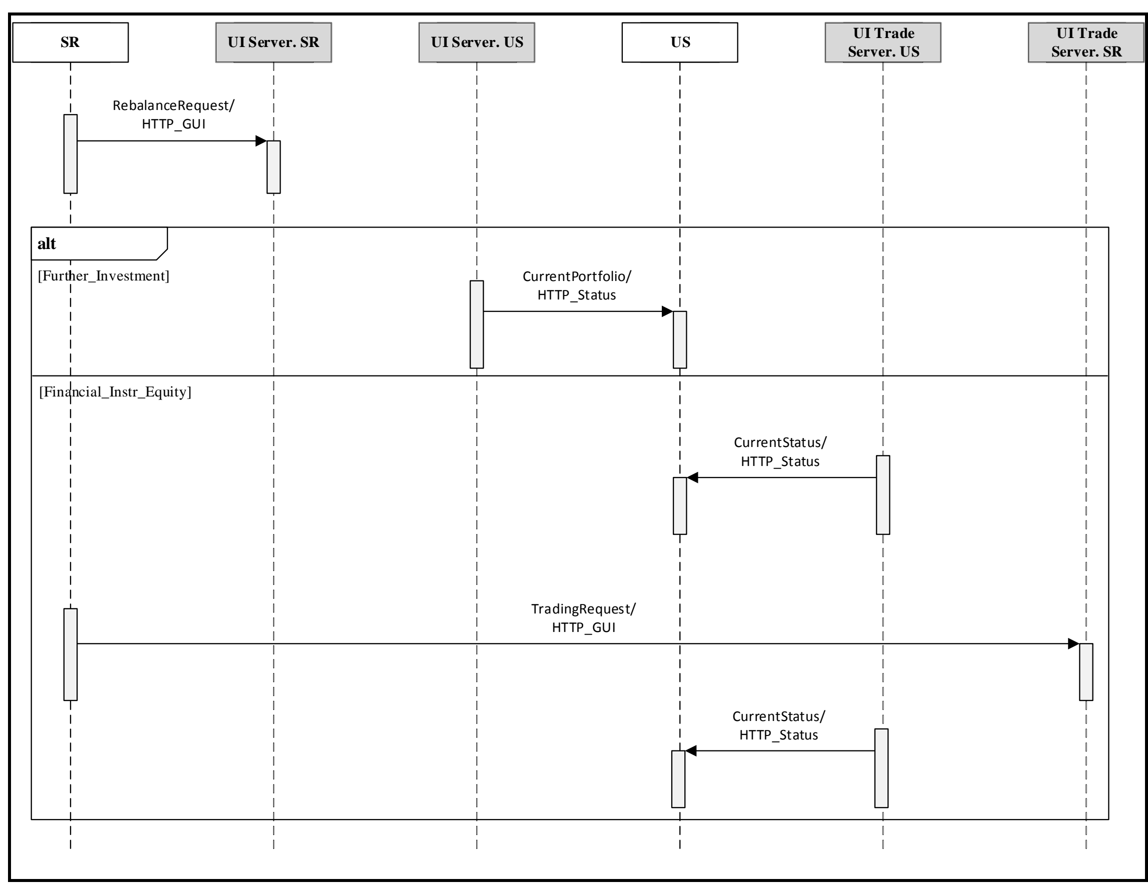


AMS Graphical Structural Representation of Transaction Domain *PortfolioRebalance*

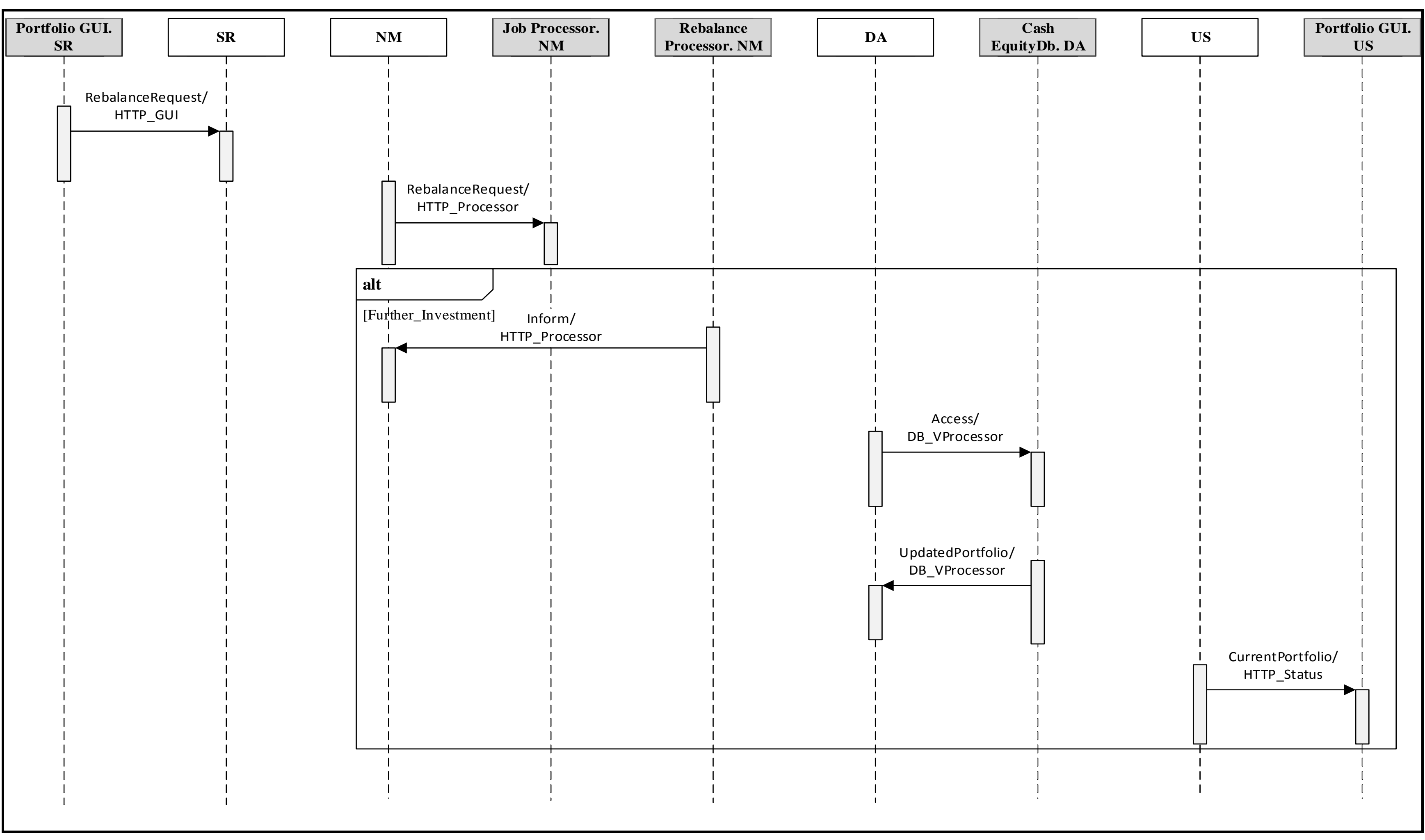


Interactions of the components in the transaction domain *PortfolioRebalance*

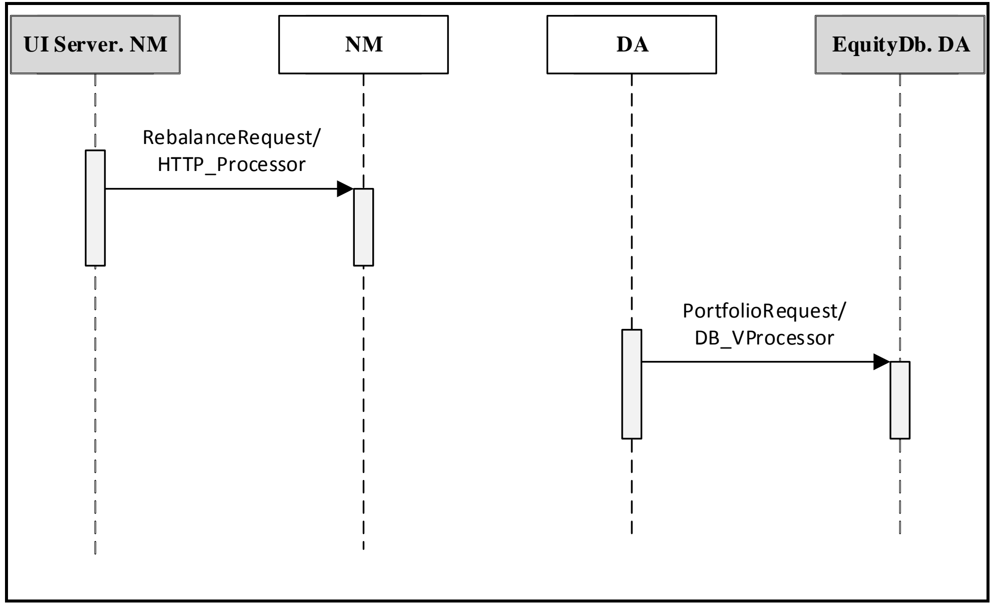
Portfolio GUI



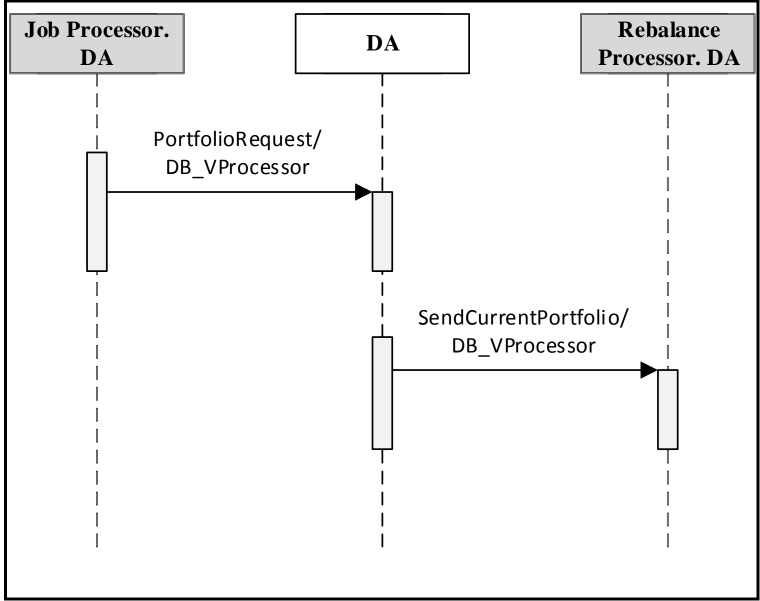
UI Server



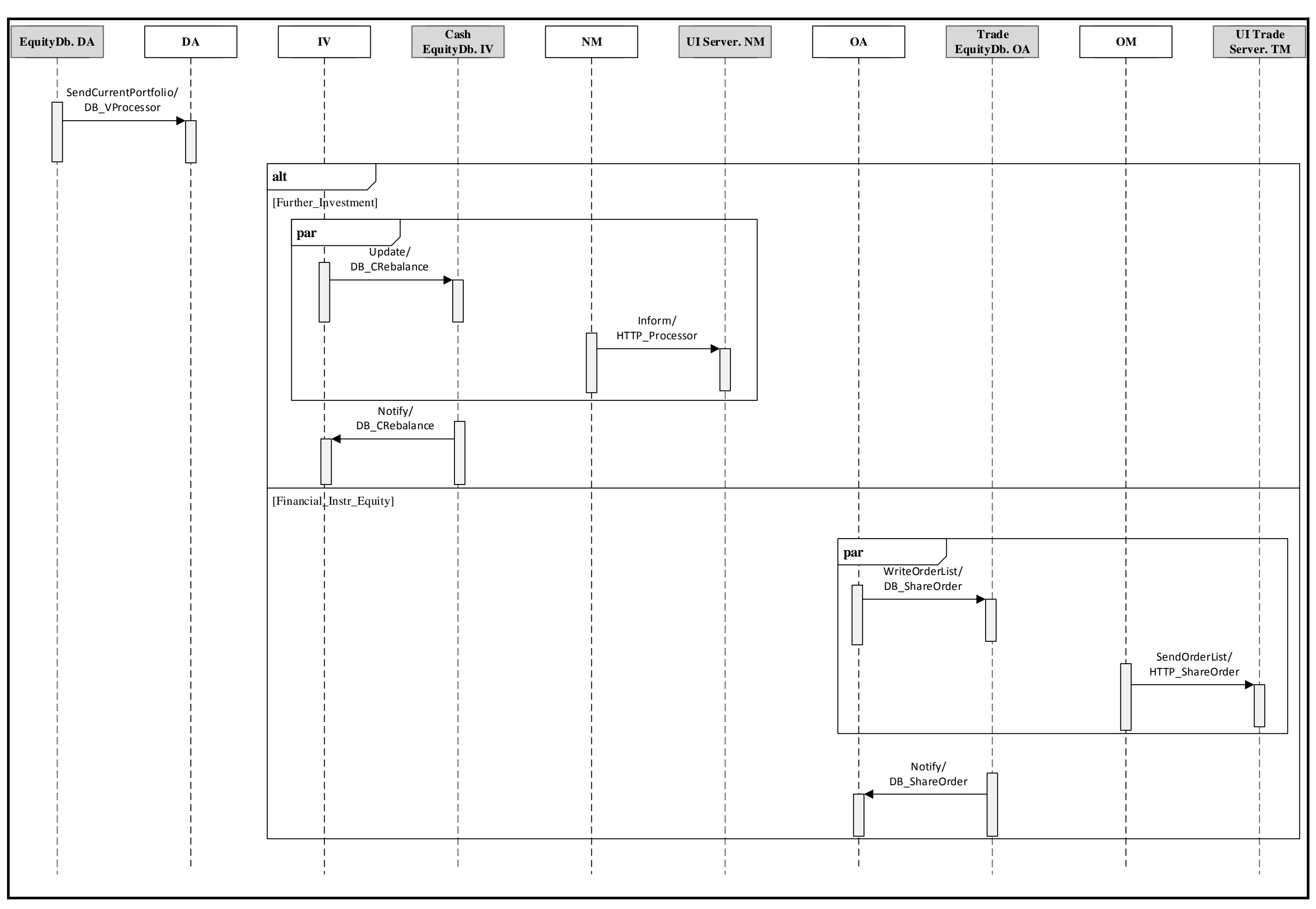
Job Processor



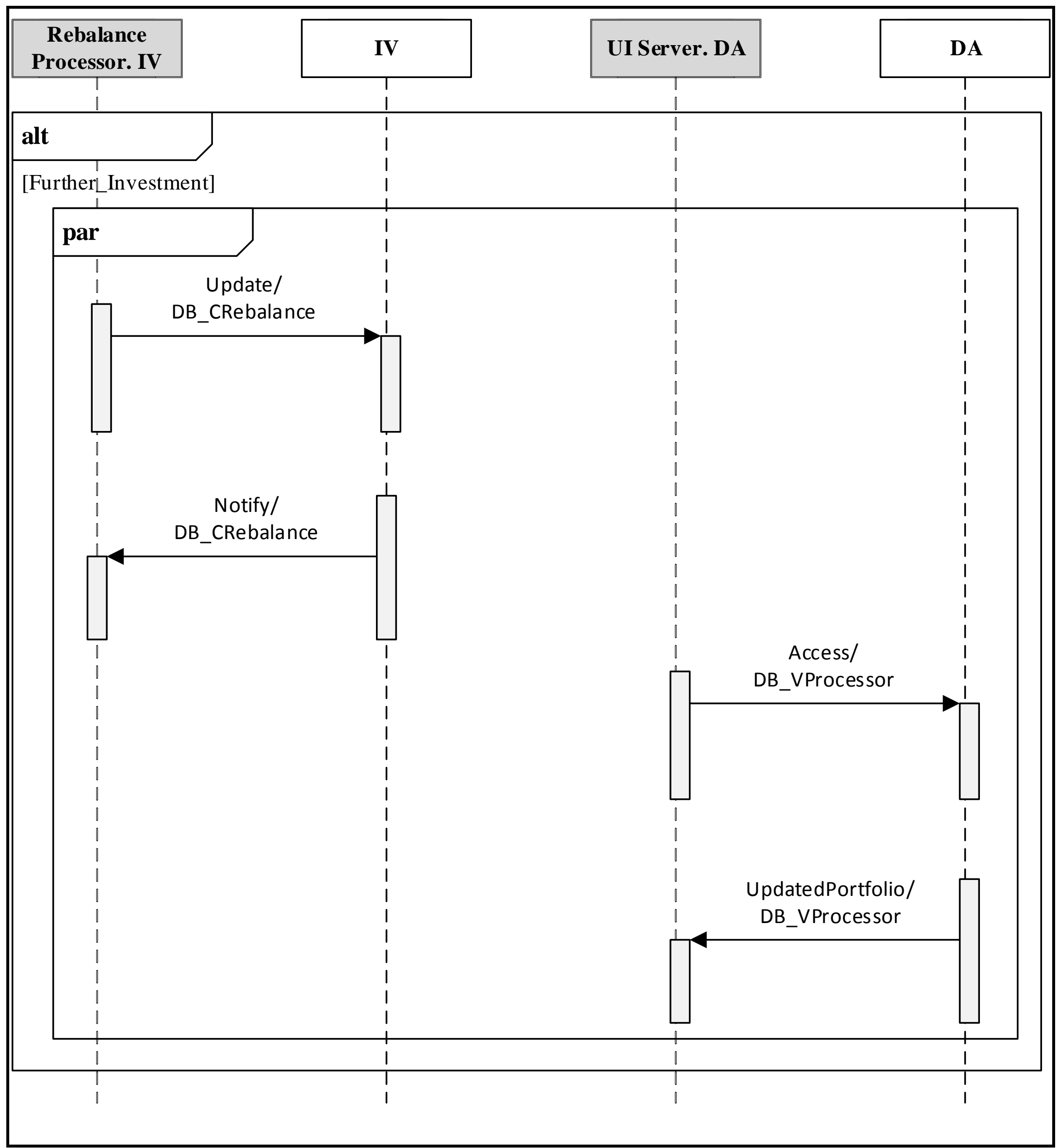
EquityDb



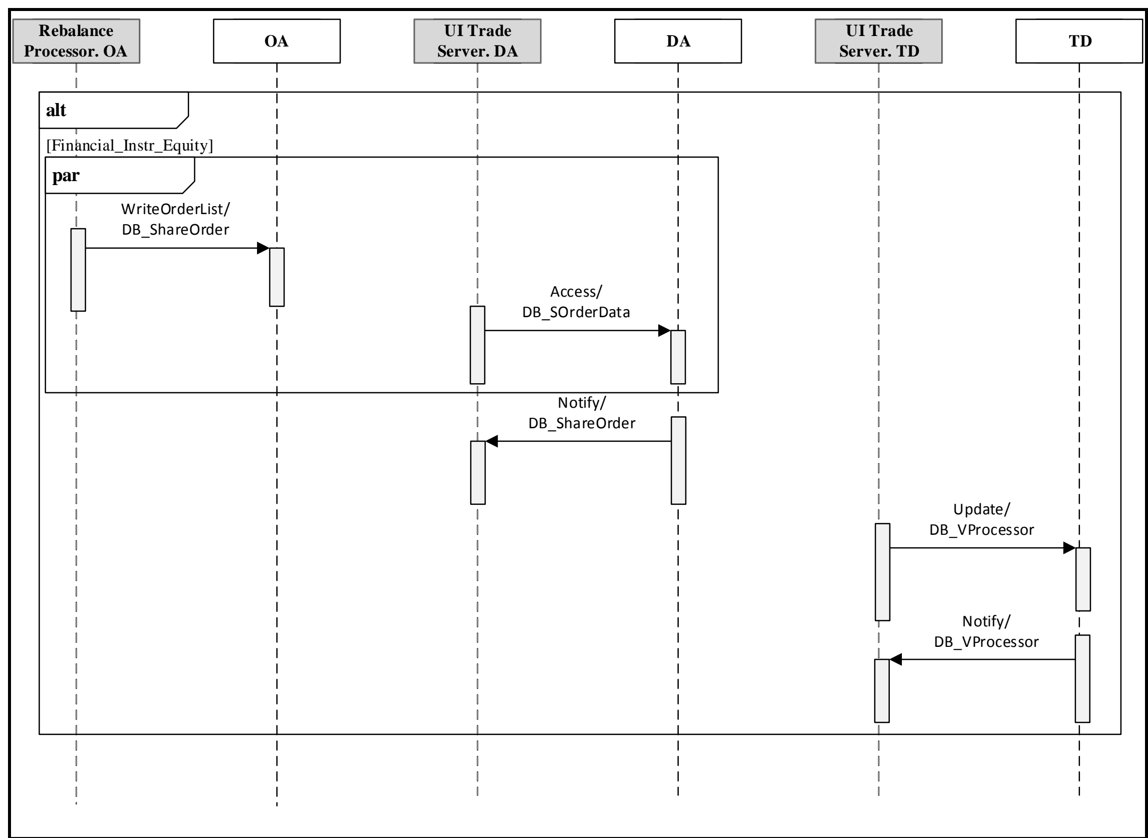
Rebalance Processor



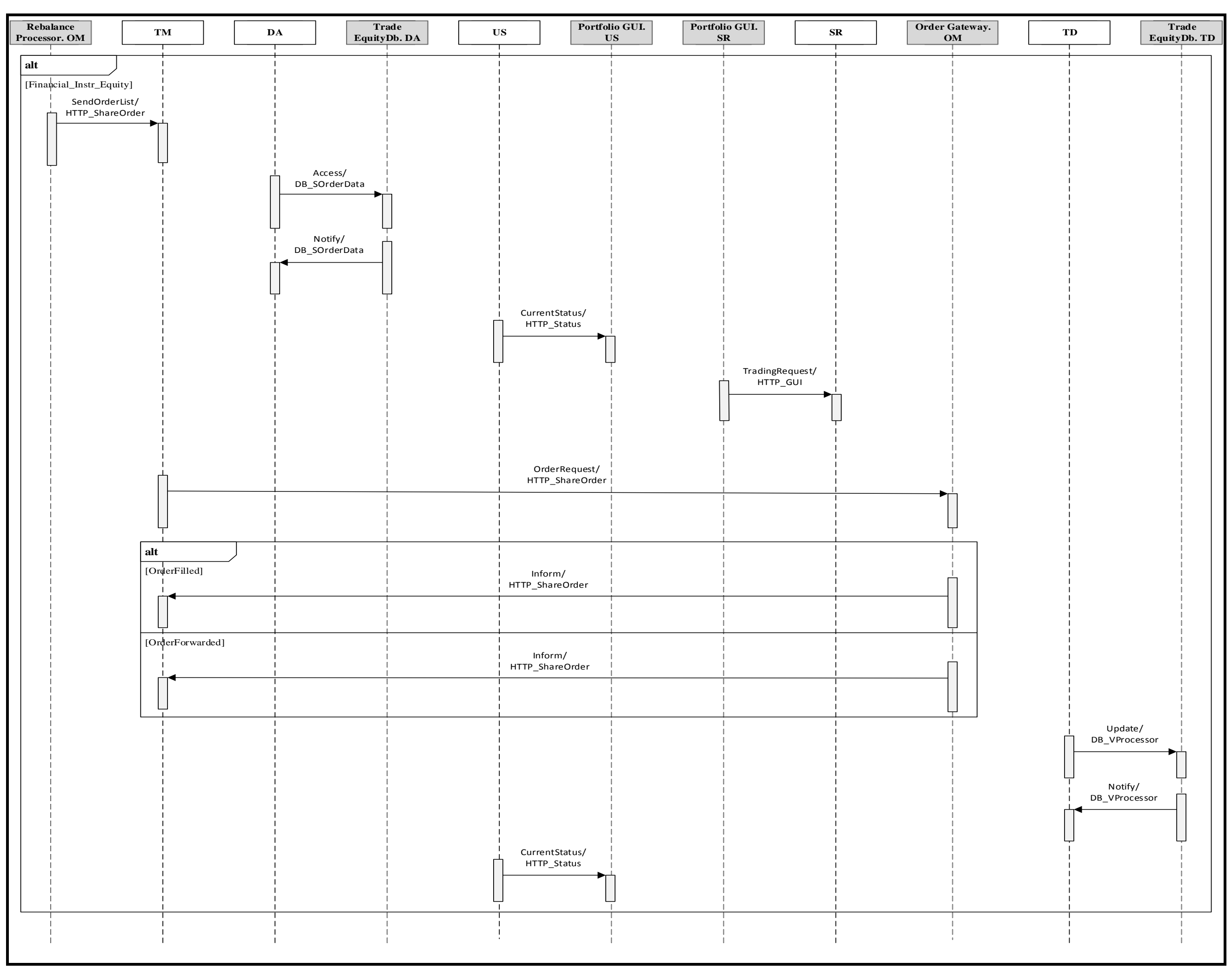
Cash EquityDb



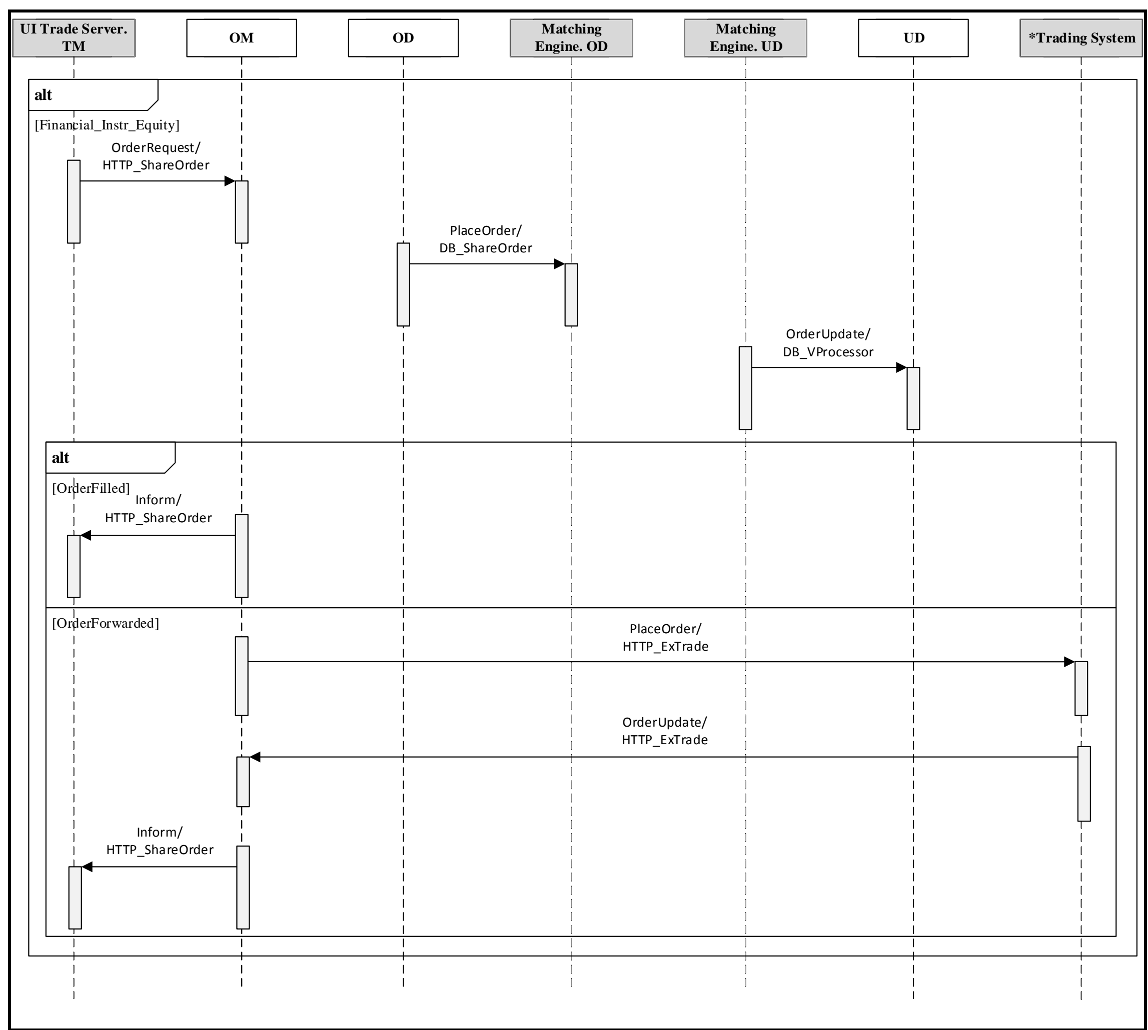
Trade EquityDb



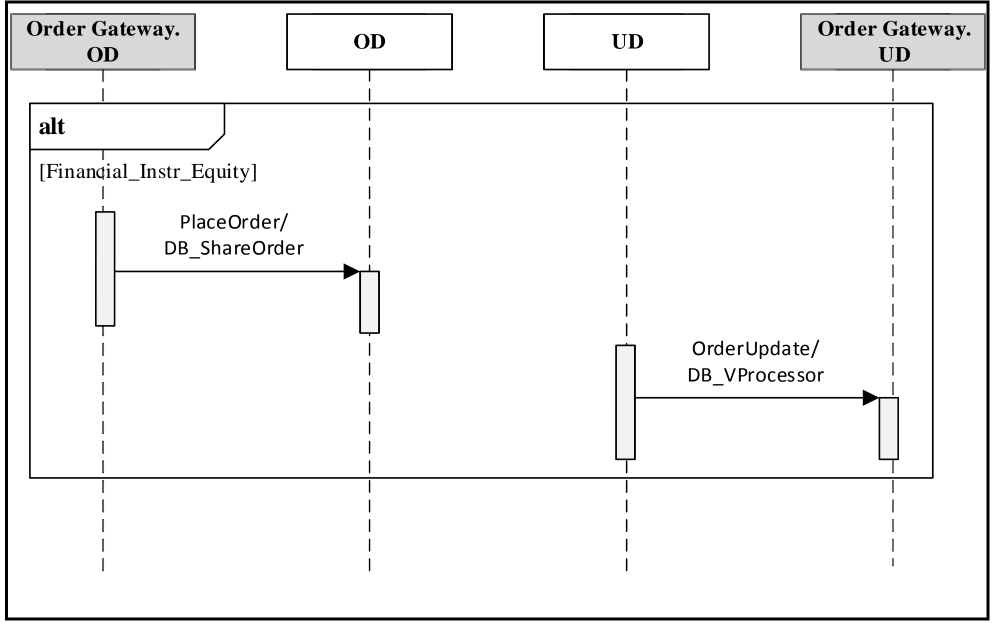
UI Trade Server



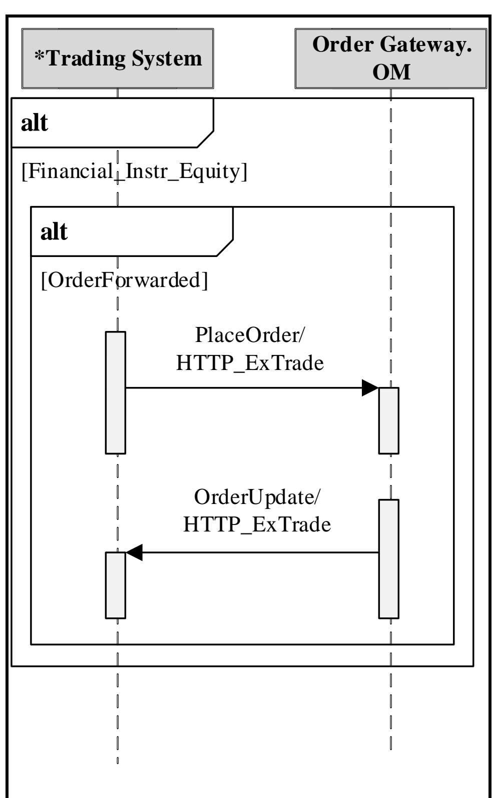
Order Gateway



Matching Engine



****Trading System***



D10: Asset Management System (AMS)

```
system
 {
  components {
    Portfolio_GUI<>: PortfolioAMS_GUI;
    UI_Server<false, false, false>: Portfolio_EquityUIServer;
    Job_Processor<false, false, false, false, false, false>:
                                                Portfolio_Processor;
    EquityDb<true, false, false, false, false, false>:
                                                       AMS_EquityDb;
    if (supported(Equity_Share)){
     // portfolio valuation
      Value_Processor<false, false, false, true, true, false>:
                                                Portfolio_Processor;
    if (supported(MarkToMarket_Method) &&
         unsupported(Share_Company_Method)){
     Market_Share_Data<false, false, false, true, false, false>:
                                                       AMS_EquityDb;
     *Stock_Market;}
    else {
     *Company_Financial_Account;
     UI_Price_Server<false, false, true>: Portfolio_EquityUIServer;}
    }
   Equity_Market_Data<false, false, false, true, true, false>:
                                                       AMS_EquityDb;
   Portfolio_Value_Calculator<true, MarkToMarket_Method,
             Share_Company_Method, false>: Portfolio_EquityValuator;
   *P/L_System;}
    // portfolio rebalancing
   if (supported(Cash_Investment || Share_Investment))
     Rebalance_Processor<Cash_Investment, Financial_Asset,
        Share_Investment, false, false, false>: Portfolio_Processor;
   if (supported(Cash_Investment))
     Cash_EquityDb<true, true, false, false, false, false>:
                                                       AMS_EquityDb;
   if (supported(Share_Investment || Financial_Asset)){
     Trade_EquityDb<true, false, true, false, false, false>:
                                                       AMS_EquityDb;
     UI_Trade_Server<true, false, false>: Portfolio_EquityUIServer;
     Order_Gateway<true, true>: Order_Generator;
     Matching_Engine<true>: Internal_EquityTrade;
```

```
    *Trading_System;}
} // end of components

connectors {
  HTTP_GUI<true, false, false, false>: HTTP_AMSUserInterface;
  HTTP_Processor<true, false>: HTTP_Equity;
  DB_VProcessor<false, false>: ODBC_EquityPortfolio;
  HTTP_Status<false, false, false, false>: HTTP_AMSUserInterface;
  if (supported(Equity_Share)){
   if (supported(MarkToMarket_Method))
    HTTP_ExMRate<false, true, false>: HTTP_ExternalSystem;
   if (supported(Share_Company_Method)){
    HTTP_ExCRate<false, false, true>: HTTP_ExternalSystem;
    HTTP_CRate<false, true, false>: HTTP_EquityValuator;}
  HTTP_Price<true, true, false>: HTTP_EquityValuator;
  Cal_Processor<false, false, false>: Calculator_Equity;
  HTTP_External<false, false, false>: HTTP_ExternalSystem;}
   if (supported(Cash_Investment))
    DB_CRebalance<true, false>: ODBC_EquityPortfolio;
   if (supported(Share_Investment || Financial_Asset)){
    DB_ShareOrder<true, true, Equity_Share>; ODBC_EquityTrade;
    DB_SOrderData<false, true, Equity_Share>; ODBC_EquityTrade;
    HTTP_ShareOrder<false, true>; HTTP_EquityTrade;
    HTTP_ExTrade<true, false, false>: HTTP_ExternalSystem;}
} // end of connectors

 arrangement {
   //similar to component type arrangement
  connect Portfolio_GUI.ServiceRequest with HTTP_GUI.requestport1;
  connect UI_Server.ServiceRequest with HTTP_GUI.requestport2;
  connect UI_Server.UpdationStatus with HTTP_Status.requestport3;
  connect Portfolio_GUI.UpdationStatus with
  HTTP_Status.requestport4;
  connect UI_Server.NotificationMessage with
  HTTP_Processor.messageport1;
  connect Job_Processor.NotificationMessage with
  HTTP_Processor.messageport2;
  connect Job_Processor.DataAccess with DB_VProcessor.dataport1;
  connect EquityDb.DataAccess with DB_VProcessor.dataport2;
  if (supported(Equity_Share)){
   // portfolio valuation
   connect EquityDb.DataAccess with DB_VProcessor.dataport1;
```

```
 connect Value_Processor.DataAccess with
 DB_VProcessor.dataport2;
 // connections for valuation methods
 if (supported(MarkToMarket_Method) &&
    unsupported(Share_Company_Method)){
  connect Value_Processor.CalculationMessage with
  HTTP_Processor.messageport1;
  connect Market_Share_Data.CalculationMessage with
  HTTP_Processor.messageport2;
  connect Market_Share_Data.MarketValue with
  HTTP_ExMRate.valueport1;
  connect *Stock_Market with HTTP_ExMRate.valueport2;
  connect *Stock_Market with HTTP_ExMRate.valueport1;
  connect Equity_Market_Data.PriceStatus with
  HTTP_ExMRate.valueport2;}
 else {
  connect Value_Processor.PriceStatus with
  HTTP_ExCRate.valueport1;
  connect *Company_Financial_Account with
  HTTP_ExCRate.valueport2;
  connect *Company_Financial_Account with
  HTTP_ExCRate.valueport1;
  connect UI_Price_Server.PriceStatus with
  HTTP_ExCRate.valueport2;
  connect UI_Price_Server.PriceStatus with
  HTTP_CRate.valueport1;
  connect Equity_Market_Data.PriceStatus with
  HTTP_CRate.valueport2;}
// connections for portfolio revaluation
connect Equity_Market_Data.PriceStatus with
HTTP_Price.valueport2;
connect Portfolio_Value_Calculator.PriceStatus with
HTTP_Price.valueport1;
connect Portfolio_Value_Calculator.NumericalValue with
Cal_Processor.valueport4;
connect Value_Processor.OperationalValue with
Cal_Processor.valueport1;
connect Value_Processor.NotificationMessage with
HTTP_External.messageport1;
connect *P/L_System with HTTP_External.messageport2;
connect Value_Processor.NotificationMessage with
HTTP_Processor.messageport1;
```

```
connect UI_Server.NotificationMessage with
HTTP_Processor.messageport2;}
if (supported(Cash_Investment || Share_Investment)){
  connect EquityDb.DataAccess with DB_VProcessor.dataport1;
  connect Rebalance_Processor.DataAccess with
  DB_VProcessor.dataport2;}
if (supported(Cash_Investment)){
  connect Rebalance_Processor.InvestmentValue with
  DB_CRebalance.dataport4;
  connect Cash_EquityDb.InvestmentValue with
  DB_CRebalance.dataport3;
  connect Rebalance_Processor.NotificationMessage with
  HTTP_Processor.messageport1;
  connect UI_Server.NotificationMessage with
  HTTP_Processor.messageport2;
  connect Cash_EquityDb.DataAccess with DB_VProcessor.dataport2;
  connect UI_Server.DataAccess with DB_VProcessor.dataport1;}
if (supported(Share_Investment || Financial_Asset)){
  connect Rebalance_Processor.OrderAccess with
  DB_ShareOrder.dataport3;
  connect Trade_Equity_Db.OrderAccess with
  DB_ShareOrder.dataport4;
  connect Rebalance_Processor.OrderMessage with
  HTTP_ShareOrder.messageport1;
  connect UI_Trade_Server.TradeMessage with
  HTTP_ShareOrder.messageport2;
  connect UI_Trade_Server.DataAccess with
  DB_SOrderData.dataport1;
  connect Trade_EquityDb.DataAccess with
  DB_SOrderData.dataport2;
  connect UI_Trade_Server.UpdationStatus with
  HTTP_Status.requestport3;
  connect Portfolio_GUI.UpdationStatus with
  HTTP_Status.requestport4;
  connect Portfolio_GUI.ServiceRequest with
  HTTP_GUI.requestport1;
  connect UI_Trade_Server.ServiceRequest with
  HTTP_GUI.requestport2;
  connect UI_Trade_Server.TradeMessage with
  HTTP_ShareOrder.messageport2;
  connect Order_Gateway.OrderMessage with
  HTTP_ShareOrder.messageport1;
```

```
    connect Order_Gateway.OrderData with DB_ShareOrder.dataport3;
    connect Matching_Engine.OrderData with
    DB_ShareOrder.dataport4;
    connect Matching_Engine.UpdatedData with
    DB_VProcessor.dataport1;
    connect Order_Gateway.UpdatedData with
    DB_VProcessor.dataport2;
    connect Order_Gateway.OrderMessage with
    HTTP_ExTrade.messageport1;
    connect *Trading_System with HTTP_ExTrade.messageport2;
    connect UI_Trade_Server.TradeData with
    DB_VProcessor.dataport3;
    connect Trade_EquityDb.TradeData with
    DB_VProcessor.dataport4;}
  } // end of arrangement

  viewpoints {
    PortfolioInvestment;
  } // end of viewpoints
} // end of AMS
```

Appendix E: WBS Case Study

This section contains the remaining architectural description of the WBS case study presented in Chapter 8 using ALI V2 notations (discussed in Chapter 6).

E1: WBS Meta Types

```
meta type Meta_Brake {
   tag monitored_by, application: text;
   tag battery_charged_on*: date;
 }

meta type Meta_BrakePump {
   tag responsible_technician, failure_rate: text;
   tag threshold_value: number;
 }

meta type Meta_BrakeValve {
   tag average_life, placed_by: text;
   tag service_duedate: date;
 }

meta type Meta_BrakeCU {
   tag processor_manufacturer*, processing_time,
       stand_by_time: text;
   tag processor_version: number;
   tag power_supply_backup: boolean;
 }

meta type Meta_DecelerationDomain {
   tag purpose, minimum_wheels_active: text;
 }
```

E2: WBS Features

```
features {
        … // defined in Section 8.3.2
       Electrical_Brake: {
          alternative names: {
             Designer.AF1, Developer.EB, Evaluator.F12;
           }
          parameters: {
             {Pedal_Value = number};
           }
         }

        Electrical_Power: {
           alternative names: {
              Designer.AF2, Developer.EP, Evaluator.F13;
             }
           parameters: {
              {Voltage = string};
            }
         }

        Mechanical_Brake: {
           alternative names: {
              Designer.AF3, Developer.MP, Evaluator.F14;
            }
           parameters: {
              {Max_Pedal_Force = string};
            }
         }

        Piston_Pressure: {
           alternative names: {
              Designer.AF4, Developer.PP, Evaluator.F15;
             }
           parameters: {
              {Maximum = string,
               Minimum = string};
            }
         }
```

```
      Accumulator_Pressure: {
          alternative names: {
             Designer.AF5, Developer.AP, Evaluator.F16;
          }
          parameters: {
             {Pressure_Supplied = string};
          }
       }
} // end of features
```

E3: WBS Interface Types

```
interface type {
    … // defined in Section 8.3.4
    ElectricOperation: MethodInterface {
        Provider: {
         function SupplyPowerVoltage
                {
                 impLanguage: Java;
                 invocation: voltage;
                 parameterlist: (string);
                 return_type: void;
                }
         }
       }
        Consumer: {
           Call: voltage (string);
        }
     }

     CommandOperation: MethodInterface {
        Provider: {
         function GenerateBrakeCommand
                {
                 impLanguage: Java;
                 invocation: command;
                 parameterlist: (string);
                 return_type: void;
                }
         }
       }
        Consumer: {
           Call: command (string);
        }
     }

     PressureOperation: MethodInterface {
        Provider: {
         function BrakePressureValue
                {
                 impLanguage: Java;
                 invocation: pressure;
```

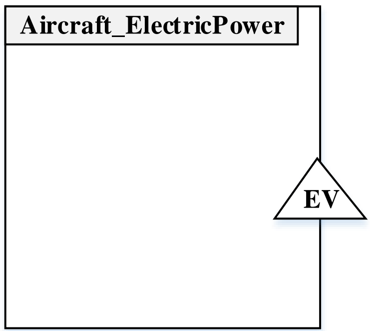
```
                parameterlist: (long_int);
                return_type: void;
              }
        }
      }
       Consumer: {
          Call: pressure (long_int);
       }
    }

    ValueOperation: MethodInterface {
       Provider: {
        function GetPedalValue
              {
                impLanguage: Java;
                invocation: getValue;
                parameterlist: (void);
                return_type: long_int;
              }
        }
       Consumer: {//nothing consumed}
    }

    Notifier: MethodInterface {
       Provider: {
        function PressureCall
              {
                impLanguage: Java;
                invocation: message;
                parameterlist: (string);
                return_type: void;
              }
        }
       Consumer: {
          Call: message (string);
       }
    }
} // end of interface types
```

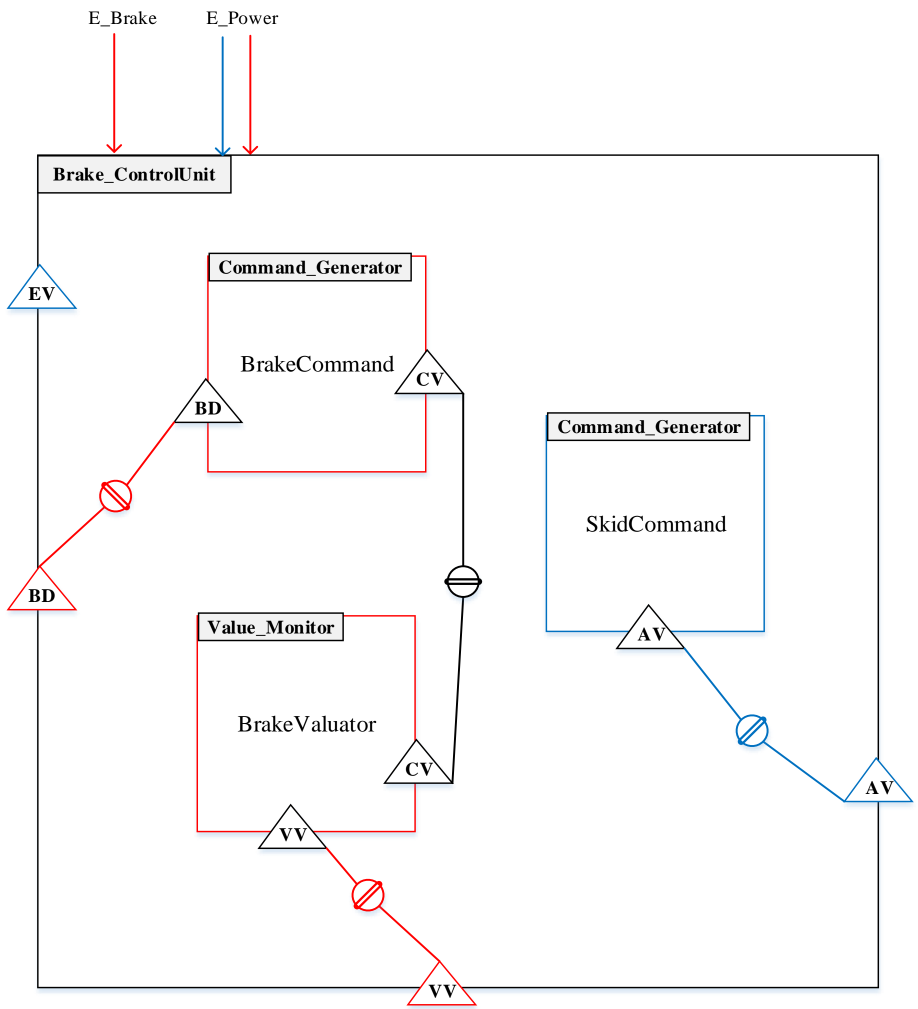
E4: WBS Component Types

Aircraft_ElectricPower




```
component type Aircraft_ElectricPower
  {
   meta: Meta_Brake {
     monitored_by: "David Christopher";
     application: "Provide electrical voltage to brake control
                  unit";
     battery_charged_on: 01-04-2016;
    }
   features: { }
   interfaces: {
     definition: { // no need to define any interface/s
     }
     implements:{
        ElectricVoltage: ElectricOperation;
     }
   } //end of interfaces
   sub-system: {
     components { }
     connectors { }
     arrangement { }
 } // end of sub-system
} // end of component type
```

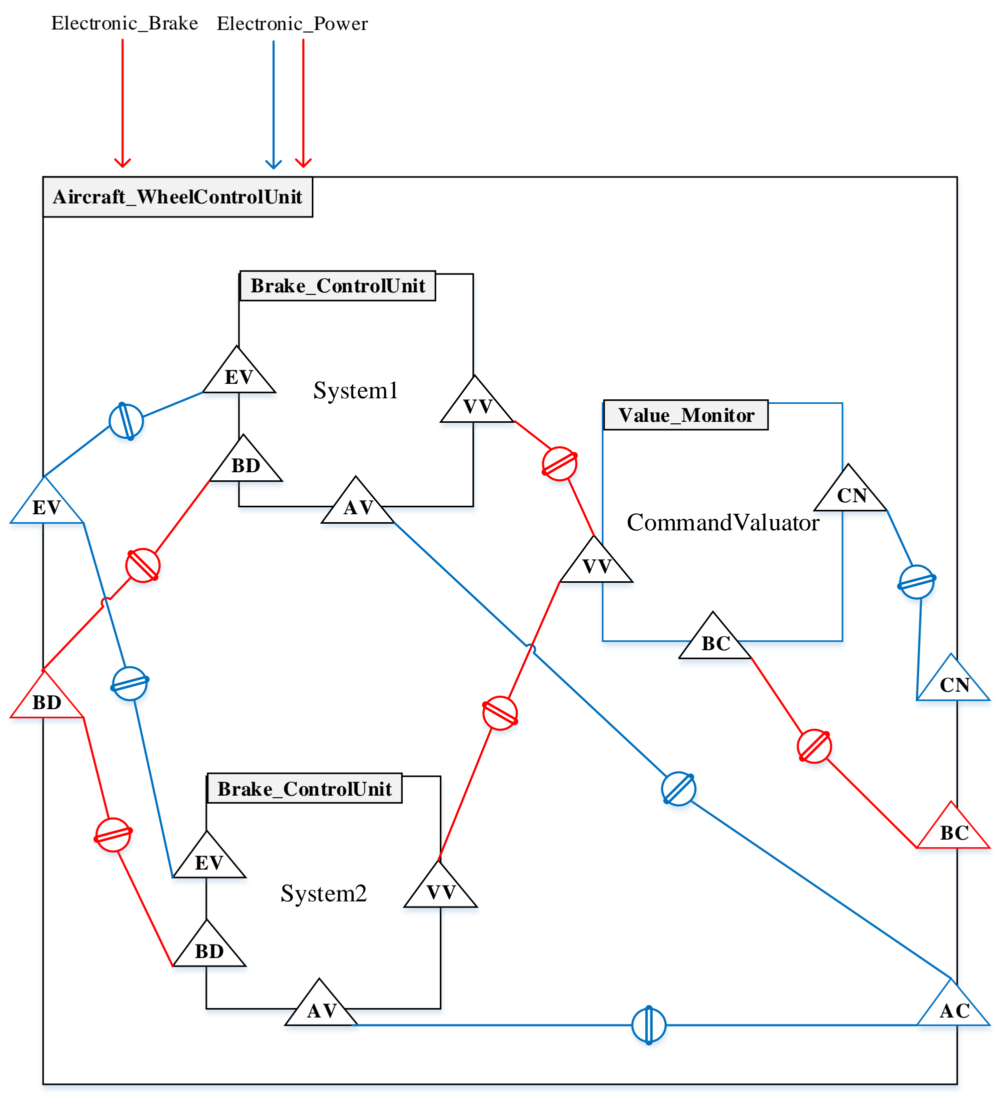
Brake_ControlUnit




```
component type Brake_ControlUnit
  {
   meta: Meta_BrakeCU {
      processing_time: "10bytes/sec";
      stand_by_time: "20 minutes";
      processor_version: 1.1;
      power_supply_backup: true;
    }
   features: {
      E_Brake: "Electrical pedal used to stop the aircraft wheel",
      E_Power: "Electric power supplied to the braking control
                unit system";
    }
```

```
interfaces: {
  definition: {
    // no need to define any interface/s
  }
  implements:{
     if (supported(E_Power)){
       {ElectricVoltage: ElectricOperation;
        AntiskidValue: CommandOperation;}
       if (supported(E_Brake)){
         BrakeData: DataOperation;
         ValidatedValue: ValueOperation;}
     }
} //end of interfaces
sub-system: {
  components {
    if (supported(E_Power)){
      SkidCommand<false, true, false, false, false>:
                                        Command_Generator;
    if (supported(E_Brake)){
      BrakeCommand<true, true, false, false, false>:
                                        Command_Generator;
      BrakeValuator<true, true, false, false>: Value_Monitor;}
   }
  }
  connectors { }
  arrangement {
    bind BrakeCommand.CommandValue with
    BrakeValuator.CommandValue;
    if (supported(E_Power)){
       bind SkidCommand.AntiskidValue with my.AntiskidValue;
       if (supported(E_Brake)){
         bind BrakeCommand.BrakeData with my.BrakeData;
         bind BrakeValuator.ValidatedValue with
         my.ValidatedValue;}
    } // end of arrangement
} // end of sub-system
} // end of component type
```

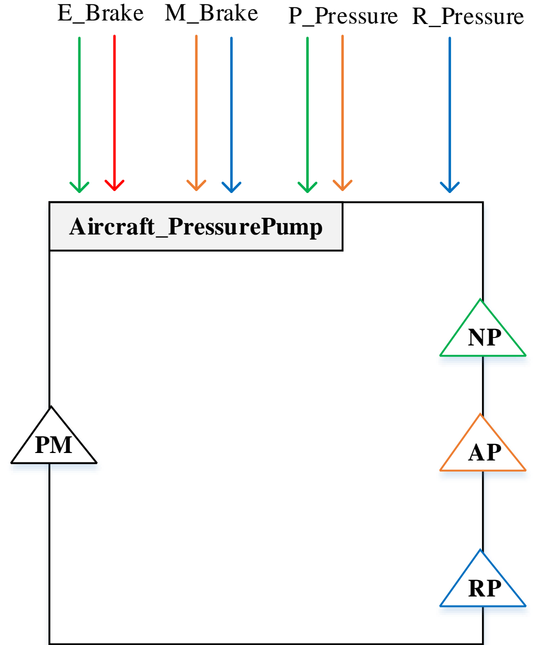
Aircraft_WheelControlUnit




```
component type Aircraft_WheelControlUnit
  {
   meta: Meta_BrakeCU {
      processor_manufacturer: "Intel";
      processing_time: "15bytes/sec";
      stand_by_time: "30 minutes";
      processor_version: 1.3;
      power_supply_backup: true;
    }
   features: {
      Electronic_Brake: "Electrical pedal used to stop the
                         aircraft wheel",
      Electronic_Power: "Electric power supplied to the braking
                         control unit system";
    }
```

```
  interfaces: {
    definition: { // no need to define any interface/s }
    implements:{
       if (supported(Electronic_Power)){
         {ElectricVoltage: ElectricOperation;
          AntiskidCommand: CommandOperation;
          CommandNotification: Notifier;}
         if (supported(Electronic_Brake)){
           BrakeData: DataOperation;
           BrakeCommand: CommandOperation;}
       }
    }
  } //end of interfaces
  sub-system: {
    components {
      System1<Electronic_Brake, Electonic_Power>,
      System2<Electronic_Brake, Electonic_Power>:
                                                  Brake_ControlUnit;
      if (supported(Electronic_Brake && Electonic_Power))
        CommandValuator<true, true, false, false>: Value_Monitor;
    }
    connectors { }
    arrangement {
       if (supported(E_Power)){
         bind System1.ElectricVoltage with my.ElectricVoltage;
         bind System2.ElectricVoltage with my.ElectricVoltage;
         bind System1.AntiskidValue with my.AnitskidCommand;
         bind System2.AntiskidValue with my.AnitskidCommand;
         bind CommandValuator.CommandNotification with
         my.CommandNotification;
         if (supported(E_Brake)){
           bind System1.BrakeData with my.BrakeData;
           bind System2.BrakeData with my.BrakeData;
           bind System1.ValidatedValue with
           CommandValuator.ValidatedValue;
           bind System2.ValidatedValue with
           CommandValuator.ValidatedValue;
           bind CommandValuator.BrakeCommand with
           my.BrakeCommand;}
      } // end of arrangement
  } // end of sub-system
} // end of component type
```

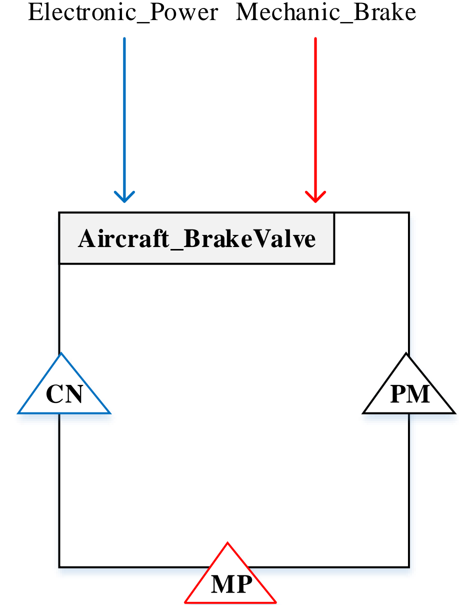
Aircraft_PressurePump




```
component type Aircraft_PressurePump
  {
   meta: Meta_BrakePump {
      responsible_technician: "Keo Yang";
      failure_rate: "0.2% in a year";
      threshold_value: 10.1;
     }
   features: {
      E_Brake: "Electrical pedal used to stop the aircraft wheel",
      M_Brake: "Mechanical pedal applied to stop the aircraft
                wheel",
      P_Pressure: "Pressure supplied by hydraulic pistons",
      R_Pressure: "Supplies stored pressure to the wheel";
    }
   interfaces: {
     definition: {
        // no need to define any interface/s
     }
```

```
      implements:{
         PressureMessage: Notifier;
         if (supported(E_Brake && P_Pressure) unsupported
             (R_Pressure))
           NormalPressure: PressureOperation;
         if (supported(M_Brake)) {
            if (supported(P_Pressure))
              AlternatePressure: PressureOperation;
            else
              ReservePressure: PressureOperation;
           }
        }
    } //end of interfaces
    sub-system: {
      components { }
      connectors { }
      arrangement { }
   } // end of sub-system
 } // end of component type
```

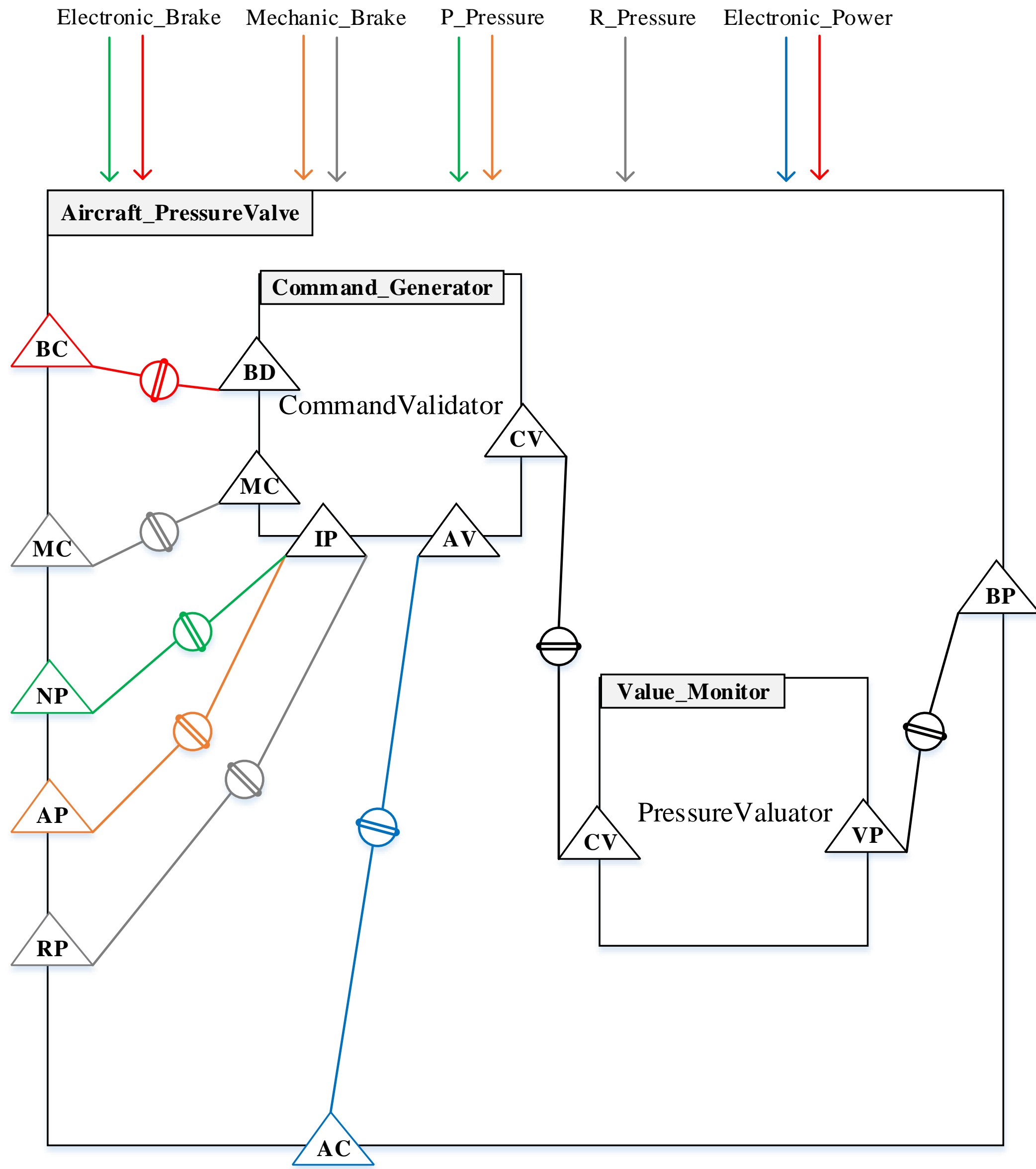
Aircraft_BrakeValve




```
component type Aircraft_BrakeValve
  {
   meta: Meta_BrakeValve {
      average_life: "1.5 years";
      placed_by: "Zach Automotive";
      service_duedate: 22-06-2018;
     }
   features: {
      Electronic_Power: "Electric power supplied to the braking
                         control unit system",
      Mechanic_Brake: "Mechanical pedal applied to stop the
                       aircraft wheel";
    }
   interfaces: {
     definition: {
        // no need to define any interface/s
     }
     implements:{
        PressureMessage: Notifier;
        if (supported(Electronic_Power))
          PressureMessage: Notifier;
        if (supported(Mechanic_Brake))
          MechanicalPosition: ValueOperation;
       }
     }
```

```
    } //end of interfaces
    sub-system: {
      components { }
      connectors { }
      arrangement { }
   } // end of sub-system
} // end of component type
```

Aircraft_PressureValve

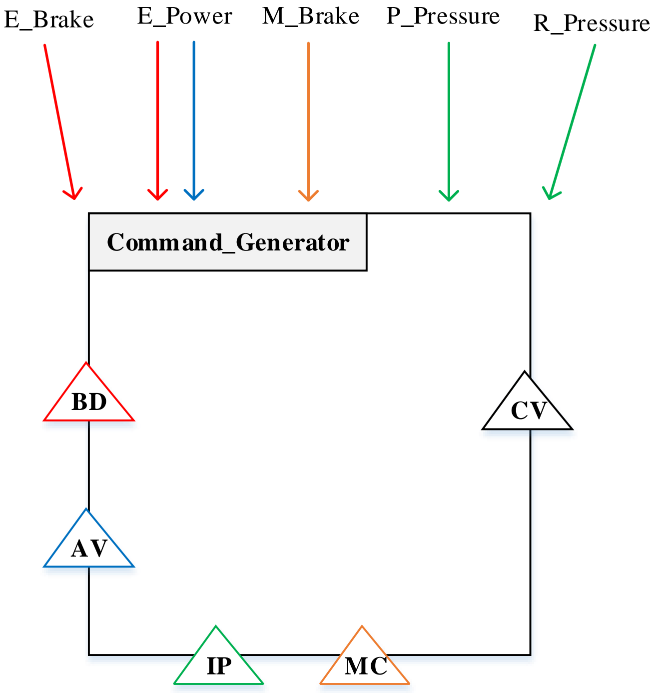



```
component type Aircraft_PressureValve
  {
   meta: Meta_BrakeValve {
       average_life: "2 years";
       placed_by: "RTC Company";
       service_duedate: 23-03-2017;
     }
   features: {
      Electronic_Brake: "Electrical pedal used to stop the
                         aircraft wheel",
      Mechanic_Brake: "Mechanical pedal applied to stop the
                       aircraft wheel",
      P_Pressure: "Pressure supplied by hydraulic pistons",
```

```
   R_Pressure: "Supplies stored pressure to the wheel",
   Electronic_Power: "Electric power supplied to the braking
                      control unit system";
 }
interfaces: {
  definition: {
     // no need to define any interface/s
  }
  implements:{
     BrakePressure: PressureOperation;
     if (supported(Electronic_Brake && Electronic_Power)){
       BrakeCommand: CommandOperation;
       if (supported(P_Pressure))
       NormalPressure: PressureOperation;}
     if (supported(Mechanic_Brake)){
       MechanicalCommand: CommandOperation;
       if (supported(P_Pressure))
          AlternatePressure: PressureOperation;
       else
          ReservePressure: PressureOperation;}
     if (supported(Electronic_Power))
       AntiskidCommand: CommandOperation;
    }
  }
} //end of interfaces
sub-system: {
  components {
    CommandValidator<Electronic_Brake, Electronic_Power,
                    Mechanic_Brake, P_Pressure, R_Pressure>:
                                          Command_Generator;
    PressureValuator<Electronic_Brake, Electronic_Power,
                     P_Pressure, R_Pressure>: Value_Monitor;
  }
  connectors { }
  arrangement {
    bind CommandValidator.CommandValue with
    PressureValuator.CommandValue;
    bind PressureValuator.ValidatedPressure with
    my.BrakePressure;
    if (supported(Electronic_Brake && Electronic_Power)) {
      bind CommandValidator.BrakeData with my.BrakeCommand;
```

```
          if (supported(P_Pressure))
            bind CommandValidator.InputPressure with
            my.NormalPressure;}
         if (supported(Mechanic_Brake)){
           bind CommandValidator.MechanicalCommand with
           my.MechanicalCommand;
           if (supported(P_Pressure))
             bind CommandValidator.InputPressure with
             my.AlternatePressure;
           else
             bind CommandValidator.InputPressure with
             my.ReservePressure;}
         if (supported(Electronic_Power))
           bind CommandValidator.AntiskidValue with
           my.AntiskidCommand;
       } // end of arrangement
    } // end of sub-system
} // end of component type
```

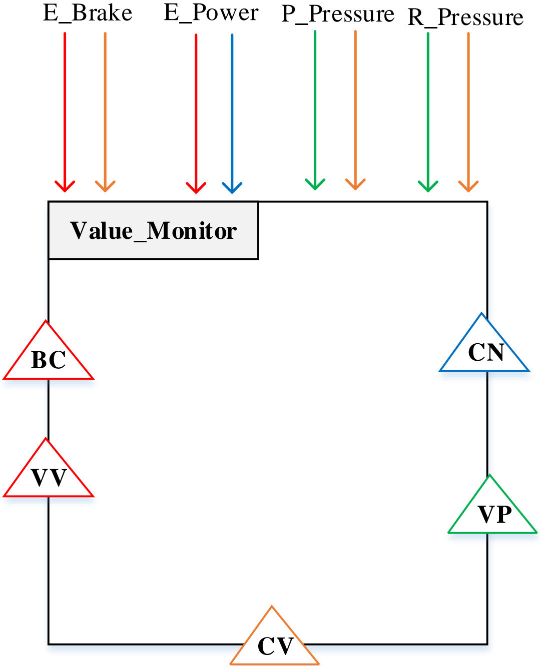
Command_Generator




```
component type Command_Generator
  {
   meta: Meta_Brake {
     monitored_by: "Matthew Johnson";
     application: "To generate brake command value/s";
    }
   features: {
      E_Brake: "Electrical pedal used to stop the aircraft wheel",
      E_Power: "Electric power supplied to the braking control
                unit system",
      M_Brake: "Mechanical pedal applied to stop the aircraft
                wheel",
      P_Pressure: "Pressure supplied by hydraulic pistons",
      R_Pressure: "Supplies stored pressure to the wheel";
    }
   interfaces: {
     definition: {
        // no need to define any interface/s
     }
```

```
    implements:{
       CommandValue: CommandOperation;
       if (supported(E_Power)){
         if (supported(E_Brake))
           BrakeData: DataOperation;
         else
           AntiskidValue: CommandOperation;}
       if (supported(M_Brake))
         MechanicalCommand: CommandOperation;
       if (supported(P_Pressure || R_Pressure))
         InputPressure: PressureOperation;
      }
  } //end of interfaces
  sub-system: {
    components { }
    connectors { }
    arrangement { }
  } // end of sub-system
} // end of component type
```

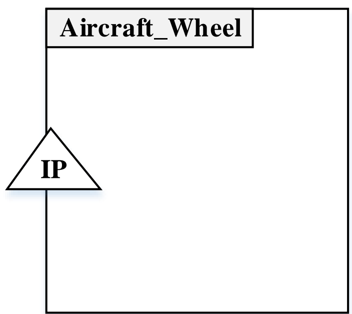
Value_Monitor




```
component type Value_Monitor
  {
   meta: Meta_Brake {
     monitored_by: "Mark James";
     application: "To validate brake command values";
    }
   features: {
      E_Brake: "Electrical pedal used to stop the aircraft wheel",
      E_Power: "Electric power supplied to the braking control
                unit system",
      P_Pressure: "Pressure supplied by hydraulic pistons",
      R_Pressure: "Supplies stored pressure to the wheel";
    }
   interfaces: {
     definition: {
       // no need to define any interface/s
     }
```

```
    implements:{
       if (supported(E_Power)){
        if (supported(E_Brake)){
          {BrakeCommand: CommandOperation;
           ValidatedValue: ValueOperation;}
        else
          CommandNotification: Notifier;}
       else
         CommandValue: CommandOperation;}
       if (supported(P_Pressure || R_Pressure))
        ValidatedPressure: PressureOperation;
    }
  } //end of interfaces
  sub-system: {
    components { }
    connectors { }
    arrangement { }
  } // end of sub-system
} // end of component type
```

Aircraft_Wheel




```
component type Aircraft_Wheel
  {
   meta: { }
   features: { }
   interfaces: {
     definition: {
       // no need to define any interface/s
     }
     implements:{
        InputPressure: PressureOperation;
       }
     }
   } //end of interfaces
   sub-system: {
     components { }
     connectors { }
     arrangement { }
  } // end of sub-system
} // end of component type
```

E5: Scenarios

```
scenarios {
     … // defined in Section 8.3.9
   AlternateOperation {
      Description: "WBS is in alternate mode with Antiskid
                    command";
      Parameterisation {
                        BSCU_Active = true;
                        GreenPressure_Failed = true;
                        BluePressure = true;
                        AccumulatorPump = false;
                       }
     }

   BSCUFailureOperation {
      Description: "WBS is in alternate mode without Antiskid
                    command";
      Parameterisation {
                        BSCU_Failed = true;
                        GreenPressure = true;
                        BluePressure = true;
                        AccumulatorPump = false;
                       }
     }

   EmergencyOperation {
      Description: "WBS is in emergency mode";
      Parameterisation {
                        BSCU_Failed = true;
                        GreenPressure_Failed = true;
                        BluePressure_Failed = true;
                        AccumulatorPump = true;
                       }
    }
  } // end of scenarios
```

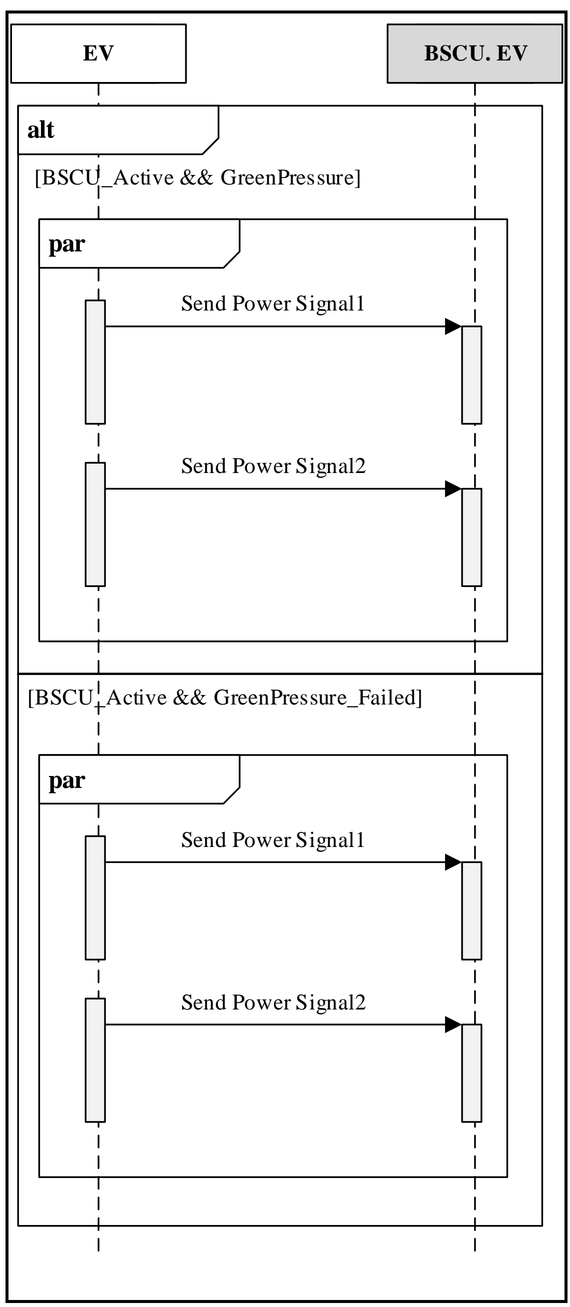
E6: WBS Transaction Domain

Interactions of the components in the transaction domain *WheelDecelerationOnGround*

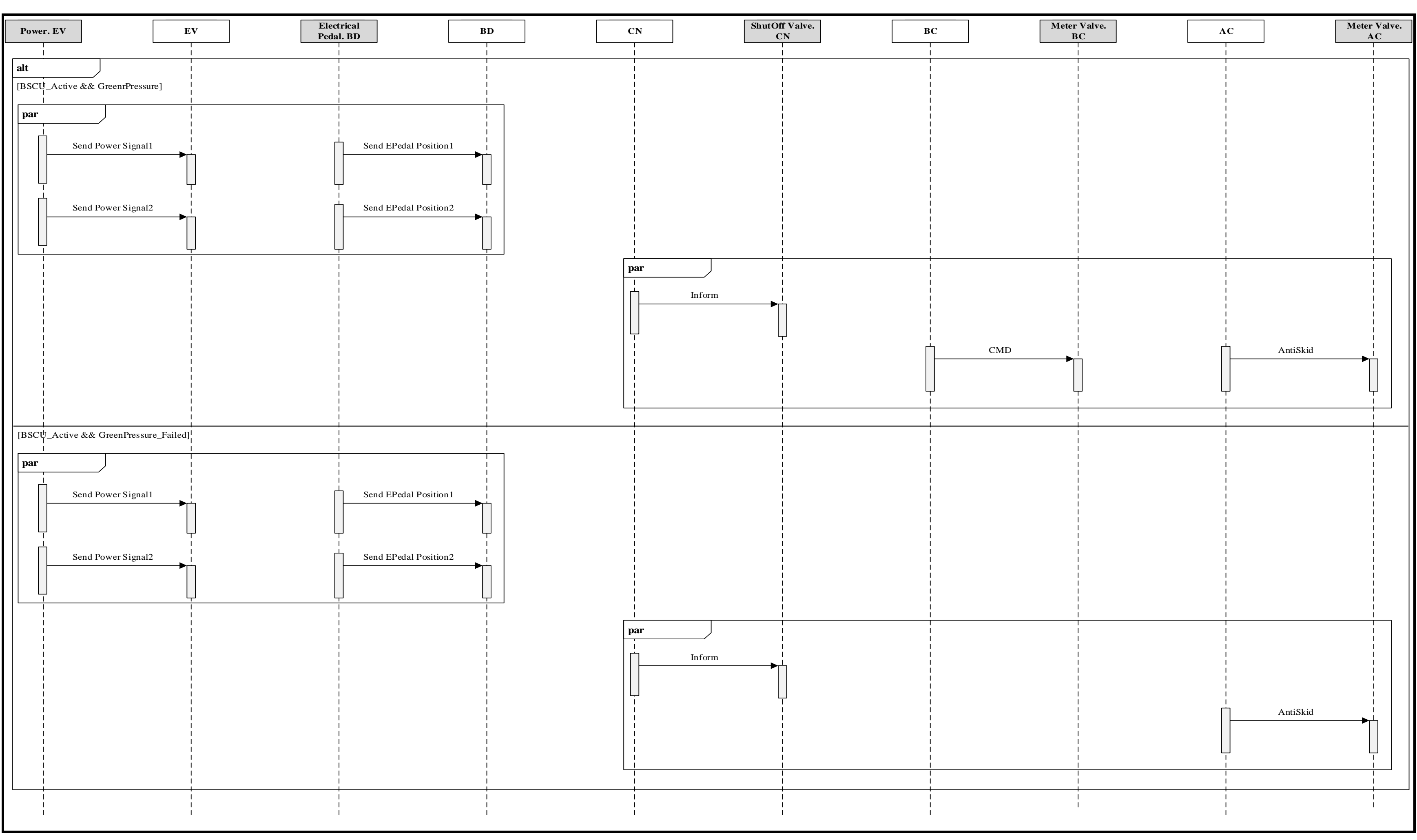
Electrical_Pedal

Provided in Section 8.3.10

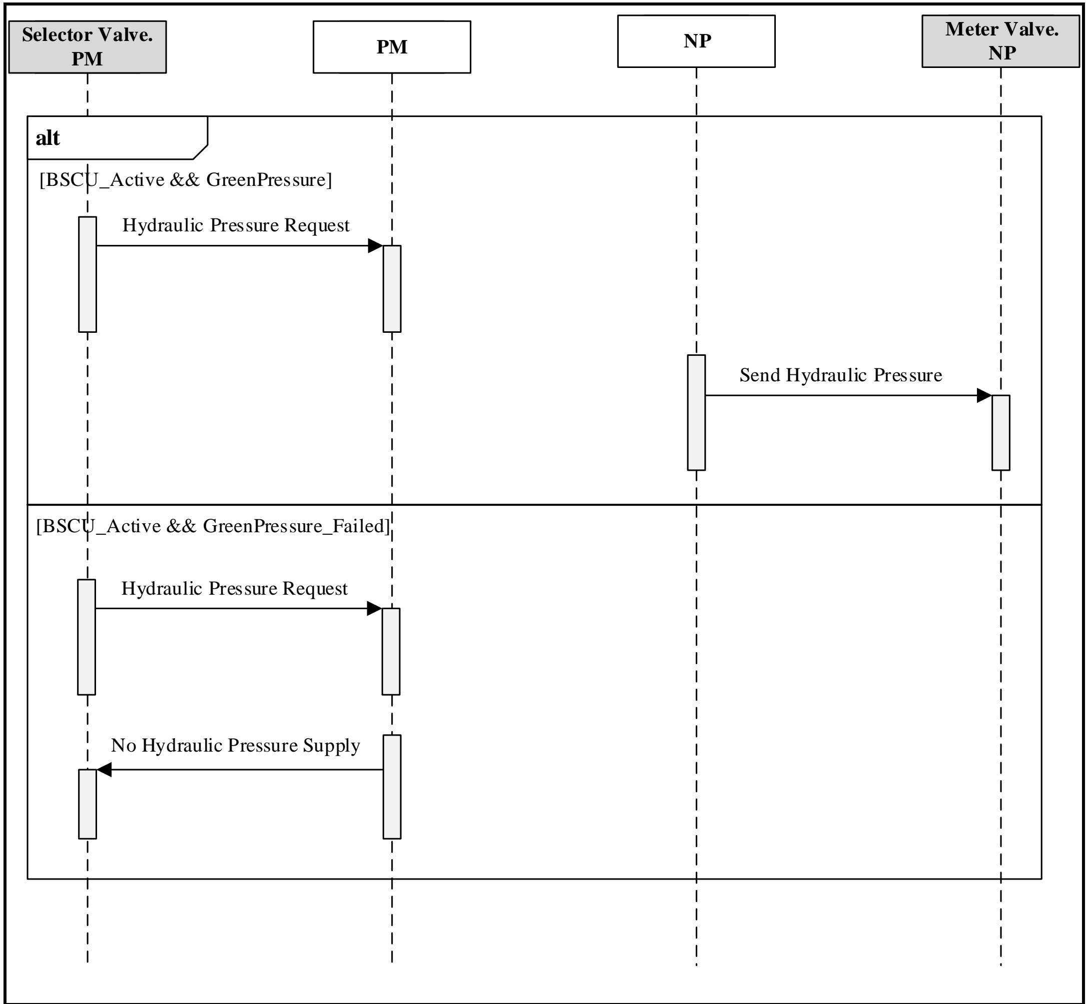
Power



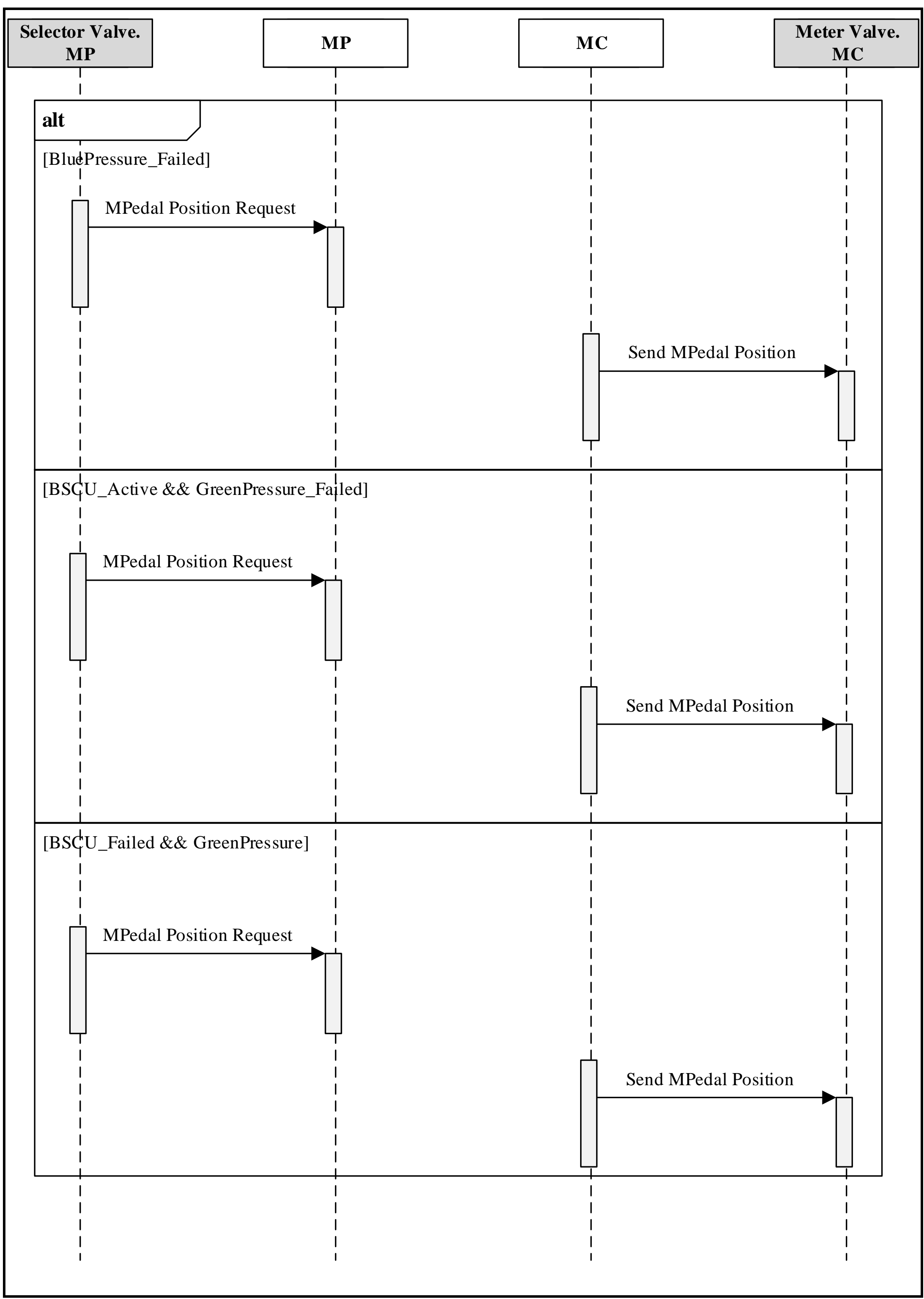
BSCU



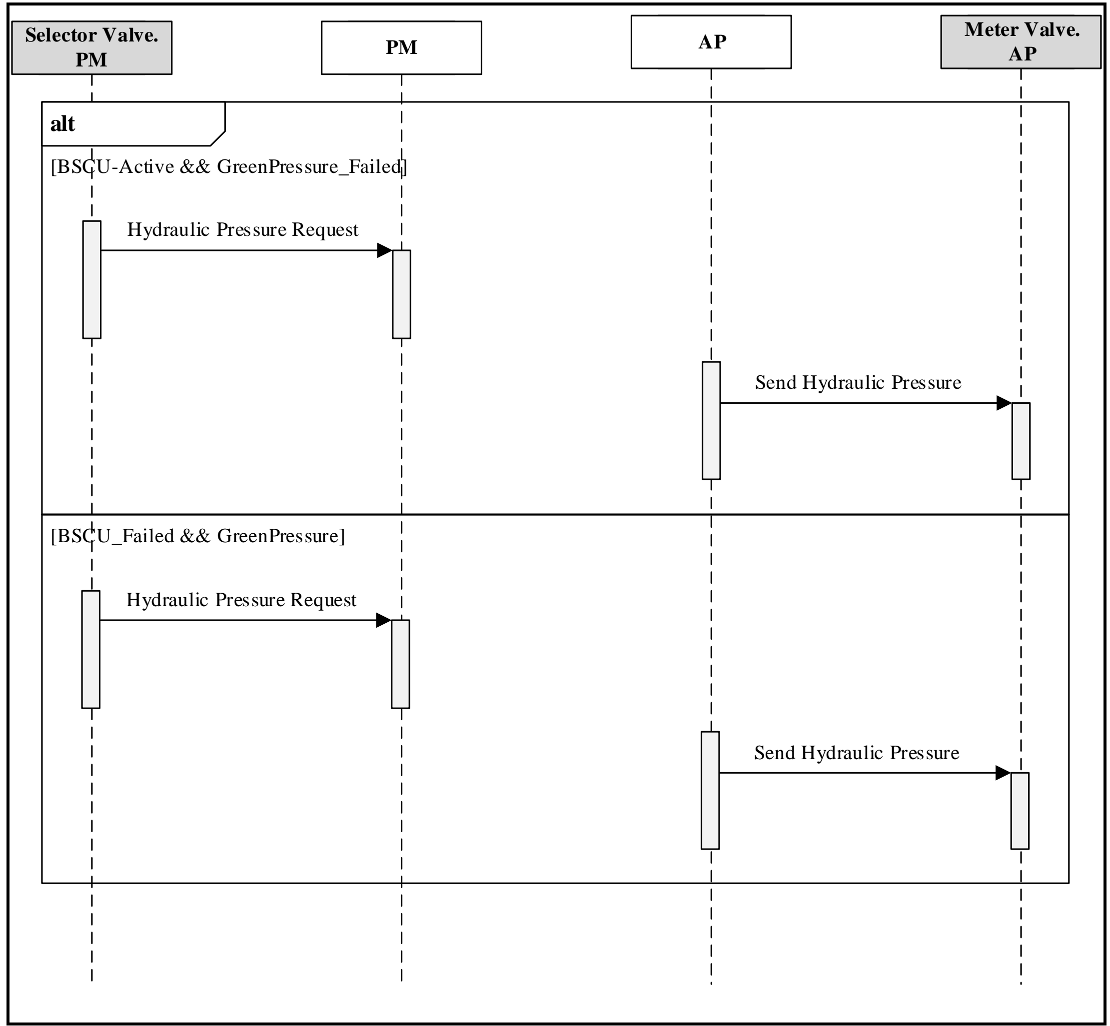
Green_Pump



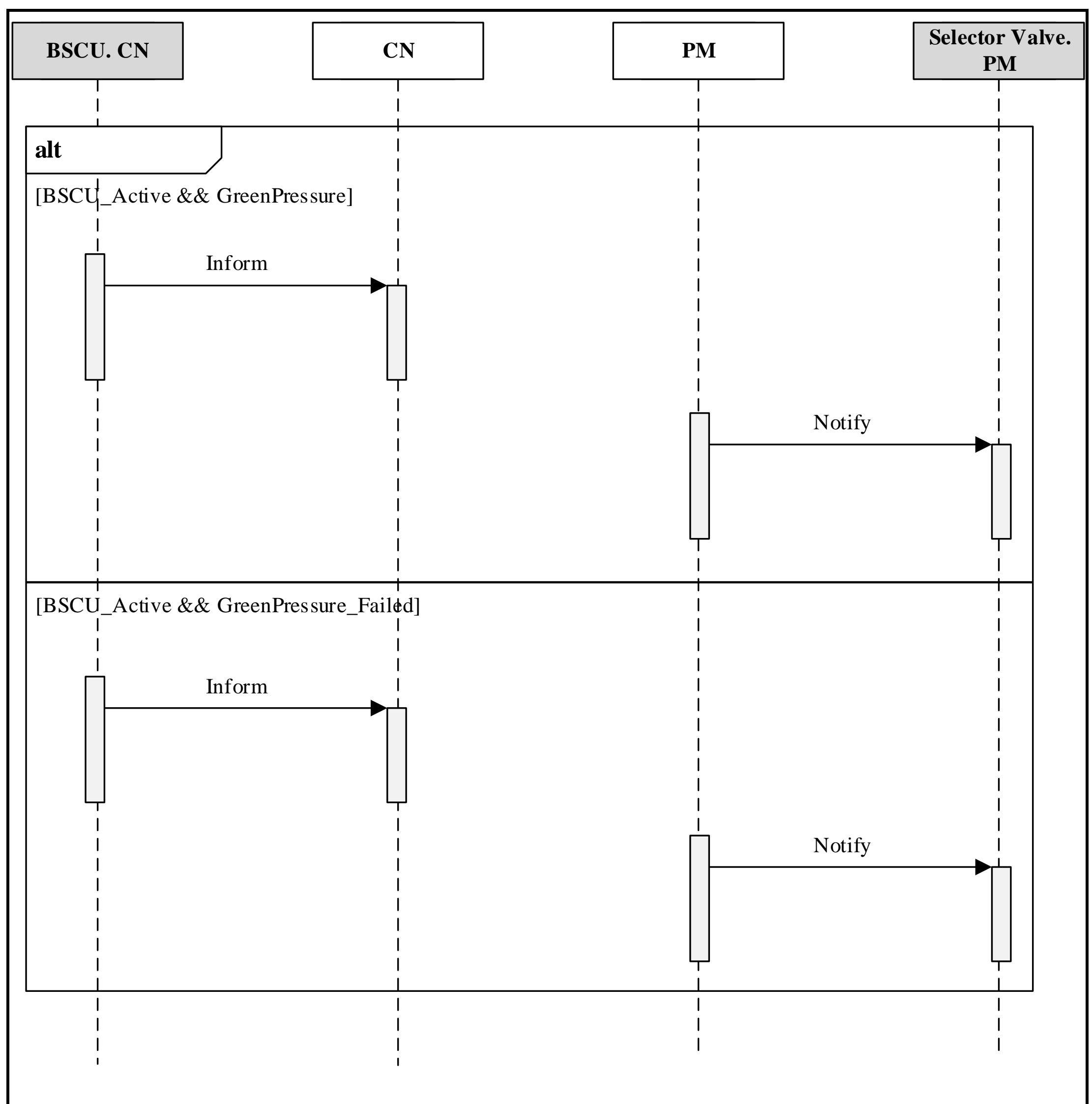
Mechanical_Pedal



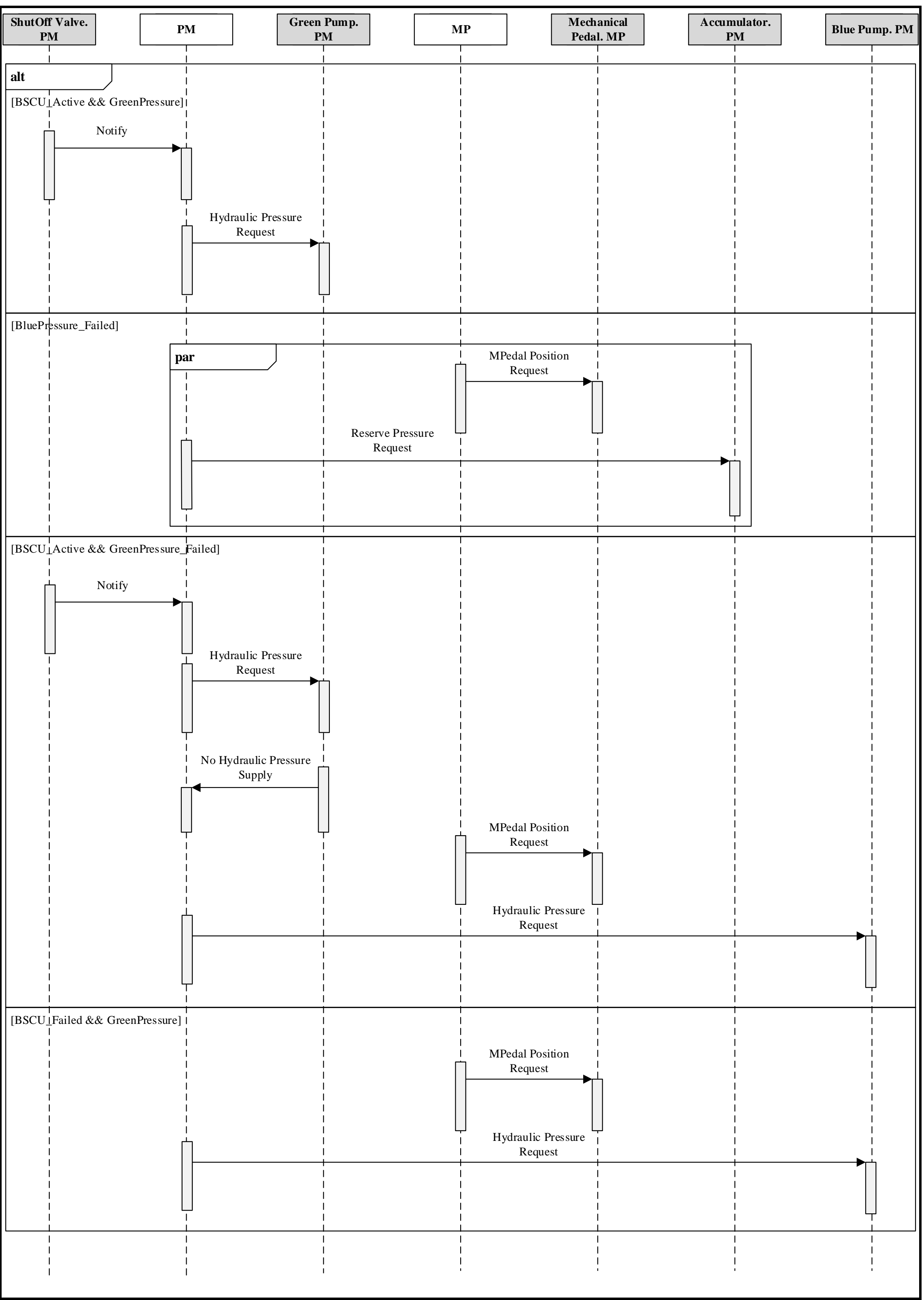
Blue_Pump



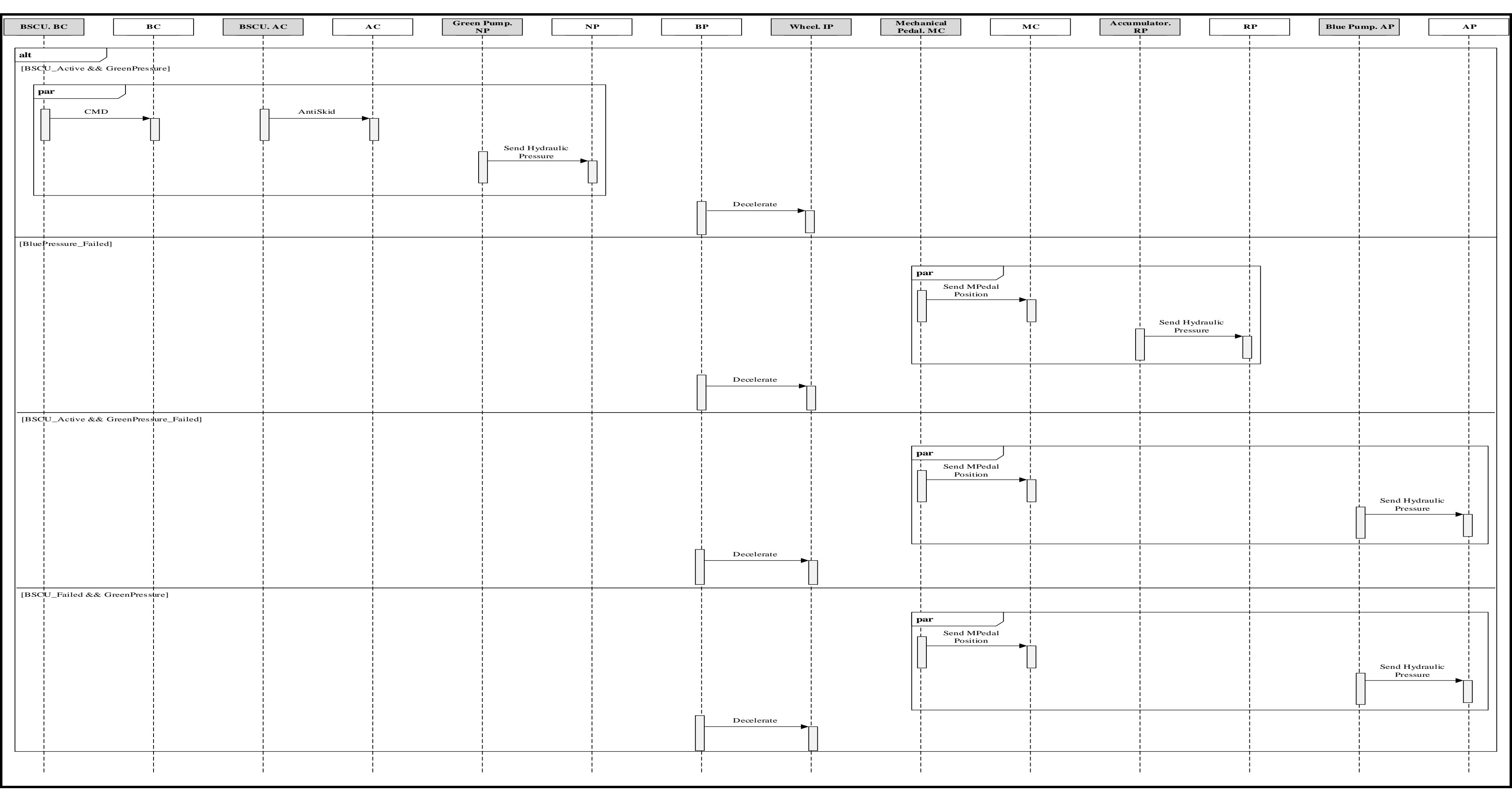
ShutOff_Valve



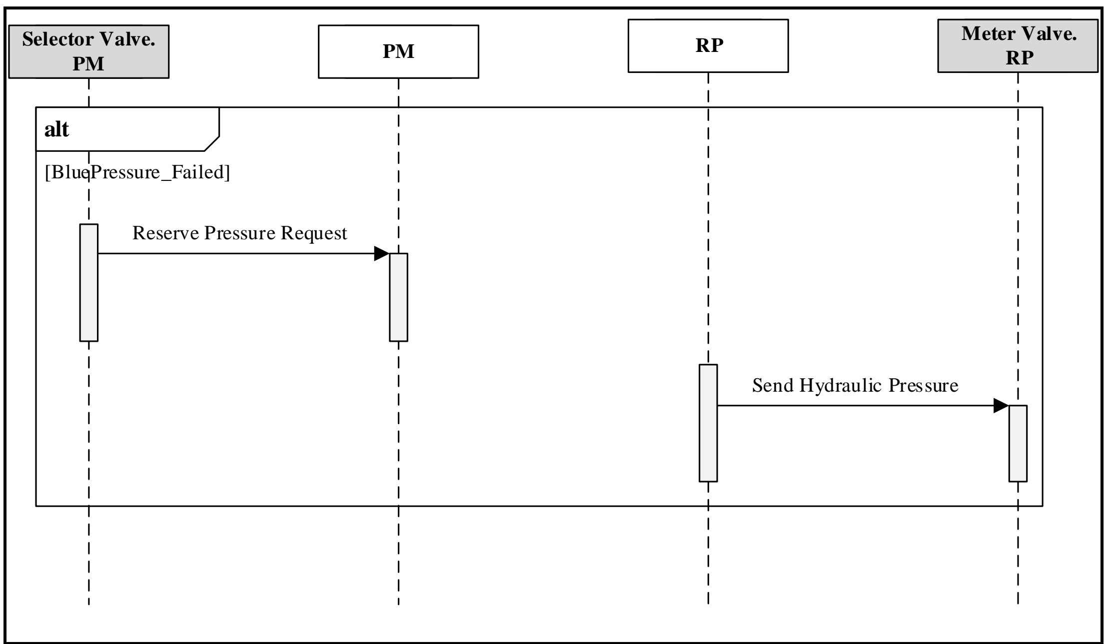
Selector_Valve



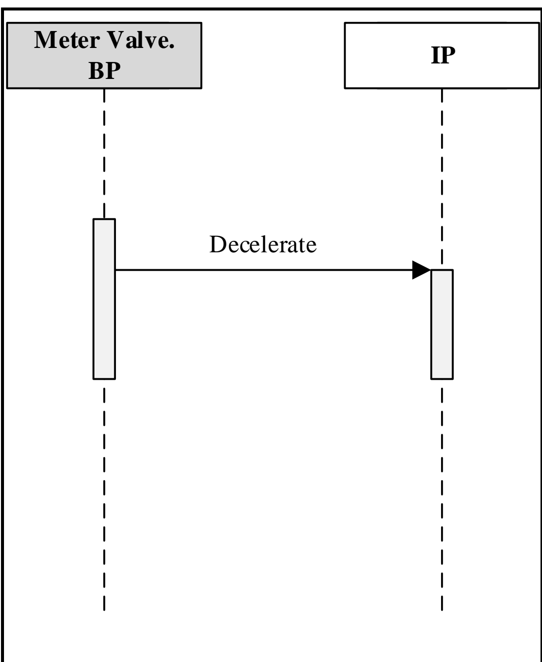
Meter_Valve



Accumulator



Wheel



E7: Wheel Brake System (WBS)

```
system
 {
  components {
    Selector_Valve<Electrical_Power>: Aircraft_BrakeValve;
    Wheel<>: Aircraft_Wheel;
    Meter_Valve<Electrical_Brake, Mechanical_Brake, Piston_Pressure,
                Accumulator_Pressure, Electrical_Power>:
                                                Aircraft_PressureValve;
    if (supported(Electrical_Power)) {
      {Power<>: Aircraft_ElectricPower;
       BSCU<Electrical_Brake, Electrical_Power>:
                                             Aircraft_WheelControlUnit;
       Shutoff_Valve<true>: Aircraft_BrakeValve;}
      if (supported(Electrical_Brake))
        Electrical_Pedal<true, false>: Aircraft_BrakePedal;
     }
    if (supported(Mechanical_Brake))
      Mechanical_Pedal<false, true>: Aircraft_BrakePedal;
    if (supported(Electrical_Brake && Piston_Pressure)){
      Green_Pump<true, false, true, false>: Aircraft_PressurePump;
    else if (supported(Mechanical_Brake && Piston_Pressure))
      Blue_Pump<false, true, true, false>: Aircraft_PressurePump;
    else
      Accumulator<false, true, false, true>: Aircraft_PressurePump;}
   } // end of components

  connectors { }

  arrangement {
    bind Meter_Valve.BrakePressure with Wheel.InputPressure;
    if (supported(Electrical_Power)) {
      {bind Power.ElectircVoltage with BSCU.ElectircVoltage;
       bind BSCU.CommandNotification with
        Shutoff_Valve.CommandNotification;
        bind BSCU.AntiskidCommand with Meter_Valve.AntiskidCommand;
        }
        if (supported(Electrical_Brake)){
          {bind Electrical_Pedal.BrakeData with BSCU.BrakeData;
           bind BSCU.BrakeCommand with Meter_Valve.BrakeCommand;}
       }
```

```
  if (supported(Mechanical_Brake)){
    bind Mechanical_Pedal.MechanicalPosition with
    Meter_Valve.MechanicalPosition;
    bind Mechanical_Pedal.MechanicalCommand with
    Meter_Valve.MechanicalCommand;
   }
  if (supported(Electrical_Brake && Piston_Pressure)){
    {bind Shutoff_Valve.PressureMessage with
     Selector_Valve.PressureMessage;
     bind Selector_Valve.PressureMessage with
     Green_Pump.PressureMessage;
     bind Green_Pump.NormalPressure with
     Meter_Valve.NormalPressure;}
  else if (supported(Mechanical_Brake && Piston_Pressure))
     {bind Selector_Valve.PressureMessage with
      Blue_Pump.PressureMessage;
      bind Blue_Pump.AlternatePressure with
      Meter_Valve.AlternatePressure;}
  else
     {bind Selector_Valve.PressureMessage with
      Accumulator.PressureMessage;
      bind Accumulator.ReservePressure with
      Meter_Valve.ReservePressure;}
 } // end of arrangement

viewpoints {
   WheelDeceleration;
} // end of viewpoints
} // end of WBS
```