Generative Modelling

with

Software Product Line Architectures

Christopher Kenneth Holmes

This thesis is submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

The University of York
York YO10 5DD
UK

Department of Computer Science

December 2007
Abstract

One of the approaches to manage the ever increasing need to improve software productivity, identified as long ago as 1968, is that of the development of software systems following a family-based view. The explicit graphical representation of families of related software systems evolved from the publication of Feature Oriented Domain Analysis, and has been refined subsequently; however there is still disagreement in the literature on what comprises a feature and how collections of features may be related. More recently, the OMG’s Model-Driven Architecture initiative and supporting technologies, such as meta-modelling and QVT have raised models to the forefront of software development artefacts. This thesis addresses the gap between the informal semantics of features and feature models by adopting a meta-modelling approach, based on OMG standards, to the definition of Software Product Line Architecture and the constituent components, thereby making explicit the missing semantics and model relationships. The hypothesis of this thesis is that the application of a meta-modelling approach brings both rigour and flexibility to the modular definition of the Software Product Line Architecture, sufficient to support tool validation. This hypothesis is evaluated via two case studies drawn from disparate domains, and the results discussed.
List of Contents

1 INTRODUCTION ........................................................................................................................................................................15
  1.1 BACKGROUND ...........................................................................................................................................................................15
  1.2 MOTIVATION ...................................................................................................................................................................................16
  1.3 CONTRIBUTION ...............................................................................................................................................................................18
    1.3.1 Novel Aspects ...........................................................................................................................................................................24
    1.3.2 Hypothesis ..................................................................................................................................................................................25
      1.3.2.1 A Metamodelled Foundation for the SPLA ..........................................................................................................................26
      1.3.2.2 A Semantics for Features ......................................................................................................................................................26
      1.3.2.3 Is the Metamodelled SPLA Domain Agnostic? ....................................................................................................................27
    1.3.3 Structure of Thesis .................................................................................................................................................................27

2 LITERATURE REVIEW .......................................................................................................................................................................29
  2.1 AN OVERVIEW OF THE UML ...................................................................................................................................................29
  2.2 META-MODELLING ........................................................................................................................................................................34
    2.2.1 What is Meta-Modelling? .........................................................................................................................................................34
    2.2.2 Why is Meta-Modelling Useful? ................................................................................................................................................36
    2.2.3 Abstract Syntax .........................................................................................................................................................................37
    2.2.4 Concrete Syntax .........................................................................................................................................................................38
    2.2.5 Relating Concrete Syntax and Abstract Syntax ......................................................................................................................39
    2.2.6 Semantics ....................................................................................................................................................................................41
  2.3 MODEL-DRIVEN LANGUAGE ENGINEERING .............................................................................................................................44
    2.3.1 Model-Driven Architecture ....................................................................................................................................................44
    2.3.2 Language-Driven Development .............................................................................................................................................45
    2.3.3 Model-Based Software Development .......................................................................................................................................50
    2.3.4 Language-Oriented Programming .........................................................................................................................................50
    2.3.5 Summary of Model-Driven Approaches ....................................................................................................................................51
  2.4 AN OVERVIEW OF THE SYML ....................................................................................................................................................53
  2.5 THE CASE FOR STRATEGIC REUSE ...........................................................................................................................................55
  2.6 SOFTWARE PRODUCT LINE ARCHITECTURES ............................................................................................................................57
    2.6.1 Model-Driven Product Line Architecture ......................................................................................................................................57
    2.6.2 The Feature Model .......................................................................................................................................................................57
      2.6.2.1 What is a Feature? ....................................................................................................................................................................58
      2.6.2.2 Why Are Features Useful? .....................................................................................................................................................60
      2.6.2.3 How are Features Related? ....................................................................................................................................................60
      2.6.2.4 A Simple Example of a Feature Model ....................................................................................................................................62
  2.7 CURRENT APPROACHES TO MODELLING THE SPLA ..................................................................................................................64
  2.8 SOFTWARE FACTORIES .................................................................................................................................................................67
  2.9 ARCHITECTURE DESCRIPTION LANGUAGES ............................................................................................................................69
    2.9.1 The ADL II ....................................................................................................................................................................................70
3 SPLA DEVELOPMENT PROCESS ................................................... 87

3.1 APPLYING THE SPLA APPROACH ................................................... 88
  3.1.1 Domain Engineering ................................................................. 88
    3.1.1.1 Use of COTS Products .......................................................... 90
  3.1.2 Product Development ................................................................. 91
    3.1.2.1 Feature Configuration ............................................................ 91
    3.1.2.2 System Specification ............................................................. 92
    3.1.2.3 Product Generation ............................................................... 93
  3.2 SUMMARY .................................................................................... 94
    3.2.1 Illustration via Case Studies ..................................................... 94

4 INTRODUCTION TO MODELLING OF THE SPLA ................................. 97

4.1 MODELLING THE SPLA ................................................................. 97
4.2 RATIONALE FOR THE DEVELOPMENT OF THE SPLA METAMODELS ................................................. 98
4.3 WHAT IS THE PROBLEM TO BE Addressed? ........................................ 101
4.4 INTRODUCTION TO THE SPLA COMPONENTS ...................................... 102
4.5 AN ABSTRACT DEFINITION OF THE SPLA ........................................ 103
  4.5.1 Feature Models ............................................................................ 104
  4.5.2 Configuration Models ................................................................. 108
  4.5.3 Reference Architectural Components ............................................. 112
    4.5.3.1 An Abstract Definition of an Example Layered Architecture .............. 113
  4.5.4 Specification Models ..................................................................... 115
  4.5.5 Product Generators ..................................................................... 116
  4.5.6 Conclusions .................................................................................. 117
    4.5.6.1 Connecting Features to Reality .................................................. 118

5 SUPPORTING THE DEVELOPMENT OF THE SPLA ....................... 119

5.1 BUILDING A METAMODEL FOR FEATURE MODELS ........................................ 119
  5.1.1 Example Feature Model .................................................................. 130
  5.1.2 Demonstration of Error Checking of Feature Models ......................... 134
    5.1.2.1 Violation of Anti-Symmetry Constraint ........................................ 134
    5.1.2.2 Mal-Formed Alternative Sub-Features ........................................ 136
    5.1.2.3 Mal-Formed Feature .................................................................. 137
    5.1.2.4 Violation of Single Parent Constraint .......................................... 139
5.1.2.5 Extension of Leaf Feature

5.2 Building a Metamodel for Configuration Models

5.2.1 Constraints on Feature/Sub-Feature Relationships

5.2.2 Constraints on Side-Effects

5.2.3 Example Configuration Model

5.2.4 Demonstration of Error Checking of Configuration Models

5.2.4.1 Violation of Dependent Features Constraint

5.2.4.2 Violation of Mandatory Alternative Feature Selection

5.2.4.3 Violation of Mandatory Variation Point Feature Selection

5.2.4.4 Violation of Mandatory Inclusive-Or Features

5.2.4.5 Configuring Features from Different Models

5.3 Building a Metamodel for Specification Models

5.3.1 Example Specification Model

5.3.2 Reference Architectural Components Model

5.4 Generating the Product

5.5 Conclusions

5.5.1 Domain Engineering

5.5.2 Feature Model

5.5.3 System Configuration

5.5.4 Specification Model

5.5.5 Product Generator

5.5.6 The Cardinality of Features

5.5.7 Normalising Feature Models

5.5.8 Modularity of the SPLA

5.5.9 Development and Validation of Metamodels

6 Case Studies and Evaluation

6.1 A Concrete Syntax for the SPLA

6.2 Tactical Data Links

6.2.1 Introduction to the TDL Domain

6.2.2 Domain Engineering of TDL Platforms

6.2.3 Feature Modelling of TDL Platforms

6.2.4 Configuration Modelling of TDL Platforms

6.2.5 Specification Modelling of TDL Platforms

6.2.6 Product Generation of TDL Platform

6.2.7 Domain Engineering of TDL Function and Message Structure

6.2.8 Feature Modelling of TDL Function and Message Structures

6.2.9 Configuration Modelling of TDL Function and Message Structure

6.2.10 Specification Modelling of TDL Functions and Message Structure

6.2.11 Product Generation of TDL Function and Message Structure

6.2.12 Related Work

6.2.13 Related Work
List of Figures

Figure 1 – Example Heavyweight Profile..............................................................................................20
Figure 2 – Model to Model Transformation ..........................................................................................21
Figure 3 – UML 4-Layer Metamodel Architecture ..............................................................................23
Figure 4 – Models Comprising the SPLA ............................................................................................26
Figure 5 – Application of Lightweight Stereotypes ............................................................................30
Figure 6 – Application of Lightweight Stereotypes with Graphical Support ........................................30
Figure 7 – Components of the MML Metamodel Architecture............................................................32
Figure 8 – The UML Model Stack........................................................................................................35
Figure 9 – Abstract Syntax Model (GenVoca) ......................................................................................38
Figure 10 – For-Loop (Concrete Syntax) ...............................................................................................38
Figure 11 – For-Loop (Abstract Syntax) ...............................................................................................39
Figure 12 – Example Program (Concrete Syntax) ................................................................................39
Figure 13 – For-Loop Example Grammar ............................................................................................40
Figure 14 – Generated Abstract Syntax ...............................................................................................40
Figure 15 – Example Components of a Metamodel ............................................................................43
Figure 16 – Relationship of the PIM to the PSM ................................................................................45
Figure 17 – Metamodel Relationships in the LDD Approach ...............................................................46
Figure 18 – Model Transformation (overview) ....................................................................................47
Figure 19 – Model Transformation using XMF-Mosaic .......................................................................48
Figure 20 – Structure of XMap Transformation ...................................................................................48
Figure 21 – XMap Specification of Model Transformation ....................................................................49
Figure 22 – Relationship Between Model-Driven Approaches ............................................................52
Figure 23 – Relationship of SysML to UML .......................................................................................53
Figure 24 – SysML Diagram Types ......................................................................................................54
Figure 25 – SysML Requirement Element .........................................................................................55
Figure 26 – The Influence of COTS ......................................................................................................56
Figure 27 – Lightweight Feature Metamodel ......................................................................................61
Figure 28 – Steering Strategies ...........................................................................................................63
Figure 29 – Flight Management Feature Diagram ..............................................................................63
Figure 30 – Lightweight Features Metamodel .....................................................................................64
Figure 31 – Lightweight Feature Group Metamodel ............................................................................65
Figure 32 – Example Lightweight Feature Model ................................................................................65
Figure 33 – Program Construction Environment ...............................................................................66
Figure 34 – Feature/Product Line Asset Map .....................................................................................69
Figure 35 – Component Sections in II ................................................................................................70
Figure 36 – Aesop Generic Architectural Entities ...............................................................................72
Figure 37 – Pipe-and-Filter Style ........................................................................................................73
Figure 38 – Example Architecture Fragment Composition ....................................................................75
Figure 39 – Metamodel of the GenVoca Model ...................................................................................78
Figure 40 – Snapshot of GenVoca INAV Model ..................................................................................78
Figure 41 – Metamodel of the GenVoca System Specification .............................................................79
Figure 42 – Snapshot of GenVoca INAV Specification .......................................................................80
Figure 43 – Type Compatible Components Constraint Error ...............................................................82
Figure 44 – GenVoca Component Adaptation .......................................................................................83
Figure 45 – The Iron Triangle of Quality Improvement .........................................................................87
Figure 46 – Domain Engineering .........................................................................................................89
Figure 47 – Product Engineering – Feature Configuration ....................................................................92
Figure 48 – Product Engineering – System Specification ......................................................................93
Figure 49 – Product Engineering – Product Generation .......................................................................93
Figure 50 – Case Study Process View ................................................................................................95
Figure 51 – Outline of SPLA Development Process ..........................................................................96
Figure 52 – Example of Lightweight Feature Stereotypes ..................................................................100
Figure 53 – Flight Management Configuration Model .........................................................................109
Figure 54 – Core Concepts of the Feature Model ................................................................. 120
Figure 55 – Feature Dependency .................................................................................... 122
Figure 56 – Feature Dependencies and Composition Constraints .................................. 122
Figure 57 – Binding Time Ordering ............................................................................. 123
Figure 58 – Feature/Sub-Feature Relationships ............................................................. 125
Figure 59 – Example of a Variation Point (Graphical Concrete Syntax) ......................... 126
Figure 60 – Example of a Variation Point (Abstract Syntax) ......................................... 126
Figure 61 – Example Normalization of Feature Models (Exclusive-Or) ......................... 127
Figure 62 – Optional Alternative Sub-Features ............................................................... 127
Figure 63 – Mandatory Alternative Sub-Features ......................................................... 128
Figure 64 – Example Inclusive-Or Feature Relationship .............................................. 129
Figure 65 – Example Normalization of Feature Models (Inclusive-Or) ......................... 129
Figure 66 – A Metamodel for Features ....................................................................... 130
Figure 67 – Example Feature Model (Textual Concrete Syntax) .................................... 131
Figure 68 – Example Feature Model (Graphical Concrete Syntax) ............................... 132
Figure 69 – Car Domain Model .................................................................................. 133
Figure 70 – Car Body Components ............................................................................. 134
Figure 71 – Feature Model with Anti-Symmetry Violation ............................................ 135
Figure 72 – Results of Constraint Check (Anti-Symmetry Violation) ............................ 135
Figure 73 – Feature Model with Mal-Formed Alternative Sub-Features ......................... 136
Figure 74 – Results of Constraint Check (HomogeneousSubFeatures) ........................ 137
Figure 75 – Erroneous Feature Declaration ............................................................... 138
Figure 76 – Constraint Violation (SelectionIsDefined) .............................................. 138
Figure 77 – Constraint Violation (BindingTimeIsDefined) ......................................... 139
Figure 78 – Relating a Feature to a Parent ................................................................. 139
Figure 79 – Erroneous Feature Constructor Function ............................................... 140
Figure 80 – Feature with Multiple Parents ................................................................. 140
Figure 81 – Results of Constraint Check (HasAtMostOneParent) ............................... 141
Figure 82 – Results of Constraint Check (ExtensionOfLeaf) ....................................... 142
Figure 83 – Semantic Domain for Features ............................................................... 142
Figure 84 – Configuration Model .............................................................................. 143
Figure 85 – Configuration Model with Orphaned Choice ......................................... 144
Figure 86 – Example Configuration Model (Textual Concrete Syntax) ......................... 148
Figure 87 – Results of Constraint Check (Configuration Model OK) ......................... 148
Figure 88 – Example Feature Configuration ............................................................. 149
Figure 89 – Configuration Model with Dependent Feature Violation......................... 150
Figure 90 – Results of Constraint Check (Dependent Features Violation) .................... 151
Figure 91 – Configuration Model with Mandatory Alternative Feature Violation (1) ... 152
Figure 92 – Configuration Model with Mandatory Alternative Feature Violation (2) ... 152
Figure 93 – Configuration Model with Mandatory Variation Point Feature Violation .... 153
Figure 94 – Configuration Model with Mandatory Inclusive-Or Feature Violation ....... 154
Figure 95 – Example Secondary Feature Model ....................................................... 155
Figure 96 – Configuring a Product from Multiple Feature Models ......................... 155
Figure 97 – Features Contained In Feature Model Constraint Violation ..................... 156
Figure 98 – Specification Model .............................................................................. 157
Figure 99 – Example Specification Model (Textual Concrete Syntax) ......................... 157
Figure 100 – Example Car Layered Architecture (Specification) .............................. 158
Figure 101 – Example Car Layered Architecture (Model) ......................................... 158
Figure 102 – Profile for a Layered Architecture ....................................................... 159
Figure 103 – Three-Tier Architecture Description .................................................... 159
Figure 104 – Three-Tier Architecture ..................................................................... 160
Figure 105 – Querying the Three-Tier Architecture ............................................... 160
Figure 106 – Overview of Product Generation Transformation ................................ 161
Figure 107 – Screenshot of Generated Model (Outline) ............................................. 164
Figure 108 – Product Generation ............................................................................ 165
Figure 163 – Generated Component Mission Management ...........................................................228
Figure 162 – Source Component Package Steering Mode ls...........................................................227
Figure 161 –Avionics Components Model Element to Ar chitecture Mapping ..............................226
Figure 160 – Avionics Components Source Model Eleme nts ........................................................224
Figure 159 – Partitioning of Avionics Navigation Sy stem Architecture.................................. ......223
Figure 157 – Erroneous Configuration Model for Avio nics Navigation System ...........................222
Figure 156 – Avionics Navigation System Configurati on..............................................................221
Figure 155 – Avionics Navigation System Configurati on Error (Constraint Report) ....................220
Figure 154 – Avionics Navigation System Configurati on............................................................219
Figure 153 – Example Avionics Navigation System Con figuration...............................................218
Figure 152 – Display Formats Feature Model ................................................................................216
Figure 149 – Mapping Search Patterns to Model Eleme nts............................................................214
Figure 148 – Example TDL Function and Message Struc ture Architecture...................................193
Figure 147 – Steering Model Feature Model .......... ........................................................................213
Figure 146 – Feature Description of Mission Managem ent (Steering Model) ...............................213
Figure 145 – Transient Objects Model Feature Model. ..................................................................212
Figure 144 – Earth Models Feature Model............ .........................................................................210
Figure 143 – Air Vehicle Feature to Model Element M apping ......................................................208
Figure 142 – Feature Description of Ellipsoidal Models................................................................209
Figure 141 – Atmosphere Model Feature Description .. .................................................................206
Figure 140 – Sensors Feature Model
Figure 139 – Air Vehicle Model Feature Description
Figure 138 – Sensors Feature Model (Concrete Syntax) ................................................................205
Figure 137 – Top-Level Feature Model of the Navigation Function..............................................204
Figure 136 – Mission System Domain Model ................................................................................201
Figure 135 – TDL Unit Configuration Error (Constraint Report) ...................................................192
Figure 134 – TDL Unit Configuration Error (Error) .......................................................................190
Figure 133 – Configuration Model for TDL Functions .................................................................189
Figure 132 – Generated TDL Functions Components ...................................................................186
Figure 131 – Generated Model of a TDL Function Configuration..................................................184
Figure 130 – Generated Model of a TDL Unit Configuration ..........................................................183
Figure 129 – Link 16 Source Model Elements ..............................................................................182
Figure 128 – Example TDL Function and Message Structure Architecture...................................181
Figure 127 – TDL Function Configuration Error (Constraint Report) ................................................180
Figure 126 – Errorneous Configuration Model for TDL Function ..................................................179
Figure 125 – TDL Function Configuration (Error) .........................................................................178
Figure 124 – Configuration Model for Selected TDL Functions ...................................................177
Figure 123 – Example TDL Function Configuration .......................................................................176
Figure 122 – Top-Level TDL Function Feature Model.................................................................175
Figure 121 – A Feature Model for TDL Functions..... ....................................................................174
Figure 120 – Generated Model of a TDL Unit Configuration .........................................................173
Figure 119 – TDL Unit Model Element to Architecture Mapping ..................................................172
Figure 118 – Example TDL Unit Target Architecture.....................................................................171
Figure 117 – Errorneous Configuration Model for TDL Unit ..........................................................170
Figure 116 – TDL Unit Configuration Error (Constraint Report) ...................................................169
Figure 115 – Example of Erroneous TDL Unit Configuration Model ............................................168
Figure 114 – A Configuration Model for a TDL Unit ....................................................................167
Figure 113 – Example C JU Configuration .....................................................................................166
Figure 112 – A Feature Model for TDL Units (graphical) ...............................................................165
Figure 111 – A Feature Model for TDL Units (textual) ..................................................................164
Figure 110 – TDL Linkage ..............................................................................................................163
Figure 109 – Model Validation Strategy ..........................................................................................162
Figure 108 – Generated TDL Message Catalogue Compon ents.....................................................161
Figure 107 – Generated TDL Functions Components.... ................................................................160
Figure 106 – Generated Model of a TDL Function Configuration..................................................159
Figure 105 – Example TDL Function and Message Structure Architecture...................................158
Figure 104 – Top-Level TDL Function Feature Model...................................................................157
Figure 103 – A Feature Model for TDL Units ..............................................................................156
Figure 102 – TDL Function Configuration Error (Error) ................................................................155
Figure 101 – Erroneous Configuration Model for TDL Function ..................................................154
Figure 100 – Example TDL Function and Message Structure Architecture...................................153
Figure 99 – TDL Linkage ................................................................................................................152

10
List of Tables

Table 1 – Development Artefact Grid ........................................................................................................68
Table 2 – Example Path Expressions in $\Pi$ ..........................................................................................71
Table 3 – Avionics Mission System Domains .........................................................................................200
Table 4 – Domain to Feature Mapping ..................................................................................................203
Table 5 – Summary of Opportunities for Future Work ........................................................................250
Table 6 – C2 Platform Minimum Implementation Requirements ..........................................................286

List of Equations

Equation 1 – Example Constraint (GenVoca) ..........................................................................................37
Equation 2 – GenVoca Grammar ............................................................................................................77
Equation 3 – GenVoca Model of INAV Domain ....................................................................................79
Equation 4 – GenVoca Specification of an INAV System .......................................................................80
Equation 5 – Type Checking of GenVoca Type Equations ....................................................................81
Equation 6 – GenVoca Type Incompatible INAV System .....................................................................81
Equation 7 – Example GenVoca Specification .......................................................................................82
Equation 8 – Example GenVoca with Adapted Interface .......................................................................83
Equation 9 – Leaf Features may not be Extended .................................................................................121
Equation 10 – Feature Model has One Root .........................................................................................121
Equation 11 – Feature has at most One Parent ......................................................................................121
Equation 12 – Features are Anti-symmetric .........................................................................................121
Equation 13 – Anti-symmetric Helper Operation .................................................................................121
Equation 14 – Binding Time is Equal to or Later Than Super-Feature .................................................123
Equation 15 – Ordering Relation on Feature BindingTimeType ............................................................123
Equation 16 – Feature Selection is Defined ..........................................................................................123
Equation 17 – Feature Binding Time is Defined ..................................................................................124
Equation 18 – Dangling Node Constraint ............................................................................................125
Equation 19 – Well-Formedness of Alternative Features ......................................................................128
Equation 20 – Enforcing Feature Selection in the Configuration Model .............................................143
Equation 21 – Selection of All Super-Features .....................................................................................144
Equation 22 – VariationPoint Sub-Feature Selection ...........................................................................145
Equation 23 – Optional Alternative Sub-Feature Selection ...................................................................146
Equation 24 – Mandatory Alternative Sub-Features Selection .............................................................146
Equation 25 – Mandatory Inclusive-Or Sub-Feature Selection .............................................................147
Equation 26 – Optional Inclusive-Or Sub-Feature Selection ..................................................................147
Equation 27 – Identification of Dependent Features ...........................................................................148
Equation 28 – Identification of Target Architectural Style ..................................................................162
Equation 29 – Transformation into Layered Architectural Style .........................................................162
Equation 30 – Helper Operation bindTo() ..........................................................................................163
Equation 31 – Helper Operation getTargetLayer() ............................................................................163
Equation 32 – Helper Operation insertClassHierarchy() ...................................................................164
Equation 33 – Feature/Sub-Feature Binding Time Constraint .............................................................247

List of Accompanying Material

None.
Acknowledgement

The author thanks BAE SYSTEMS for funding aspects of this work, and making the work possible. Thanks also go to my supervisors: Dr. Andy Evans and Prof. Richard Paige. Dr. Evans helped set the course for the initial work, making available the emerging metamodelling technology in the form of the MMT, XMT, and finally the XMF tool. Prof. Paige took up the role of supervisor following the departure of Dr. Evans who went on to start up the company Xactium Ltd. The author thanks Prof. Paige for his continued support and encouragement. Thanks must go also to those with whom I’ve worked along the way, in particular Dr. Martin Hargreaves who helped to manage the Software Productivity programme, providing opportunities to work in new domains, and also Dr. Julian Johnson for his support during the final phases of this work. Last, and not least, thanks go to my wife and family for their support and patience, especially for enduring my many late nights and periods away from home.
Declaration

The case studies described in this thesis make use of information in the public domain and relevant work undertaken in collaboration with colleagues from BAE SYSTEMS; all other work is that of the author. This thesis makes use of material published previously by the author (within both the public domain and BAE SYSTEMS) as detailed below:

- Holmes, C., ‘Methods of attacking the requirements backlog within the avionics domain – Results of a literature survey’, University of York, Department of Computer Science, YCS 335 (2001)
1 Introduction

The use of software pervades a diverse set of application domains, from domestic refrigerators and games consoles, to large business information systems. Software is also used in hard real-time domains to control machines used in the manufacturing industry, and to assist with the flight control of some aircraft\(^1\); management of the avionics mission system is also often a software intensive domain. Customer demands for increasing levels of functionality, at a lower cost of ownership, imply that systems can no longer be built from scratch. It has long been known that the software industry is failing to keep pace with demand, and that mass-produced techniques are required [McI68]. The situation is exacerbated when one considers Yourdon’s findings, reporting that there is both a visible\(^2\) and invisible backlog of requirements\(^3\) for software systems [You89]. The size of the invisible backlog being estimated at the time as in excess of five times that of the visible backlog; and the industry is failing to keep pace even with the visible backlog. The result of this shortfall in software productivity is not only dissatisfied customers, but also missed business opportunities both for the customer and the developer. One common approach to tackling the productivity problem is to utilise more staff, however, as Brooks points out, staff and time are only interchangeable when there is no communication between them and this is seldom the case, concluding that ‘adding manpower to a late project makes it later’ [Bro82]. Therefore, a more scalable solution is required; it should be possible to construct new systems by reusing artefacts developed for other systems; this is the industrialisation of the software industry to which McIlroy appealed in 1968.

1.1 Background

The hardware world is addressing the productivity issue via the development and use of standard components, architectures, and protocols in the form of a product line, e.g. PowerPC processors on VME cards hosted in a rack. Clements and Northrop cite Boeing, Ford, Dell, and even McDonalds as practitioners of the product line approach [CN02]. This approach supports the development of new systems by plugging together standard components in specific configurations whilst adhering to certain rules (e.g. the bus master should be at slot 1). Software application developers are beginning to adopt an analogous approach in certain areas. It is now relatively rare for a project to design and produce a bespoke operating system (OS) to support each application, the purchase and configuration of a commercial off-the-shelf (COTS) OS (e.g. VxWorks, LynxOS, etc.) is now relatively commonplace, although not necessarily a trivial exercise. Hence, we are beginning to see the migration of the software engineering function within companies from a vertically integrated

---

\(^1\) Unstable aircraft such as the Typhoon would be unflyable without the aid of real-time software.

\(^2\) Requirements identified to the developer.

\(^3\) Requirements not identified to the developer due to the size of the visible backlog.
structure (where expertise is required at all levels of the software architecture) towards value chain specialisation [Ver00] where a revenue stream may be leveraged by economies of scale. The success (or failure) of the Software Product Line Architecture (SPLA) depends on the variability required of the products to be supported; too wide a variation results in a SPLA of singletons that is likely to be more costly to develop than a traditional approach, Clements and Northrop assert that the SPLA takes advantage of the fact that products to be built are, by design, very similar [CN02].

Interest has also been focussed on realising the aspirations voiced by McIlroy via the development of families of reusable software components. Most modern programming languages provide support for the commonly used transcendental functions that were the focus of McIlroy’s seminal paper, and some also provide support for domain-specific extensions, e.g. the annexes of the Ada95 programming language [Int95]. Whilst the level of abstraction of such components envisaged by McIlroy today seems rather low, the combinatorial nature of the problems relating to variants (i.e. precision, robustness, generality, and time/frame) is still very relevant, as is the requirement to provide some form of cataloguing of the family if we are to support libraries of any significant size. McIlroy also identified the need to avoid unwanted generality and provide components tailored to a customer’s exact needs. The reuse of such components, it was hoped, would lead to reduced development life-cycles and improved quality (since the components would be reused many times), and, ultimately, result in a lower cost of ownership. However, it appears that such families have only been successful in relatively narrow domains, such as math libraries and GUIs, e.g. the Numerical Algorithms Group (NAG) math libraries, Java Swing, etc. However, standard frameworks are of increasing popularity, such as Eclipse, providing a common look and feel to a number of related software tools.

The potential benefits of the SPLA are significant. For example, by following the adoption of the SPLA approach Cummins Inc. report that in excess of 75% of the software in an average application comes from the core asset base; average development timescales have decreased from 250 man-months to ‘a few man-months’; quality has improved; applications have a common look and feel; project timescales are more predictable; and staff mobility across projects is greater. The overall productivity reported by Cummins Inc. as a result of the implementation of a managed adoption of the SPLA is a remarkable 360% improvement [CN02].

1.2 Motivation

Interest in the software engineering community has been focused on the application of reuse via object and component based technologies with the intention of improving productivity and quality. However, the relatively fine grained level of reuse provided by approaches such as class

4 A manufacturer of diesel and natural gas engines.
specialisation and generics has been shown to provide little in the way of benefits \cite{Gut89}, \cite{Bas97} and \cite{CN02}. Furthermore, use of certain programming features, such as generics or class-wide programming, may not be favoured in certain domains where specific language profiles are enforced – e.g. the Ravenscar profile \cite{Bur99} in the domain of hard-real-time systems. Such approaches to reuse are also reported to have no impact on programme costs and tend to cause library scalability issues as features are added \cite{Pen99}, with developers being faced with a sea of look-alike components. It has also been shown that the maintenance of reusable components is expensive and difficult both intellectually\footnote{Attributed to a study performed by Boehm in \cite{PP99}.} and managerially\footnote{E.g. who pays for the development and maintenance of reusable components?}. Hence, there is a need to support the construction of software systems at levels of abstraction above that of the class/module.

Bosch identifies four primary goals for the development of software; low development cost, high quality (reliability, efficiency), short time to market, and low maintenance cost \cite{Bos00}. The results of research and practical experience suggest that only by moving away from the development of single systems towards the development of families of systems will significant advances against these goals be achieved \cite{Cza02}. However, the \emph{one size fits all} approach that may result from diversity in requirements (e.g. MS Office components are aimed at supporting a broad user base) tends to lead to the deployment of baroque components \cite{Ver00}; as an example, the operation signatures exported via the COM interface of the MS Office components can, in some cases, include tens of arguments, many of which may not be required for general usage. This runs counter to the need for the efficient use of computing resources, although this may be more of a concern in particular domains, e.g. embedded systems.

The UML is arguably the \textit{lingua franca} for the modelling of software intensive systems, and Ada95 is the programming language of choice for embedded avionics applications; hence, the use of standard modelling and implementation languages makes software artefacts accessible to a wide user base. The OMG’s recent Model-Driven Architecture (MDA) initiative seeks to raise the level of reuse from code to models by decoupling platform-independent and platform-specific details, thereby achieving its goals of \textit{interoperability} and \textit{portability} \cite{OMG03}. Technologies such as Generative Programming \cite{CE00}, Frame-based reuse \cite{Bas97}, and Mixin-Layers \cite{SB98} are claimed to support the development of efficient bespoke components from a reusable kit of parts, however, these approaches operate primarily at the level of the application code. Feature diagrams \cite{KCH+90} provide an abstract view of the components that may be constructed within the Software Product Line Architecture (SPLA), expressed in terms of commonality and variability. A general problem with the above approaches is that, whilst each provides support (either implicitly or explicitly) for the SPLA, each relies on its own notation and understanding of the semantics and
constraints of the SPLA, and such approaches do not integrate well into a coherent, standardised modelling environment.

1.3 Contribution

This body of work is motivated by the problems outlined above in supporting an integrated approach to the modelling and development of families of software intensive systems within the framework provided by an industry standard modelling language; the almost ubiquitous UML. A family-oriented approach to modelling, such as feature modelling, focuses attention on the commonality and variability exhibited between members of the family, such that we may differentiate between individual members. The notion of the SPLA provides the architectural framework within which the software application may be deployed; however, in order to achieve this view, we must provide an explicit definition of the meaning of the concepts involved, and support transformations between such views; the OMG’s recent initiative to support QVT (Queries, Views, and Transformations) provides some level of support in respect to the latter [OMG08].

Meta-modelling provides one approach to the definition of Domain-Specific Languages (DSLs) in terms of the abstract syntax and semantics – the abstract syntax describes the concepts of relevance to the domain, whilst the semantics gives these concepts meaning. A concrete textual or graphical syntax may also be specified to provide a user-friendly interface to the DSL (there may even be multiple concrete syntaxes defined for a given DSL). Instead of using a large, general purpose modelling language (such as the UML) and applying local interpretations to generic concepts, a core set of simple language modelling elements are expressed in the form of a metamodel, and richer languages are then constructed on top of this metamodel; e.g. MOF [OMG06] is the metamodel upon which the UML is built, introducing the main concepts of the language such as namespaces (packages), classes and associations. The development of the DSL concepts from the small, well-defined core language (such as the MOF) allows us to define a modelling language that is tailored to the target domain whilst avoiding the vagueness of applying local interpretations to overly general concepts. If it is the case that a foundation concept requires refinement to represent a domain concept more accurately then we are able to extend the underlying metamodel (e.g. MOF) via a profile, and build our DSL in terms of the extended metamodel. The adaptation of the underlying metamodel via profiles takes the form of a collection of stereotyped model elements derived from the underlying metamodel (which may also include profiles); such stereotyped model elements are considered to be either lightweight or heavyweight. A lightweight profile covers the introduction of a new model elements that are simply a renaming of existing model elements, e.g. one might wish to develop a metamodel of IDEF1X [NIST93]. In such a situation, one might stereotype the meta-class `Class` as an `IDEF1X Domain`; this is fine as (hopefully) we know what we mean by an `IDEF1X Domain`, we might even use the stereotype to render instances in some
domain-specific icon, but it doesn’t help explain what we mean to a third party as the stereotyped class has exactly the same characteristics as the meta-class Class, clearly they may apply their own (different) interpretation to the same concept. A heavyweight profile, by contrast, provides domain-specific model elements as stereotypes of existing model elements and also describes additional attributes, constraints and (possibly) a specific semantics; e.g. the meta-class Class is extended to provide the domain-specific class IDEFIX Domain, and this class extends the meta-class with additional attributes and constraints that ensure that only valid instances may be created, the rules for validity being different (more refined) to those applied to the meta-class Class. An example heavyweight profile is shown in Figure 1.
Figure 1 – Example Heavyweight Profile
The abstract syntax of the DSL is defined in terms of a core set of simple language modelling elements which are expressed in the form of a ‘meta-model’ (e.g. MOF and [possibly] one or more profiles); richer languages are then defined in terms of this small, well-defined core (e.g. the UML infrastructure). Mappings support the transformation of instances of one (meta) model into instances of another via a set of explicitly modelled transformation rules, in the case of the example model in Figure 1, the author has also developed a metamodel of the KFL language [OW08] and defined a set of mappings from the IDEF1X metamodel to the KFL metamodel such that a model expressed in IDEF1X may be transformed automatically into a model expressed in KFL and then turned into program source code that may be uploaded into the Ontology Works IODE tool as illustrated in Figure 2. This model-based approach was developed as part of a larger body of work and is described more fully in [DSB+08]. This is an example of horizontal model transformation, although vertical transformations are also possible and indeed receive much attention in the MDA literature [CSW08]. Metamodelling provides an approach by which we may provide a well-defined modelling language tailored to a specific domain; when supported by tools this language may be checkable against well-formedness rules and semantics, providing an opportunity for increased rigour. In the case of the example cited above, a parser has been written to load an instance of the IDEF1X metamodel into the XMF tool; we are then able to check validity of each model prior to generation of the KFL source code.

Figure 2 – Model to Model Transformation

The development of (meta) models from a common foundation modelling language (such as the MOF) also provides the ability to transform instances of one model into instances of another
following defined mapping rules; once again, an executable tool can enforce the rigour of the mapping specification. The use of metamodelling to define modelling languages provides the benefit of rigour in terms of the definition of constraints and semantics and is seen to offer some advantages over the more classical formal specification techniques, such as Z [Spi92], such as increased accessibility to a wide audience, the availability of a number of suitable industry strength tools, and increased scalability. The DSL may be bootstrapped from the small, well-defined core; tools used to reason about the core may also be brought to bear on the DSL as similar concepts and techniques are used throughout the language definition, leading to a structured and more coherent language. Models developed in this way are also more open because they are expressed in languages with the same underlying constructs. The definition of mappings between (meta)models allows instances of one meta-model to be transformed into instances of another; such mappings are also provided in the form of a model. Such an approach epitomises the goal of the OMG’s concept of MDA, where a coherent modelling approach is applied to support portability and interoperability of both applications and (meta)models.

The contribution of the thesis is the definition of a profile that will support the extension of the MOF metamodel to provide support for the rigorous modelling of the SPLA. The components forming the SPLA profile are grounded firstly by an informal description of the semantics required, this description is subsequently refined by a specification expressed in Z [Spi92], and finally translated to a MOF-based architecture and realised in an executable metamodelling tool. The benefit of specifying the SPLA component models in Z is that it provides a technology-independent view, making it accessible beyond the realm of the UML modelling community, whilst the benefit of the metamodelled, MOF-based view is that is accessible to the UML modelling community and is demonstrable by an executable metamodelling tool (XMF-Mosaic). The benefits of such a profile are primarily related to the ability of the user to check the implementation against constraints that are expressed in a form suitable for execution by a tool. However, the partitioning of the SPLA into a number of composable parts follows the MDA approach of modelling both the components and the transformations between them, and this is seen as providing leverage from both the feature model and target architectural model; viewed this way the SPLA becomes an SPLA of SPLAs. Hence, the metamodel of the SPLA offers a future-proofed approach to product line development, not least because we can change the constituent components at both fine and coarse levels of fidelity, e.g. ranging from (say) extension of the feature model(s) to substitution and re-mapping to alternative target architectures.

The UML comprises a layered architecture that is often condensed into a four key layers; known as layers m0 through m3 [Kob99] (see Figure 3). The casual UML modeller is likely only to every see the top two layers (m0 & m1). The Model layer (m1) is the layer within which the user will
generally define the model of the system to be constructed (e.g. a library database) in terms of instances of UML model elements (e.g. packages, classes, and associations), and the User Objects layer (m0) is the layer within which instances of the elements declared in the model layer will be instantiated (e.g. a particular instance of the library database); hence the Model layer (m1) provides the schema to the User Objects layer (m0). The Metamodel layer (m2) is the point at which most UML modelling tools (e.g. Rational Rose) provide an implementation, providing the user with access to the building blocks required to create business models, e.g. providing access to the UML package, class, and association model elements – one might think of this as the API for the UML. The UML metamodel (m2 layer) provides the necessary well-formedness rules and semantics to be applied to the UML models, and the UML metamodel is itself an instance of a model, the MOF, shown at layer m3; and is the meta-meta-model to the User Objects layer. The MOF provides a metadata management framework and a set of metadata services to enable the development and interoperability of model and metadata-driven systems [OMG06], and provides a common (and open) backbone to a number of the OMG’s technology standards, such as the CWM – hence, any model that can be expressed in the MOF is potentially interoperable with any other MOF-based model.

![UML 4-Layer Metamodel Architecture](image)

**Figure 3 – UML 4-Layer Metamodel Architecture**

The MOF at version 2.0 provides support for the UML at version 2.0, the previous version of MOF (version 1.4) providing the foundation for the UML at version 1.4 through 1.5. One can see from the layered metamodel architecture of the UML (Figure 3) that extension of the modelling concepts
provided by the MOF in the form of a profile will provide a package of additional functionality that may be adopted by the UML (layer m2) of any MOF-aware tool; alternatively, the definition of a MOF-based profile could provide a specification of functionality that may be implemented by a tool vendor and re-exported to user via the UML API.

This profile will be based on a meta-modelling architecture approximating to the MOF 2 as exported by an appropriate commercial meta-modelling tool (XMF-Mosaic), will support the specification of feature models, and provide explicit mappings to the target architecture (which will also be described in the form of a metamodel). The meta-modelling approach followed provides the ability to define models that are correct, verifiable and amenable to tool checking. The thesis will investigate the support required to facilitate the generation of conforming refinements to the feature model (configurations) and the subsequent generation of models (or model fragments) from a configuration into the form of an architectural blueprint. The work will be validated within the context of two small case studies, one relating to the domain of tactical data links, the other relating to that of an avionics navigation system. Form this it is proposed that feature modelling and the SPLA is a generally applicable domain-independent technology; therefore it is anticipated that the resulting profile will also be applicable to additional domains.

1.3.1 Novel Aspects

This work is novel because it adopts a meta-modelling approach in the style advocated by the 2U Consortium [2UC02], and more recently [CSW08] to the description of the components comprising the SPLA: features, their permissible relationships, the creation of configurations and specifications, and, ultimately, the generation of products (models). The meta-modelling approach to be adopted will support the definition of SPLA models primarily in terms of an abstract syntax and semantics; a textual concrete syntax for feature models will also be described and mappings between models will be modelled explicitly. This approach is amenable to tool support and will enable the models to be validated against constraints defined in the metamodel. The advantage of providing separate models of the abstract syntax and concrete syntax is that a common model of (e.g.) features could be deployed supporting a variety of renderings, e.g. textual, tabular or graphical, and offering support for different dialects tailored to the user’s domain or toolset; one might envisage an overarching SPLA of SPLAs, describing such choices. In addition a more abstract implementation-independent model of the SPLA is provided, introduced initially in narrative and subsequently recast in the formal specification language Z [Spi92], thereby providing an additional level of accessibility of the metamodels presented to support the SPLA.

The explicit modelling of the SPLA and the mappings from the feature model into an appropriate architecture is a necessary foundation from which one could define product configuration rules that
could support the generation of bespoke components from a kit of parts. This kit of parts could be model fragments or code fragments (or both); hence the proposed technology is tentatively termed *generative modelling*. This work leverages from metamodelling tools and techniques developed under the RFP supporting the re-architecting of the UML for UML 2; it is, however, independent of UML 2 insofar as it does not place any requirements on the modelling language itself, deferring instead to the underlying metamodel MOF, from which it will be required to derive some of the feature modelling components. The relationship to the UML is decoupled further by the use of the commercial metamodelling tool XMF-Mosaic that is, itself, derived from MOF, providing an executable meta-modelling environment. Furthermore, an abstract specification of the SPLA models based on nothing more than predicate logic provides separation from both the MOF and the XMF toolset.

### 1.3.2 Hypothesis

As models take on more prominence in the development of software intensive systems they become the currency of reuse, taking the place formerly occupied by the source code. Development of systems from the family-oriented view has the potential to provide significant leverage; productivity benefits of up to 300% have been reported, with some organisations reporting as much as 75% of product software being built from a core asset base, although the cost of setting up the SPLA requires significant investment [CN02]. Code generation techniques such as generative programming also show promise, with the potential of improving productivity by supporting the generation of custom components. Code generation techniques, such as Frame Technology, have been reported to yield productivity increases in the order of 100% [Bas97] and are reported elsewhere [HE03a]. The effectiveness of the SPLA is highly dependent on the distance separating family members. If this distance is too great then the problem reduces to a collection of one-off developments, since there is little commonality between products and the SPLA does not offer any significant benefits, it may even hinder development as developers grapple with unwieldy components in an attempt to enforce commonality across disparate family members [CE02]. The OMG’s MDA initiative seeks to provide leverage via the transformation of models (essentially mappings provide the glue that binds together models, whilst insulating such models from each other).

The hypothesis to be tested by this thesis is expressed as a number of research questions regarding the design, implementation, and use of the SPLA following a metamodelling approach. The application of such an approach raises a number of issues that are to be investigated and reported in the thesis, each of which is described below.

---

7 This term has been used by other authors in the context of computer graphics and hardware fabrication.
8 Available from www.ceteva.com
1.3.2.1 A Metamodelled Foundation for the SPLA

It is asserted that the notion of the SPLA is not well-founded in the literature. In particular, the feature model appears to perform multiple roles, defining the relationship between the rather abstract concepts represented by features whilst also implicitly defining the relationship between domain elements. This is seen as blurring the distinction between the problem space (the feature model) and the solution space (the domain model). The thesis will investigate the applicability of a metamodelled description of the SPLA following the modular design illustrated by the package diagram at Figure 4. The SPLA metamodel will comprise a number of related components with the intention of providing a modular and coherent description, such that the overloaded purpose we see reported in the literature regarding the feature model is removed, resulting in an SPLA metamodel exhibiting a strong separation of concerns. The hypothesis will be tested by the development of metamodels elaborating the packages illustrated in Figure 4 (see chapters 4 and 5), and the subsequent application within two case studies drawn from different domains (see chapter 6).

Figure 4 – Models Comprising the SPLA

1.3.2.2 A Semantics for Features

The abstract nature of the concept of a feature has undergone only minor refinement since its introduction due to Kang’s publication of Feature-Oriented Domain Analysis (FODA) in [KCH+90], and there is generally agreement amongst authors (see section 2.6.2). FODA
characterises the relationship between features in the form of the well-known programming language constructs:

- Aggregation/decomposition
- Generalisation/specialisation
- Parameterisation

Given the overloaded role played by the feature model it is reasonable to question that the use of three relationships on the feature diagram will be adequate for all possible situations. In particular, use of the aggregation and specialisation relationships may be viewed as introducing design decisions at an arbitrarily early stage in the description of the SPLA; a more suitable semantics should be described that is independent of the typical programming constructs. Hence, it is asserted that the relationships between features are simply that [relationships between features], what this means at the level of the application program components may vary depending on the context and even upon the features supported by individual programming languages. It is asserted that the application of meta-modelling techniques following the style popularised by Language-Driven Development (LDD) and Language Oriented Programming (LOP) approaches will help to make explicit the semantics of the SPLA and clarify the transformation of the SPLA into a valid system configuration. Furthermore, support for model transformation via appropriate mappings facilitates the generation of views of the SPLA appropriate to many domains, e.g. generation of code, documentation, validation constraints, etc., adapted for specific customers/disciplines (publications, development, safety, testing). This hypothesis will be tested via the development of two case studies in chapter 6.

1.3.2.3 Is the Metamodelled SPLA Domain Agnostic?

Although there is evidence in the literature of the application of SPLA-based software development programs in diverse domains, e.g. engine controllers (Cumins) [CE02] and avionics mission systems (Boeing) [Sha00], however there is no supporting evidence to suggest use of a common underpinning SPLA metamodel, or even a common understanding of the concept of the SPLA. Hence, it is asserted that the metamodelled SPLA components should be domain agnostic, this will be tested via the application of the metamodelled SPLA on two case studies addressing model reuse in significantly different domains.

1.3.3 Structure of Thesis

This thesis is structured as follows: Chapter 1 (this chapter) provides an introduction to the problem to be addressed, the motivation for the work, and a brief statement on the technology to be used and the hypothesis to be tested. Chapter 2 provides background information and a review of the most relevant literature relating to the proposed metamodelling of feature model and, SPLAs. Chapter 3 describes the process that will be used to develop and populate the SPLA, having designed and
implemented a suitable set of metamodels. Chapter 4 provides an informal description of the metamodels proposed to support the SPLA, moving on to a more robust implementation-independent specification in Z [Spi92]. Chapter 5 describes the work undertaken in support of the development of a UML profile to provide modelling facilities for the SPLA, and to offer a potential solution to the problem of large-grained software (model) reuse. Chapter 6 provides an evaluation of the SPLA metamodels via two small case studies in the domains of Tactical Data Links (TDLs) and an avionics navigation system. Chapter 7 summarises the work undertaken, identifying the contribution of the work, lessons learned and potential opportunities in the future, it then reassesses and evaluates the extent to which the research questions raised in section 1.3.2 have been achieved.
2 Literature Review

This section provides a context for the work described in chapters 4, and 5 by providing a motivating case for the explicit modelling of the SPLA and providing a review of the most relevant information in the literature aimed at supporting significant levels of improvement in software productivity.

This section begins with an introduction to the UML before investigating its structure in the form of the MOF. The concept of language engineering via meta-modelling is described, leading to the definition of Domain-Specific Languages adapting and extending standard modelling languages via profiles; both lightweight and heavyweight approaches to profile definitions are discussed, including a brief description of the SysML as an example of a popular extension to the UML.

The case for reuse is investigated, covering both fine-grained and large-grained approaches, leading to the identification of the (Software) Product Line Architecture. The review of relevant technologies is concluded with a summary of popular approaches to large-grained software fabrication; the Architectural Description Language and its application within the UML, and Model-Driven Design, a paradigm within which models become the currency of reuse.

2.1 An Overview of the UML

The UML has been chosen as the modelling language for this research because of its wide-spread adoption throughout the software engineering community and general accessibility, the UML is generally considered to be the lingua franca for the modelling of software intensive systems [Tho04]. The UML underwent a significant re-architecting exercise (in evolving from UML v1.4 to 2.0). At v1.x, it may be argued that the UML had become bloated and inconsistent, containing support for many domain-specific modelling requirements, and documented to various degrees of rigour: from exemplar diagrams, to narrative descriptions, to formal specification with the Object Constraint Language (OCL). Furthermore, UML (at v1.x) was not truly extensible as a result of a lack of tools providing an open and extensible implementation of the underlying metamodel. A typical approach to the application of a domain-independent extension of the UML at v1.x would be to identify a user-visible model element (such as the meta-class Class) and define some appropriate stereotype (e.g. «Factory») that could then be adopted by users to convey some additional (but informal) semantics. Depending upon the level of tool support, the stereotyped element could be rendered in some domain-specific form (e.g. Rational Rose supported such extensions).
In the example below the stereotypes «Factory» and «Aircraft» are associated with the metaclass `Class`, using Rational Rose 98, this allows the tool to add these domain-specific stereotypes to the default stereotypes provided by the tool for all user-declared classes (see Figure 5).

![Figure 5 – Application of Lightweight Stereotypes](image)

The tool allows us to assign custom icons to each stereotype (one icon per stereotype), such that the class diagrams may be rendered using the graphical language extensions provided with the intention of providing additional information to the reader (see Figure 6).

![Figure 6 – Application of Lightweight Stereotypes with Graphical Support](image)
There are a number of problems relating to the weak semantics of the language extensions invited by this approach:

1. The stereotypes are simply labels applied to existing meta-elements, hence the semantics inferred by the stereotype Factory may not be understood to be the same concept by all readers.

2. The icons applied are non-standard, they would not be re-rendered if the model were to be exported to another tool- even another instance of Ration Rose would require that the necessary tool configuration extensions (and it may even have its own, possibly conflicting, definitions).

3. The tool does not support any form of structuring of the language extensions – all users can see all extensions all of the time.

4. The tool lacks fidelity; Stereotypes may be applied in possibly unintended situations (e.g. Class Utility is defined by the standard [OMG00] to be a stereotype «utility» applied to the metaclass Class that the tool vendor has chosen to implement as a first class citizen).

The general approach to the development and application of a lightweight profile (as indicated above) was to try to gain confidence in one’s adaptation of the metamodel by applying stereotypes to existing modelling concepts (e.g. applying a stereotype «Factory» to the core concept Class), and, depending upon the level of tool support, providing a custom icon for the stereotyped (Factory) concept. Finally, there was no tool-based test of correctness of the metamodel either pre or post-extension, with the metamodel generally being hidden behind an opaque vendor-specific API, it is such an approach that helps to admit item 4 (above).

It was the intention that the UML 2 provide a small core set of functionality with a precise semantics (the Infrastructure), this serves as the kernel to the language superstructure [OMG07a, OMG07b]; arguably, this aim has only been partially realised – the structure is there, however some authors argue that the precise semantics is not, e.g. [GS04] [ZDD06] and [DD06]. In [BCD+06] the authors provide a description of a collaborative programme to define a formal semantics for UML 2, noting the motivation for the activity as being a result of the complexity of the existing language specification using a combination of notations, including natural language text. The authors note that a consequence of the complexity and ambiguity is that it is difficult to detect and correct subtle errors, incompleteness and inconsistency, these issues, in turn, complicate development of UML-compliant tools and compromises interoperability between tools (as each tool vendor may attach a subtly different semantics to the same language concept). This deficiency is also identified by Thomas in an article primarily concerning MDA and model transformations, concluding that the UML lacks both a reference implementation and an operational semantics, resulting in tool implementation difficulties [Tho04]. In this instance the author is making the point...
that the lack of a clear, and checkable, semantics will lead to interoperability problems and these will, in turn, militate against the widespread adoption of MDA by the software engineering community.

The 2U Consortium’s proposal for the re-architecting of the UML was predicated on the use of metamodeling techniques as a mechanism for the specification of a family of modeling languages. Key to this approach was the definition of a metamodeling language (MML) which provided the building blocks and mechanisms to support the definition of higher-level modeling languages (e.g. UML). MML\(^9\) was the first in a line of metamodeling tools developed, initially, in academia, and subsequently as a commercial enterprise. MML enforced a strict separation of Model concepts (e.g. Class, Attribute, etc.) from Instance Concepts (e.g. Object, Slot, etc.), Model concepts being mapped to Instance concepts through Semantics; this is where well-formedness rules were provided (e.g. to prevent the instantiation of abstract classes). Model and Instance components comprised an Abstract Syntax and a Concrete Syntax, the Abstract Syntax describes what is meant by a modeling concept (e.g. Class), and the Concrete Syntax describes the rendering of the concept (e.g. a visual or textual representation of a Class). A mapping from the concrete syntax to the abstract syntax was then defined; this package is analogous to a parser, instances of the concepts introduced by the abstract syntax reside in the semantic domain. An outline of the MML metamodel is shown in Figure 7. The 2U Consortium’s approach to meta-modelling was supported by a tool (MMT) that had the ability to check the correctness of model components expressed in a programming language similar to Java; it also provided drawing facilities via Java Swing components and also offered documentation facilities.

\[\text{Concrete Syntax} \rightarrow \text{Abstract Syntax} \rightarrow \text{Semantic Domain} \]

\[\text{Parser} \rightarrow \text{Semantics} \]

**Figure 7 – Components of the MML Metamodel Architecture**

\(^9\) MMT was a prototype meta-modelling tool that (after a number of iterations) resulted in the commercial meta-modelling tool XMF-Mosaic.
The design of the MML recognised that language concepts may exhibit common structures or patterns, support being provided in the form of templates (analogous to the Generics of Ada95 [Int95]). Hence, one could specify some general pattern (e.g. a Container pattern) and simply 

*stamp-out* the pattern (e.g. the Container-Contains relationship) within a context specified by the client. MML also provided support for package specialisation, based on the notion of package imports taken from Catalysis [DW99]. The adoption of the metamodelling approach using a suitable metamodelling tool, such as XMF, will allow the expression and verification of the concepts required to support feature and configuration models, and facilitate the embodiment in a suitably well defined core of the UML, such as is provided by XCore (roughly analogous to MOF 2). However, it is expected that a requirement will be identified for further architectural mechanisms to support PLAs and product configurations. Of primary concern is the narrowing of the interface from a PLA (the superset of products) to a single Configuration (one specific product) based on user-directed choices. It is clear that, whilst the template approach is a powerful concept, it does not support a PLA to Configuration mapping since this will generally lead to a restriction in the scope admitted by the PLA as we go from a more generic view to a more specific view that may require the use of (e.g.) predicate logic and/or some form of metalanguage to identify legal optional components for the PLA interface. The application of user choices to the PLA will lead to the generation of a configuration that is a subset of the PLA (the source model); furthermore it is asserted that the semantics of the relationship between features is not the same between to the two models. This aspect is discussed in more detail in subsequent chapters.

It is worth noting that Greenfield and Short provide a discussion of the reasons why the UML was considered unsuitable for the automation of the software development artefacts described for the Software Factories approach [GS04], concluding that it ‘falls well short of being a suitable foundation for formal domain specific languages’. The justification for Greenfield and Short’s conclusion stems largely from their assertion that there is a lack of a well-defined semantics afforded by the specification of UML 2, and the definition is not structured appropriately to support the definition of new languages. Both of these issues were raised against UML v1.x, and, although the specification of UML 2 has been strengthened, it still lacks a clear and consistent semantics and is not expressed in a form that may be easily executed (and validated); hence tool vendors may choose different interpretations (both of which would be arguably correct, since the standard may be ambiguous). The authors go on to try to demonstrate why the definition of a profile in the UML would also fail to provide sufficient rigour, however the example profile cited (EAI profile) is relatively lightweight. Whilst these are valid criticisms of the UML, they are believed to be surmountable via the application of the meta-modelling approach that will be described in Chapter 5, which is based on heavyweight extension of a well-defined core language, such that it is
possible to provide extensions to the language in a well-defined form without the unnecessary baggage that might come from lightweight language extension.

2.2 Meta-modelling

Meta-modelling has gained significantly in popularity over the past ten years, with a corresponding increase in the size of the meta-modelling community, fuelled largely by the market uptake of modelling languages with meta model-based foundations (such as the UML and related technologies like MDA). A metamodel is a precise definition of the constructs and rules needed for creating semantic models\(^{10}\); hence the metamodel underpins the model by defining some domain of relevance, constraining the scope of those models that are declared valid within this domain. Significantly, a metamodel may be used to attach semantics to a model, such that not only are the concepts to be modelled described, but also their meaning. Metamodels may or may not be executable, in the work that follows our primary interest is in executable metamodels, such that models (instances of the metamodels) may be checked by a suitable tool for validity against the metamodel. A metamodel may be considered analogous to the schema used to define the structure of a database in terms of the entities and relations.

It is also interesting to note the overlap between metamodels and ontologies as both seek to identify a domain and characterise it in terms of the concepts it comprises and the relations between such concepts such that well-formed statements in the domain may be declared. Pidcock defines an ontology as a controlled vocabulary expressed in an ontology representation language, noting that this language has a grammar to define what it means to be a well-formed statement \([\text{Pid03}]\). Pidcock asserts that, if we view a metamodel as a model of some domain of interest, then it may be compared to an ontology. The author notes that a valid metamodel is an ontology, but not all ontologies are modelled explicitly as metamodels, and concludes that a metamodel is an ontology used by modellers. Czarnecki et al. take this view further by raising the comparison between ontologies and feature models and describing mapping rules from the feature model to the ontology, although they warn of potential complications in such mappings if the ontology is closer to implementation and the feature models are closer to requirements \([\text{CHK+06}]\), i.e. mismatch in levels of abstraction of the models.

2.2.1 What is Meta-Modelling?

Meta-modelling is defined in the Wikipedia\(^{11}\) at its most general as:

> The analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for the modelling a predefined class of problems.

\(^{10}\) See [http://www.metamodel.com](http://www.metamodel.com)

\(^{11}\) [http://en.wikipedia.org/wiki/Metamodel](http://en.wikipedia.org/wiki/Metamodel)
An analogy is drawn between the correspondence of a model to its metamodel and the conformance of a computer program to the grammar of the programming language in which it is written, and this trait is clearly evident in the MOF, underpinning as it does a number of higher level user models such as UML, SysML, CWM, etc., the models being validated against the constraints defined in the metamodel, i.e. the metamodel constrains the permissible structure of the models, the model being an instance of the metamodel. MOF may thus be viewed as a DSL for creating metamodels.

The UML comprises a layered architecture; this is often condensed into the four key layers (although it should be noted that the UML 2.0 introduces sub-layers within the model layer – m1), see Figure 8.

![Diagram of the UML Model Stack](image)

**Figure 8 – The UML Model Stack**

The left-hand-side of Figure 8 shows the UML model stack in the form of the major constituent packages, whilst the right-hand-side provides an example used to express an instance of a Cat (the object Tiddles), showing the implicit usage of the underlying model, meta-model and meta-meta-model. It should be noted that the rather obscure relationships on the meta-meta-artefacts Object and Class prevents the infinite recursion that would otherwise occur as we descend to
yet deeper meta-levels. Hence, by adapting the components exported by the metamodel (m2) level it is possible to capture and express the properties required of our models (languages) that are instances of this metamodel, this may be achieved via extension (or restriction) of the components with which this layer is pre-populated. In this way, meta-modelling supports the heavyweight extension of the modelling language, leading to the definition of a highly-focussed DSL, comprising abstract syntax, concrete syntax and semantics, adapted to a specific domain, such as the modelling of features, GenVoca architectures, etc. Such metamodels may then be composed via well-defined transformations to support broader domains; this is in stark contrast to the one-size-fits-all approach provided by earlier versions of the UML, although there are still relatively few commercial tools available to take advantage of such an approach.

2.2.2 Why is Meta-Modelling Useful?

Although the UML provides many modelling features, serving a broad spectrum of domains, it is possible that there may not be a clean fit between the modelling language and the modelling requirements in some particular domain; this may be because the language is insufficiently precise, or the necessary modelling features may be missing altogether. Fortunately, the UML provides an extension mechanism via the notion of the profile that is intended to allow language designers to adapt the language to fit their requirements; this adaptation is performed by either lightweight or heavyweight meta-modelling. Profiles aren’t new to UML; however the definition of the extension mechanism has been strengthened somewhat for UML 2.0 [SK06]. Unfortunately, problems remain with regard to the creation and instantiation of heavyweight profiles, since access to the underlying metamodel is required, and most tool vendors consider this to be proprietary, we may get access to the metamodel via a tool-specific API, but the resulting solution tends not to be very portable. As a result of these restrictions, profiles often provide little more than an extension to the graphical syntax of the language based on particular stereotypes or roles, such profiles are considered to be lightweight extensions to the language. However, the OMG publishes a number of profiles to extend the UML for particular domains, such as the Real-Time Profile\(^\text{12}\) [OMG05a], CORBA Profile [OMG02], etc.

The XMF-Mosaic tool is amongst a small number of tools differing in this respect, the metamodel (in this case an approximation to MOF 2) is provided by the tool in an open, extensible and executable form, providing full support for metamodel extension via the profile, and meta-modelling via MOF models. Meta-modelling allows us to design models that are well-grounded by being explicit about the semantics required, indeed in [SV06] Stahl and Völter

\(^{12}\) Replaced by MARTE [OMG07d].
describe meta-modelling as being one of the most important aspects of Model-Driven Software Development (MDSD), being of particular relevance to:

- Definition of abstract syntax (DSLs)
- Improved rigour – models are validated against the constraints of the metamodel
- Mappings – model transformations
- Code generation
- Tool customisation/integration – via the explicit metamodel

This is a view that is reiterated by Clark et al. [CSW08], stating that meta-modelling is the way to achieve the rapid design and integration of semantically rich languages in a unified way (because the languages are based on the same meta-model).

A well-defined metamodel comprises an Abstract Syntax, Concrete Syntax and Semantics, the relationship between these aspects is shown in Figure 7.

Each of the components of the metamodel plays a well-defined and encapsulated role, each is described in the sections that follow.

2.2.3 Abstract Syntax

The Abstract Syntax of the language corresponds to the taxonomy of the concepts to be encapsulated, defining the relationships between such concepts and identifying the well-formedness rules that must be applied to any valid instance of the metamodel. As an example, the abstract syntax of a metamodel of a GenVoca architecture is likely to comprise the concepts of a Realm and TypeEquation, and enforce the type compatibility when declaring instances of the model.

The metamodel is further likely to define constraints across instances, such as enforcing type compatibility:

```context GenVoca::Specifications::TypeEquation
self.component.requiredInterface->forAll(realm |
  self.subExpressions->exists(te | 
    realm.layers->includes(te.component)))
```

Equation 1 – Example Constraint (GenVoca)
2.2.4 Concrete Syntax

Although the abstract syntax is concise, it is not often convenient for the user to have to instantiate models in these terms, a more user-friendly view is required; this view is provided by the Concrete Syntax. The concrete syntax is expressed (typically) in either a textual or graphical form, and may include additional items that do not relate directly to the concepts of the abstract syntax, but, instead, add readability, e.g. the for-loop found in many programming languages comprises an identifier, a guarding expression and a body (Figure 11).

It’s not very convenient to write programs in such a terse style, and, hence, the concrete syntax may include a number of keywords to improve legibility (see Figure 10, keywords shown in bold) and provide the language with a general style:

```
for Item in Min .. Max loop
  X := X + Item;
end loop
```

Figure 10 – For-Loop (Concrete Syntax)

Figure 9 – Abstract Syntax Model (GenVoca)
The concrete syntax provides the user’s view of the specification of the model/program and, as such, provides the grammar to which it must conform. As an example, Figure 12 provides a simple program expressed in an Ada-like concrete syntax based on the metamodelled language described above.

The syntax is Ada-like, it’s not Ada; e.g. a semi-colon isn’t specified following the key-word end loop.

2.2.5 Relating Concrete Syntax and Abstract Syntax

Clearly, there must be some relationship between the Abstract Syntax and Concrete Syntax, it is necessary to be able to generate models in abstract syntax from specifications expressed in the concrete syntax. The relationship between the two is that of a parser, the parser parses the concrete syntax (if it complies with the grammar defined for the concrete syntax) and generates the necessary objects in the abstract syntax; Figure 13 provides an illustration of the grammar (expressed in BNF using the XMF-Mosaic tool) against which the program (concrete syntax) is parsed.

---

13 The syntax is Ada-like, it’s not Ada; e.g. a semi-colon isn’t specified following the key-word end loop.
context Statement
@Grammar extends Expression.grammar
Statement ::= 
  'declare' 
  decs = SequenceOfDeclarations 
  'begin' 
  body = SequenceOfStatements 
  'end' 
  { BlockStatement(decs, Block(body)) }.

SequenceOfDeclarations ::= DeclarativePart*.
<snip>
</snip>
LoopStatement ::= 
  WhileLoop | ForLoop.

WhileLoop ::= 
  'while' pred = Expression 'loop' 
  body = SequenceOfStatements 
  'end' 'loop' 
  { WhileLoop(pred, Block(body)) }.

ForLoop ::= 
  'for' identifier = Name 'in' range = Range 'loop' 
  body = SequenceOfStatements 
  'end' 'loop' 
  { ForLoop(identifier, range, Block(body)) } | 
  'for' identifier = Name 'in' 'reverse' range = Range 'loop' 
  body = SequenceOfStatements 
  'end' 'loop' 
  { ForLoop(identifier, Seq{range->last, range->head}, Block(body)) }.

Range ::= 
  from = SimpleExpression '.' '.' to = SimpleExpression 
  { Seq{from, to} }.

Figure 13 – For-Loop Example Grammar

Parsing the example program shown in Figure 12 results in the generation of the abstract syntax shown in Figure 14, this may be subsequently executed if we attach an appropriate semantics to the language. This also serves to illustrate the benefits of a more human readable concrete syntax supported by a parser.

BlockStatement(Seq{ 
  ObjectDeclaration(Min,Integer,Constant(MyInt(0))), 
  ObjectDeclaration(Max,Integer,Constant(MyInt(7))), 
  ObjectDeclaration(X,Integer,Constant(MyInt(0))))}, 
Block(Seq{ 
  ForLoop(Item,Seq{ 
    Variable(Min), 
    Variable(Max)}, 
  Block(Seq{ 
    Assignment(X,BinaryExpression(+,Variable(X),Variable(Item))))}))})

Figure 14 – Generated Abstract Syntax
2.2.6 Semantics

Whilst the Abstract Syntax and Concrete Syntax provide a description of the concepts within the model and their rendering, they provide little in the way of describing the meaning of the concepts; this is the semantics of the model; such meaning may relate to the behaviour, static properties, or the relation of a language to some other language [CSW08]. Clark et al. describe the semantics of programming languages and modelling languages as being similar to (but more formal than) the semantics of a natural language, taking the form of a correlation or mapping between language concepts and thoughts and experiences of concepts in the world around us; and assert that semantics should be of practical use in understanding the meaning of a language. Such semantic descriptions of programming and modelling languages may be provided via a number of popular formalisms, e.g. translational, operational, extensional or denotational; each of these approaches to the definition of a semantic is summarised below.

- **Translational** – A translational semantics involves the translation of the source language into a target language with a well-defined semantics. A translational approach may be achieved via a mapping from the source language into the target language, or via the parsing of the source language directly into the target language – not this approach was taken by the GNAT Ada compiler such that the Ada source code was parsed into C and subsequently compiled. Whilst this approach allows reuse of tried and tested artefacts the translation will become more difficult as the source and target languages diverge.

- **Operational** – An operational semantics is analogous to the definition of an interpreter, typically one would write an operation \( \text{eval} \) on each concept in the source language describing the steps to be taken to execute the concept in terms of operations on the language itself (this was the approach taken in making the language shown in Figure 13 executable, see Figure 15).

- **Extensional** – An extensional semantics is achieved by taking building the semantics of the source language on top of the semantics of an existing foundation language with a clear semantics such that we extend the semantics of the foundation language only where necessary. This is a fairly natural inheritance-based approach and is aligned closely to the notion of a heavyweight language profile.

- **Denotational** – A denotational semantics is the classical abstract, mathematically based description of the semantics of a language; mathematical objects are the denotation of the language concepts.

In the Language Reference Manual for the UML (at v1.3), Rumbaugh et al. take the view that the UML must support the description of a wide variety of systems, and, hence, the definition of the metamodel in terms of a single semantics would constrain the appeal of the language unnecessarily, describing this lack of a clear semantic underpinning as providing different interpretations in the
form of semantic variation points [RJB99]. The authors choose to describe the semantics of the language in the form of natural language with supporting examples, following the general idiom of introduction, semantics, notation, and discussion (e.g. see the definition of an association class). This form of semantics description is fundamentally flawed because the user must interpret the description, and the intentional openness of the semantics appears to allow users from different domains to infer different (and possibly incompatible) meanings for the same concept. Furthermore, there is no litmus test to verify that a concept is being used (or implemented) correctly, hence this issue affects both users and tool vendors – since the tool vendor must infer some semantics in order to implement the concept. The meta-modelling approach described by Clark et al. allows us to make the semantics of the language clear and checkable by a tool, furthermore the semantics of the language is integrated with the definition of the language itself (concrete syntax, abstract syntax, mappings, etc.).
43

Figure 15 – Example Components of a Metamodel
2.3 Model-Driven Language Engineering

Following the widespread adoption of the UML as the *lingua franca* for the modelling of software intensive systems, interest has diverged from the reuse of software artefacts at the source code level, to the reuse and transformation of models. Various initiatives have emerged in support of this goal, the most notable of which are described below.

2.3.1 Model-Driven Architecture

Probably the most prominent approach to model-driven language engineering is the OMG’s Model-Driven Architecture (MDA). The MDA seeks to address two primary concerns that may be viewed as orthogonal aspects of reuse:

1. Reuse of models across different modelling platforms – *model interoperability*
2. Reuse of models in different applications and (possibly) domains – *model portability*

Although the interoperability of models across UML-compliant tools provided by various manufacturers is clearly a significant challenge, particularly as there is currently no litmus test of compliance of a tool to the UML standard (e.g. as was provided for the Ada community via the ACVC suite), the second goal of *model portability* is more relevant to this discussion. The approach advocated by the OMG in support of model portability is via the separation of modelling aspects into clearly defined domain-specific models known as the Platform Independent Model (PIM) and the Platform Specific Model (PSM); as an analogy, one might consider the PSM as being roughly equivalent to an execution platform using the CORBA architecture, whilst the PIM might be equivalent to an application that expressed independently of the execution architecture in terms of the business rules that are to be implemented. However the PIM may depend on the services of more than one PSM, and the PSMs may themselves be nested such that the PSM becomes the PIM to a more target-specific/concrete PSM, see Figure 16.
In the case of MDA, the notion of model dependency is represented via the mapping concept; a mapping takes one or more elements of one type (the *domain*) and provides a transformation into one or more elements of another type (the *range*); in this way the model dependencies are *soft* insofar as there is no direct reference between models, this is provided by the mapping. An application may thus be ported from one platform to another (regenerated) by the specification of new mappings to the appropriate PIM. Mappings form one part of the OMG’s specification for Queries, Views and Transformations (QVT) that provides the supporting infrastructure upon which the concepts of MDA may be constructed, see [OMG05b]. Although MDA represents a bold step in the direction of model reuse, it is not without its problems [CSW08]:

- The weak definition of MDA means that tool support for MDA is difficult to quantify.
- The mappings between models tend to be too strongly focussed on vertical mappings.
- MDA is built on weak and inflexible architecture – a result of an imprecise semantics.

### 2.3.2 Language-Driven Development

In [CSW08] the authors identify languages as being the central abstraction, rather than the higher-level abstractions usually implied by *models*, and are, therefore, considered to be more widely applicable than MDA; the authors term this approach Language-Driven Development (LDD). The LDD approach seeks to address the shortcomings of MDA and the use of poorly defined high level modelling abstractions by the use of both vertical and horizontal mappings between coherent and well-grounded models; the definition of such models being based on meta-modelling techniques, see Figure 17.
The authors argue that the LDD approach is open and extensible and, therefore, supports the evolution of new languages in addition to integration of existing domain-specific languages (once captured in a suitably semantically rich metamodel) in contrast to the one-size-fits-all approach seen with UML v1.x. An example of the use of LDD in the domain of the Tactical Data Link (TDL) is shown in Figure 18. In this example, the aim is to transform instances of a model relating to the structure of the message catalogue belonging to a particular member of the family of TDLs, into an XML document that may be rendered as HTML within a web-browser. Package JMessages describes the structure of the elements of the message catalogue of the Link 16 TDL, and provides well-formedness checks to verify the validity of instances of the model. Package DocumentModels provides a model of a standard document (in terms of sections, figures, tables, paragraphs, etc.), and the relevant well-formedness checks to validate model instances. Finally, package XML provides a model of the XML. In each case, the models focus on a particular domain, and know nothing of each other’s existence. In order to transform instances of JMessages into XML, we first transform instances of JMessages into instances of DocumentModels via the mappings in package JMessages2DocumentModels; we then transform instances of DocumentModels into XML via package DocumentModels2Xml. Instances of the model XML may be output in the form of a string (that we stream to a file); we then transform the XML to HTML via XSLT (a text based XML transformation language with an
implicit model of HTML). Clearly, the example could be extended to incorporate a model of HTML and provide a mapping from XML to HTML, however XSLT is perfectly adequate for the task, and there are a number of freely available XSLT standard compliant tools (e.g. Xalan\textsuperscript{14} and Saxon\textsuperscript{15}).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{model_transformation_overview.png}
\caption{Model Transformation (overview)}
\end{figure}

The domain modelling toolkit XMF-Mosaic\textsuperscript{16} supports the principles of LDD, and, in particular, the transformation between models (via the language XMap – which has both a textual and graphical concrete syntax). The XMap language allows us to identify one or more Element objects as the domain and one or more Element objects as the range for the transformation (see Figure 19), we must then describe the mechanics of the transformation using the XMap language (an extension of Xactium’s XOCL).

\textsuperscript{14} \url{http://xalan.apache.org/index.html} - Xalan home page.
\textsuperscript{15} \url{http://saxon.sourceforge.net/} - Saxon home page.
\textsuperscript{16} \url{http://jessica.xactium.com/web/} - Xactium Ltd.
In the TDL example, a Catalogue contains a collection of Message objects, and a JMessage is a specialisation of the Message concept. Hence, in order to transform an instance of a Catalogue into an instance of a DocumentModel we begin by transforming the Catalogue object and then descend through the object structure, transforming the constituent objects as we go. Figure 21 provides the description of the top-level transformation Catalogue2MainSection as an example in the use of the XMap textual language. In this example, we can see how the mapping queries and navigates the domain model to create an appropriate instance of the target model (the range). The XMap transformation follows the pattern shown in Figure 20.

```
@Clause <className>
  <var> = <patternMatch>;
  ...
  when <booleanExpression>
  do
    <xoclExpression>;
    ...
  where
    <var> = <xoclExpression>;
  ...
end
```

Figure 20 – Structure of XMap Transformation
@Clause toDocumentView

cat = Messages::Catalogue[name = N]
do
  DocumentModels::MainSection[docId = ID, edition = ED, title = TITLE, subTitle = SUBTITLE, date = DATE, writtenBy = AUTHORS, structuredBy = SECTIONS]
where
  ID = "MIL-STD-6016";
  ED = "C";
  TITLE = "Tactical Data Link (TDL) 16 Message Standard";
  SUBTITLE = "Message Description";
  DATE = xmf.date();
  AUTHORS = Set{DocumentModels::Author("Auto-generated by JMessages2DocumentModels")};
  SECTIONS = self.createCannedComponents() -
              append(Seq{DocumentModels::Section{
                name = "", number = 0,
                contains = cat.getMessagesInAscendingOrder() -
                           collect(msg |
                             Seq{JMessage2WordApplicabilityTable() (msg)} -
                             append(msg.getSubLabelsInAscendingOrder() -
                                    collect(subLabel |
                                             SubLabel2MessageSummary() (subLabel))) | flatten)
              }) - flatten
end

Figure 21 – XMap Specification of Model Transformation
2.3.3 Model-Based Software Development

Stahl and Völter describe a more general approach to model-based reuse in [SV06], termed Model-Based Software Development (MDSD). Whereas the LDD approach talks only about aspects relating to model transformation, MDSD is aimed at supporting automated code generation from domain-specific models, using techniques similar to MDA & LDD, but extended to support reuse of non-generated artefacts and also code frameworks via both model-to-model transformations and model-to-code transformations; in this respect MDSD may be viewed as being a lightweight form of LDD. The collection of models and mappings needed to transform a domain model into an application is known as the domain architecture, and is defined relative to a platform on which it is to execute. The advantage of the model-to-code transformation is that the model is aware of the target of the transformations (the platform) resulting in simplified transformations as there is no need to have a complete metamodel of the target language, one can generate directly into a code framework, such as Struts; in this instance one could view the model-to-code mapping as being frame-like. Furthermore, the model-to-code mappings may make use of specific idioms to generate more efficient code, hence MDSD is not as pure an approach as LDD since model dependencies at the lower layers of the domain architecture tend to be hard, but it’s very pragmatic, relying on an implicit dynamic architecture (the target implementation framework), instead of a fully elaborated target language metamodel with the necessary action language transformations.

2.3.4 Language-Oriented Programming

Language Oriented Programming (LOP) is the name given to a comparable paradigm described in [Dmi04] and supported by the Meta Programming System (MPS) marketed by JetBrains. The LOP approach seeks to free the developer from thinking in terms of the implementation language (e.g. Java or Ada95) because such languages are, by their very nature, general purpose and tend to evolve slowly as a consequence (because enhancements to the language and tool set must satisfy many disparate groups working in, possibly, highly diverse domains). LOP recognises the complexity of the intellectual mapping from a conceptual model into a target programming language, and in a style similar to that of LDD, seeks to simplify this complex transformation via one (or more) DSLs that are modelled explicitly; the layered transformational approach is a refinement of MDA.
Dmitriev focuses his interest on three main problems attributed to current programming-oriented approaches:

- It takes a long time to describe the mapping from the conceptual (domain-specific) model to the implementation (the general purpose programming language).
- High-level (problem-oriented) documentation is lost to documentation of the implementation, hampering maintenance.
- Language extensions (class libraries) add (sometimes significantly) to the developer’s learning curve (e.g. consider the MFC).

It is claimed that LOP provides a solution to these issues by allowing the developer to express the design in terms of domain-specific concepts that may be rendered graphically rather than textually where appropriate, e.g. the design of a Graphical User Interface (GUI); in this respect Dmitriev is articulating a case for the clear separation of the Concrete Syntax and Abstract Syntax, providing, possibly, multiple views of the design. Dmitriev argues that a single text-based concrete syntax for a programming language hampers language evolution because new concepts must be fitted into an existing unambiguous grammar, leading to the introduction of complex rules, additional keywords, brackets, etc., whereas an intelligent editor could operate directly on the abstract syntax and render specific views as required – an idea that resonates strongly with Symonyi’s Intentional Programming [CE00]. In Dmitriev’s parlance, a language is characterised by a triad of aspects: \textit{structure} (abstract syntax), \textit{editor} (concrete syntax), and \textit{semantics}.

Dmitriev describes MPS as a \textit{universal platform for designing DSLs along with their supporting tools and environments}, its approach shares many similarities with that of XMF-Mosaic by Xactium. MPS provides three core languages to support the construction of DSLs, the structure language, editor language, and transformation language; the structure and editor languages provide support for the definition of the abstract syntax and concrete syntax, whilst the transformation language provides the necessary mapping between the closely integrated sets of DSLs – the transformation language is equivalent to XMap of XMF-Mosaic, although it is augmented by additional DSLs to support navigation of the model (the Model Query Language and the model pattern-matching language) and the generation of artefacts via stereotypical and extendable templates (the template language). MPS is bootstrapped with a number of low-level languages to support general-purpose programming, collections, and the GUI – again, there is a strong parallel to the XCore and XTools languages provided with XMF-Mosaic.

2.3.5 Summary of Model-Driven Approaches

From the discussion above, one can see that, whilst MDA provides the basic framework to support the transformation between models, MDSD, LDD and LOP take this idea further, describing the domain architecture as comprising (possibly) many closely integrated models requiring both
horizontal and vertical transformations. Furthermore, the model everything approaches of LDD and LOP seek to underpin the role of models as the currency of reuse. MDSD and LOP share the idea of a form of target language wrapper (model-to-code transformations) to reduce the development effort required to provide support for code generation, although neither is particularly vocal on the subject of the definition of the behavioural semantics, although MDSD makes pragmatic use of common frameworks (e.g. struts), and the provision of protected code regions where the programmer can insert code that will not be overwritten by the code generator in subsequent generations (a similar approach is used in other less capable tools supporting code generation, e.g. Rational Rose Ada95\(^{17}\)). On reflection, one could view the approaches of MDA, LDD, LOP, and MDSD as a collection of refinements with MDSD as the most generic (integrating with frameworks, code, and models), whilst MDA is the most specific (being aimed at a UML-based modelling framework), this view is illustrated in Figure 22.

![Figure 22 – Relationship Between Model-Driven Approaches](image)

The motivation behind all of the above approaches is a consequence of the generality and lack of precision offered by popular modelling languages, such as the UML. Significant improvements in productivity and quality are promised via the move to domain-specific modelling languages that allow users to model in concepts relevant to the domain, rather than the possibly sterile notion of packages and classes. The use of meta-modelling techniques allows us to be precise in the specification of the DSL, whilst model transformation provides an adaptable and open modelling architecture, allowing the exchange of models across different DSLs (language profiles) sharing a common metamodelled backbone. Although one might suggest that the unprecedented success of the UML in wide variety of domains has ultimately made it too large and unwieldy, one might view the model-driven technologies described above as realising the aim of the UML as being a family of related modelling languages.

\(^{17}\) For code generation read package spec generation.
2.4 An Overview of the SysML

The SysML was created in order to provide the systems engineering domain with access to a general purpose, model-based approach as had been achieved within the software engineering community via a variety of modelling languages of which the UML is the most notable. The SysML is the result of an initiative by the Model Driven Systems Design workgroup of INCOSE in 2001 to customise the UML for Systems Engineering Applications [OMG07c], the specification of the SysML 1.0 being adopted by the OMG in May 2006. The resulting graphical modelling language is based on UML 2 and is expressed as lightweight profile that both restricts and extends the underlying language (UML 2), see Figure 23.

![Figure 23 – Relationship of SysML to UML](image)

The most notable aspect of SysML involves the nine diagram types of which two are new (i.e. not part of UML), three are modified from the UML, and four are reused. These diagrams provide support for the four main dimensions: structure, behaviour, requirements, and parametrics – see Figure 24 (taken from [FMS08]).
Use of the above diagrams is provided by a number of new and reused model elements, e.g. the Use Case diagrams are unchanged from the UML, depicting actors, use cases and associations. However the ‘new’ model elements are provided in the form of lightweight stereotypes of existing UML 2 model elements, primarily the meta-class Class. This is clearly pragmatic from the perspective of the UML too vendor as it minimises the effort required to make a UML 2 tool SysML aware, however there remain issues regarding the semantics of the model elements since the semantics are exactly as per the stereotyped model element taken from the base language (UML 2). To try to counter this criticism, and to make the SysML model elements appear to have relevance to the domain the concrete (graphical syntax) has been amended. We can illustrate one of the shortcomings of the lightweight approach to the definition of SysML via the requirements diagram. In this context a requirement specifies a capability or condition that must (or should) be satisfied, a requirement may specify a function that a system must perform or a performance condition a system must achieve [OMG07c]. The requirements diagram may be used to render requirements as a graphical, tabular or tree structure. The Requirement model element is provided as a stereotyped class, and may be rendered on diagrams in addition to the requirement diagram to show the requirement’s relation to other elements. The specification of Requirement identifies a need for language support for requirement hierarchies in the form of composites, and also the need to reuse requirements in different contexts. From the UML specification we know that the model element Class has attributes and operations and participates in inheritance hierarchies, allowing multiple inheritance; a class is also able to access the private features of its ancestors (super-classes) [OMG07b]. In order to conceal the obvious mismatch between the semantics of Basic::Class and the description of Requirement, SysML introduces the notion of a slave requirement to allow the same requirement to appear to exist in multiple namespaces - slave
requirements are requirements whose text property is a read only copy of the text property of the master requirement. The concept `SysML::Requirements::Requirement` is defined with two attributes: text and Id, both of type String (see Figure 25). The attribute `text` provides the text relating to the requirement it represents and the attribute `Id` provides an identifier for the requirement (e.g. as might be generated by a requirements management tool such as Telelogic’s DOORS). Clearly, we would like to enforce a rule of the form that all requirement Ids should be unique, however this isn’t possible within the semantics of the underlying base class `Basic::Class`. Furthermore, the SysML element Requirement does not have operations, however its underlying base class does allow operations, hence this restriction must be enforced directly by the tool’s GUI; the tool can’t rely on the underlying language metamodel.

<table>
<thead>
<tr>
<th>«requirement»</th>
<th>RequirementName</th>
</tr>
</thead>
<tbody>
<tr>
<td>text=“The system shall …”</td>
<td>Id=“1234a1”</td>
</tr>
</tbody>
</table>

Figure 25 – SysML Requirement Element

### 2.5 The Case for Strategic Reuse

The in-service life of an embedded avionic system is often very long; a period of up to 20 years is not uncommon. The development and certification of such systems is very costly, and may require many person-years of effort. Pasetti and Pree report that there is a tendency by such projects to develop their software from scratch, whereas in other domains (e.g. desktop applications) Component Based Development (CBD) is often used [PP99]. Pasetti and Pree suggest that a real-time system may often be characterised as a processor and memory with little or no growth capability running an application optimised to support a specific mission and the constraints of the system. The use of specialised hardware also serves to militate against reuse across systems, (e.g.) by enforcing the use of different and sometimes specialised programming languages (e.g. CMS-2M and Coral-66) across projects, there being no suitable common target/cross-compiler. Since no two applications appear to be quite the same, redevelopment, rather than reuse, is often seen as the only solution. Recent years have seen the migration from bespoke vertically integrated systems to horizontally integrated systems based on high performance, market driven COTS products [Lui99]; see Figure 26. Hence, it would appear that the performance issues to which Pasetti and Pree refer are being addressed within this application domain, and the application of CBD technologies to real-time embedded systems is now a practical proposition [Sha00]. This is underpinned by the emergence of a relatively small number of preferred application programming languages within the domain of real-time embedded systems, such as Ada95, C++, and, more recently, Java; it is worth
noting that, although differing in syntax, these languages share many of Meyer’s criteria identifying an object-oriented programming language, such as modularity, encapsulation, inheritance, polymorphism [Mey88].

Key:
- **Product-specific**
- **Family-specific**
- **Domain-specific**
- **Bought-in/3rd party components**

**Figure 26 - The Influence of COTS**

It is known that reuse at the level of specialisation of components and generics does not impact programme costs and tends to cause library scalability issues as features are added [Pen99]. In order to admit the possibility of large-scale reuse at the application level it should be possible to model and develop systems at a more abstract level. Over the years many approaches have been proposed, such as patterns, frameworks, generative programming, and Component-Based Development (CBD); these approaches are discussed in detail by [Hol01a] and [Hol01b]. In contrast to these approaches to accelerating software development, the Software Product Line Architecture (SPLA) shifts the focus to the development of a family of related products. This approach was first popularised by Kang [KCH+90] in his seminal work on Feature-Oriented Domain Analysis (FODA), although the underpinning ideas can be traced back much further to Parnas [Par76] and McIlroy [McI68]. Using the SPLA approach, software systems are modelled in terms of the commonality and variability of prominent or distinctive features (identified via domain analysis), features may also be related in some way such that selection of one feature may necessitate or preclude the selection of one or more other features. Hence, the feature model represents the superset of products that may be fabricated from the product line. The definition of common features and capabilities of systems in the domain serves to leverage large-grained reuse. Whilst FODA introduces a number of key components to support the modelling of features and

---

18 Reused artefacts may extend to specifications, architecture, designs, source code, tests, etc.
their relationships, the semantics of these components are described only informally, being supported only by narrative and examples.

2.6 Software Product Line Architectures

The Software Product Line Architecture (SPLA) and software architecture are different concepts, Clements & Northrop summarise the major difference between the two as being the SPLA’s aim of describing which variations (product configurations) are to be admitted, concluding that ‘identifying the allowable variations is part of the product line architecture’s responsibility, as is providing built-in mechanisms for achieving them’ [CN02]. The authors further define the Product Line Architecture (PLA) as:

- A description of the structural properties for building a group of related systems (that is, a Product Line), typically the components and their relationships. The inherent guidelines about the use of components must capture the means for handling required variability among the systems (sometimes called a reference architecture).

And the Software Product Line (SPL) as:

- A set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

2.6.1 Model-Driven Product Line Architecture

The notion of the domain architecture comprising a number of closely integrated models with defined transformations between components is a view that, it is argued, is absent from the development of the PLA, with the PLA being viewed generally as separate entity outside the domain architecture. Although in their discussion [SV06] Stahl and Völter describe the integration of Product Line Engineering (PLE) into MDSD, the authors view appears to be that the product line is more or less outside the scope of the MDSD, either providing the analysis to guide the definition of the domain architecture, or viewing MDSD as an implementation of the product line; in neither case does there appear to be explicit, model-based linkage between the technologies.

2.6.2 The Feature Model

Feature Modelling provides an approach to the modelling of a family of related systems (a product family) in terms of commonality and variability, such that the feature set allows us to differentiate between members of the family. The features and capabilities common to a class of system are identified by a domain analysis [KCH+90]; this activity should also identify the relationships between features and is crucial to the success of the SPLA. If the features chosen are too general we will not be able to differentiate between individual family members; too specific and each
member will be sufficiently different as to militate against the reuse of any artefacts, degenerating to a standard one-off development.

Feature modelling offers the developer a means to specify custom solutions via reusable assets such that the engineered solution comprises only those features actually required, thereby avoiding the development of baroque components [Ver00] incorporating many unused features, and consuming system resources unnecessarily – this is clearly an attractive trait for the development of embedded applications where systems resources are often at a premium [PP99].

2.6.2.1 What is a Feature?

A feature is defined variously in the literature as:

- A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems. [KCH+90]
- A use case, part of a use case or a responsibility of a use case … any distinguishable characteristic of a component, component system, or application system that customers or reusers can use to select between available options. [JGJ97]
- A property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances. [CE00]
- User-visible aspects or characteristics of a system organised into a tree of And/Or nodes to identify the commonalities and variabilities within the system. [CN02]
- A requirement or characteristic that is provided by one or more members of the product line. [Gom04]

Whilst there is general agreement on the definition of a feature, we shall adopt the relatively broad definition provided by Jacobson [JGJ97] and Czarnecki [CE00] as it is clear that it addresses the needs of both internal and external customers. Hence, a feature shall be defined as:

- An abstract, implementation-independent entity, and the feature model provides a configuration road map that identifies the configuration decisions to be made and the consequences of these decisions in terms of related features.

Czarnecki et al. emphasise that whilst feature modelling is used to capture commonality and variability in systems, it is not appropriate in all cases and should be used model configuration choices; such choices tend to be more static than the runtime data being processed [CHE05]. Hence, whilst an algorithm may contain commonality and variability in the form of its definition (sequence, selection, and iteration), this would not be captured in a feature model. The feature model would be expressed at the level of identifying candidate algorithms for some particular purpose, e.g. a family of algorithms applicable to calculating the ballistic trajectory of some projectile.

The feature diagram is generally represented in the form of a Directed Acyclic Graph (DAG), most commonly a tree structure, although some authors permit orthogonal secondary feature trees.
and some authors provide support for modularity and even recursion via feature references [CHE05]. In this body of work we restrict the structure of the feature diagram to that of a tree, although this could be extended if necessary.

Kang et al. describe the Feature-Oriented Domain Analysis method (FODA) in terms of a number of core modelling concepts: aggregation/decomposition, generalisation/specialisation, and parameterisation [KCH+90]. These concepts are used to support the modelling (and structuring) of a family of systems in the form of feature diagrams. Since their inception, the set of core concepts of feature diagrams has diverged from that expressed in the form a parlance similar to that of a programming language and evolved to cover the following types of feature:

- Optional (may or may not be present in a family member)
- Mandatory (must be present in all family members)
- Parametric (generic components that may be adapted by supplying appropriate parameters)

In [LKL02], Lee et al. describe a Feature-Oriented Reuse Method (FORM) in which the feature model comprises layers indicating the level of abstraction of the relevant feature. The layers of abstraction provided are:

- Capability Layer – *user visible characteristics that can be identified as distinct services*
- Operating Environment Layer – *represents environments in which applications are used*
- Domain Technology Layer – *represents the way of implementing services and operations*
- Implementation Technique Layer – *generic functions or techniques that are used to implement services, operations and domain functions*

The feature diagram described by Lee et al. captures the structural or conceptual relationship between features in terms of three simple relationships:

- Composed-Of – *this is used to model the whole-part relationship between features*
- Generalisation/Specialisation – *used to model the generalisation of sub-features*
- Implemented-By – *used to model the case where a feature is necessary to implement another feature (generally appearing to be used to show feature dependency between layers)*

The FORM approach is supported by A System Analysis and Design Aid tooL (ASADAL). Use of ASADAL is described in [SEL00]; the tool supports the modelling of feature diagrams as trees, although duplicate features are modelled on separate branches of the tree. The notion of parametric parameters does not appear to have survived the transition from FODA [KCH+90]. ASADAL supports the expanding and eliding of branches of the tree, presumably in order to combat the size of a fully enumerated feature diagram; however there is no support for hierarchical structuring, models being rendered on a single diagram with visual complexity being managed simply by the expand/collapse functionality.
The abstract nature of the concept of a feature is generally agreed amongst authors, hence it is reasonable to question that the use of three simple relationships on the feature diagram will be adequate for all possible situations. In particular, use of the Generalisation/Specialisation and Composed-Of relationships may be viewed as introducing design decisions at an arbitrarily early stage in the description of the SPLA; a more suitable semantics should be described that is independent of the typical programming constructs. Hence, it is asserted that the relationships between features are simply that [relationships between features], what this means at the level of the application program components may vary depending on the context and even upon the features supported by individual programming languages.

2.6.2.2 Why Are Features Useful?

As has been described above, features support the modelling of a family of related systems in terms of their commonality and variability of domain stakeholder characteristics such that individual family members may be discerned; feature modelling therefore provides a powerful and abstract concept with which to describe a product family without committing to implementation details. The feature model lends itself well to a graphical syntax to describe the relationship between features in terms of commonality and variability in a clear and succinct form (although large feature models appeal for some form of support for hierarchical structuring); it also shows the dependency between features, however annotation via some kind of constraint language may be required to describe feature composition rules succinctly; such as if the size of an engine is greater than 2 litres then it must be configured with an alternator rated at n Amps. Clearly, it would be possible to refactor the feature model to allow such a configuration constraint to be described via a «requires» relationship, but this risks introduction of additional abstract features to provide the necessary structure. In it’s fully instantiated form, the feature model may be combined with other tools to validate user-defined choices of features when instantiating specific instances of product family members – such approaches have been demonstrated in the form of design wizards, such as Batory’s P3 generator [BCR+00] and constraint-based configuration systems as early as the R1/XCON expert system (a configurator for VAX computers) in 1981, although the latter style of expert system is known to have suffered from maintenance issues [FS99]. Therefore, one can view features and the feature model as the kernel of the SPLA, however it should be reiterated that the feature model is just one component of the SPLA, other components (models) are required to complete the view; however such models aren’t made explicit in the literature.

2.6.2.3 How are Features Related?

Having provided an informal description of a feature, we now summarise the informal descriptions of the kinds of relationships between features that have been described in the literature as necessary
to support the modelling of a family of systems in terms of their discriminating characteristics. The relationship between features is modelled as [KCH+90]:

- **Dependency** (feature A requires feature B in order to fulfil its responsibilities)
- **Mutual Exclusion** (feature A requires that feature B be excluded in order to fulfil its responsibilities)
- **Alternative** (exclusive-OR) (the responsibilities of feature A may be provided by sub-feature C or sub-feature D but not both)
- **Inclusive-OR** (the responsibilities of feature A may be provided by sub-feature C or sub-feature D or both sub-features C & D)
- **Mandatory and Optional features** (features that must or may be incorporated in the configuration of any valid family member)
- **Rationale** (the reason for choosing (or not choosing) feature A over feature B)

Czarnecki and Eisenecker extended the semantics of feature relationships to cover both the inclusive-OR and exclusive-OR (alternative) relationship in [CE00], and, in common with Hein et al. [HSV00] provide support for the notion of multiple sets of related sub-features. Whilst Czarnecki & Eisenecker do not describe their definition of the feature model components via a metamodel (a narrative description is provided in [CE00]), Hein et al. describe the feature model via a lightweight UML profile (Figure 27). However, the profile lacks rigour, leaving some of the concepts represented open to (mis)interpretation, e.g. what is the meaning intended of the association stereotype «appears»?

![Figure 27 – Lightweight Feature Metamodel](image)

Gomaa further refines the semantics of feature relationships in [Gom04] to include:

- **Prerequisites** (equivalent to Dependency above)
- **Mutually Inclusive** (an overloaded form of the Dependency relationship)
In addition to this, Gomaa introduces a classification of related sub-feature sets:

- Mutually Exclusive/Zero-Or-One-Of (semantics as per Alternative)
- Exactly-One-Of (#selected sub-feature sets = 1)
- At-Least-One-Of (#selected sub-feature sets ≥ 1)
- Zero-Or-Many-of (#selected sub-feature sets ≥ 0)

Whilst Gomaa asserts that the extended feature and feature group classification provides access to useful modelling constructs, it would appear to admit no structures that could not be realised via the approach advocated by Czarnecki & Eisenecker; although the resulting structures might be a little more concise; hence these feature groups are considered syntactic sugar and not discussed further.

Features may be selected at various points in the configuration of a system from the feature model; this is defined as the binding time, and (typically) may be a member of the sequence: { generation, compile, load, run }. This is augmented by the constraint that a sub-feature may not have a binding time earlier than that of the super-feature (e.g. we don’t wish to admit the situation where feature A is selected at run time, and feature A₁ has a binding time of run, and its sub-feature A₁₁ has a binding time of compile, implying that the sub-feature A₁₁ must be configured before its super-feature A₁). It should be noted that there is some variation in the binding times supported by various authors.

### 2.6.2.4 A Simple Example of a Feature Model

As an example we will consider the navigation of a pre-planned route by an air vehicle, represented by a feature model within the Mission Management domain [HE03b]. For the sake of this example we are merging the domains of Navigation¹⁹ with Mission Management²⁰. A Navigation Route comprises a number of Route Points and may be utilised by a Navigation Strategy to provide steering commands and/or cues to allow the air vehicle to fly the pre-planned route. Route Points may be either fixed geographical locations (e.g. waypoints, ground-based TACAN stations), or moving objects (e.g. aircraft carriers, tankers, etc.), but it is assumed for the sake of simplicity that a Route cannot comprise both fixed and moving Route Points. The Navigation Strategy feature may be decomposed into the sub-features Direct-To-Point and Course-To-Point, and it may be reasonable for a product to embody either or both sub-features. The Course-To-Point feature requires an Air Vehicle Model feature with a G-Available sub-feature in order to provide realistic steering commands, whereas the Direct-To-Point feature may not require any other features to fulfil its responsibilities (see Figure 28).

---

¹⁹ The Navigation domain is responsible for answering the question: ‘where are we?’
²⁰ The Mission Management domain is responsible for answering the question: ‘where do we want to go?’
The feature diagram for this model is represented below in Figure 29 using the graphical syntax described in [CE00]; the feature diagram would be augmented with additional information (such as more formal descriptions of the features and the associated rationale) in the feature model.
2.7 Current Approaches to Modelling the SPLA

In Section 2.6.2 a description was provided, summarising the most prominent descriptions of the feature model that are used to capture the commonality and variability of the members of the product line. It has been shown that whilst some authors have described a metamodel to underpin the feature model, such metamodels are either informal, or are expressed implicitly in the form of a programming language. The former case implies that the metamodel can’t be used to verify the correctness of a feature model under the control of a tool. Whilst in the latter case an executable semantics will support validation of the models, however the metamodel is expressed in a standard programming language or generated into a tool framework (e.g. EMF), and is thus, less tractable, less abstract, and less open to adoption by alternative technologies. The advantage of specifying the SPLA metamodels in a common modelling language, such as the UML of MOF is that it is more open and extensible, such that it may be combined with other models to provide a more powerful system (such as combining the feature model with a configuration model) whilst enforcing modularity.

In [Gom04] the author describes an informal semantics for features using stereotyped UML classes; this is a lightweight approach to the specification of a features profile as the lightweight stereotypes do not introduce any semantics or constraints over and above that of the metaclass to which they are applied (the metaclass Class). Although the author advocates metaclass modelling, representing the features and feature groups as meta-classes, there is no indication of how this ‘metamodel’ is to be integrated into the source metamodel, e.g. which meta-classes are used to derive the feature meta-classes? Figure 30 provides an illustration of the feature metamodel proposed; this provides little more than a simple taxonomy of the feature related modelling concepts required (a similar metamodel is used to describe the concept of feature groups).

Figure 30 – Lightweight Features Metamodel
A similar metamodel is used to describe the concept of feature groups (Figure 31), feature groups are simply a collection of features that have some relevance in the domain, the semantics of the kind of group is provided only by the illustrative meta-class name.

Figure 31 – Lightweight Feature Group Metamodel

Whilst the feature groups may be given self-explanatory names, there is little a tool could do to enforce the necessary constraints on models expressed using this lightweight profile.

Figure 32 provides an example feature model expressed using this approach (taken from [Gom04]). The feature model describes two alternative hotel feature groups; one for conventional hotels comprising a group of optional features (Automated Cancellation, Automated No Show Billing, and Block Booking), the other for residential hotels comprising another group of optional features (Meal Plan, and Maid Service).

Figure 32 – Example Lightweight Feature Model
Although the above metamodels are lacking in terms of the rigour provided by executable semantics and tool support, this forms only a small part of the work, which is aimed at providing support for the development and evolution of product lines using UML in a pragmatic form, such as might be attempted with a conventional UML tool with no exported (and customisable) metamodel. Griss, et al. take a similar approach to the adoption of a PLA in the telecoms domain using Rational Rose; in this case the lightweight stereotyped classes are rendered as custom graphical icons [GFA98].

Jarzabek and Sevoira argue that the plug-and-play paradigm is inappropriate in circumstances where components are subjected to frequent changes, and appeal for higher levels of flexibility [JS00]. The authors propose a program construction environment based on the generative programming techniques (frame technology) proposed by Bassett [Bas97]. The construction environment is shown in Figure 33.

![Figure 33 – Program Construction Environment](image)

The environment proposed by Jarzabek and Sevoira seeks to separate the generic architecture, the DSSA, from the Customisation Decision Tree (CDT), the feature model. This is in contrast to the template metaprogramming approach of Czarnecki and Eisenecker, where the two have a much closer (implicit) relationship through the generator. However, both approaches seek to differentiate
between the Domain Specific Language (DSL) and the Implementation Components Configuration Language (ICCL), although this separation is more distinct in the frame technology. It is the role of the CDT\textsuperscript{21} to provide the necessary support for variants in the product family and the means by which artefacts in the generic architecture may be specified and customised. It is the role of the customisation tool to perform the adaptation and connection of generic components. The CDT is described as an extension of a feature diagram, and the authors use the feature modelling notation of Czarnecki and Eisenecker [CE00], however no attempt is made to strengthen the associated semantics. Each variation point in the CDT has an associated customisation script that is to operate on the necessary components in the generic architecture. Hence, customisation of components in the generic architecture is automated and may be repeated on demand, thereby removing the burden of the management of multiple versions of components. Backward compatibility may be provided by the annotation of the customisation scripts (frames) with version identifiers. Therefore, whilst the approach of Jarzabek & Sevoira demonstrates a clear separation between the feature model and the target architecture, the approach is not underpinned by a metamodel, and requires scripts to adapt the frame-based SPLA.

Although frame technology is clearly able to support the generation of specific family members from a product line, the technology is essentially a text-based mark-up language that directs operation of a pre-processor. There is no underpinning metamodel, and the feature model is implicit in the structure of the frames themselves, hence there is no clear separation (although this is a deficiency Jarzabek and Sevoira seek to address). Some of the limitations in the text-based concrete syntax of the fame commands have reportedly been addressed more recently by Netron Inc. with the introduction of a graphical toolset to support the design and debugging for frames, however it’s difficult to comment further as no examples are provided on the company’s web-site\textsuperscript{22}.

2.8 Software Factories

In [GS04] Greenfield and Short describe the Software Factories approach as the industrialisation of software development, motivated by the catch-22 situation of widespread use of slow and error-prone manual approaches against increasing demand and decreasing time to market. The aim of the approach is to raise the level of abstraction at which the software developers operate, by bringing them closer to the real-world problems to be solved; in essence by the application of (amongst other things) the DSL. The authors identify a recurring pattern comprising four practices to automate software development tasks:

\[\text{http://www.netron.com}\]

\textsuperscript{21} This is analogous to the DSL described by Czarnecki & Eisenecker in [CE00].

\textsuperscript{22} See http://www.netron.com
- Development of a common framework to support product development in a common architectural style
- Development of an assembly language automated by tools
- Tools to support incremental and rapid builds
- Tools to capture design decisions in an executable form

Although a pattern to support successful evolution has been identified, the authors report that such automation is costly and therefore restricted to businesses with products in the high-volume value-chain specialisation domain. Consequently, the focus of the Software factories is to support narrow vertical domains, such as health care and financial services.

One of the more interesting aspects of the Software Factories approach is that it is pragmatic, and the authors clearly write from experience of real projects. Greenfield and Short are not proposing the adoption of this technology only for new projects, but, rather, accept that, in the case of the definition of the SPLA, the features described in the feature/asset map may be rich and comprise an heterogeneous collection of artefacts. However, there appears to be some loss of traceability between the feature model and the asset interfaces; furthermore there is a lack of a transformational semantics from the feature model to the implementation. The main artefact of the approach is the Software Factory Schema, this is used to categorise and summarise development artefacts in the form of a grid, each cell of which identifies a viewpoint from which some aspect of the software may be constructed (Table 1 – taken from [GS04]).

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Use Cases and Scenarios</th>
<th>Business Entities and Relationships</th>
<th>Business Processes Service Factoring</th>
<th>Service Distribution Quality of Service Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflows</td>
<td>Message Schemas and Document specifications</td>
<td>Service Interactions Service Definitions Object Models</td>
<td>Logical Server Types Service Mappings</td>
<td></td>
</tr>
<tr>
<td>Role Definitions</td>
<td>Database Schemas Data Access Strategy</td>
<td>Detailed Design Technology Dependent Design</td>
<td>Physical Servers Software Installed Network Layout</td>
<td></td>
</tr>
</tbody>
</table>

### Table 1 – Development Artefact Grid

The contents of the cells may be quite diverse, but are generally formally expressed artefacts, such as source code, SQL, and models (expressed in the relevant DSL). Mappings are defined between and within cells to fully or partially automate model transformations; it is for this reason [the lack of a clear semantics] that the authors eschew the UML as a suitable vehicle to support Software
Factories. Hence, the authors describe the software factory schema as a directed graph with the nodes as artefacts and the edges as transformations between artefacts. The implementation of the software factory schema is the software factory template, and it is this that is loaded into the IDE\textsuperscript{23} to provide the tools necessary to support the development of applications of this kind. Although the approach is intended to be generic, Stahl & Völter report that the public perception is that it is a Microsoft-centric approach [SV06].

The Software Factories approach provides traceability between the feature and the product line assets (an heterogeneous collection of artefacts ranging from component specifications to patterns and fragments, sufficient to define the realisation of the feature in the target domain) via the $N \times M$ feature/asset map (Figure 34). Whilst this approach recognises that features may have a very broad scope, and may cross-cut components in the target architecture, there is no explicit modelling of the transformation of semantics from the feature model to the implementation\textsuperscript{24}; it’s also difficult to see traceability from the feature model to the interfaces of the assets (and vice versa) – hence, what confidence do we have that the components identified by the feature composition will work together correctly?

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{feature_product_line_asset_map.png}
\caption{Feature/Product Line Asset Map}
\end{figure}

One can observe that the Software Factories approach is informed by the practicalities of instantiating an SPLA-based approach; e.g. very few projects start from a clean sheet of paper, resulting (presumably) in the need to capture such a diverse set of construction information for each product line asset. However, this may also be viewed as one of the drawbacks with Software Factories; it seems to partially miss the opportunity to define a set of coherent, inter-related models, on a solid (meta-modelled) architecture\textsuperscript{25}.

2.9 Architecture Description Languages

High-level design concepts such as patterns and frameworks are generally abstract, implementation independent descriptions of (part of) a system. Automated correctness checking of functional and

\textsuperscript{23} The Software Factories approach is currently supported by an extension to MS Visual Studio.

\textsuperscript{24} e.g. what does a Variation Point mean to the style in which the product is to be generated?

\textsuperscript{25} This might also go some way to satisfying the authors’ reservations concerning use of the UML and related technologies.
non-functional system properties is clearly desirable and will lead to applications that are more likely to be correct [FSJ99]. In order to express the semantics of architectures unambiguously for the benefit of both practitioners and tools, the Architectural Description Language (ADL) has been devised [SG96]. Unfortunately, there is little agreement in the literature on what is an ADL, and, as a consequence, the ADL exists in a number of forms. Medvidovic & Taylor offer a number of criteria to support the classification and comparison of ADLs [MT97], the primary requirements being support from the ADL for the architectural features: Component, Connector, and Architectural Configuration. Shaw and Garlan argue the need for the treatment of connectors as first class objects and propose six key attributes of an ADL [SG96]:

- Composition: systems composed of components and connectors
- Abstraction: prescription of abstract roles of components
- Reusability: reuse of components, connectors, and patterns (possibly in different contexts)
- Configuration: separation of system structure from component structure, dynamic reconfiguration
- Heterogeneity: composition of heterogeneous components, and architectural descriptions
- Analysis: support for reasoning (both manual and automated)

Medvidovic and Taylor also warn that an ADL may subsume some formal semantic theory (e.g. statecharts), and, as a result, may only be suitable for modelling certain types of system.

2.9.1 The ADL $\Pi$

The approach recommended by Fayad et al. is that of the ADL $\Pi$ which may be used to describe software components [GM98]. In $\Pi$ each component is represented by four sections (see Figure 35):

- Export: data types exported
- Body: component realisation
- Import: data types required (including performance requirements)
- Common Parameters: data types imported and exported unchanged (including performance requirements)

![Figure 35 – Component Sections in $\Pi$](image)

Systems are constructed by matching a clients import section to a suppliers export section (the use relation), and are called configurations and may be composed hierarchically to support larger
systems. The similarity between this structure and that of CORBA is discussed. The component would depend upon an import IDL (Interface Definition Language) module and export an IDL module (which would also contain the necessary factory methods to support object construction), whilst the body would represent the object implementation. Common parameters would appear in both the import and export IDL modules, with the object implementation providing the necessary linkage.

Four *views* are used to specify each section of the component:

- **Type**: component invariants
- **Imperative**: imperative operation signatures and algorithms
- **Concurrency**: specific ordering of execution
- **Interaction**: component distribution

Components are parameterised by their formal imports, and may be considered to be generic (the generic form is referred to as the *Concurrently Executable Module* (CEM) and the instantiated generic referred to as a *component incarnation*). *Path expressions* are constructed from a set of formally defined operators to specify the ordering of execution of component operations (see example based on a simple database – Table 2).

<table>
<thead>
<tr>
<th>Path Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>create; [write_ops</td>
<td>read_ops]</td>
</tr>
<tr>
<td>write_ops := Add</td>
<td>Remove</td>
</tr>
<tr>
<td>read_ops := {Found + Empty}</td>
<td>Query operations Found and Empty may be performed simultaneously in any order</td>
</tr>
</tbody>
</table>

*Table 2 – Example Path Expressions in Π*

The authors use Queuing Specification and Description Language (QSDL) for both functional and performance related evaluation. Performance requirements may be evaluated on a component or system basis, and stochastic measures obtained.

2.9.2 A Generic Ontology

Shaw and Garlan provide a description of the ADLs Aesop, UniCon, and Wright [SG96]. Whilst Aesop is reportedly no longer under development, with interest shifting to Acme [CMU00], it is interesting to note the stylised approach to the representation of architecture based on ‘a generic ontology of seven entities’ [SG96]:
• Components
• Connectors
• Configurations: topologies of components and connectors
• Ports: the component’s interfaces
• Roles: connector’s interfaces
• Representations: the architectural style
• Bindings: the component’s external interface

This is shown graphically in Figure 36.

Aesop supports a variety of architectural styles:
• Pipe-and-Filter Style
• Pipeline Style
• Real-Time Style
• Event-Based Style

The Pipe-and-Filter architectural style may be characterised by the components that take a set of inputs, apply some transformation and stream output to some other component such that output typically commences before all the input has been consumed. Output from a component (filter) is streamed (or piped) to other components via a pipe. This style of architecture supports an asynchronous data flow model (see Figure 37).
Aesop ensures that source roles can only attach to input ports and sink roles can only attach to output ports.

The Pipeline architectural style is a specialisation of the Pipe-and-Filter style with the restriction that connection between filters is in series.

The Real-Time architectural style is characterised by the synchronous and asynchronous inter-task communication and the requirement to satisfy some form of timing constraints. This architectural style comprises three forms of component:

- Devices – provide inputs/outputs to/from the system
- Processes – perform data transformations
- Resources – support for shared resources (access must be synchronous)

The Event-Based architectural style is similar to the Observer pattern described by Gamma et al. in [GHJ+95]; components register their interest in events and can announce events over an event bus without explicit knowledge of their dependants. Shaw and Garlan utilise an event bus connector to broadcast events to interested parties.

In their description of ADLs, Shaw and Garlan go on to argue the need to recognise the difference between implementation and interactions relationships. This approach may be likened to the separation of static and dynamic views of architecture enforced by more recent object-oriented methodologies (e.g. as described by Texel and Williams [TW97a]). Shaw and Garlan note that implementation relationships are usually supported directly by the implementation language (e.g. the with & use of Ada), whereas the interaction relationship requires the use of abstractions (protocols) not directly supported, e.g. the Event-Based or Pipeline style. Shaw and Garlan describe the ADL Wright to illustrate how the ADL supports the architectural styles.

2.9.3 Architectural Fragment Composition

Bosch suggests that, whilst design patterns and frameworks have seen wide-scale adoption across the software engineering industry, their implementation is often difficult due to the semantic gap...
between the pattern/framework and the implementation language [Bos98a]. Bosch states that the problem is due to the inability of programming languages to support an architectural view that focuses only on part of the required behaviour of a group of components. It is further argued that, whilst inheritance and mixin-inheritance go some way to alleviating the problem, there is no support for architectural structures comprising more than one component (although this view is challenged by Smaragdakis & Batory [SB98]). Furthermore, Bosch states that ADLs are based on premise of a dichotomous relationship between architecture and components, and do not readily support the composition of architectural structures. From this the following aims are identified:

- First class, reusable representation of architecture
- Specification of architecture-specific behaviour
- Instantiable (for a family of systems) and configurable (for the peculiarities of a particular family member)
- Specification in a single language

In response to the above aims, Bosch proposes ‘the notion of architectural fragments for specifying object-oriented architectures’, and provides linguistic support in the form of the research language LAYOM (Layered Object Model). The architectural fragment consists of partially implemented ‘roles’ (e.g. describing only the behaviour particular to the component’s part in a design pattern); classes are then composed with the roles (using ‘superimposition’ – the ability to apply predefined, configurable functionality on a component [Bos99]). A role is said to comprise instance variables, methods, acquaintances, layers and an interface, and may override or extend the services of its instantiating application class. In the example provided by Bosch, a process measurement framework contains the role ‘sensor’, this role is played by the application class Camera. However, in order to integrate with an MMI, it is required that the Camera should notify the MMI when some event of interest occurs. This is achieved by composing both the measurement framework and the Observer framework in order to provide a notifying Camera subject that may be observed by the MMI. The ‘partial implementation’ may range from abstract interface methods to a complete component. It is possible to compose different architectural fragments when systems are to be instantiated; Bosch illustrates this with an example of the following form (see Figure 38).
Bosch elaborates on the problems of implementing design patterns in conventional object-oriented programming languages in [Bos98b]. The problems identified are:

- **Traceability** – the design pattern is often lost during the transition from design to implementation
- **The self problem** – delegation of responsibility leads to the reference to the original receiving object being replaced by the delegating object
- **Reusability** – the mixing of domain dependent and independent classes tends to make reuse of the implementation difficult
- **Implementation overhead** – implementation of the design pattern may necessitate the implementation of some trivial methods, this also decreases the understandability of the implementation

Again, the research language L\text{AY}\text{OM} is described (it provides language support for the expression of design patterns), and a number of design patterns are described in terms of ‘layers’ in the language (the language is extensible and is not limited only to the patterns described in the paper). The aim is to be able to support design patterns as first-class entities, such that the design pattern is implemented as an identifiable unit in the code.

The L\text{AY}\text{OM} applies a series of ‘wrappers’ around an object, messages to the object must pass through the interfaces to the wrappers and are evaluated and converted before being passed on to lower layers (if necessary). The layers may also provide a specification of certain relations; three kinds of relation are identified:

- **Structural** (e.g. inheritance)
- **Behavioural** (e.g. binding of behaviour to categories of client)
- **Application domain** (e.g. X controls Y)

Using the above layers, it is possible to define the superclass and set of operations to be inherited. Design patterns are defined as layer types in the L\text{AY}\text{OM} (a number of examples are provided), and
may be composed by the mixing of layer types. Finally, the LAVOM specification is then used to generate C++ that may be used in conjunction with legacy code.

2.9.4 The UML as an ADL

Architecture Description Languages (ADLs) provide an opportunity for the incorporation of formal methods and engineering models into the analysis of software and system architectures [FLV03]. In addition to this a formal description of the software system architecture also provides support for the automated generation of glue code required to interface heterogeneous software components.

Use of the UML as an ADL is highly desirable as a result of the large user base that brings with it tools and processes.

In [SW99], Schürr and Winter argue that the UML has the potential to replace the plethora of ADLs and Module Interconnection Languages (MILs) to provide the foundation for a standard ADL. Although they warn that the state of flux of the constructs and semantics of earlier versions of the UML (pre 1.3) has not helped. Schürr and Winter identify a number of deficiencies of the UML as an ADL against those described by Shaw and Garlan [SG96]; the failure to treat connectors as first class objects and the lack of generality of the UML component diagram. The authors’ focus is on exploring and clarifying the semantics of the UML package concept, particularly with respect to the import mechanism. In UML, packages provide the mechanism for the structuring of systems and provide management of the namespace. Schürr and Winter argue that the revised package import semantics of UML 1.3 are both inconsistent and dangerous (from the perspective that they may warp the software architectural design).

UML 2 introduced the concept of package merge to provide a flexible approach to model extension; the main intent being to provide a mechanism to support the notion of compliance levels within the four layers that comprise the UML metamodel. The package merge concept is intended to ensure compatibility such that if the result of merging the receiving package $P_1$ with the merged package $P_2$ is package $P_3$ then the package merge concept must ensure that any model instance that was valid with respect to $P_1$ is also valid with respect to the resulting package $P_3$.

\[
P_3 = \text{merge}(P_1, P_2)
\]

\[
M_1 = \text{inst}(P_1) \Rightarrow M_1 = \text{inst}(P_3)
\]

The merging algorithm is relatively naive, being based on the correspondence of model elements, if a correspondence is found in the merge package then the model elements are merged, otherwise the model elements are deep copied. Such an approach to model merging based on name correspondences is known to be problematic [PB03].
However, Zito et al. develop a taxonomy covering four possible scenarios for use of package merge and illustrate that whilst the examples in [OMG07a] appear to use package merge in such a way as to ensure compatibility between merged models, there are a number of possible usages that could lead to incompatible models [ZDD06]. Furthermore, the authors highlight that the popular view that the semantics of package merge are “complex and tricky” and only recommended for use in specific circumstances, leading the authors to question the clarity of the concept.

Harkki presents a feature matrix for a number of popular ADLs in [Har04]. The author notes that a number of approaches may be appropriate in order for the UML to support an ADL; however each has its drawbacks:

1. Extension of UML via MOF (heavyweight language profile) – lack of tool support
2. Extension of UML via stereotypes (lightweight language profile) – lack of support for OCL in tools (difficult to enforce the profile)
3. Use of UML ‘as is’ – too loose

In summary, it appears that whilst the ADL was an active area of research until approximately 2003~2004, interest in the integration of an ADL into the UML appears to have peaked with the proposal for a profile for software architecture [FLV03]. Also, problems remain with the language definition features of the UML (package merge) that would be necessary to support integration of the ADL into the UML, even following the most recent incarnation of the language (UML 2.1).

2.9.5 The GenVoca Model

GenVoca represents a synthesis of two earlier approaches to supporting improvements in software productivity via large-grained reuse produced independently in 1988: Genesis (Batory) and Avoca/x-kernel [SWK+95]. GenVoca views a software system as comprising a number of generic virtual machines (called realms); each realm exports an interface that may be used by clients. Each realm provides a number of components (called layers) that implement the interface of the realm; hence each layer in a realm may be substituted by any other (although there may be semantic constraints). It is clear from this that the GenVoca model is more abstract that the typical class-based reuse approach. Product families are represented in the compact (textual) GenVoca grammar of the form shown in Equation 2.

\[
\text{realm} = \text{layer}[, \text{layer}] \\
\text{layer} ::= \text{parameterisedComponent} | \text{atomicComponent} \\
\text{atomicComponent} ::= \text{component} \\
\text{parameterisedComponent} ::= \text{component}[\text{parameter}[,, \text{parameter}]] \\
\text{parameter} ::= x : \text{requiredType}
\]

\textbf{Equation 2 – GenVoca Grammar}
A system may be described from the GenVoca architecture in the form of a type equation identifying the components (layers) required to be composed to form the system; it’s a simple matter to type check the composition to ensure syntactic correctness of the specification.

A metamodel of GenVoca is has been developed in XMF-Mosaic, Figure 39, shows the model elements and their relationships; it should be noted that the relationship between the Layer and its required Realms is modelled as an ordered sequence to allow for the case where a layer takes a number of parameters of the same type.

![Metamodel of GenVoca Model](image)

**Figure 39 – Metamodel of the GenVoca Model**

A simple parser may be written to read a textual description of a GenVoca grammar (model) and instantiate the relevant structure as an instance of the above metamodel. To illustrate this, an instance of the hypothetical avionics architecture supporting the inertial navigation domain (INAV) will be described and instantiated. The system will be described as comprising three realms (INS, GPS, and INAV) as shown by the object diagram at Figure 40.

![Object Diagram](image)

**Figure 40 – Snapshot of GenVoca INAV Model**

The family of realms and components may be expressed in the form of the GenVoca grammar shown in Equation 3.
Equation 3 – GenVoca Model of INAV Domain

The INS realm contains four components; the components \( \text{ins\_type\_1} \) and \( \text{ins\_type\_2} \) represent different kinds of INS equipment, whilst the composite components \( \text{ins\_select} \) and \( \text{ins\_blend} \) are each parameterised by two further components of type INS. The INAV realm comprises three composite components; \( \text{gps} \) and \( \text{ins} \) are simply interface adapters for components of type GPS and INS respectively, e.g. \( \text{gps} \) adapts the interface exported by a component of the layer GPS to the interface INAV. The component \( \text{gps\_ins} \) is parameterised by components of type GPS and INS, e.g. this component might provide clients with a blended GPS/INS solution. Components in the realm INAV could be used by yet more abstract components in the architecture; e.g. see the example reference architecture for avionics in [BCG+95].

The specification of a system from the GenVoca model is performed via a type equation, the type equation identifies the components required and defines the relevant component instances to be provided to satisfy the necessary interface requirements; e.g. the component \( \text{ins\_select} \) must be specified with two sub-components, each conforming to the \( \text{INS} \) realm interface. Again, a metamodel may be defined to capture the notion of a GenVoca system specification (Figure 41).

Figure 41 – Metamodel of the GenVoca System Specification

In this metamodel, a system specification comprises a type equation that may be nested to account for the arbitrary structure of the sub-components required. Constraints may be added to the model to ensure that the specification is well-formed, e.g. ensuring that each layer is composed with interfaces of the required type. A simple parser can be defined to support the textual definition of
GenVoca type equations and populate instances of the metamodel, e.g. Equation 4 provides an example specification of an INAV system.

```
Root::inavSpec :=
@Specification(INAV_1, inavModel)
gps_ins[gps_select[gps_type_1, gps_type_2],
    ins_select[ins_type_1, ins_type_1] ];
end;
```

Equation 4 – GenVoca Specification of an INAV System

The example at Equation 4 creates an instance of a GenVoca specification observing the composition rules defined by inavModel at Equation 3, such that (e.g.) the component gps_ins is configured with a component x of the required type GPS and a component y of the required type INS. This system specification is illustrated by the object diagram shown at Figure 42.

Figure 42 – Snapshot of GenVoca INAV Specification

Note that the specification calls for two instances of the component ins_type_1 (e.g. this may be a dual-redundant equipment, with the ins_select component providing the output of the primary instance).

The GenVoca system specification may be type checked via the constraints within the context of class TypeEquation shown in Equation 5.
Equation 5 – Type Checking of GenVoca Type Equations

Constraint SubExpressionCardinality confirms that there are as many sub-expressions as there are interfaces required by the component to which the type equation relates, and constraint TypeCompatibleComponents confirms that there is a sub-expression containing a component from each of the realms required by the component to which this type equation relates. The combination of these constraints is sufficient to confirm type correctness of each TypeExpression object. In the event that either constraint should fail an error message is reported by the system.

In the following example, a GenVoca system is specified as a single gps_ins component. In contrast to the example at Equation 4, the specification shown below in Equation 6 defines an illegal composition of the component gps_ins by configuring it with two components of type INS (rather than one of type GPS and one of type INS as is required by the GenVoca realm definitions at Equation 3):

```
  Root::inavSpecErr :=
    @Specification(INAV_1, inavModel)
    gps_ins[ins_type_1, ins_type_2];
  end;
```

Equation 6 – GenVoca Type Incompatible INAV System

The type error in the system specification at Equation 6 is identified correctly via evaluation of the model constraints shown in Equation 5 and is reported to the user (Figure 43).
The concrete syntax for the definition of GenVoca models and specifications is defined in BNF; this along with the relevant parsers may be found at Appendix 1.

2.9.6 GenVoca as an ADL

The style of component communication in GenVoca is investigated in [BSC99]. It is argued that component replication occurs when the style of component operations are combined with the communication style (e.g. RPC), and, by treating components and adapters as orthogonal, they may be combined in an arbitrary manner. It can be seen from the realm and type equations shown at Equation 7 that in System component d communicates with component b via the interface S, and the style of this communication is advocated by the interface (S).

\[
S = \{ a, b, c \} \\
T = \{ d[ S ], e[ S ], f[ s ] \} \\
System = d[ b ]
\]

Equation 7 – Example GenVoca Specification

In order to allow the communication style to be modified without affecting components d or b, Batory et al. apply the adapter pattern [GHJ+95] in the form of an intermediate interface G (i.e. an additional realm is introduced), and two new components, s2g[G] (to convert interface S to interface G) and g2s[S] (to convert back again). By combining component d with adapter s2g a new component d’ (exporting interface T and using interface G) may be produced. By combining component b with g2s a new component b’ may be produced (exporting interface G). Hence, it is
possible to adapt the style of communication between the two components to use the G-style interface without affecting either component b or d (illustrated in Equation 8).

\[ S = \{ a, b, c, s2g[ G ] \} \]
\[ T = \{ d[ S ], e[ S ], f[ s ] \} \]
\[ G = \{ g2s[S] \} \]
\[ \text{New\_System} = d[ s2g[ g2s[ b ]]] \]

**Equation 8 – Example GenVoca with Adapted Interface**

![Diagram of GenVoca Component Adaptation]

Such an approach may be appropriate when (e.g.) components d & b are mounted on remote nodes and need to communicate via some network protocol.

### 2.10 Development Tools

The development tools required to support meta-modelling activities resulting in executable metamodels are highly specialised. Whilst XMF-Mosaic\(^{26}\) has been used to develop the metamodels supporting this thesis there are a number of competing tools that may be equally well-suited to such a task:

- XMF is a commercial metamodelling tool has evolved from MMT and XMT. MMT was originally developed as a research tool to support the 2U Consortium proposal for the re-architecting of UML at UML 2 using the metamodeling language MML [2UC02].

---

\(^{26}\) And, in some cases, it’s predecessors MMT & XMT.
MMT spawned XMT and then XMF as commercial tools developed and marketed by Xactium Ltd. and, more recently, Ceteva Ltd. XMF-Mosaic comprises two main components, the execution engine and the Eclipse-based GUI. The execution engine is written (mainly) in XOCL, which is also used as the application level programming language. The core language components exported by the tool are closely related to MOF 2, and may be extended by the user to support both user-models and heavyweight profiles. The tool provides support for the application of user-defined constraints over model instances and provides the user with building blocks from which grammars may be defined to support population of model instances in the form of a concrete syntax. The most recent version of XMF provides an open-source version of the execution engine (XMF v2.2).

- The Eclipse Modeling Framework\textsuperscript{27} began life as an implementation of MOF that has evolved into an Eclipse-based modelling framework and code generation facility for building tools and applications from a structured data model expressed in (e.g.) XMI or Java with specific annotations, from which EMF is able to build the necessary components to support visualisation and editing of the model. In common with XMF-Mosaic, the core metamodel of EMF is also an approximation to MOF. The use (and extension of) the EMF has been reported elsewhere in support of model merging [KPP06], whilst Czarnecki has demonstrated use of an Eclipse plug-in to support feature modelling and template generation.

- Meta Programming System (MPS) is under development by JetBrains and has not yet reached the market in a commercial form. MPS is described as an implementation of LOP [Dmi04], being a meta-modelling IDE with the usual language support features – code completion, refactoring, etc. However, in keeping with this genre of meta-modelling tools it is also open and extensible to support the creation of custom editors, extend and adapt languages, etc. MPS has been ‘in the research phase’ for some years, and it is unclear when (or if) it will be released formally.

- Generic Modeling Environment (GME) was developed by the Institute for Software Integrated Systems (ISIS) of Vanderbilt University, and is another metamodel based tool for the development of domain-specific modelling and program synthesis environments [LMB+01]. The domain-specific language is defined in terms of syntactic, semantic and presentational information, and is expressed using the GME’s own metamodel, which, in this case, isn’t based on MOF; although it contains many of the features one would expect, containment (namespace), references, connections, sets, constraints, etc.

\textsuperscript{27} See http://www.eclipse.org/emf
• MetaEdit+ is a commercial tool marketed by MetaCase and is aimed at supporting full code generation directly from models. MetaEdit+ comprises two main graphically oriented components, a meta-modelling tool (MetaEdit+ Workbench) that is used to develop the domain-specific language and deploy in the form of XML language definition, and a modelling tool (MetaEdit+ Modeller) that developers use to model in the language. The meta-modelling concepts used are described by the acronym GOPRR (Graphs, Objects, Properties, Relationships and Roles) [Met06].

2.11 Conclusions

In 1968 McIlroy identified the need for industrialism in the software engineering domain [McI68]. Recognising the duality of the demand to meet the needs of individual customers and the need to maintain the cost and productivity benefits of mass-production, he called for a software components sub-industry. This goal has been achieved to varying degrees in some domains, e.g. operating systems and math libraries; however the value chain specialisation we see here is not typical of the more context-sensitive domains. The same-as-except paradigm that underppins Frame Technology is novel and extremely powerful. Judicious use of frames may support the application of OO design principles to non-OO implementation languages, or more restrictive language profiles. Furthermore, the ability to suppress redundant features and inject new features provides robustness to the PLA, resisting the need to refactor unnecessarily, and leads to significantly more compact products [PM03]. Frame Technology is reported as providing productivity increases over conventional methods, including OO [Bas97] and [PM03], however it is also reported to incur a steep learning curve [PM03]. Frames may also be difficult to read and the scope of variables difficult to identify without the support of an intelligent editor. Although Frame Technology supports the evolution of product lines and adaptation of product line components; care is needed to define (externally) the composition rules implicit in the Feature Model. Hence, frame technology requires underpinning with a formal description of the component composition rules to validate configurations. For example, cross-links of the following form would not normally be embedded within the frame hierarchy; such constraints belong in the feature model.

• feature X is incompatible with feature Y
• if feature A is selected then feature B must also be selected

Hence, feature modelling and frame technology can be viewed as complimentary technologies supporting a PLA approach; frame technology is discussed in detail in [HE03a].

The SPLA has been introduced and a number of technologies supporting its implementation have been discussed. The main drawback observed in these technologies is the failure to support an

\[28\] This is sometimes referred to as horizontal integration.
holistic view of the SPLA, primarily as a result of the lack of a cleanly separated view of the feature model and its relationship to the artefacts to be transformed into a specific family member. These drawbacks are exacerbated by the failure to build semantically rich tools from a common modelling language, such as the UML and/or MOF; such tools will form an integrated and extensible solution; for example how might one integrate the Netron Fusion toolset for frame development into an MDA programme? The toolset will be built on a proprietary (and possibly implicit) metamodel, and integration with third party tools would (presumably) be via a bespoke API using some open connectivity standard, e.g. COM – because the interfaces aren’t expressed from the basis of a common meta-architecture, and this inhibits the interchange of models between tools. In subsequent chapters we will demonstrate the application of metamodelling to the definition of a collection of models comprising an integrated and flexible view of the SPLA. This approach will be shown to alleviate many of the disadvantages discussed above.
3 SPLA Development Process

A context for this body of work and a motivating case based on an evaluation of the current state of the art has been presented. The SPLA technology that will be discussed in Chapters 4 and 5 forms one side of the iron triangle [KCT95] shown in Figure 45 is the primary area of interest in this body of work. Leadership is required to in order to effect change successfully, e.g. as provided by Ron Temple at Cumins Inc. It is reported by Clements & Northrop that, as a result of Cumins’ commitment to the adoption of the SPLA, they now build more than 75% of all product software from core assets, software quality has improved (as a result of common corrections applied to the core assets), applications have greater functionality, and development timescales are more predictable [CN02].

![Figure 45 – The Iron Triangle of Quality Improvement](image)

Clements and Northrop identify three essential activities that map to the Iron Triangle: Management (leadership), Core Asset Development, and Product Development (process); it is noted that order of core asset development (sometimes called domain engineering) and product development depends on the context of the implementation of the SPLA; products may be developed from core assets, and core assets may be extracted from existing products [CN02] – in this respect the SPLA may be considered to be analogous to a framework, a common approach to the solution of a commonly recurring problem, that of building applications in similar or related domains.

This chapter provides an outline of the development process adopted in the creation of the example models provided to support the description of the Software Product Line Architecture metamodels described in detail in Chapter 5, the intention being to justify the steps taken and provide the reader
with an awareness of how each of the components comprising the family of SPLA metamodels fit together; this also serves to underpin the steps taken in the development of the case studies described in Chapter 6. We also identify the differences one might expect when developing components of the SPLA from scratch (as illustrated in the examples that follow in Chapter 5) versus when developing the SPLA using third party components; the key message being that different approaches are likely to be appropriate in different contexts of use.

3.1 Applying the SPLA Approach

The SPLA is likely to be utilised in at least two separate contexts, (1) to support the establishment of an in-house development over which the developers have complete control, and (2) to support the establishment of an in-house development over which the developers do not have complete control. The contrast between these two approaches is that in the latter context the developers are likely to buy in third party components that must simply be integrated into the product family ‘as-is’, and over which the developers have little or no control – such components will be realised as leaf features (i.e. further extension is not permitted). The identification of components comprising the core asset base may be achieved via a number of routes, the developers may have produced a number of similar products from which core components could be identified and extracted (data mining), or the developers may wish to produce an entirely new architectural framework that is to used to provide the foundation for all new products (forward engineering). Once the SPLA has been developed and becomes the basis for product development a tightly coupled process of interaction between the core asset base and the products being developed occurs with each informing the other of changes and opportunities; hence the SPLA will continue to evolve as it is informed of new core components from products being developed, and these core components become available to all projects based on the SPLA.

3.1.1 Domain Engineering

Domain Engineering encompasses the process of analysing the domain of interest in order to bound the scope of the SPLA to be developed, e.g. one might identify the domain of Navigation for avionics systems. Scoping of the SPLA requires care, as too broad a scope will make the SPLA difficult to develop as it will be difficult to genericise the problem domain adequately resulting in an overly complex core component base; too narrow a scope will reduce the applicability of the SPLA and impact upon its profitability as there will be fewer applicable products across which to amortise development costs and gain leverage from reuse. The domain engineering activity results in the definition of a domain model that characterises problems in the domain and supports the definition of a feature model of the domain. Legacy assets may be mined to identify candidate reusable components with which to bootstrap the SPLA, and subject matter experts will be able to provide guidance on the scoping of the SPLA and construction of the domain engineering artefacts.
Hence the domain engineering activity provides the basis for the development of the SPLA in the form of key features and their relationships. It should be stressed that, at this point, we have not made any commitment to a particular architectural style, such as the adoption of some form of message passing middleware. The domain engineering activity is illustrated in Figure 46.

The feature model is of central importance to the SPLA as it is used to define the generic platform independent architecture (framework) and identify the points at which specific products may provide customisations by way of variation points. The set of core model components are those components that may be used by any configuration of the SPLA; we would expect find a subset of these components present in any product developed from the SPLA in accordance with the semantics of the feature model.

![Figure 46 – Domain Engineering](image-url)
The feature model is layered in nature, as features will exist at various levels of abstraction, typically a feature will fall into one of the four layers identified by Kang et al. [KCH+90] and [LKL02]:

- Capability
- Operating Environment
- Domain Technology
- Implementation Techniques

Layering of features helps to provide an indication of opportunities for reuse, e.g. features in the implementation techniques layer are more likely to be domain-independent, and, hence, may be applicable to other domains. In contrast to this approach, Czarnecki et al. propose a viewpoint based feature analysis process resulting in multiple feature models [CHE05].

3.1.1.1 Use of COTS Products

The general move away from vertically integrated project structures to structures supporting value chain specialisation makes it more likely that the SPLA will include COTS or 3rd party components; e.g. the SPLA may need to support the configuration of products with a 3rd party operating system, middleware technology, persistence, GUI components or some math package. Greenfield and Short refer to this phenomenon as a Software Supply Chain [GS04], each participant in the supply chain consuming resources from an upstream supplier and adding value to the product before providing it to a downstream consumer. Although the use of COTS products may be viewed as accelerating development it also brings with it some challenges, e.g.

- At one end of the spectrum the COTS product may simply be a black box component, with little or no scope for adaptation and/or configuration by the user, e.g. a DLL.
- At the other end of the spectrum the product may be highly configurable, user configuration of the product with an optimum memory/performance footprint may be a challenge and interdependence of configuration options may not be obvious, e.g. configuration of an RTOS with a platform BSP.
- There may be certain restrictions on its use, e.g. up to some specified SIL, or compatibility issues with other COTS components.
- Use of COTS components may impose architectural constraints, e.g. use of the PyDev editor is dependent upon an Eclipse environment and is only available for a Windows platform.
- Lack of control of the development process, e.g. updates to COTS components are unlikely to be synchronised to the product development process.
- Use of COTS is likely to introduce a dependency on the COTS vendor.
3.1.2 Product Development

Product Development is the ultimate objective of the SPLA; this is the activity by which products are developed by configuring the SPLA, specifying the target architecture and generating the product.

3.1.2.1 Feature Configuration

The initial activity to be undertaken as part of product development is the configuration of the feature model; this activity removes all variability from the feature model, resulting in a definition of the features required to be embodied in the end product described in the form of a configuration model. Czarnecki et al. identify the need for staged configuration, admitting the possibility of partial refinements or configurations of the feature model as components are made available; or as decisions are made regarding the configuration options [CHE05]. Staged Configuration seeks to constrain the scope of the systems that may be configured from the feature model resulting in a feature model that is a refinement of the previous feature model; in this respect the approach is strikingly similar to that of the contextual diagrams proposed by Felfernig et al. [FJZ00]. One can appreciate that staged configuration may be appropriate when configuring a large feature model, or when COTS products are selected (or mandated) to provide a specific implementation of (a set of) features, as part of a software supply chain, however Czarnecki et al. also suggest that optimisation and the definition of policy standards provide additional contexts motivating use of staged configurations. We simply note that the Feature Configuration activity may be iterative in order to allow (possibly) repeated refinements of the feature model (see Figure 47), and the use of COTS may impact the development process by constraining the model or even propagating variability as a result of options in the supply chain.
3.1.2.2 System Specification

The relationships between features in the feature model are not intended to convey a structural semantics in the form of generalisation or composition; rather these are used simply to represent variability in the model. Features are linked to components (model elements), and, prior to system generation, a target architecture must be specified, e.g. a layered architecture (see Section 5.2.4.3). The process of system specification involves the binding of a system configuration to an architectural blueprint as illustrated in Figure 48.
3.1.2.3 Product Generation

The final activity in the process of product development is to apply the specification model to the architectural model and generate the product in the required architectural style as illustrated in Figure 49. The resulting product generated is itself another model, this could be turned directly into code as illustrated in the example in Figure 2, or, following an MDA approach, the generated model may be transformed further by one or more PIM to PSM mappings.

Product generation takes the form of constructing the product model by transforming (merging) the model elements identified in a specific configuration with a target architectural style (the architecture blueprint), the product generator contains the knowledge of how each particular model element is to be transformed into the target architecture. The separation of the model components,
which provide the meta-architecture, and the product generator, which provides the component architecture, provides considerable flexibility; the product generator could be as simple as a package to layer mapping, or as complex as providing architectural optimisations. The examples illustrated in the case studies in Chapter 6 will adopt the former, simpler approach.

3.2 Summary

The steps required to define and exploit the SPLA have been described above and are summarised in Figure 51. Whilst the steps are generic, the manner in which they are executed will depend on the context of the development, e.g. integration within a software supply chain versus in-house design and development of the SPLA. Feedback loops will occur where (e.g.) the configuration model provides a requirement for the introduction of a new feature. The feature set will evolve as a result of other subscribing product development programmes, management of the development of the core feature set is known from anecdotal evidence to be a non-trivial task, since development and maintenance of a core (and hence) reusable components is more costly than one-off development [Hol01a]. The construction of an SPLA should only be undertaken following a business case analysis confirming market opportunities for multiple products based on the SPLA, development of an SPLA for a product family of one will not be cost-effective as the SPLA requires a substantial start-up investment [CN02].

3.2.1 Illustration via Case Studies

Chapter 6 describes two case studies based on a Tactical Data Link (TDL) specification (section 6.2) and an avionics navigation system (section 6.3). The case studies are used to inform and validate the SPLA metamodels described in Chapters 4 and 5. The SPLA view of the TDL is without precedent and is based on an analysis of the domain as described by a very large standard in the public domain [DoD04]. The SPLA view of the avionics navigation system is derived from work based on the outputs of the ADAGE project [CTB+96] (also in the public domain) and from the author’s experience in the navigation domain, e.g. as described in [HE03b]. In each case the features identified tend to be at the Capability level, a more detailed analysis of features would be likely to reveal more features in the lower layers, e.g. Implementation Techniques.

The TDL domain is characterised by large, opaque standards expressed, primarily, in narrative form. The focus of the SPLA case study is the domain engineering phase that is required to identify candidate features to bootstrap the feature model, although we also demonstrate the generation of product models from the TDL SPLA. We use the SPLA approach to help to clarify the intent of the standard (which is in excess of 7300 pages of text) and demonstrate the concise view provided. We also demonstrate the configuration of specific family members within the constraints of the SPLA.
This work feeds into the more general model-based approach used to describe the semantics of the TDL domain, reported elsewhere, e.g. [HJR07a].

In contrast, the case study based on the navigation domain uses a more mature set of features identified by the ADAGE project, e.g. [BCG+95] and extended from the author’s experience in this domain. Features are linked to application models such that it is possible to demonstrate system configuration and the generation of such a configuration into a specified architectural style. In this example the domain model is more stable and the product generation capabilities are exercised more fully.

The two case studies provide a complimentary view of the SPLA development process as illustrated in Figure 50.

![Figure 50 – Case Study Process View](image-url)
Figure 51 – Outline of SPLA Development Process
4 Introduction to Modelling of the SPLA

The literature relating to technologies of relevance to the modelling of the SPLA have been reviewed in Chapter 2 to summarise the current state of the art. This chapter summarises deficiencies with the current descriptions of the SPLA and identifies a number of improvements that will be addressed in the sections that follow. This chapter begins with a recap of the evolution of a product line view of a family of software systems leading to the point at which metamodelling emerges as a candidate approach to define the meaning of the SPLA. An informal semantics is attached to the components that will form the metamodels underpinning the SPLA, and an abstract, implementation-independent definition of the metamodels is then presented. Chapter 5 goes on to describe an executable implementation of this definition using the XMF-Mosaic tool.

4.1 Modelling the SPLA

The Software Product Line Architecture has been the subject of academic study for many years; first suggested by McIlroy in 1968 as a parallel to advances made in hardware design, and, subsequently, by Parnas in the form of program families [Par76]. The rationale for the use of the SPLA is predicated on the existence of a number of artefacts (we deliberately avoid the term programs) whose common properties are so extensive that it is advantageous to study the common properties of the artefacts before analysing individual members. The New Technique to which Parnas refers in [Par76] is that of evolving a new family member from a more abstract view of an existing member, and may be seen to be appealing for an approach similar to FODA, with extension being taken from the points of commonality, rather than adapting from the leaf features of an existing system. FODA was introduced by Kang et al. in 1990 [KCH+90] and was first integrated into UML-based projects by Griss et al. in the form of FODAcom [GFA98]. Whilst these works publicised the use of FODA, and feature modelling, there had been no attempt to develop an explicit metamodel to support features. In August 2000 Hein et al. published a UML profile in the form of a collection of related stereotypes to underpin the feature model, the motivation being to evaluate the degree to which the UML could be used; the attractions being the generality of the UML and the absence of suitable feature modelling tools [HSV00]. The authors conclude that the UML lacks support for abstract views required early in the life-cycle, and does not have the expressive power required. Furthermore, the authors cite the inability to create an appropriate rendering of feature models as a result of the failure of flexible view definition on UML models. More recently, other authors have sought to develop more heavyweight integration of the feature model into the UML [GLC05] and [VS06], however these approaches do not expose

29 Paraphrasing Parnas in [Par76].
a fully integrated (meta)model-based solution. Despite this, there is a general focus on the feature model without the necessary holistic view of the SPLA.

This chapter identifies the deficiencies observed in existing approaches to the definition of the feature model and introduces a new family of models expressed using the Z specification language as described by Spivey in [Spi92] such that the abstract specification is independent of a particular implementation technology. We will subsequently describe one particular implementation of this abstract model (see Chapter 5) and provide a more detailed executable specification based on a meta-modelling approach and implemented using the XMF-Mosaic tool.

4.2 Rationale for the Development of the SPLA Metamodels

Feature models were first popularised by Kang et al. as part of the description of the FODA approach to system development [KCH+90], and, even at this early stage, one can see that the objective of the approach is code generation. Although there have been some minor changes to the components comprising a feature metamodel there is general agreement in the literature that features should provide a view of a product family to enable stakeholders to discriminate between individual family members (see section 2.6.2.1). Since Kang’s initial description of the components one might expect to find in a metamodel of the feature model there has been some debate in the literature regarding further extensions, most significantly leading to the introduction of the Alternative [exclusive-OR] feature/sub-feature relationship by Czarnecki & Eisenecker [CE00]. It is also interesting to note that whilst Czarnecki & Eisenecker initially advocated the view that the feature model should be devoid of cardinalities on feature/sub-feature relationships (because feature models should be abstract), in more recent work Czarnecki retracts this view [CBU+02]; presumably as the focus of the work is aimed once again at code generation. The authors hold the view initially that cardinality could be modelled within the existing feature modelling framework via multiplicity features, the authors claim subsequently that this approach only holds when applied to leaf features. In [CBU+02] the authors extend the metamodel of features by introducing feature attributes; the aim here is to try to constrain the size of the feature diagram – e.g. each attribute might otherwise have been modelled as sub-feature.

Czarnecki’s change of heart on the inclusion of cardinality information on feature diagrams is primarily to support feature cloning [CHE05]. Whilst not unique, this serves to illustrate that the feature model is serving a dual purpose; discrimination between different members of a product line, and design of the family members. We assert that the underlying reason for this is the desire to achieve a point from which code generation is achievable as soon as possible – one of the main arguments made by this thesis is that this is neither helpful nor necessary, and that a metamodelled MDA-based approach offers a more satisfactory solution; each model playing just one role in
support of the modelling of the feature hierarchy, identification of a particular configuration, specification of a target architectural style, and, finally, the transformation of the specification into the finished artefact. Such an approach is an advance over previous definitions, offering more scope for customisation to adapt to changing requirements, e.g. evolution of the SPLA in the face of changing target architectures.

One of the main reasons for the overloading of the feature model seen in the literature is a result of the underlying modelling language (e.g. as reported by Hein et al. in [HSV00]). Although more recent developments of the UML have largely addressed the lack of expressiveness of the language, there are comparatively few tools available that are able to take advantage of this development and provide the increased richness to the user. Although now somewhat dated, one can see the lack of adequate tool support manifest in the description of FODAcom as described by Griss et al. in [GFA98], in which the feature modelling components are represented as custom graphical icons attached to common classes exported by the metamodel (m2 layer). Such a lightweight stereotyping of (e.g.) the meta-class Class means simply that the stereotype inherits all the properties of the stereotyped class and, hence, provides no alternative semantics (see Section 2.1). Modelling via such lightweight stereotypes reduces the modelling tool to a simple drawing package, e.g. how is one able to confirm that a feature model is well-formed? Furthermore, the authors do not define an explicit mapping to the semantic domain for the feature model; the feature model exists only as a stereotyped class diagram, hence an object diagram would provide the semantic domain, but how are we to check the correctness of an instance of the feature model, and what do the objects in the instance diagram mean? As an example of lightweight feature modelling, consider Figure 52 taken from [Cla01]; how could a tool ensure that the payment feature is not configured as both Payment_on_Delivery and Credit_Card?
Figure 52 – Example of Lightweight Feature Stereotypes
More recently, Gomaa provides a description of a design method for the development of software systems from the SPLA using the UML [Gom04]. The PLUS method (Product Line UML-based Software engineering) is underpinned with extensions to the UML to allow the modelling of artefacts of the SPLA; however the language extensions are simply a collection of lightweight stereotypes (see Figure 30 to Figure 32). Once again, the components of the feature model do not have a clear underpinning semantics, other than that provided in the form of an informal narrative description and also that of the underlying stereotyped class – unfortunately, no guidance is provided to indicate which meta-classes are to be extended, or how. Although arguably pragmatic insofar as use of the design method is not predicated on anything more advanced than the ability of a tool to support lightweight stereotyping (e.g. Rational Rose would provide an adequately capable platform), there are no executable constraints defined that could be brought to bear on feature models constructed from such descriptions – therefore, once again, how is one able to confirm that a feature model is well-formed, or that a particular configuration is valid with regard to the feature model? Furthermore, how can we be sure that one implementation of Gomaa’s feature model is comparable to another? There is no guarantee that the developers will share a common understanding of the meaning of the feature model component definitions, hence it will be more difficult to bring tools to bear on the SPLA as they have subtly different (implied) meanings.

4.3 What is the Problem to be Addressed?

Software use is pervasive and demand is outstripping supply. Potential solutions to this problem include software reuse and code generation. However, reusable software must be developed with reuse in mind; which is known to be more costly than development for one-off use [Pre97], furthermore the one-size-fits-all approach results in baroque components making inefficient use of machine resources [Ver00], whilst fine grained reusable components don’t scale sufficiently to impact development costs, e.g. [CN02].

The SPLA provides a solution to the software productivity problem by providing product family-oriented view, identifying individual family members in terms of commonality and variability. Whilst the development costs incurred in setting up the SPLA are high and also loaded towards the front of the development, productivity benefits of up to 300% have been reported [CN02]. Code generation techniques such as generative programming also offer a solution to the software productivity problem by supporting the generation of custom components. Code generation techniques have been reported to yield productivity increases in the order of 100% [Bas97].

Clearly, the combination of the SPLA and code generation has the potential to provide a powerful approach to the rapid development of differing members of a family of related software systems. However there are a number of weaknesses observed in the literature with both approaches that
should be addressed if they are to be made open and accessible, and amenable to tool support in
domain-agnostic form:

1. The description of the SPLA must allow us to discriminate between different family
   members easily; however the description of the SPLA in the literature does not have clear
   semantics.
2. The SPLA is sometimes modelled explicitly in the form of a feature model; however the
   definition of features lacks rigour.
3. Feature models provide the dual role of describing commonality and variability, and also
   provide an inherent view of the underlying software architecture; this limits the reusability
   of the feature model by binding it to the architecture unnecessarily.
4. Code generation techniques such as generative programming are language and even
   compiler-dependent, they may also make use of restricted language features.\textsuperscript{30}

The MDA approach advocates a strong separation of concerns via highly focussed models that may
be composed to yield more complex artefacts, hence leverage is gained via reuse of models – and
more highly focussed models are clearly more amenable to reuse. Model-Driven Development
extends the MDA approach, placing even more emphasis on models such that models are
considered to be ‘the code’; the advantage being that the models may relieve us of language-
specific constraints, e.g. the models raise the level of abstraction and may allow us to eliminate
certain language features that may be undesirable in some domains, and emulate powerful code
generation features across different languages. As an example, one could envisage the generic
instantiation being provided at the model level, and then the subsequent transformation of the
instantiated models into C (which doesn’t natively support generics). The weaknesses of the
existing descriptions of the SPLA summarised above will be addressed via an increasingly robust
model-driven definitions in the form of a number of constituent component models with defined
inter-relationships; together these models will describe the SPLA in a more robust and flexible
manner than has been presented in the literature to date. The models are described in Section 4.5
using Z [Spi92], such that they are independent of any particular implementation technology, these
models will then be developed further in the form of an executable specification in Chapter 5.

4.4 \textbf{Introduction to the SPLA Components}

A review of the literature relating to the technologies of most relevance to the definition of the
SPLA has been presented in Chapter 2. It can be seen that the SPLA and feature model are closely
related in the literature, and definitions provided by the various authors gives an insight into the
artefacts that might comprise the SPLA; but, clearly, there’s more to the SPLA than simply a

\textsuperscript{30} Use of Ada generics would not be permitted in a SPARK environment.
feature model, as this says nothing about the implementation of the features (i.e. to what concept the feature relates) or how we should achieve allowable variation required by the description of the SPLA provided by Clements and Northrop [CN02], indeed Bosch views the SPLA as living in a three dimensional problem space, developing along each axis [Bos00]:

1. Software architecture, component and system.
2. Business, organisation, process and technology.

In this section we build upon the work in the literature by proposing a number of component metamodels to describe the SPLA and distil an informal semantics for features that will become the basis of more detailed implementation-independent modelling in section 4.5 and subsequently realised in the form of a collection of executable models in chapter 5.

At its core, one may hypothesise (echoing Clements and Northrop’s view) that the SPLA should provide a number of compatible views such that one may discriminate between different products (members of the product family) and then construct a specific product in accordance with a set of configuration rules, whilst providing support for the evolution of the SPLA as new products are identified and embodied within it. One may follow a Model-Driven approach (such as MDD) to try to provide separation between concerns, with mappings between artefacts being defined to (e.g.) support differing architectural implementations and scalability. It is proposed in this thesis that the SPLA be composed of the following views (models):

1. Feature Model
2. Configuration Model
3. Specification Model
4. Reference Architectural Components Model
5. Product Generator Model

Each of the above, their rationale, and interrelationship is described in the following sections.

4.5 An Abstract Definition of the SPLA

An executable model of the SPLA is presented in Chapter 5, this model is predicated on the availability of a suitable tool (XMF Mosaic is used) and the availability of a few basic building blocks in the form of the meta-classes: Package, Class, Specialisation and Attribute that are available from the MOF [OMG06]. However, in advance of this it is necessary to provide an abstract definition of the metamodel of the SPLA using nothing more complex than predicate calculus and a simple package diagram – package dependency is used to indicate that one package depends on some component within a subordinate package. The reason for the definition of such an abstract model is to maximise the utility of this body of work, making it accessible to a wider
audience, not just the community of XMF users and developers. An outline (package level) view of the architecture of the SPLA metamodel is provided in Figure 4.

An abstract logic-based specification is provided for the components of the SPLA using the formal specification language Z [Spi92]. The specifications described have been developed using the tool CADiZ [TM95] and subjected to type checking. Hence, these specifications are both implementation independent and grounded in an ISO standard language [ISO02].

The description of the contents of each of the packages begins with FeatureModels, the lowest layer of the SPLA metamodel.

4.5.1 Feature Models
The role of the feature model is to provide the mechanism by which we can specify the commonality and variability of the pertinent features of some collection of artefacts comprising all or part of a product family. In section 2.6.2.1 a feature was defined as:

- An abstract, implementation-independent entity, and the feature model provides a configuration road map that identifies the configuration decisions to be made and the consequences of these decisions in terms of related features.

Some features will be defined as being open to extension, whilst others will be closed to prevent further extension, as might be the case if the feature were to relate to some bought-in third party artefact. There are a number of approaches to the implementation of such features, e.g. we may have a meta-class Feature with an attribute defining its extensibility type (e.g. Open or Closed, or we may have separate classes to represent each concept). Features that are open to extension may be extended by the addition of sub-features; hence we see that features are tree-structured such that all non-root features must have a parent (super) feature. The relationship between features is transitive and anti-symmetric – (e.g.) a feature can’t be its own sub-feature. Furthermore the tree structure mandates that there must be only one root feature in any feature model; we do not admit the possibility of cross-links as described by Hein et al. in [HSV00]. The inclusion of a feature in a valid configuration may be optional or mandatory; however a feature’s optionality is only relevant if its super-feature is selected for inclusion in the configuration. Hence, the feature model must support the notion of both Optional and Mandatory features, by default the root feature is considered to be mandatory. Figure 29 provides an example feature diagram expressed using the graphical concrete syntax described by Czarnecki & Eisenecker [CE00].
We now introduce the basic type definitions required for the Feature Model:

The set of all features:

\[ \text{FEATURE} \]

The set of all model elements:

\[ \text{MODELELEMENT} \]

The set of all extension points:

\[ \text{EXTENSIONPOINT} \]

Features are either optional or mandatory; hence we introduce the appropriate free variables:

The optionality type comprises optional and mandatory.

\[ \text{OPTIONALITY} ::= \text{optional} \mid \text{mandatory} \]

Features are either open (to allow subsequent refinement and/or decomposition) or closed (as might be the case for some bought-in third party component):

\[ \begin{align*}
\text{open, closed} & : \mathbb{P} \text{FEATURE} \\
\{\text{open, closed}\} & \text{partition FEATURE}
\end{align*} \]

Divide the set of all features into two types: open and closed. Every feature is either open or closed, but no feature is both open and closed.

A feature may or may not have a super-feature (the only feature without a super-feature is the root); hence we model the relation as a function. Furthermore, the feature/super-feature relation is antisymmetric; the set of all ancestor features can be modelled as the transitive closure of the super-feature relation:

\[ \begin{align*}
\text{superfeature, ancestor} & : \text{FEATURE} \rightarrow \text{FEATURE} \\
\text{ancestor} = \text{superfeature}^+ 
\end{align*} \]

superFeature is a function from a feature to its parent feature, the relation is acyclic (a feature can’t be a super-feature of itself). A feature may have a maximum of one super-feature - hence we model it as a function. The relation ancestor is the transitive closure of the superfeature relation.

Features may be extended via the notion of an extension point, however only features defined as open may be extended, whilst closed features may not. The feature/sub-feature relationship conveys a specific semantics, hence we allow an open feature to be extended via any number of extension points (including none at all).
The extension points provide the link between a feature and its (possibly empty set of) sub-features, we model this via the relation subfeature. The composition of the relations extensionpoint and subfeature may then be used to define the relation child from a feature to its sub-features.

\[
\text{child} = \text{extensionpoint} \; \circ \; \text{subfeature}
\]

the relation child is the composition of the extensionpoint and subfeature relations

Features are extended via extension points, however we wish to apply differing semantics to certain forms of feature/sub-feature relationships; we do this via a partition:

\[
\begin{align*}
\text{variationPoint, alternative, inclusiveOr} & : \mathbb{P} \text{ EXTENSIONPOINT} \\
\text{partition EXTENSIONPOINT}
\end{align*}
\]

Divide the set of extension points into three types: variationPoint, alternative, and inclusiveOr. Every extension point is either a variationPoint, alternative, or inclusiveOr, but no extension point is more than one.

In any feature model some features will always be required to be selected in any valid configuration (if the parent feature is also included). The notion of optional and mandatory features is modelled via the relation optionality:

\[
\text{optionality} : \text{FEATURE} \rightarrow \text{OPTIONALITY}
\]

optionality is a function from a feature to its optionality type, this function allows us to infer if a feature is optional or mandatory.

It is now possible to define the necessary well-formedness rules for the various forms of extension point:

1. The variation point relationship (vp) is the simplest, we wish to be able to attach any form of feature to a variation point.

\[
\text{vp} : (\text{variationPoint} \leftrightarrow \text{EXTENSIONPOINT}) \leftrightarrow \text{FEATURE}
\]

\(\text{vp}\) is a relation from a variation point to a set of features

2. The alternative (alt) and inclusive-or (incor) relationships may only be attached to features of the same type, i.e. either optional or mandatory, but not a combination of both.
alt : (alternative ∈ EXTENSIONPOINT) → FEATURE

∀f₁,f₂ : FEATURE • optinality(f₁) = optinality(f₂)

opt is a relation from an alternative extension point to a set of features with
the constraint that all features mapped have the same optinality type (i.e.
either all optional or all mandatory)

incor : (inclusiveOr ∈ EXTENSIONPOINT) → FEATURE

∀f₁,f₂ : FEATURE • optinality(f₁) = optinality(f₂)

opt is a relation from an alternative extension point to a set of features with
the constraint that all features mapped to have the same optinality type (i.e.
either all optional or all mandatory)

We now define a selection of utility relations:

dependency : FEATURE → ℘ FEATURE

dependency is a relation from a feature to a set of features upon which the
feature depends.

dimension : FEATURE → ℕ

∀f : FEATURE • dimension(f) = # ran extensionpoint

The dimension of a feature is the cardinality of its extension points

allparents : FEATURE → FEATURE

allparents = ancestor

The allparents relation is the set of all the ancestors of a feature up to the root

descendant : FEATURE → FEATURE

This section on the specification of the Feature Model is concluded with the schema defining the
relationships that must hold over the model and also the schema defining the initial state of the

Feature Model:

FeatureModelDB

features : ℘ FEATURE
superfeature : FEATURE → FEATURE
links : FEATURE → MODELELEMENT
dependency : FEATURE → FEATURE

dom links ⊆ features
dom dependency ⊆ features
#features > 0 ⇒ ∃₁ f : FEATURE | f ∈ features •
# ran superfeature = 0

FeatureModelDB schema

(1) the links relation is a subset of the set of features in the feature model,
i.e. features don’t have to be mapped to model elements, but model elements
have to be mapped to by a feature

(2) the dependsOn relation is defined similarly

(3) if the feature model contains any features then one must be the root
### 4.5.2 Configuration Models

The configuration model is used to describe the user’s chosen configuration of the feature set described by the feature model and may be validated against the inherent structural constraints to ensure that the specified configuration is well-formed with respect to the underlying feature model: selection of all mandatory sub-features, selection of only one alternative sub-feature, etc., in this respect the Configuration Model may be viewed as roughly equivalent to the Solution Domain Feature Model as utilised by the Software Factories approach of Greenfield, et al. [GS04]. One can envisage that the configuration model might also be supported by a tool to provide the user with guidance of features based on selection criteria, such as the identification of a parsimonious solution, resonating with Batory’s P3 design wizard [BCR+00]. One might also envisage partially completed configuration models pre-defined to describe a particular sub-range of the SPLA, e.g. an subset of features relating to export versions of some avionics software system; Czarnecki and Eisenecker describe such an approach via the notion of a product information brochure providing a graphical rendering of a car product line [CE99] – it should be noted that such a graphical rendering is simply another concrete syntax for feature models, whilst the main contribution of the example is to illustrate that feature models are used (implicitly) in a variety of domains.

Returning to the example feature model at Figure 29, we may wish to define a Flight Management configuration supporting a Direct-to-Point strategy; feature Direct-to-Point is a sub-feature of Strategy, and this feature requires that the optional feature Route is also included in the configuration. The feature Route has a mandatory sub-feature Route Point, hence this must also be included in the configuration, and feature Route Point has a group of mandatory alternative sub-features, hence we must also choose one of these (in the case of this example, the feature Fixed is selected). Finally, the root feature Flight Management has a mandatory sub-feature Flight Management, so this must also be included in the configuration. The resulting configuration model is illustrated in Figure 53 with all selected features shaded (in blue).
Hence, the role of the configuration model is to define the choices made by the user of the features to be used to define a particular incarnation of the family of products admitted by the feature model. The configuration model provides the vehicle by which we are able to partition the dual role played by the feature model in typical definitions, e.g. [CE00], [Gom04]. The feature model describes what is possible, whilst the configuration model realises one particular system configuration, hence the role of the configuration model is to ensure that the features selected by the user conform to the configuration rules of the various extension points such that it is well-formed with regard to the constraints of the feature model. In this way, the configuration model provides the semantic domain for the feature model.

We introduce the basic type definitions required for the Configuration Model:

The set of all choices.

\[ CHOICE \]

A selection is a relation between a Choice and a Feature, and selection of a Feature requires that all parents of that feature (up to the root) also be included in the set of features selected:
If a feature with a variation point attached is selected then all mandatory sub-features attached to that variation point must also be selected:

VariationPoint

\[
\begin{align*}
\text{feature} : \text{FEATURE} & \leftrightarrow \text{open} \\
\text{vpSelection} : \text{CHOICE} & \leftrightarrow \text{FEATURE} \\
\exists sf : \mathbb{P} \text{FEATURE} \mid \\
\forall f : \text{FEATURE} \mid f \in sf \land \text{optionality}(f) = \text{mandatory} \bullet \\
& \quad sf \subseteq \text{child(feature)} \land \\
& \quad \text{extensionpoint(feature)} \in \text{variationPoint} \bullet \\
& \quad sf \subseteq \text{ran vpSelection}
\end{align*}
\]

all mandatory sub-features attached to a variation point must be included in the set of selected features

By implication selection of optional sub-features of a variation point is optional.

If a feature with an alternative extension point is selected then one sub-feature from the set of mandatory alternatives must be selected:

MandatoryAlternatives

\[
\begin{align*}
\text{feature} : \text{FEATURE} & \leftrightarrow \text{open} \\
\text{mandatoryAlternative} : \text{CHOICE} & \leftrightarrow \text{FEATURE} \\
\exists sf : \mathbb{P} \text{FEATURE} \mid \\
\forall f : \text{FEATURE} \mid f \in sf \land \text{optionality}(f) = \text{mandatory} \bullet \\
& \quad sf \subseteq \text{child(feature)} \land \\
& \quad \text{extensionpoint(feature)} \in \text{alternative} \bullet \\
& \quad sf \subseteq \text{ran mandatoryAlternative} \land \#sf = 1
\end{align*}
\]

exactly one sub-feature from a set of mandatory alternative sub-features must be selected if the parent is selected

If a feature with an alternative extension point is selected then a maximum of one sub-feature from the set of optional alternatives must be selected:
The specification of the Feature Model mandates that the set of sub-features attached to an alternative extension point are homogeneous (i.e. all mandatory or all optional, never a combination of both), see Section 4.5.1.

If a feature with an inclusive-or extension point is selected then at least one sub-feature from the set of mandatory sub-features must be selected:

\[
\exists sf : \mathbb{P} \text{FEATURE} \\
\forall f : \text{FEATURE} \mid f \in sf \land \text{optionality}(f) = \text{mandatory} \\
sf \subseteq \text{child(feature)} \\
\text{extensionpoint(feature)} \in \text{inclusiveOr} \\
sf \subseteq \text{ran mandatoryInclusiveOr} \land \#sf \geq 1
\]

a minimum of one sub-feature from a set of mandatory inclusive-or sub-features must be selected if the parent is selected

If a feature with an inclusive-or extension point is selected then any number of sub-features from the set of optional sub-features must be selected:

\[
\exists sf : \mathbb{P} \text{FEATURE} \\
\forall f : \text{FEATURE} \mid f \in sf \land \text{optionality}(f) = \text{optional} \\
sf \subseteq \text{child(feature)} \\
\text{extensionpoint(feature)} \in \text{inclusiveOr} \\
sf \subseteq \text{ran mandatoryInclusiveOr} \land \#sf \geq 0
\]

any number of sub-features from a set of optional inclusive-or sub-features must be selected if the parent is selected

The specification of the Feature Model mandates that the set of sub-features attached to an inclusive-or extension point are homogeneous (i.e. all mandatory or all optional, never a combination of both), see Section 4.5.1.
Finally, if a feature depends on other features then selection of that feature must result in the selection of the set of dependent features:

\[
\text{Dependent feature : FEATURE} \\
\text{dependent : CHOICE} \leftrightarrow \text{FEATURE} \\
\text{run dependent} = \text{dependency(feature)}
\]

all dependent features attached to a feature must be included in the set of selected features.

The Configuration Model builds on the Feature Model, introducing the selection relation providing the linkage between choices and features, the initial the Configuration Model is empty:

\[
\text{ConfigurationModelDB} \\
\Downarrow \text{FeatureModelDB} \\
\text{choices} : \text{P CHOICE} \\
\text{selection} : \text{CHOICE} \leftrightarrow \text{FEATURE} \\
\text{run selection} \subseteq \text{features} \\
\text{dom selection} \subseteq \text{choices}
\]

ConfigurationModelDB schema
1. the relation selection maps a choice to a feature in the feature model
2. the binary relation selection maps a choice to a feature

\[
\text{InitConfigurationModelDB} \\
\Delta \text{ConfigurationModelDB} \\
\text{choices'} = \emptyset \\
\text{selection'} = \emptyset
\]

the initial configuration model is empty

4.5.3 Reference Architectural Components

The notion of a feature has been described above (section 4.5.1). Clearly, there must be some relationship between the features provided by the product family and the realisation of the concept that the feature represents. However, it is generally the case in the literature that the application architecture is implicit in the feature model, e.g. Czarnecki and Eisenecker demonstrate the implementation of a feature diagram relating to a car, however dependencies between features are derived at an early stage to support implementation using the GenVoca layered architecture [CE99]. Whilst the FORM approach described by Lee et al. in [LKL02] describes the feature model overlaid onto a four-tiered architectural model. Hence, the implementation style is being used to drive the structure of the feature model, although the features are abstract and (ideally) should say nothing themselves about the implementation. Furthermore, the semantics of the feature relationships is related (implicitly) to programming language constructs such as inheritance and association, although this is not entirely surprising, since the aim of the authors is usually to
support software system configuration and generation (possibly augmented by language-specific
technologies, e.g. C++ and template meta-programming). However, Czarnecki and Eisenecker note
that the semantics of the relationships between features depends on the nature of the product to be
generated, such as a Use Case or an executable program; furthermore the authors note that a
number of styles of program could be used to implement a specific set of features [CE00]; however
they don’t go as far as to make the programming style a parameter of the SPLA.

The advent of the OMG’s recent MDA technology and the associated model-driven technologies
spawned from this initiative advocate the use of models as the currency of reuse, and the
partitioning of applications into the PIM and PSM (see 2.3.1). In this thesis a similar approach is
brought to bear on the feature model and the associated application architecture, hence the
relationship between the feature model and the application architecture may be modelled explicitly
and executed by tool-supported technologies such as XMap and Epsilon [KPP06]. This approach
offers a number of advantages, the most significant being:

- Explicit semantics – we describe the meaning of the SPLA in models
- Modularity – the feature model is relieved of the burden of expressing architecture
- Adaptability – the feature model may be composed with different architectural models

Hence, there may be a number of differing architectural models describing various architectural
styles, such as the GenVoca layered architecture, IMS, or DDS. Such an approach may be useful in
order to support reuse of feature models across a family of application architectures, providing both
upgrade paths and also multiple realisations of the family members, e.g. deployment onto multiple
platforms, and also multiple views of family members such as one might wish to provide in the
form of auto-generated product documentation; one view might be the synthesised system, whilst
another might be the associated documentation.

The reference architectural components model is really a placeholder for architectural styles that
may be adopted within the domains of interest, e.g. a layered architecture to support GenVoca or
IMS-type systems, or J2EE to support a distributed multi-tier application. An example in the form
of a layered architecture is described below.

4.5.3.1 An Abstract Definition of an Example Layered Architecture

A layered architectural model comprises (not surprisingly) layers, layers contain components and
each layer may contain other layers (i.e. nesting). Each layer has a controller and layers realise
interfaces. Layers may depend on sub-layers via their interfaces, except for the bottom-most layer
(the foundation layer) that has no dependencies. This is a (brief) description of a layered
architecture of the form found in GenVoca, e.g. [BCG+95], we will leave the notion of a
component undefined at this point – although it will (typically) be realised by the meta-class `Class`.

Firstly, we introduce the basic type definitions for the layered model:

The set of all Architectural models.

\[[ARCHITECTURE]\]

The set of all Layers.

\[[LAYER]\]

The set of all Components.

\[[COMPONENT]\]

The set of all Interfaces.

\[[INTERFACE]\]

The set of all Controllers.

\[[CONTROLLER]\]

There are two forms of layer, a foundation layer and a layer; hence we divide the set of layers into the two distinct groups:

\[
\begin{align*}
\text{foundation, layer} & : \text{FLAYER} \\
\{\text{foundation, layer}\} & \text{partition LAYER}
\end{align*}
\]

Divide the set of all architectural layers into two types: foundation and layer. Every architectural layer is either a layer or a foundation layer, but no architectural layer is both a layer and a foundation layer.

Now we define the basic axioms describing the layering pattern:

\[
\begin{align*}
\text{contains} & : \text{ARCHITECTURE} \leftrightarrow \text{LAYER} \\
\text{sublayer} & : \text{LAYER} \leftrightarrow \text{LAYER}
\end{align*}
\]

A layered architecture contains a set of layers.

Layers may be nested.

\[
\begin{align*}
\text{realises} & : \text{INTERFACE} \leftrightarrow \text{LAYER} \\
\text{dependsOn} & : (\text{LAYER} \leftrightarrow \text{layer}) \leftrightarrow \text{INTERFACE}
\end{align*}
\]

A layer realises a set of interfaces. A layer depends on a set of interfaces. Hence a foundation layer does not depend on a set of interfaces.
Finally, we define the layered architecture model schema:

\[ \text{controls} : \text{CONTROLLER} \rightarrow \text{LAYER} \]

A controller controls a single layer (total function from controller to layer).

\[ \text{controlledBy} : \text{LAYER} \rightarrow \text{CONTROLLER} \]

\[
\forall \text{layer} : \text{LAYER}; \text{controller} : \text{CONTROLLER} \mid
\text{layer} \in \text{dom controlledBy} \land \text{controller} \in \text{ran controlledBy} \impliedby
\text{controls}(\text{controller}) = \text{layer}
\]

A layer is controlled by a single controller (total function from layer to controller).

Finally, we define the layered architecture model schema:

\[ \text{ArchitectureModelDB} \]

\[ \text{layers} : \mathbb{P} \text{LAYER} \]
\[ \text{interfaces} : \mathbb{P} \text{INTERFACE} \]
\[ \text{controllers} : \mathbb{P} \text{CONTROLLER} \]
\[ \text{components} : \mathbb{P} \text{COMPONENT} \]
\[ \text{architecture} : \text{ARCHITECTURE} \leftrightarrow \text{LAYER} \]

\[ \text{ran architecture} = \text{layers} \]
\[ \text{ran controls} = \text{layers} \]
\[ \text{ran dependsOn} = \text{interfaces} \]
\[ \text{ran realises} = \text{layers} \]
\[ \text{dom realises} = \text{interfaces} \]
\[ \text{dom classes} = \text{layers} \]
\[ \text{ran classes} = \text{components} \]

\[ \text{InitArchitectureDB} \]
\[ \Delta \text{ArchitectureModelDB} \]

\[ \text{layers}' = \emptyset \]
\[ \text{interfaces}' = \emptyset \]
\[ \text{controllers}' = \emptyset \]
\[ \text{architecture}' = \emptyset \]

4.5.4 Specification Models

In the description of the feature model (above) it was stated that the definition of the relationship between features in terms of programming constructs is considered to be misplaced, for example specific programming constructs (such as inheritance) are sometimes prohibited when
implementing high-integrity systems, and a restricted language profile may be mandated, e.g. the Ravenscar Profile [Bur99] or the SPARKAda subset [Bar03]. But why should this be an implicit concern for the feature model? One might introduce a set of features to describe target constraints, such as the target language profile required to satisfy a particular SIL, or one might delegate this decision to a separate model; we see this as a more flexible approach because the feature model is not polluted with implementation details and the realisation of the feature relationships is made explicit, thereby catering for the varying degrees of abstraction one might find in the feature model; the FORM approach suggests that each feature will fall into one of four levels of abstraction [LKL02], although other approaches such as GenVoca provides a user-defined layered model in the form of any number of realms (see Section 2.9.5). The specification model makes the relationship between the components in the configuration model and the target architecture explicit, and provides a customisation point such that we could substitute target architectural styles without recourse to amending the underlying configuration or feature models; this design approach resonates well with the intentions of MDA [OMG03].

The specification model is very simple, it merely relates a configuration model to an architectural model that is to be used (subsequently) to realise the configuration within a given architectural style.

A specification model contains a configuration model and a reference architectural style (model):

\[
\begin{align*}
\text{SpecificationModelDB} \\
\Xi \text{ConfigurationModelDB} \\
\Xi \text{ArchitectureModelDB}
\end{align*}
\]

4.5.5 Product Generators

Having modelled the features in the product line, identified the relevant choices for a particular configuration and specified the architectural style to be applied the final step in the process is to define the transformation rules required to generate the resulting model; this process is modelled via the Product Generators component. Each Product Generator (there may be many in the SPLA) defines the relevant transformations rules to map each selected feature onto the specified architecture. The Product Generator contains the explicit knowledge of (e.g.) the traversal rules to identify relevant components and also information required to identify the target location of the transformation (i.e. where in the resulting architecture a component should be placed).

The product generator contains the specification model and applies the necessary transformations required to generate the specified product in the style of the specified architectural model, e.g. generation of an Earth Model component in the style of a layered architecture. The product
A product generator simply contains a specification model and defines the projections to bind the
selected features to the appropriate architectural component.

\[
P_{\text{ProductGeneratorModelDB}}
\begin{align*}
\Xi & \text{ConfigurationModelDB} \\
\Xi & \text{SpecificationModelDB} \\
\Xi & \text{ArchitectureModelDB} \\
\text{components} : & \text{P COMPONENT} \\
\text{projection} : & \text{FEATURE} \leftrightarrow \text{COMPONENT} \\
\text{dom projection} & = \text{ran selection} \\
\text{ran projection} & = \text{components}
\end{align*}
\]

The generation of a model from a specification is provided by navigating to the model element and
transforming it into the appropriate component as defined by the transformation rules. The
transformation rules will depend upon the architectural style being applied and are specified below
in general terms only:

\[
\text{GenerateModel}
\begin{array}{l}
\Delta \text{ProductGeneratorModelDB} \\
\text{transformation} : \text{MODELELEMENT} \leftrightarrow \text{COMPONENT}
\end{array}
\quad
\forall f : \text{FEATURE};
\quad c : \text{COMPONENT} \\
f \in \text{dom projection} \land c \in \text{ran projection} \land
\quad \text{links}[f] \in \text{dom transformation} \land
\quad c \in \text{ran transformation}
\]

A model is generated by transforming MODELELEMENT objects into COMPONENT objects.

We can now describe the transformation operation as simply:

\[
\text{DoGenerateModel}
\begin{array}{l}
\text{GenerateModel} \land \text{Success} \lor \text{Fail}
\end{array}
\]

The complete specification is provided in Appendix 2.

4.5.6 Conclusions

Whilst some approaches to the definition of the SPLA are clear about the need for various views of
the artefacts that it may comprise (e.g. [Bos00]), and some authors even provide predefined
architectural styles or frameworks for the implementation\(^{31}\), there does not appear to be an
approach leveraging from the model-driven paradigms that have emerged over recent years to

\(^{31}\) [http://gp.uwaterloo.ca/files/OOPSLAdemo/OOPSLAdemo-part1.htm](http://gp.uwaterloo.ca/files/OOPSLAdemo/OOPSLAdemo-part1.htm) (e.g.)
support the definition of a model of the SPLA, describing the artefacts and their relationships clearly.

The set of models described above bridges the gap between the feature model and the resulting product via clearly defined relationships and model transformations expressed in the implementation-independent language Z [Spi92]. Not only does this approach make the resulting metamodels underpinning the description of the SPLA more open to implementation in a variety of modelling (or even programming) languages, it also affords more clarity to the description of the component comprising the SPLA and, ultimately, the SPLA itself. The models provide considerable flexibility to adapt the semantics to suit product requirements, thereby offering additional support for the evolution of the product line. The relationship between the models described above is shown (informally) in Figure 4, an implementation of which will be described in Chapter 5 using the tool XMF-Mosaic.

The composition of the models illustrated in Figure 4 results in the generation of a product from a feature model with all notion of static variability removed\(^{32}\), expressed within the constraints of the reference target architecture (Reference Architectural Components).

### 4.5.6.1 Connecting Features to Reality

It can be seen from the above description of the feature models described in the literature and reviewed in section 2, that there is one vital omission – the features in the feature model must relate to some concept with the domain being modelled, such that the collection of features described by the configuration model relates to a (set of) component(s) that may be composed and/or transformed into (one or more artefacts to support the generation of) the end product – one approach was outlined using a frame technology-based metamodel in [HE03a]. Traceability from features to domain concepts is provided via the Feature/Product Line Asset Map in the Software Factories approach [GS04], whilst FODA provides implicit traceability via separate feature and domain models [KCH+90]. In the case illustrated by Czarnecki & Eisenecker in [CE99] and [CE00], the components in the feature model are the domain concepts with the corresponding limitation that the semantics of the relationships between features are restricted to programming language constructs: Variation Point \(\Rightarrow\) association, and Inclusive/Exclusive-Or \(\Rightarrow\) inheritance. In contrast, the models described in this chapter make the relationship from feature to domain concept explicit and flexible such that the semantics of the feature diagram are decoupled from the dual role of providing both discrimination between different members of the product line and design of the family members.

---

\(^{32}\) Dynamic variability may still exist in the form of run-time dispatching.
5 Supporting the Development of the SPLA

Chapter 4 described an abstract specification of the components comprising the SPLA in an implementation-independent form via the ISO standard formal specification language Z [ISO02]. This chapter describes an executable implementation of the SPLA specification using a metamodelling approach following the style advocated by the 2U Consortium [2UC02], and more recently [CSW08]. The implementation presented in this section provides a description of the components comprising the SPLA: features, their permissible relationships, the creation of configurations and specifications, and, ultimately, the generation of products (models). The metamodelling approach adopted defines the SPLA models primarily in terms of an abstract syntax and a semantics; a textual concrete syntax for the models is also described as necessary (the language grammars for which may be found in the Appendices), and mappings between models are also implemented and described. The component models of the SPLA described below are implemented using the metamodelling tool XMF-Mosaic via the XOCL language, example executions demonstrate both correct operation of the models and also resilience in the face of erroneous data via a collection of constraints over the models. Chapter 6 presents two case studies that have been used to inform and validate the implementation of the SPLA models described below.

Whilst the component models of the SPLA are described and implemented in XMF-Mosaic using XOCL, the specifications are sufficiently open and accessible that they should be amenable to implementation in any MOF-aware language.

5.1 Building a Metamodel for Feature Models

An informal description of a Feature and a supporting semantics has been derived from the work of other authors and summarised elsewhere in this document. However, this informal description is unnecessarily clouded by the lack of a clear distinction between the concrete syntax and the abstract syntax, and a desire that the feature model should support code generation (see, for example, the ASADAL tool that supports the FORM approach [SEL00], or Czarnecki’s feature modelling plug-in for Eclipse). This section of the thesis seeks to tease apart the concrete syntax, abstract syntax and semantics such that the problem becomes amenable to re-definition using a commercial meta-modelling technology, resulting in a well-defined and machine checkable definition. Such a definition is essential if we are to reduce (and hopefully eliminate) ambiguity from the specification; the approach of having such a well-defined and machine checkable definition is also of vital importance in other (meta) modelling domains, e.g. UML/SysML. The metamodel underpinning the Feature Model is presented in the form of a series of views, each

33 http://itcentre.tvu.ac.uk/~clark/XMF/
34 http://gsd.uwaterloo.ca/projects/fmp-plugin/
illustrating a particular component of the metamodel, this approach makes the supporting
descriptions more accessible; the complete metamodel is then presented at the end of this section
before a demonstration of the error checking capabilities that may be applied to instances of the
model.

The informal definition of a feature that has been adopted is due to the work of Czarnecki &
Eisenecker [CE00] and Jacobson, et al. [JGJ97]; that a feature represents some domain concept that
may be used to discriminate between concept instances. From this it is inferred that features do not
necessarily exhibit a one-to-one relationship with application classes, hence it is concluded that a
feature simply represents some domain concept by having an explicit relationship with zero (or
more) components in the domain model. Furthermore, some features may be refined, whilst
others are leaves in the feature tree and may not be refined. This leads to the identification of the
core of the Features metamodel, the FeatureModel contains a collection of Features, a
Feature relates to a set of components in the domain model and a feature may be represented by
either the of the concepts Feature or ExtensibleFeature, depending upon whether the
feature is to be open or closed to subsequent extension (Figure 54). There is one root feature in any
feature model, all other features are related to their parent feature via the association named
superFeature; clearly the root feature has no parent feature and hence the collection (set) of
super-features is empty. Whilst the name superFeature may seem unnatural, given that feature
models are generally expressed as trees, the name is used to avoid confusion with the predefined
attribute parents on the metaclass Classifier.

Figure 54 – Core Concepts of the Feature Model

The concept represented by the class Feature is that of a leaf feature, i.e. a feature that is closed
to subsequent extension, therefore it is an error for any feature to have a parent (superFeature)
that is of type Feature. We introduce the operation isLeaf on class Feature (this operation

35 Metaclass XCore::Element is the root of the hierarchy of classes in the metamodel and thus may be used to
represent any class (since every class will be a kind of Element)
always returns true), and override the operation in class ExtensibleFeature such that it always returns false; we can then define the constraint ExtensionOfLeaf as illustrated in Equation 9.

context Feature
    self.superFeature() ->forall(f | not f.isLeaf())

Equation 9 – Leaf Features may not be Extended

Since the hierarchy of features in the feature model is tree structured, it is reasonable to enforce the following constraints:

There must be one and only one root feature in the feature model:

context FeatureModel
    self.features ->select(f | f.superFeature() ->isEmpty) ->size = 1

Equation 10 – Feature Model has One Root

A feature may have at most one parent super-feature:

context Feature
    self.superFeature() ->size <= 1

Equation 11 – Feature has at most One Parent

Features are transitive and antisymmetric:

context Feature
    self.isAntisymmetric(Set{})

Equation 12 – Features are Anti-symmetric

Where the helper operation isAntiSymmetric is defined in Equation 13.

@Operation isAntiSymmetric(children : Set(Feature)) : Boolean
    if children ->collect(child | child.name) ->includes(self.name)
    then false
    else if self.isRoot()
        then true
    else let parent = self.superFeature -> sel
        in parent.isAntiSymmetric(children -> including(self))
    end
end
end

Equation 13 – Anti-symmetric Helper Operation

The components to which a feature is associated may have certain constraints relating to their selection for any valid configuration, for example feature A may depend on some other feature B in
some way, such that if we select feature A then we must also select feature B. In this example, features A and B are located in different parts of the feature model, i.e. features A and B do not share a common path to the root feature; this relationship is modelled via the class Dependency. In the example shown in Figure 55, feature $f_2$ depends on feature $f_6$; hence any valid configuration containing feature $f_2$ must also include feature $f_6$.

![Figure 55 – Feature Dependency](image)

Furthermore, the selection of certain features may be optional or mandatory, and there may be specified binding times associated with individual features such that a child feature cannot be specified with a binding time earlier than that of its parent. Figure 56 provides an illustration of the components provided to support feature composition constraints.

![Figure 56 – Feature Dependencies and Composition Constraints](image)
It can be seen that not all of the rules mentioned above relate directly to the context of the feature model, clearly some may be applied as well-formedness rules for the feature model, however some relate to the validation of feature compositions, and that aspect is now deferred to a later model (the Configuration model).

Within the context of the components contained within the feature model we can assert that no feature should have a binding time earlier than its parent:

\[
\text{context Feature}
\text{self.superFeature()}
\text{->forAll(parent |}
\text{self.bindinTimeEqualToOrLaterThan(parent))}
\]

**Equation 14 – Binding Time is Equal to or Later Than Super-Feature**

Where the ordering relation `bindingTimeEqualToOrLaterThan` on class `Feature` is defined as illustrated in Figure 57.

![Figure 57 – Binding Time Ordering](image_url)

The implied sequence of the enumeration literals of `BindingTimeType` is enforced via the helper operation `bindingTimeEqualToOrLaterThan`, see Equation 15.

\[
\text{context Feature}
@Operation bindingTimeEqualToOrLaterThan(f : Feature) : Boolean
\text{let seqOfEnumLiterals = FeatureModels::BindingTimeType.name}
\text{in seqOfEnumLiterals->}
\text{indexOf(self.bindinTime().name.toString() >=}
\text{seqOfEnumLiterals->}
\text{indexOf(f.bindinTime().name.toString())}
\]

**Equation 15 – Ordering Relation on Feature BindingTimeType**

A feature’s selection type is defined to be either optional or mandatory simply by enforcing that it is bound to an object of type `SelectionType`, where `SelectionType` provides the two enumeration literals: Mandatory and Optional.

\[
\text{context Feature}
\text{self.selection()<>null}
\]

**Equation 16 – Feature Selection is Defined**
A feature’s binding time is defined to be one of the set of enumeration literals Compile, Generate, Link, Run; as provided by the enumeration type BindingTimeType. Therefore, we must ensure that each feature object is bound to a BindingTimeType object via the slot bindingTime.

```
context Feature
    self.bindingTime() <> null
```

**Equation 17 – Feature Binding Time is Defined**

The final dimension of the feature metamodel is the requirement that it must support the various forms of feature/sub-feature relationship that are necessary to support the partitioning of the sub-features as disjoint subsets facilitating the modelling of a family of systems in terms of their discriminating characteristics. The feature/sub-feature relationships are used to constrain the choices that may be made when defining a specific configuration of the feature model. Drawing from the list of feature relationships described informally elsewhere in this thesis, the following core set of requirements has been identified:

- **Dependency** – feature A requires feature B (provided via class Dependency as discussed above)
- **Alternative** – the responsibilities of feature A may be provided by either sub-feature B or C, but not both
- **Inclusive-Or** – the responsibilities of feature A may be provided by either sub-feature B or C or both sub-features B & C
- **Optionality** – we need to be able to specify that some sub-features must be selected, and that others may be if desired (provided via the enumeration SelectionType)

In addition to the informal semantics described elsewhere in this thesis, Czarnecki & Eisenecker demonstrate in [CE00] that it is useful to be able to cluster specific sets of sub-features beneath a parent feature, such that a feature may have multiple dimensions (this aspect is not restricted to the clustering of alternative sub-features). From this it may be inferred that the necessary well-formedness rules may be achieved via the addition of an appropriately typed Node relating the feature to the respective sub-features (see Figure 58).
Clearly, only ExtensibleFeature objects should be allowed to possess Node objects, and, as has been noted above, a number of types of Node are required to meet the modelling requirements; and dangling nodes are not allowed.

context Node
not(self.subFeatures->isEmpty)

Equation 18 – Dangling Node Constraint

Nodes of type VariationPoint are the simplest and provide the ability for a feature to be described by a set of sub-features, the selection of which depends only upon their individual optionality (i.e. there are no additional well-formedness rules to be applied). The VariationPoint node provides a modelling abstraction to support feature relationships expressed in the form of a meronym\(^{36}\). An example of a VariationPoint is shown using the graphical syntax of Czarnecki & Eisenecker in Figure 59; this example depicts the case where a Car feature must be selected with a Body, Gearbox, and Engine features, and may also be selected with an optional Tow Bar feature.

\(^{36}\) A meronym denotes the part-of relationship, e.g. Gearbox is part-of Car.
The equivalent Abstract Syntax is shown in Figure 60. From this it can be seen that the Car feature contains a single extension point (named Meronym) that binds it to the relevant sub-features, and also that the sub-features retain links back to the parent (super-feature). Also shown are the links from the features to the singleton optionality enumeration components.

Figure 60 – Example of a Variation Point (Abstract Syntax)

The semantics of the Alternative node class is self-evident; any valid configuration must comprise one and only one sub-feature from the set of alternatives. However, there are some considerations required of the well-formedness rules that one might wish to apply to the feature model. In [CE00], Czarnecki and Eisenecker present a number of normalization rules to transform
feature models with non-intuitive semantics into simpler feature models, such as the transformation of a feature with one (or more) optional alternative sub-features into a feature with all optional alternative sub-features, e.g. see Figure 61.

The example of feature normalisation shown in Figure 61 requires some consideration of what is actually being presented by the model. It appears to say (in the first instance) that feature A has a set of alternative sub-features \{B, C, D\}, of which selection of B is optional and selection of C & D mandatory. The semantics of the alternative relationship are such that one and only one sub-feature from the set of alternatives must be selected; therefore the notion of optionality takes on a different semantics, and two forms of Alternative feature/sub-feature relationship emerges; optional alternatives, and mandatory alternatives. We would wish to admit the following situations as valid instances of a feature model:

- A feature **may** be extended by selection of any one of the set of alternative sub-features (optional alternatives – see Figure 62)

![Figure 62 – Optional Alternative Sub-Features](image)

Valid configurations:
\{A\}, \{A, B\}, \{A, C\}, \{A, D\}

Optional alternatives

And:

- A feature **must** be extended by selection of any one of the set of alternative sub-features (mandatory alternatives – see Figure 63)
Therefore, any combination of optional and mandatory sub-features under the same Alternative node has ambiguous semantics and is an error case.

Hence, it is concluded that (in this instance at least), the feature normalisation activity is being proposed because the semantics of the feature relationships are not defined sufficiently rigorously to support execution/checking by a tool. Well-formedness is enforced by the constraint shown in Equation 19, ensuring all sub-features are homogenous (either all Optional or all Mandatory) by checking that for each sub-feature, all siblings are of the same type.

\[
\text{context Homogeneous}
\]  
\[
\text{self.subFeatures->forAll(sf1 | self.subFeatures->excluding(sf1)->forAll(sf2 | sf1.selection = sf2.selection))}
\]

Equation 19 – Well-Formedness of Alternative Features

The final component provided to support the required relationships between features and sub-features is the InclusiveOr. The notion of the InclusiveOr relationship between features was identified independently by both Czarnecki & Eisenecker, and Griss et al., although the rationale behind each was subtly different [CHE05]. It can be seen that the well-formedness rules of this component are identical to that the Alternative node; it may be bound to any number of optional or mandatory features, but the features must be of the same type (either all mandatory or all optional).

The semantics of the InclusiveOr relation are described (somewhat loosely) by Czarnecki & Eisenecker in [CE00] as:

1. \textit{If the parent of a set of or-features is included in the description of a concept instance then any non-empty subset from the set of or-features is included in the description; otherwise none are included.}

This definition appears to reduce to allowing any set of the sub-features to be included, which would be equivalent to the semantics of the VariationPoint. However, it is clear from the
examples and also the normalisation rules that this is not what was intended. In the example fragment shown in Figure 64 we see an Engine feature that has an inclusive-or relation with two sub-features, Petrol and Electric, the intention here is to admit the possibility of choosing any of the following engine combinations: \(\text{\{Petrol | Electric | Hybrid\}}\), where Hybrid is a combination of both Petrol and Electric engine features.

![Figure 64 – Example Inclusive-Or Feature Relationship](image)

The obvious interpretation of this diagram fragment would be to infer that the mandatory sub-features Petrol and Electric should be present in any valid configuration, but, clearly, this isn’t what’s intended! The (somewhat counter intuitive) normalisation rules help to clarify the original intent of this relationship, allowing the inclusive-or relationship to be reduced to a variation point under certain conditions (see Figure 65).

![Figure 65 – Example Normalization of Feature Models (Inclusive-Or)](image)

In this example of feature normalisation, Czarnecki & Eisenecker assert that a feature with one (or more) optional (inclusive) inclusive-or sub-features may be normalised into a feature with all optional sub-features; the transformation of mandatory or-features to optional features inferring a curious semantics to the or-feature relationship.
Despite the peculiarities of the semantics inferred by Czarnecki & Eisenecker, the intention is that the inclusive-or relationship is only valid when all of the sub-features are of the same type (i.e. all optional or all mandatory); hence the semantics are similar to those of the Alternative relationship, hence both the Alternative and Inclusive-Or relationships derive from the abstract parent class Homogeneous. The InclusiveOr relationship is analogous to the hyponym of linguistics, denoting the kind-of relation. For this reason, the InclusiveOr relation is provided, but with semantics equivalent to the Alternative; both of which are inherited from their abstract parent class (Homogeneous).

Each of the major components of the metamodel of the FeatureModel has been discussed above, hence we now combine each into a completed view of the model – see Figure 66.

**Figure 66 – A Metamodel for Features**

5.1.1 Example Feature Model

A textual concrete syntax for feature models has been developed (the listing may be found at Appendix 2); this greatly simplifies the task of creating example feature models. A well-formed example feature model is presented in Figure 67 based on an example due to Czarnecki & Eisenecker in [CE99] and [CE00].

---

37 For example, Coupe, Saloon, Estate are each a kind-of Car Body
<imports elided>
Root::carFM :=
@FeatureModel(CarFeatureModel)
   Root Feature(Car) has a {Body, Gearbox, Engine, TowBar};
   Optional Feature(TowBar) is leaf;
   Mandatory Feature(Body) is one of {Coupe | Estate | Saloon};
   Mandatory Feature(Coupe) is leaf;
   Mandatory Feature(Estate) is leaf;
   Mandatory Feature(Saloon) is leaf;
   Mandatory Feature(Gearbox) is one of {Automatic | Manual};
   Mandatory Feature(Automatic) is leaf;
   Mandatory Feature(Manual) is leaf;
   Mandatory Feature(Engine) is
      any combination of {Petrol | Electric};
   Mandatory Feature(Petrol) is leaf;
   Mandatory Feature(Electric) is leaf;
   Feature Coupe requires {Manual, Petrol};
end;

Figure 67 – Example Feature Model (Textual Concrete Syntax)

This example is presented in Figure 68 in the form of the graphical concrete syntax described in [CE00].
Figure 68 – Example Feature Model (Graphical Concrete Syntax)
Each of the features in the feature model may be related to zero or more model element from a source model; and it is this model (the domain model) that provides the set of components from which we wish to configure a specific family member. It is also this model that defines the relationships between the components of the PLA, rather than the approach taken by Czarnecki & Eisenecker where it is the relationship between features in the feature model that implies the relationship between components. The structure of the source model used in the example of the car PLA is shown in the car domain model (Figure 69).

![Car Domain Model](image)

**Figure 69 – Car Domain Model**

The components comprising the Car Body are provided as illustrated in Figure 70. A car body is represented by the abstract root class `Body`, with each of the three body styles identified in the feature model being represented by a concrete subclass (`Coupe`, `Estate`, and `Saloon`). Car body accessories are provided in the child package `Bodies::Accessories`, and this comprises the abstract root class `Accessory` with a single subclass `TowBar`, and an `Accessory` has an association relation with a `Body`. 

133
In section 5.4 we will demonstrate the usage of these model elements as a reusable kit of parts in conjunction with the specification model to generate a custom model of a Car.

5.1.2 Demonstration of Error Checking of Feature Models

In this section a number of erroneous feature models are described; instances are created and then checked for validity by the metamodel implementation in the XMF-Mosaic tool. Where possible, examples are expressed using the concrete syntax for the sake of brevity, however this precludes the evaluation of some erroneous situations as the parser will not allow the creation of a feature model with certain errors (e.g. the grammar of the concrete syntax does not support the explicit specification of binding time, defaulting all features to having a binding time of CompileTime (see Appendix 2)). Hence, we can see that the grammar of the concrete syntax can prevent some errors reaching the model, being caught by the parser; if we were describing instances of the feature model in the (less succinct) abstract syntax, then this is not the case. In the sub-sections that follow we will demonstrate error checking of the feature model via models expressed in concrete syntax, supported by further models expressed in abstract syntax to illustrate error cases that would not be admitted by the concrete syntax.

5.1.2.1 Violation of Anti-Symmetry Constraint

In this example we define a feature model for a PLA of Cars, however the optional feature TowBar is extended to include a number of types of tow bar, one of which (LowLoading) includes an optional sub-feature called TowBar, introducing a circularity into the feature model (see Figure 71).
The above feature model is parsed into the model and then checked against the set of constraints on the Feature Model, resulting in the error report in Figure 72 (exported from the tool in the form of an HTML document). The error report identifies that the optional LowLoading sub-feature TowBar is not unique within this part of the tree, failing the constraint AntiSymmetric and thereby invalidating the feature model.

Figure 72 – Results of Constraint Check (Anti-Symmetry Violation)
5.1.2.2 Mal-Formed Alternative Sub-Features

The set of alternative sub-features attached to a feature must be of the same type, i.e. either all optional or all mandatory (see Equation 19). The example feature model constructed by the description shown in Figure 73 creates a feature Body and then (erroneously) binds to it a collection of both optional and mandatory sub-features.

```<package includes elided>
Root::carFMError1 :=
@FeatureModel(CarFeatureModel)
  Root Feature(Car) has a {Body, Gearbox, Engine, TowBar};
  Optional Feature(TowBar) is leaf;
  Mandatory Feature(Body) is one of {Coupe | Estate | Saloon};
  Optional Feature(Coupe) is leaf;
  Optional Feature(Estate) is leaf;
  Mandatory Feature(Saloon) is leaf;
  Mandatory Feature(Gearbox) is one of {Automatic | Manual};
<unchanged>
end;
```

Figure 73 – Feature Model with Mal-Formed Alternative Sub-Features

The above feature model is parsed into the model and then checked against the set of constraints on the Feature Model, resulting in the error report in Figure 74 (exported from the tool in the form of an HTML document); this identifies that the set of alternative sub-features {Saloon, Estate, Coupe} do not form an homogenous group, failing the constraint HomogeneousSubFeatures and thereby invalidating the feature model.
It should be noted in the example above that the name ‘Hypernym’ assigned to the Node of type Alternative is created by default as part of the parsing of the textual concrete syntax.

5.1.2.3 Mal-Formed Feature

The metamodel for the feature model identifies that a Feature (or sub-classes) must have an associated binding time and selection, each of these elements are enumerations and, therefore, have a defined set of permissible enumeration literal values. The XMF type system prevents us from declaring undefined enumeration literal values, and also associating invalid instance types in the slots; hence we need only check that the relevant slot is not null, as provided via Equation 16 & Equation 17. The parser for the feature modelling language concrete syntax forces the declaration of the feature selection type (optional or mandatory), and sets the binding time for all features to Compile, hence the only way to create a feature with no defined binding time and/or selection type is via the abstract syntax of the form illustrated in Figure 75.
@Operation createErroneousFeatures() : Set(Feature)
let features = Set{
  ExtensibleFeature
  [name = "Car",
   //selection = SelectionType::Mandatory,
   //bindingTime = BindingTimeType::Compile,
   extensionPoints = Set{
     VariationPoint [name = "Car System Components",
                    subFeatureNames = Set("Tow Bar",
                                             "Body",
                                             "Gearbox",
                                             "Engine")]]},
  <further declarations>}

Figure 75 – Erroneous Feature Declaration

Declaring an instance of the erroneous feature Car allows us to confirm that the constraints over
the feature model are able to identify the error(s) correctly; the constraint report fragment is shown
in Figure 76 and Figure 77.

![Constraint Report](image)

**Constraint SelectionIsDefined**

**Candidate Car::ExtensibleFeature**

- extensionPoints: Set(Car System Components: VariationPoint)
- components: Set()
- superFeature: Set()
- selection: null
- bindingTime: null
- name: Car

**Invariant**

self.selection() <> null

**Reason**

Feature: Car does not have a selection defined.

Figure 76 – Constraint Violation (SelectionIsDefined)
5.1.2.4 Violation of Single Parent Constraint

The relationship between a feature and its parent (the super-feature) is modelled via the attribute `superFeature` and is expressed as a set relationship (Figure 78). We wish to admit the possibility of a feature having no parent to allow for the root feature; however we wish to ensure that all other features have one and only one parent feature. We could have chosen to model the `superFeature` relation as a single object of the type `Feature`, such that the root feature would have a null attribute, however the normal style of modelling association relations with a cardinality of 0..1 is via a set, hence we need a to enforce this with cardinality constraint over such relations (Equation 11).

The grammar of the textual concrete syntax for describing feature models prevents us from instantiating a feature with multiple parents; hence the constraint must be checked via the instantiation of an error case in the abstract syntax (Figure 79).
@Operation makeFeatureWithTooManyParents() : Feature
let p1 = Feature(name = "Parent 1",
    components = Set{},
    selection = SelectionType::Mandatory,
    bindingTime = BindingTimeType::Compile,
    superFeature = Set{});

p2 = Feature(name = "Parent 2",
    components = Set{},
    selection = SelectionType::Mandatory,
    bindingTime = BindingTimeType::Compile,
    superFeature = Set{});

in let child = Feature(name = "Child",
    components = Set{},
    selection = SelectionType::Optional,
    bindingTime = BindingTimeType::Compile,
    superFeature = Set(p1, p2))
in child
end
end

Figure 79 – Erroneous Feature Constructor Function

This constructor function creates the erroneous feature instance shown in the snapshot at Figure 80.
It can be seen that the feature Child has two parent features (Parent 1 and Parent 2) identified via the relationship superFeature, and our constraint check should identify such relationships as erroneous.

Figure 80 – Feature with Multiple Parents

The result of the evaluation of the constraints over the instance identifies correctly that the cardinality constraint of the feature/parent relationship (superFeature) has been exceeded, and results in the error report in Figure 81.
5.1.2.5 Extension of Leaf Feature

Under some circumstances we may wish to prevent a feature from being decomposed further, e.g. the feature may relate to some bought-in item that must be used as-is and not adapted/configured in any way. Such features are said to be closed to adaptation and are represented in the metamodel of features by the class Feature; features open to adaptation are represented by the class ExtensibleFeature (see Figure 58); hence the relationship extensionPoints to class Node. The grammar of the feature description language prevents us from describing a feature as an extension to a leaf feature; hence we illustrate the situation via the abstract syntax described in section 5.1.2.4 (above). In this example we can see that feature Child extends both the parent features Parent 1 and Parent 2, both of which are instances of the class Feature and are therefore closed to adaptation. The result of the evaluation of the constraints over the instance identifies correctly that the features: Parent 1 and Parent 2 are leaf features and should not be extended by feature Child (Figure 82).
5.2 Building a Metamodel for Configuration Models

Having described a feature model for some domain, it is possible to apply the constraints described above in Section 5.1 to check that the feature model is well-formed, e.g. the feature model has one root feature, no leaf features are extended etc. However, the semantics of the feature model can only be checked in the semantic domain of the model, in this case the semantic domain of the feature model is provided by the Configuration Model (Figure 83).

Figure 83 – Semantic Domain for Features
The Configuration Model is little more than a container of choices, i.e. those features presented in the feature model that we wish to use to configure some specific family member from the space of all family members admitted by the Feature Model. The structure of the Configuration Model is shown in Figure 84. It should be noted that the set of choices identified by the Configuration Model should be fully enumerated, i.e. we do not admit the possibility of a configuration with implicit feature choices. However, this does not preclude some pre-processing stage to enumerate the configuration from a partial set of choices, e.g. one could simply identify the outer-most features required and have the system infer the fully enumerated set of choices.

![Diagram of Configuration Model](image.png)

**Figure 84 – Configuration Model**

The Configuration Model contains a reference to a collection of the selected features, and a reference to the source Feature Model, from which the choices have been derived. Clearly, an initial constraint on the Configuration Model is that all choices must be contained within the same source Feature Model (Equation 20).

```plaintext
context ConfigurationModel
self.choices->forAll(choice |
   self.featureModel.features->includes(choice))
```

**Equation 20 – Enforcing Feature Selection in the Configuration Model**

The well-formedness rules of the feature model ensure that the feature/sub-feature relationships are valid, e.g. all homogeneous nodes are connected to features with the appropriate selection type (optional or mandatory). However, it not until a configuration of features is identified that we can check the validity of the set of chosen features against the semantics of the feature model, e.g. that one feature from a set of mandatory alternative features is chosen. The constraints applied to the Configuration Model may be partitioned into two categories, constraints relating to the feature/sub-feature relationships (vertical view), and constraints relating to side-effects such as feature dependencies (horizontal view).
5.2.1 Constraints on Feature/Sub-Feature Relationships

The constraints to be applied to the feature/sub-feature relationship relate to the incorporation of all relevant features into the configuration as necessitated by the correct use of the concrete Node subclasses: VariationPoint, Inclusive-Or, and Alternative. The initial constraint is that for each selected feature, all ancestor features from the selected feature to the root must also be selected (Equation 21).

\[
\text{context ConfigurationModel}
\]

\[
\text{let parents = self.choices->collect(choice | choice.allParents())->flatten}
\]

\[
\text{in parents->forAll(parent | self.choices->includes(parent))}
\]

\[
\text{end}
\]

Equation 21 – Selection of All Super-Features

Hence, the example configuration illustrated in Figure 85 would be invalid because feature \( f_1 \) is not chosen in the configuration whilst its sub-feature \( f_2 \) has been chosen, i.e. selection of feature \( f_2 \) requires that its parent feature \( f_1 \) also be selected for the configuration to be valid against this constraint.

As stated above, the semantics of the VariationPoint node is such that all mandatory sub-features should be included in the set of choices (Equation 22), hence in addition to the example configuration at Figure 85 being invalid as a result of the failure to include the parent feature of \( f_2 \).
in the configuration, it would also be declared invalid because feature $f_1$ is a mandatory sub-feature of $f_0$, related via an extension point of type VariationPoint.

```swift
context ConfigurationModel
let extensibleFeatures = self.choices->select(choice | choice.isKindOf(ExtensibleFeature))
in let vps = extensibleFeatures->collect(ef | ef.extensionPoints->select(ep | ep.isKindOf(VariationPoint)))->flatten()
in let mandatorySubFeatures = vps->collect(vp | vp.getMandatorySubFeatures())->flatten()
in mandatorySubFeatures->forAll(msf | self.choices->includes(msf))
end
end
end
```

**Equation 22 – VariationPoint Sub-Feature Selection**

Whilst a little verbose, the above constraint simply identifies the set of Variation Points relevant to the set of chosen features and then checks that all mandatory sub-features are included in the set of selected features.

The semantics of the Alternative (Exclusive-Or) feature/sub-feature relationship are such that there should only ever be a maximum of one chosen sub-feature. The well-formedness rules of the feature model ensure that the set of sub-features attached to an extension point of type Alternative are homogeneous; either all optional or all mandatory, never a combination of both (Equation 19). If the set of sub-features are optional then the configuration may include a minimum of zero and a maximum of one sub-feature in the set of choices (Figure 62). However, if the set of sub-features are mandatory then the configuration must include exactly one sub-feature in the set of choices (Figure 63). The constraints required for extension points of type Alternative are described for optional alternative sub-features in Equation 23 and for mandatory alternative sub-features in Equation 24.
context ConfigurationModel
let extensibleFeatures = self.choices->select(choice |
    choice.isKindOf(ExtensibleFeature))
in let alts = extensibleFeatures->collect(ef |
    ef.extensionPoints->select(ep |
        ep.isKindOf(Alternative)))->flatten()
in alts->forAll(alt |
    not alt.getOptionalSubFeatures()->isEmpty implies
    (alt.getOptionalSubFeatures()->|
        intersection(self.choices))->size <=1)
end
end

Equation 23 – Optional Alternative Sub-Feature Selection

The constraint for configurations including mandatory alternative sub-features is:
context ConfigurationModel
let extensibleFeatures = self.choices->select(choice |
    choice.isKindOf(ExtensibleFeature))
in let alts = extensibleFeatures->collect(ef |
    ef.extensionPoints->select(ep |
        ep.isKindOf(Alternative)))->flatten()
in alts->forAll(alt |
    not alt.getMandatorySubFeatures()->isEmpty implies
    (alt.getMandatorySubFeatures()->|
        intersection(self.choices))->size = 1)
end
end

Equation 24 – Mandatory Alternative Sub-Features Selection

Both constraints work in the same manner. We identify the set of Alternative nodes relevant to the
features chosen for inclusion into the configuration and then check that for each set the number of
sub-features selected is within the required cardinality constraints applicable to the selection type
of the set of sub-features.

The semantics of the Inclusive-Or feature/sub-feature relationship are partitioned in to those for the
Optional Inclusive-Or relation and the Mandatory Inclusive-Or relation. In the case of the
Mandatory Inclusive-Or relation, at least one member of the set of possible sub-features must be
included in the set of selected features (Equation 25).
context ConfigurationModel
let extensibleFeatures = self.choices->select(choice |
  choice.isKindOf(ExtensibleFeature))
in let incOrs = extensibleFeatures->collect(ef |
  ef.extensionPoints->select(ep |
    ep.isKindOf(InclusiveOr)))->flatten()
in incOrs->collect(incOr | incOr.getMandatorySubFeatures())
  ->select(fs | not fs->isEmpty)
  ->forAll(fs | fs->intersection(self.choices)->size >= 1)
end
end

Equation 25 – Mandatory Inclusive-Or Sub-Feature Selection

In the case of the Optional Inclusive-Or relation, zero or more members of the set of possible sub-
features must be included in the set of selected features (Equation 26).

context ConfigurationModel
let extensibleFeatures = self.choices->select(choice |
  choice.isKindOf(ExtensibleFeature))
in let incOrs = extensibleFeatures->collect(ef |
  ef.extensionPoints->select(ep |
    ep.isKindOf(InclusiveOr)))->flatten()
in incOrs->collect(incOr | incOr.getMandatorySubFeatures())
  ->select(fs | not fs->isEmpty)
  ->forAll(fs | fs->intersection(self.choices)->size >= 0)
end
end

Equation 26 – Optional Inclusive-Or Sub-Feature Selection

5.2.2 Constraints on Side-Effects

The selection of a particular feature from the feature model must satisfy the semantics of the
relationship to its parent (assuming it’s not the root); however in addition to this the feature may be
dependent upon some other feature elsewhere in the tree of features in the Feature Model. In this
eventuality we must identify all such dependencies and confirm that all dependent features are also
in the configuration. Equation 27 describes the constraint required to check for the inclusion of all
dependent features.
context ConfigurationModel
let ds = self.featureModel.dependencies->select(d | self.choices->includes(d.source))
in ds->collect(d | d.target)->flatten->forAll(f | self.choices->includes(f))
end

Equation 27 – Identification of Dependent Features

This constraint identifies the set of all dependency objects from the feature model having a source feature in the set of choices included in the configuration and then confirms that the target feature of all such dependencies is also contained in the set of choices included in the configuration.

5.2.3 Example Configuration Model

A textual concrete syntax for configuration models has been developed (the listing may be found at Appendix 4); again, this simplifies the task of creating example configuration models. An example configuration for a sport car variant derived from the feature model in Figure 67 is presented textually in Figure 86.

parserImport Root::ProductFamilies::ConfigurationModels;
import Root::ProductFamilies::ConfigurationModels::ConfigurationModel;
import Root::ProductFamilies::FeatureModels;
Root::sportsCarCM :=
  @ConfigurationModel(sportsCarConfigurationModel)
  From carFM select {Car, Body, Coupe, Gearbox, Manual, Engine, Petrol};
end;

Figure 86 – Example Configuration Model (Textual Concrete Syntax)

The features chosen for this configuration are shown as shaded in the graphical view of the feature model in Figure 88, and the results returned by the constraint checks confirm that this is a valid instance of a Configuration Model (Figure 87). Subsequent examples will not include a screen shot of such valid constraint checks and will instead simply state that all constraints have been satisfied.

Figure 87 – Results of Constraint Check (Configuration Model OK)
Features selected for the example configuration shaded in blue.
5.2.4 Demonstration of Error Checking of Configuration Models

As shown in Figure 83 the Configuration Model provides the semantic domain for the Feature Model, hence we are now able to provide semantic checks on Configuration Models, based on the semantics of the components of the Feature Metamodel. In this section a number of erroneous configuration models are described\(^38\) and then checked for validity by the metamodel implementation in the XMF-Mosaic tool. Example configurations are described using the concrete syntax where possible; synthetic (abstract syntax) models are used to illustrate certain error cases that are precluded by the grammar of the configuration model concrete syntax.

5.2.4.1 Violation of Dependent Features Constraint

In this example we configure a Sports Car from the feature model of Cars described by Figure 67. The feature model for the Car PLA specifies that the Body sub-feature Coupe requires both the Engine sub-feature Petrol and the Gearbox sub-feature Manual, however we configure a Sports Car substituting the Gearbox sub-feature Automatic for the required feature Manual and generate the resulting configuration model (the configuration is shown in Figure 89).

```
<imports elided>
Root::sportsCarCMError1 :=
@ConfigurationModel(sportsCarConfigurationModel)
  // Coupe requires Manual Gearbox
  From carFM select {Car,
    Body, Coupe,
    Gearbox, Automatic,
    Engine, Petrol};
end;
```

**Figure 89 – Configuration Model with Dependent Feature Violation**

The above configuration model is parsed into the model and checked against the set of constraints, resulting in the error report in Figure 90. The error report identifies the set of choices selected for the configuration and states that the configuration model does not include all dependent features, identifying that the dependent feature Manual has not been selected in the configuration (the car feature model illustrated in Figure 68 states that a car with a Coupe style Body requires a Gearbox feature of type Manual).

\(^38\) Erroneous with respect to the underlying Feature Model.
5.2.4.2 Violation of Mandatory Alternative Feature Selection

In this example we configure a Saloon Car from the feature model of Cars described by Figure 67. The feature model for the Car PLA specifies that the feature Gearbox has two mandatory alternative sub-features (Automatic & Manual); however we neglect to specify either of the mandatory alternative Gearbox sub-features. The well-formedness rules of the Alternative feature relation are such that all sub-features must be of the same type (either all Mandatory or all Optional, never a combination of both), and the semantics of the Alternative feature relation falls into two forms as described in Equation 23 and Equation 24; in this example we are exercising the constraint testing the cardinality of the set of selected mandatory alternative sub-features. The configuration description is shown in Figure 91.

Figure 90 – Results of Constraint Check (Dependent Features Violation)
The above configuration model is parsed into the model and checked against the set of constraints, resulting in the MandatoryAlternativeSubFeatureSelectionCardinality constraint failing. This constraint fails because the model does not exhibit the expected cardinality of the set of selected mandatory alternative (exclusive-or) sub-features selected, neither of the Gearbox sub-features (Manual or Automatic) was selected.

We now configure a Saloon Car from the feature model with a Gearbox comprising both the Manual and Automatic mandatory alternative sub-features (Figure 92). Clearly, this configuration is in error as it violates the semantics of the mandatory alternative feature/sub-feature relationship.

The above configuration model is parsed into the model and checked against the set of constraints, resulting in the MandatoryAlternativeSubFeatureSelectionCardinality constraint failing. The constraint fails again because the model does not exhibit the expected cardinality of the set of mandatory alternative (exclusive-or) sub-features selected; this time, however, the constraint fails because we have selected more than one mandatory alternative sub-feature of Gearbox, where one (and only one) of the mandatory alternatives is required.
5.2.4.3 Violation of Mandatory Variation Point Feature Selection

In this example we configure a Saloon Car from the feature model of Cars described by Figure 67. The feature model for the Car PLA specifies that the root feature Car has three mandatory sub-features (Body, Gearbox, and Engine) and one optional sub-feature (Tow Bar) related via a variation point. We define a Saloon Car configuration neglecting to select the Engine feature (and any sub-feature thereof). The semantics of the VariationPoint feature relation is described by Equation 22 and states that all mandatory sub-features of a variation point must be included in a valid configuration model. The configuration description of the erroneous Saloon Car is shown in Figure 93).

```xml
<imports elided>
Root::saloonCarCMError3 :=
(ConfigurationModel(saloonCarConfigurationModel)
   // Car requires selection of all mandatory VP features
   // we exclude the mandatory VP feature Engine
   From carFM select {Car,
       Body, Saloon,
       Gearbox, Automatic};
end;
```

*Figure 93 – Configuration Model with Mandatory Variation Point Feature Violation*

The above configuration model is parsed into the model and checked against the set of constraints, resulting in the AllReachableMandatoryVariationPointFeaturesSelected constraint failing. The constraint fails because the model does not include all mandatory sub-features reachable from the root feature Car; in this instance the mandatory sub-feature Engine has been omitted from the configuration. Selection of the mandatory feature Engine would then expose a further error as this feature has a set of mandatory inclusive-or sub-features, at least one of which would then need to be selected.

5.2.4.4 Violation of Mandatory Inclusive-Or Features

In this example we configure a Saloon Car from the feature model of Cars described by Figure 67. The feature model for the Car PLA specifies that the feature Engine has two mandatory alternative sub-features (Petrol & Electric); however we neglect to specify either of the mandatory inclusive-or Engine sub-features. The well-formedness rules of the InclusiveOr feature relation are such that all sub-features must be of the same type (either all Mandatory or all Optional, never a combination of both), and the semantics of the InclusiveOr feature relation falls into two forms as described in Equation 25 and Equation 26; in this example we are exercising
the constraint testing the cardinality of the set of selected mandatory inclusive-or sub-features (Equation 25). The configuration description is shown in Figure 94.

```gala
Root::saloonCarCMError4 :=
@ConfigurationModel(saloonCarConfigurationModel)
   // Engine requires selection of at least one sub-feature
   From carFM select {Car,
       Body, Saloon,
       Gearbox, Manual,
       Engine,
       TowBar};
end;
```

**Figure 94 – Configuration Model with Mandatory Inclusive-Or Feature Violation**

The above configuration model is parsed into the model and checked against the set of constraints, resulting in the MandatoryInclusiveOrSubFeatureSelectionCardinality constraint failing. The constraint fails because the model does not include any of the mandatory inclusive-or sub-features of Engine; at least one of the Engine sub-features (Petrol and Electric) should also have been selected for a valid configuration of the car feature model.

### 5.2.4.5 Configuring Features from Different Models

The configuration model is to be restricted to referencing only features from a single feature model. We do not wish to admit the possibility of configuring a system from a combination of features from multiple feature models as the results would be undefined; or, alternatively, we would be forced to define the relationships between features from the different models and one could view this as simply reifying two feature models into a single coherent feature model. The grammar of the concrete syntax for describing configuration models does not admit the possibility of deriving the configuration model from anything other than a single feature model, and the metamodel for the configuration model also specifies this relationship via the attribute `featureModel` (see Figure 84), however there exists the possibility that a user could refer out to features from alternative feature models, and we would want to raise an error under such situations. In this example we will use the Car feature model described in Figure 67, and we will also declare an additional (secondary) feature model as illustrated in Figure 95.
<imports elided>
Root::radioFM :=  
@FeatureModel(RadioFeatureModel)
  Root Feature(Radio) has a {Chassis, Channels, RDS, PowerOutput};

  Mandatory Feature(Chassis) is one of {Dashboard, Boot};
  Mandatory Feature(Dashboard) is leaf;
  Mandatory Feature(Boot) is leaf;

  Mandatory Feature(Channels) is any combination of {FM, AM};
  Mandatory Feature(FM) is leaf;
  Mandatory Feature(AM) is any combination of {LW, MW, SW};
  Mandatory Feature(LW) is leaf;
  Mandatory Feature(MW) is leaf;
  Mandatory Feature(SW) is leaf;

  Optional Feature(RDS) is any combination of {StationId, Traffic};
  Mandatory Feature(StationId) is leaf;
  Mandatory Feature(Traffic) is leaf;

  Mandatory Feature(PowerOutput) is one of {Low, Medium, High};
  Mandatory Feature(Low) is leaf;
  Mandatory Feature(Medium) is leaf;
  Mandatory Feature(High) is leaf;

  Feature RDS requires {FM};
end;

Figure 95 – Example Secondary Feature Model

We now attempt to configure a Saloon Car with a radio by referencing out to features from both the Car and Radio feature models, using the Car feature model as the primary model to which the configuration model is bound (Figure 96).

(1)  <imports elided>
(2)  Root::saloonCarCMError5 :=
(3)  @ConfigurationModel(sportsCarConfigurationModel)
(4)  From carFM select {Car,
(5)  Body, Saloon,
(6)  Gearbox, Manual,
(7)  Engine, Petrol,
(8)  Radio,
(9)  Chassis, Dashboard,
(10) Channels, FM,
(11) AM, MM,
(12) RDS, StationId,
(13) PowerOutput, Medium};
(14) end;

Figure 96 – Configuring a Product from Multiple Feature Models

This example fails to compile because the configuration model is bound to the feature model carFM at line 4, and the parser uses this reference to try to resolve all referenced features, clearly it

155
will fail to resolve the Radio features in the Car feature model, and a compilation error is reported. In order to demonstrate the mis-configuration of a system from multiple feature models equivalent to the intent of the description in Figure 96, an example is synthesised via the abstract syntax. An instance of the configuration model is created and checked against the set of constraints, resulting in the error report in Figure 97.

The constraint violation report identifies a violation of the constraint

FeaturesContainedInFeatureModel

and also identifies the set of features that have been selected erroneously, i.e. those features selected that are not contained within the feature model to which the configuration has been bound (\texttt{carFM}), in this case the features: Radio, Chassis, Dashboard, Channels, FM, AM, MW, RDS, StationId, PowerOutput, and Medium.

![Constraint Report - Windows Internet Explorer](image)

**Figure 97 – Features Contained In Feature Model Constraint Violation**

### 5.3 Building a Metamodel for Specification Models

Once a valid configuration model has been specified from the source feature model it is then possible to relate the selected model elements to a specific architectural style (the architectural blueprint). The architectural blueprint identifies the target architecture into which the configuration is to be mapped. Figure 98 provides the class diagram of the specification model.
5.3.1 Example Specification Model

A textual concrete syntax for the specification models has been developed (the listing may be found at Appendix 5). An example specification model for the sports car variant discussed above is presented textually in Figure 99.

```
parserImport Root::ProductFamilies::SpecificationModels;

import Root::ProductFamilies::SpecificationModels::SpecificationModel;
import Root::ProductFamilies::ConfigurationModels;

Root::sportsCarSM :=
  @SpecificationModel(sportsCarSpecificationModel)
  Bind sportsCarCM to carLayeredArchitecture;
end;
```

The architectural blueprint is represented as simply an instance of the meta-class `Package`; this is the root (enclosing) package of the target architecture to be used; again, a meta-modelling approach has been adopted to provide the specification of the architectural blueprint, see Section 5.3.2.

In the above listing the parameter `sportsCarCM` is an instance of the configuration model, this is bound to `carLayeredArchitecture`, an instance of the metaclass `Package`, via the attribute `architecturalBlueprint`. The parameter `carLayeredArchitecture` is an instance of the architectural model (the architectural blueprint) into which the configuration is to be mapped. The keyword `Bind` used to bind the specification model to the configuration model and architectural blueprint is syntactic sugar (see the definition of the grammar for the concrete syntax of the specification model in Appendix 5). The layered architecture to be used is shown in Figure 100, although a more complete description of the modelling of architecture is provided at Section 5.3.2.
<imports elided>

Root::carLayeredArchitecture :=
@Architecture(carArchitecturalModel)
  Layer(CarBody) depends on {iTransmission, iEngine}
  realises {iCarBody};
  Layer(Transmission) depends on {iEngine}
  realises {iTransmission};
  Layer(Engine) realises {iEngine};
end;

**Figure 100** – Example Car Layered Architecture (Specification)

The resulting package structure of the architectural model is shown in Figure 101.

![Figure 101 – Example Car Layered Architecture (Model)](image)

In this example architecture, each layer has been generated as an instance of the meta-class `Package`, and each layer contains a controller class named appropriately. Each layer of the architecture depends upon the interface exported by some lower layer (as described by the concrete syntax in Figure 100).

### 5.3.2 Reference Architectural Components Model

The interface required of the architectural model is that the root object must be an instance of the meta-class `Package`. The architectural style is defined most conveniently via a profile; individual architectures may then be described as instances of the profile. As an example, the profile for a layered architecture is modelled as shown in Figure 102.
The profile for a simple layered architectural model is derived from the meta-classes Package and Class, and describes the architecture as comprising a number of layers, each layer contains a Controller class that will be responsible for the orchestration of components within the layer, and each layer realises a set of Interfaces. There are two kinds of architectural layer, the FoundationLayer (the bottom most layer), and the Layer (any higher-level layer); Layer objects depend on the Interfaces provided by lower layers; and both the FoundationLayer and Layer classes may contain sub-layers (to support nesting). Profiles could be defined to provide alternative architectural styles, e.g. IMS and DDS.

A simple textual concrete syntax has been developed to provide an easy mechanism with which to describe instances of the layered architecture profile in Figure 102, the parser for which is presented in Appendix 6. As an example, one might describe the popular three-tiered architecture in the form shown in Figure 103; the instantiation of this architecture is shown in Figure 104.

```
<imports elided>
Root::threeTieredArchitecture :=
  @Architecture(ThreeTieredModel)
  Layer(Presentation) depends on {iLogic} realises {iPresentation};
  Layer(Logic) depends on {iData} realises {iLogic};
  Layer(Data) realises {iData};
end;
```

Figure 103 – Three-Tier Architecture Description
A number of helper operations are provided on the architectural profile of Figure 102 to support queries and manage the instantiation of models from the textual specification language, e.g. the operation `getLayerByName` on class Architecture returns any layer of the specified name (see Figure 105).

[1] XMF> threeTieredArchitecture.getLayerByName("Logic")->sel.dependsOn; Set(Interface(iData))

[1] XMF> threeTieredArchitecture.getLayerByName("Logic")->sel.realises; Set(Interface(iLogic))

Figure 104 – Three-Tier Architecture

Figure 105 – Querying the Three-Tier Architecture
At this point we have:

- A Feature Model bound to the source Model Elements
- A Configuration Model identifying the features required
- A Specification Model identifying the Configuration and the Reference Architectural Components Model (the architectural blueprint)

The final link in the chain is to be able to generate the required product (model) into the target architecture; this is provided by the Product Generator.

### 5.4 Generating the Product

Generation of the product specified by the SPLA is essentially a transformational issue, we have assembled all the necessary components (models), and the final activity requires the merging of the specified model elements into the specified architectural style (e.g. the layered architecture shown in Figure 100 and Figure 101). Clearly, there may be more than one product generator for each target architectural style, as it is the role of the product generator to merge the components, and this is dependant upon either implicit or explicit knowledge of both the source model elements and the target architecture. Figure 106 provides an outline of the form of the product generation transformation for the layered architecture described for the car PLA example.

![Figure 106 – Overview of Product Generation Transformation](image)

In this example, the top-level transformation `CarSpecification2Product` identifies the type of architectural blueprint in use and invokes the corresponding top-level specification transformation for that architectural style, in this case `CarSpecification2Product` will invoke `CarSpecification2LayeredProduct`, resulting in the generation of a new model conforming to the specified architectural blueprint. Alternative architectural styles (e.g. IMS, DDS, etc.) would be accommodated via alternative transformation components – the top-level transformation is shown at Equation 28.
Equation 28 – Identification of Target Architectural Style

A simplistic approach to the merging of the source model components identified via the configuration model and the architectural blueprint is applied via the subsequent transformation Specification2LayeredProduct, whereby all components (model elements) identified via the selected features are copied into the appropriate layer in the target architectural blueprint, see Equation 29.

Equation 29 – Transformation into Layered Architectural Style
The above transformation iterates across the chosen components identified via the configuration model and, via three simple helper operations (Equation 30 through Equation 32), inserts the components into the target architecture.

The helper operation at Equation 30 binds the components to the appropriate layer in the target architecture.

@Operation bindTo(a : Architecture, components : Set(Element)) : Architecture
@For component in components do
    let layer = self.getTargetLayer(a, component)
    in if not layer->isEmpty
        then ("inserting: " + component.name + 
            " into layer: " + layer->sel.name).println();
        self.insertClassHierarchy(layer->sel, component)
        else ("unable to bind " + component.name+ 
            " to a layer: " + layer.name + 
            " in: " + a.name).println()
    end
end
end;
// return a to conform to the contract
a
end

Equation 30 – Helper Operation bindTo()

The helper operation at Equation 31 identifies the appropriate layer in the target architecture based on explicit a priori knowledge of the mapping from the source model package structure to target architecture. Whilst clear, this approach could be made more flexible via the parameterising of the component to layer mapping.

@Operation getTargetLayer(a : Architecture, component : Element) : Set(FoundationLayer)
if CarDomainModels::Bodies.classes->
    collect(c | c.name)->includes(component.name)
then a.getLayerByName("CarBody")
elsif CarDomainModels::Engines.classes->
    collect(c | c.name)->includes(component.name)
then a.getLayerByName("Engine")
elsif CarDomainModels::Gearboxes.classes->
    collect(c | c.name)->includes(component.name)
then a.getLayerByName("Transmission")
else (component.of() + component.name + 
    " is NOT recognised!").println();
    Set()
end
end

Equation 31 – Helper Operation getTargetLayer()

The helper operation at Equation 32 traverses the inheritance hierarchy from the component class to its root. The test at line 2 on p.owner.isMetaPackage() identifies the point at which the model traverses meta-levels, and prevents the inadvertent incorporation of unnecessary meta-classes (such as the metaclass Class).
Equation 32 – Helper Operation `insertClassHierarchy()`

The structure of the resulting model generated by this process is shown in Figure 107.

![Screenshot of Generated Model (Outline)](image)

Figure 107 – Screenshot of Generated Model (Outline)

Revisiting the components (model elements) from the domain model that are related to the features selected via the configuration model illustrated in Figure 86 and Figure 88, it can be seen that the generated model comprises a valid subset of the domain model. The model illustrated at the top of Figure 108 shows the components of the domain model, and the model illustrated at the bottom shows the selected components merged into the specified architectural style in the appropriate layer (CarBody).
The car PLA example demonstrated above is small and relatively simple, however larger and more complex examples are illustrated in the case studies described in Chapter 6, these examples also illustrate more significant architectural transformations between the domain model and the architectural blueprint.

### 5.5 Conclusions

In this section a detailed description of the SPLA has been presented using a meta-modelling architecture following the MDD style presented in [CSW08], the overall structure of the SPLA is described in Figure 51, showing the relationship between the metamodels. The following chapter (Chapter 6) illustrates the implementation of the SPLA described above via two case studies drawn from different domains. The purpose of the case study is to demonstrate the domain-independence of the SPLA and illustrate the strengths and weaknesses of the approach.

The intention of the approach to the design of the SPLA described above is to increase the utility of the SPLA by breaking it up into a number of coherent parts, defining the relationships between the parts such that a product may be generated from a specification derived from the space of systems admitted by the product family. In Chapter 7 we will reflect upon the opportunities for future improvements to this modular design of the SPLA.
5.5.1 Domain Engineering

The Domain Engineering activity provides an analysis of the domain of interest and scopes the SPLA in terms of commonality and variability, such that it is not so specific that it precludes development of related systems, or so broad that development of the SPLA is too costly. The Domain Engineering activity is described in Section 3.1.1, the output from this activity of most relevance to the work reported in this thesis is the feature model.

5.5.2 Feature Model

The metamodel for the feature model is described in Section 5.1, a populated example is illustrated in Section 5.1.1, and a textual concrete syntax has been developed to support the description of feature models in an effective manner (see Appendix 3). The feature metamodel described takes a simple, but coherent view of the feature model components; this allows instances to be checked for well-formedness against a small number of constraints, e.g. checking that all sub-features attached to a node of type Homogeneous have the same optionality, whilst deferring all notion of choice to subsequent models. The approach taken to the structure and validation of the feature model eliminates the need to introduce refinement rules as described by Czarnecki & Eisenecker [CE00] because the problematic structures are identified by the constraint checks ad associated error messages. Relating the features in the feature model to elements in the domain model relieves the feature model of the dual role of describing both feature compositional and structural semantics; the feature model describes only feature compositional rules, whilst the domain model describes structural semantics.

5.5.3 System Configuration

System configuration is supported by the configuration model, the metamodel for which is described in Section 5.2, a populated example is illustrated in Section 5.2.3, and a textual concrete syntax has been developed to support the description of configuration models in an effective manner (see Appendix 4), although a graphical concrete syntax is also possible. The configuration model takes a valid feature model as an input and supports the description of individual family members from the description of the family of systems provided by the feature model; in this respect the configuration model may be seen as a refinement of the feature model. The configuration model supports the modular approach to the modelling of the SPLA by validating the user-defined system configuration description against the semantics of the feature relationships, e.g., ensuring that only one mandatory alternative sub-feature is selected from a group. This approach to the definition of the configuration model relieves the feature model of the dual role of validating both the structural properties of the description of a family of systems and the description of the synthesis of one particular family member. In addition to the simplification
provided to the feature model by this approach, there is also the opportunity to plug-in alternative feature configuration semantics, should this be required.

5.5.4 Specification Model

The metamodel for the specification model is described in Section 5.3, a populated example is illustrated in Section 5.3.1, and a textual concrete syntax has been developed to support the description of feature models in an effective manner (see Appendix 5). The specification model takes a valid configuration model and binds it to an explicitly modelled architectural blueprint (see Section 5.3.2), such that the architectural style to be applied to the system to be synthesised from the configuration model may be parameterised; this approach, whilst novel in the context of the SPLA, is evocative of the MDA approach advocated by the OMG [OMG03], and provides an additional level of flexibility to the design of the SPLA, such that domain models may be reused in alternative architectural styles – one might consider the domain model to be the PIM, and the architectural blueprint as defining the PSM. The approach is also consistent with the theme of enforcing modularity of component metamodel for the SPLA.

5.5.5 Product Generator

The product generator has been described in the form of a transformation from source components (accessed via the specification model) to target components (the synthesised model). Although the product generator has been demonstrated using the uni-directional pattern-matching executable language XMap[^39^], alternative technologies are emerging, e.g. see [Rad08], and the approach is believed to be generally applicable – however this would need to be confirmed by further work. The approach to product generation from the SPLA is novel from two perspectives; firstly, the semantics of the relationships between components in the domain model are provided via the explicit definition of the model conforming to a metamodel, rather than being implied from the overloaded use of the feature model, and secondly, the target architectural style is provided as a parameter via the specification model. The drawback with the product generator as described in Section 5.4 is that it requires explicit knowledge of the transformation rules for domain components (e.g. see Equation 31); it would be preferable that such transformations are implicit. One possible approach to solving this problem would be to colour the domain model via meta-attributes in some way, such that the product generator need only know about the model colourings; however this remains as future work. Although the generation of an example system derived from a product line of cars has been demonstrated, the model generation takes a broad brush approach to the generation of the product family member, assuming that all attributes and operations on each selected class are required. The feature model relates each feature to a (possibly

[^39^]: A model transformation language bundled with XMF.
empty) set of model elements from the domain model, hence the fidelity of the feature model to
domain model mapping could be decomposed to the attribute, operation, and constraint level, at the
cost of a more complex generator; supporting the idea of *colouring* the domain model; this remains
as future work.

5.5.6 The Cardinality of Features

There is some debate in the literature regarding the decoration of feature relationships with
cardinalities; those authors in favour tend to be rooted firmly on the side aiming to use the feature
model for subsequent code generation, e.g. [Hak02], whilst those against see the feature model as
being necessarily abstract, Kang et al. eschew defining the cardinality of feature relationships on
the feature diagram in the initial description of FODA [KCH+90], deferring instead to other
models, such as ERD models; one can draw a parallel between Kang’s approach and the approach
described in this chapter, although in the case presented above the linkage between the domain
model and the feature model is explicit, whilst in Kang’s approach it is not. It should also be noted
that whilst Czarnecki & Eisenecker initially side with pure view of the feature model, the authors
subsequently recant on the absence of cardinalities on the feature model, citing the results of using
feature models in a practical application [CBU+02]. In the work presented above, the cardinality of
the relationship between features is implicit, insofar as it is expressed in the component inter-
relationships of the source model components. However, the use of the feature model presented
above is not predicated on the linkage to a collection of source model components, e.g. the feature
model may be utilised to validate (and visualise) a family of artefacts, or as part of the development
of a new family of components in a feature-driven style. The point here is that by binding the
features in the feature model to the artefacts in the domain model, one can avoid the notion of
cardinality in the feature model, Kang avoids this in FODA via an informal link to other models of
the domain, such as the ERD [KCH+90], however formal linkage of the feature model to domain
model components provides increased rigour and traceability.

5.5.7 Normalising Feature Models

In the description of features provided by Czarnecki & Eisenecker [CE00], the authors provide a
number of normalisation rules for feature models. However, on inspection, it appears that the
underlying motivation for such rules is driven by the need to provide transformations from
mal-formed feature models to well-formed models. It has been demonstrated above that a more
rigorous approach to the definition of the metamodels comprising the SPLA eliminates such
transformations by providing constraints over model instances to identify error cases, provide
informative error messages and enforce well-formedness of the models.
5.5.8 Modularity of the SPLA

The SPLA has been defined in the form of a number of related metamodels. The intention is to enforce a separation of concerns and to relieve the feature model of its overloaded role of describing both the feature network and the domain architecture, by the introduction of relevant supporting metamodels. An example of the benefits of such an approach may be seen in (e.g.) the definitions of the feature model and the configuration model. The feature model concentrates on the structure of the product family in terms of feature relationships, enforcing well-formedness of the family-based view via constraints over the model. The configuration model builds upon this by providing more complex constraint checks to ensure well-formedness of specific sets of features chosen to comprise one particular family member. The definition of the specification model as a separate entity affords further flexibility by providing the reference architectural model as a parameter to the configuration, thus providing an opportunity to decouple the feature model from the target architecture, offering reuse of the feature model across both different domain models (via substitution of the component bindings) and alternate reference architectural models. The product generator completes the picture by providing the system synthesis, resulting in the generation of a target model from the space of systems admitted by the feature model.

5.5.9 Development and Validation of Metamodels

The approach to the development of metamodels was evolutionary and incremental. Having established a view of the informal semantics for features (from the literature search), an initial metamodel for the feature model was developed and tested; development and testing of the initial feature model informing the partitioning of the feature and configuration models. Some constraints on the models were obvious from the outset, e.g. leaf features shouldn’t have sub-features; however there are choices to be made regarding the implementation of such constraints, e.g. ranging from a very generic model with many constraints to identify invalid instances, to very specific models, the structure of which precludes many invalid instances from ever being created, the approach followed tending to be towards the latter as this makes the visual rendering of the model clearer to the intended audience.

A four-stage approach to the validation of models, as illustrated in Figure 109 was adopted. The first stage was simply to inspect the designs (class diagrams and XMF source code) visually to confirm that they appeared sensible. The second stage was to create simple snapshots of the models using the abstract syntax (e.g. see Figure 60); although this was found to be a useful practice that allows development and testing of model constraints (by supporting the creation of invalid snapshots as test cases), the snapshots become very large and unwieldy very quickly. The third stage was to create more challenging snapshots using the XOCL language directly (i.e. writing example models in the abstract syntax). This approach allows the creation of far more complex
model instances, however the XOCL script files can also become quite complex and long. The fourth and final stage in the approach followed was to develop a textual concrete syntax and associated parser(s) to support the creation of models in a model concise and user friendly form (e.g. see Figure 67). The advantage gained by expressing models in such a domain-specific concrete syntax is clarity, however the drawback is that the grammar of the concrete syntax may preclude the definition of invalid model instances that would be admitted by writing directly in the abstract syntax, hence it has been necessary to synthesise a number of examples demonstrating the effectiveness of the constraint checking over model instances.

The strategy adopted for the development and validation of the metamodels underpinning the SPLA was found to be successful and pragmatic and has also been used successfully on other metamodelling projects [HJ08b]. However, on reflection, it would have been better to have spent less time developing models in the abstract syntax as the productivity increase gained by working in the concrete syntax far outweighed the effort required to develop the grammar and associated parsers.
6 Case Studies and Evaluation

A summary of the SPLA as described in the literature has been presented and a number of issues with this description have been identified. Consideration of these issues has supported the development of an alternative (novel) view of the SPLA that offers advantages over its predecessors in terms of modularity and flexibility, leveraging from an architecture underpinned by metamodels. A model-based view of the SPLA has been described (initially) via an abstract, logic-based specification expressed in Z, and (subsequently) via an executable implementation using the XMF-Mosaic tool. This chapter describes two case studies undertaken to validate the utility and flexibility of the models comprising the SPLA as presented in Chapter 5, implemented in the XMF-Mosaic tool. In each case the SPLA will be developed in accordance with the process described in Chapter 3 and the results used to validate the hypothesis that the SPLA metamodel is domain-agnostic, and more flexible than its predecessors, the results will also help to identify areas of possible future development and/or improvement.

In order to demonstrate that the metamodel-based SPLA is domain-agnostic it will be applied to two small SPLAs drawn from different and unrelated domains; both domains are derived from the author’s experience are representative of real-world problems. An additional benefit of the use of disparate case studies is that it likely to highlight deficiencies in the metamodels and inform the direction of future work. The success criteria expected is to demonstrate that the Feature and Configuration models are applicable in both domains, and that it is possible to separate the target implementation architecture from the SPLA itself.

The case studies presented in sections 6.2 and 6.3 follow a common structure; each begins with a brief introduction to the domain (a detailed introduction may be found in the Appendix 7 through Appendix 9), followed by the various stages of development of the SPLA (see Figure 51), each case study concludes with a brief review of relevant related work. This chapter concludes with an evaluation of the result obtained from the two case studies.

The first case study relates to the application of the SPLA to the modelling of a Tactical Data Link (TDL); we choose Link 16 as defined by [DoD04] as this is in the public domain, and has been used elsewhere in the literature. Previous published work relates to the identification of an execution semantics (e.g. [Mak06] and [ZH07]); however this case study represents the first application of SPLA technology. A more detailed description of the application of model-based approaches to the TDL domain may be found in [HP08].

The second case study relates the application of the SPLA to the modelling of a navigation system for the avionics domain. This work is derived from [HE03b], which was, in turn, informed by
previous work in the public domain relating to the ADAGE project (e.g. see [CTB+96]). Whilst the literature reports on the application of the SPLA within the navigation domain, this was predicated on specification using the GenVoca architecture which serves the dual purpose of defining both the product line and the (layered) software architecture, both of which we seek to tease apart to make more explicit. Where possible, GenVoca type equations are compared against the corresponding feature model. A more detailed description of the navigation domain may be found in [HE03b].

6.1 A Concrete Syntax for the SPLA

Although there is a generally accepted graphical concrete syntax for the feature model (see [CE00]), there is no correspondingly accepted concrete syntax for the Configuration model (primarily because it has not been made explicit previously). The XMF-Mosaic environment provides a tool (XTools) to support the development of graphical languages, however there were found to be limitations with this facility and it was necessary to adopt a textual concrete syntax to allow models to be specified efficiently – the grammars associated with the relevant textual languages designed for this purpose may be found at Appendix 3 through Appendix 6.

6.2 Tactical Data Links

This section reports the results of a case study applying the PLA models described in Chapter 5 to the domain of the TDL. The case study comprises two demonstrations, one describes the application of the PLA to the family of TDL Units; the other describes the application of the PLA to the family of TDL functions that may be supported by a TDL unit. In both examples, we traverse the complete set of PLA models described in Chapter 5 and conclude each with an example model generated from the PLA.

6.2.1 Introduction to the TDL Domain

The TDL provides one of the backbone technologies underpinning the defence community’s goal of Network Enabled Capability (NEC) by providing the information and infrastructure to afford users with both an integrated picture of the battlefield and also provide tasking orders and responses. A number of TDLs are in service with coalition forces, and are implemented on a variety of assets, such as aircraft, ships, land vehicles, and command stations, an example of which is illustrated in Figure 110.
Although it is possible for TDL platforms to operate on multiple TDLs concurrently (e.g. in the example shown in Figure 110 the Nimrod is shown as communicating on both Link 11 & Link 16) acting as (e.g.) a bridge (known as a Forwarding Unit), the case study is restricted to single Link 16 platforms only.

The Link 16 TDL is a general purpose TDL, in contrast to some others, e.g. Link 4A or VMF (a list of data link characteristics is provided in [HJ05b]). It has evolved over a number of years, stemming from a requirement identified by the US military in the early 1970’s for a TDL offering a broad range of functions that would be applicable for use across multiple forces (e.g. Navy, Marines, Air Force, Army, etc.). The TDL comprises two distinct layers, the bearer, and the message standard – the bearer provides the services required to support the transfer of data across the network, and could in principle support many different message standards – the bearer is not considered further here. The message standard for Link 16 (or TADIL J) is defined by MIL-STD-6016C [DoD04], this document features a number of shortcomings affecting its usability as described in [HJ05a], of particular relevance to this work are the following:

- Document-based, no apparent underpinning model
- Largely narrative
- Open to (mis)interpretation
- Not checkable by machine
- Duplication of material invites inconsistency
- Comprises many interdependent sections and appendices
- It’s enormous, >7300 pages

The derivation of a layered hierarchy of models to underpin the TDL domain is described in [HJR07a] and [HJR07b], and lessons learned from the application of meta-modelling and modelling within the TDL domain are documented in [HJ08b]. Whilst the standard describing Link 16 [Dod04] suffers from a number of significant issues, as identified above and discussed elsewhere in the literature (e.g. [HJR07b], [Zei06]), it seeks to provide a description of the features to be implemented to support specific objectives, such as Air Surveillance, and, as such, one may view it as an informal (and very large) description of a PLA.

There are two aspects of the Link 16 TDL standard that appeal for the application of a PLA-based approach in order to bring increased clarity to the standard. Firstly, the standard identifies a number of types of TDL platform, the generic term for which is a JTIDS Unit (JU), however the definition of and relationship between members is not made clear from the narrative introduction. Secondly, and more interestingly, the standard defines a set of minimum implementation (MIN IMP) rules; these may be viewed as the features and feature dependencies that must be supported by any
conforming platform implementation – i.e. a platform must comply with the relevant MIN IMP rules to be considered valid and to be allowed to participate on the link. Non-conformance against the minimum implementation rules results in either re-work (which could be very costly in terms of time, effort, and reputation), or the raising of an exception\textsuperscript{40} (which must be accompanied by a supporting rationale), in this case the result may be manifest in the form of interoperability issues with the platform. Each of the above opportunities for the application of an SPLA approach is described more fully below.

6.2.2 Domain Engineering of TDL Platforms

The Link 16 TDL is described by a large document-based standard [DoD04]. The link has evolved considerably since it original inception in the early 1980’s, and the hardware required to operate on the link has also evolved over time. In parallel with the evolution of Link 16, new links have been introduced (e.g. VMF), requiring that some TDL platforms provide the capability to operate on multiple links, sometimes concurrently; e.g. Figure 110 shows the Nimrod aircraft interacting on both Link 16 and Link 11. Platforms play various roles on the network; however the Link 16 standard does not make the variety of roles or unit types very clear. The domain engineering activity guiding development of a product family of TDL platforms is guided by an analysis of the standard revealing the identification of three primary features required to define a functional Link 16 TDL platform:

1. Network Participation: connection to the network.
2. Unit Type: a coarse-grained family of TDL units.
3. Data Forwarding: the (optional) ability of a platform to communicate on other links.

Each of the above is described in more detail in Appendix 7, leading to the description of a taxonomy for TDL units.

6.2.3 Feature Modelling of TDL Platforms

The taxonomy of JUs may be transitioned to the feature model description shown in Figure 111 (textual concrete syntax); the well-formedness of this model may be checked against the constraints described by the feature model implementation in Section 5.1 – the instance satisfies all constraints.

\textsuperscript{40} This is request for exemption against the rules of the network.
Root Feature(TdlUnit) has a \{NetworkParticipation, UnitType, DataForwarding\};

Mandatory Feature(NetworkParticipation) is any combination of (DataSource, Subscriber);
Mandatory Feature(DataSource) has a \{PositionReference\};
Optional Feature(PositionReference) is leaf;

Mandatory Feature(Subscriber) is one of \{PrimaryUser, SecondaryUser\};
Mandatory Feature(PrimaryUser) is leaf;
Mandatory Feature(SecondaryUser) is leaf;

Mandatory Feature(UnitType) is one of \{JU, PU, RU\};
Mandatory Feature(JU) is one of \{C2JU, IEJU, NetworkManager\};
Optional Feature(C2JU) has a (SurveillanceSystem);
Optional Feature(SurveillanceSystem) is leaf;
Optional Feature(IEJU) is leaf;
Optional Feature(NetworkManager) is leaf;
Mandatory Feature(PU) is leaf;
Mandatory Feature(RU) is leaf;

Optional Feature(DataForwarding) is any combination of (PointToPoint, Broadcast);
Mandatory Feature(PointToPoint) is any combination of \{Link11, Link11B\};
Mandatory Feature(Link11) is leaf;
Mandatory Feature(Link11B) is leaf;
Mandatory Feature(Broadcast) is any combination of \{Link16, GenericLink\};
Mandatory Feature(Link16) is leaf;
Mandatory Feature(GenericLink) is leaf;

Feature PU requires \{PointToPoint\};
Feature RU requires \{DataSource, Link11\};
end;

Figure 111 – A Feature Model for TDL Units (textual)

For completeness, the graphical concrete syntax of the feature model specified in Figure 111 is rendered in Figure 112; this graphical concrete syntax is as described by Czarnecki & Eisenecker [CE00].
Figure 112 – A Feature Model for TDL Units (graphical)
6.2.4 Configuration Modelling of TDL Platforms

The feature model described above may now be used as the basis from which a C2 JU TDL unit may be configured having the features: Data Source, Position Reference, Secondary User, with Data Forwarding to Link 11B (see Figure 113).

The configuration model is shown graphically in Figure 114; all chosen features are shaded.

The above feature and configuration models may then be checked against the constraints on the underlying metamodels (both of which are valid instances).
Figure 114 – A Configuration Model for a TDL Unit
By contrast we could specify an invalid configuration in the form shown in Figure 115, and then exercise the model constraints to verify that the erroneous instance of the configuration model is identified.

```xml
<imports elided>
Root::ruUnitSMError :=
@ConfigurationModel(RuTdlUnitError)
From tdlUnitFM select
{TdlUnit,
   NetworkParticipation, Subscriber,
   UnitType, JU, RU,
   DataForwarding, PointToPoint, Link11};
end;
```

**Figure 115 – Example of Erroneous TDL Unit Configuration Model**

This configuration model is illustrated in Figure 117, with errors shaded in red. This model is identified correctly as being in error (Figure 116) via the application of the model constraints; failing against the constraints:

- MandatoryExclusiveOrSubFeatureSelectionCardinality
- AllDependenciesSatisfied

![Constraint Report](image)

**Figure 116 – TDL Unit Configuration Error (Constraint Report)**

The constraint error report identifies correctly that, whilst the Subscriber feature has been selected, one of the sub-features (PrimaryUser or SecondaryUser) should have been selected, and also that selection of the RU feature requires selection of both the DataSource feature and the Point-to-Point feature, the former of which has been omitted from the configuration.
One Subscriber sub-feature feature should have been chosen.

This feature should have been chosen.

Figure 117 – Erroneous Configuration Model for TDL Unit
6.2.5 Specification Modelling of TDL Platforms

The Specification model for the TDL Platform comprises the configuration model specified in Figure 113 and illustrated graphically in Figure 114, with a target architectural model. We derive a simple three layered architectural model of the form shown in Figure 118, into which we wish to generate the specified TDL unit configuration.

![Figure 118 – Example TDL Unit Target Architecture](image)

6.2.6 Product Generation of TDL Platform

The product generation stage of the process takes the TDL Unit specification model described above and merges the TDL Unit configuration with the specified target architecture based on specified correspondences. In this example we use a simple mapping from TDL Unit packages into target architectural layers (see Figure 119), although the mapping could be arbitrarily complex.

![Figure 119 – TDL Unit Model Element to Architecture Mapping](image)

The resulting model generated by the application of the architectural transformation is shown in Figure 120, demonstrating the successful generation of a valid configuration of TDL Unit in the form of an instance of a three layered architecture that is itself an instance of a metamodelled layered architecture.
6.2.7 Domain Engineering of TDL Function and Message Structure

The Link 16 TDL has a functionally-oriented message-based interface, with TDL platforms communicating with each other via sequences of message transactions. The Link 16 standard defines the complete set of messages in the form of a message catalogue, with messages organised into functional groups and decomposed into a varying number of fixed length words [DoD04]. Each Link 16 word is partitioned into a number of fields, and each field has a defined type; the definition of which is provided by the data dictionary. There is no single reference implementation of the Link 16 standard to be deployed or implemented by each participant on the link; instead the family of JTIDS units is partitioned (coarsely) into C² and non-C² units, and each of these is then described in the form of a set of minimum implementation (MIN IMP) requirements. The MIN IMP requirements go from the level of the function (e.g. position reporting) to the individual values that may be assumed by the fields of each word (these are referred to as Data Items). The MIN IMP requirements can be seen to approximate to a product family view of the TDL function and message structures, describing valid groups of messages and associated interdependencies; however the standard does not provide such a structured view, resorting to many long tables with associated prose and footnotes, covering 500 pages of documentation. A more detailed analysis of the TDL functions and message structures is provided in Appendix 8, this provides the basis for the
development of a feature-oriented view of the TDL function and message structures in the form of a model-based product family.

6.2.8 Feature Modelling of TDL Function and Message Structures

The TDL message structures may be transitioned to the feature model description shown in Figure 121 (textual concrete syntax) based on the MIN IMP rules such that related functions are modelled via the appropriate feature construct (variation point, inclusive/exclusive-or) and function dependencies are made explicit. The well-formedness of this model may then be checked by the feature model implementation.

For completeness, the graphical concrete syntax of the feature model specified in Figure 121 is rendered in full following the style described in [CE00] in Appendix 10 (Figure 198 through Figure 206), the top-level feature model is presented below in Figure 122. For illustrative purposes, we ignore the ambiguity of the message definitions for the features: System Information and Net Management, and Precise Participant Location and Identification, and simply declare that the features are mandatory. The well-formedness of the feature model of the MIN IMP rules model may be checked against the constraints described by the feature model implementation in Section 5.1 – the instance satisfies all constraints.
<imports elided>
Root::tdlFunctionFM :=
@FeatureModel(TdlFunctions)
  Root Feature(TdlFunction) has a {SIENM, PPLI,
   SupportFunctions, MessageCatalogue};

Mandatory Feature(SIENM) has a {SIENM_Tx, SIENM_Rx};
Mandatory Feature(SIENM_Tx) is leaf;
Mandatory Feature(SIENM_Rx) is leaf;

<PPLI feature elided>
Optional Feature(SupportFunctions) has a {Surveillance, EwIntelligence,
   Control, Management};

Optional Feature(Surveillance) is any combination of
   {AirSurveillance, SurfaceSurveillance, SubsurfaceSurveillance,
     LandSurveillance, SpaceSurveillance, ElectronicSurveillance};

<Surveillance features elided>
<EwIntelligence, Control features elided>
Optional Feature(Management) is any combination of
   {MissionManagement, WeaponsCoordinationAndManagement};

<MissionManagement, WeaponsCoordinationAndManagement features elided>
Mandatory Feature(MessageCatalogue) is any combination of
   {SurveillanceMessage,
     AntisubmarineWarfareMessage,
     InformationManagementMessage};

Mandatory Feature(SurveillanceMessage) is any combination of
   {ReferencePointMessage, EmergencyPointMessage, AirTrackMessage,
     SurfaceTrackMessage, SubsurfaceTrackMessage,
     LandPointOrTrackMessage, SpaceTrackMessage,
     EwProductInformationMessage};

<MessageCatalogue features elided>
Feature MissionManagement requires
   {WeaponsCoordinationAndManagement_Tx,
     WeaponsCoordinationAndManagement_Rx};
Feature Control requires
   {Air_Tx, Air_Rx, WeaponsCoordinationAndManagement_Tx,
     WeaponsCoordinationAndManagement_Rx};
Feature EwIntelligence requires       {Surveillance};
Feature Air_Tx requires
   {AirTrackMessage,
     TrackManagementMessage,
     IffOrSifManagementMessage};

<AIR_Rx required features elided>
end;

Figure 121 – A Feature Model for TDL Functions
Figure 122 – Top-Level TDL Function Feature Model

Decomposed at Appendix 10

Surveillance Message (J3)
Antisubmarine Warfare Message (J5)
Information Management Message (J7)
6.2.9 Configuration Modelling of TDL Function and Message Structure

The feature model may now be used as a basis from which we may configure a specific set of TDL message structures to support a defined set of TDL functions. As an example we will configure a system to support the following functions:

- System Information and Net Management (SIENM)
- Precise Participant Location and Identification (PPLI)
- Air Surveillance (Tx & Rx modes)
- EW Intelligence (Tx & Rx modes)

The functions identified (above) provide a small, but representative suite of TDL functionality comprising both mandatory and optional features sufficient to exercise the SPLA metamodels. A more comprehensive set of features could be configured; however this would not provide any additional benefit, other than simply providing a larger configuration model, and possibly obscuring the results.

As a result of ambiguity in the source material [DoD04] regarding the message requirements for the SIENM & PPLI functions, we have not defined message dependencies for these functions, however, the feature model captures (and enforces) message dependencies for the Air Surveillance and EW Intelligence functions. Hence, the desired configuration may be expressed via the form illustrated in Figure 123.
<imports elided>
Root::tdlFunctionCM :=
  @ConfigurationModel(TdlFunctions)
  From tdlFunctionFM select
  {TdlFunction,
    SIENM,
    SIENM_Tx,
    SIENM_Rx,
    PPLI,
    PPLI_Tx,
    PPLI_Rx,
    SupportFunctions,
    Surveillance,
    AirSurveillance,
    Air_Tx,
    Air_Rx,
    EwIntelligence,
    EwIntelligence_Tx,
    EwIntelligence_Rx,
    MessageCatalogue,
    SurveillanceMessage,
    EmergencyPointMessage,
    AirTrackMessage,
    InformationManagementMessage,
    TrackManagementMessage,
    IffOrSifManagementMessage
  };
end;

Figure 123 – Example TDL Function Configuration

The above configuration model is shown graphically in Figure 124 (due to the size of the feature model we show only the features selected). The configuration model is checked against the constraints of the Configuration metamodel and is reported as a valid instance.
Figure 124 – Configuration Model for Selected TDL Functions
We could specify an invalid configuration in the form shown in Figure 125, and then exercise the model constraints to verify that the erroneous instance of the TDL Function configuration model is identified.

```plaintext
<imports elided>
Root::tdlFunctionError1CM := @ConfigurationModel(TdlFunctions)
   From tdlFunctionFM select
   {SIENM,
    SIENM.Tx,
    SIENM.Rx,
    PPLI,
    PPLI.Tx,
    PPLI.Rx,
    SupportFunctions,
    Surveillance,
    EwIntelligence,
    EwIntelligence.Tx,
    EwIntelligence.Rx
   };
end;
```

Figure 125 – TDL Function Configuration (Error)

The configuration model is illustrated in Figure 126, with (three) errors shaded in red. This model is identified correctly as being in error via the application of the model constraints. Each of the three errors in the configuration model is of a different type and each is identified by a separate constraint as illustrated in Figure 127 under the heading ‘Dependent checks’.
The TdlFunctions root feature should have been chosen

A Surveillance subfeature should have been chosen

A Message Catalogue and subfeature should have been chosen

Figure 126 – Erroneous Configuration Model for TDL Function
The constraint checks on the Configuration metamodel identify three categories of error:

1. **AllSuperFeaturesSelected** – the violation of this constraint is caused by the failure to select the root feature `TdlFunctions`.

2. **AllReachableMandatoryFeaturesSelected** – the violation of this constraint is caused by the failure to select the feature `MessageCatalogue`.

3. **MandatoryInclusiveOrSubFeatureSelectionCardinality** – the violation of this constraint is caused by the failure to select at least one of the surveillance sub-features, such as the feature `AirSurveillance`.

It should be noted that by addressing the errors identified above by selecting the specified features we will expose further errors, e.g. selection of the feature `AirSurveillance` must be accompanied by selection of either or both sub-features `Air_Tx` or `Air_Rx`; selection of feature `MessageCatalogue` will similarly require resolution of the constraints on selection of its sub-features.

### 6.2.10 Specification Modelling of TDL Functions and Message Structure

The Specification model for the TDL Message Structures comprises the configuration model specified in Figure 123 and illustrated graphically in Figure 124, with a target architectural model. We derive a three layered architectural model similar to that described in Section 6.2.5 as illustrated in Figure 128, into which we wish to generate the specified TDL Functions and Message...
Structures. Once again, this target architecture is an instance of the layered architecture metamodel described in Section 5.3.2.

![Diagram of Layered Architecture](image)

**Figure 128 – Example TDL Function and Message Structure Architecture**

6.2.11 Product Generation of TDL Function and Message Structure

The product generation stage of the process takes the TDL Message Structure specification model described above and merges the TDL Message Structure configuration with the specified target architecture based on given correspondences. In this example we use a mapping from the source Link 16 packages shown in Figure 129 to the target architecture shown in Figure 128.

![Diagram of Mapping](image)

**Figure 129 – Link 16 Source Model Elements**

The mapping from source model elements to target architecture is shown in Figure 130; in this example we are merging the contents of the source packages `MessageConcepts` and `Messages` into a single target layer `MessageCatalogue`.
The model generated as a result of the merging of the source model elements with the target architectural blueprint is shown in Figure 131, demonstrating the successful generation of a custom model from a valid configuration of the TDL Functions feature model in a specified (and model-based) architectural style.
The contents of the layers Functions and MessageCatalogue are shown in Figure 132 and Figure 133, this shows that the models generated contain only those components selected from the source Link 16 model via the configuration model in Figure 123.
Figure 132 – Generated TDL Functions Components
Figure 133 – Generated TDL Message Catalogue Components
6.2.12 Related Work

Although there is evidence in the literature of ongoing work in both the UK and the US investigating the modelling of the semantics and behaviour of the Link 16 TDL (e.g. see [HJR07a], [Mak06], and [ZH07]), there does not appear to be any evidence of the application of the SPLA and/or feature modelling to analyses of the standard and support for generation of refinements of the standard based on custom configurations. The modelling described by Zeigler and Mak et al. relates to the behavioural modelling of the TDL to support executable simulations in the form of a testing oracle. This work is concentrated towards the back of the life-cycle and relies on the availability of an implemented TDL platform, such that this may be exercised against the oracle. The work reported by the author elsewhere (e.g. [HJR07a], [HJ08a]) seeks to derive models to describe the TDL to aid design activities undertaken during the early stages of the life-cycle, the application of a PLA approach to the TDL development has the potential to integrate well with this work.

The standard itself alludes to the prospect of the definition of platform-specific profiles in terms of a number of generic platform types, such as fighter, bomber, etc., however there is no evidence of the use of such profiles in the public domain; neither does the standard actually seek to refine the notion of a platform profile. Hence, it is concluded that the application of feature modelling to the representation of the minimum implementation requirements, supported by an appropriate metamodel, graphical concrete syntax and tool support would help to tackle the inherent ambiguity and inconsistency in this part of the standard and support interoperability at design time (i.e. early in the life-cycle). Furthermore, the expression of the TDL standard in the form of a PLA would provide the opportunity to generate custom models from specification that are themselves checkable by machine, thus the combination of the PLA approach and the model-based representation of the TDL standard as described in [HJR07a] and [HJ08a] would provide a powerful model generation capability.

Section 6.2 has demonstrated the application of a PLA approach to the modelling of a family of TDL systems leading the generation of custom models into a specified target architecture in the form of:

- A family of TDL Units (Section 6.2.2)
- A family of TDL Functions (Section 6.2.7)

Section 6.3 will describe the results of a separate case study applying the PLA to the domain of the avionics navigation system.
6.3 **Avionics Navigation System**

This section reports the results of a second case study, this time applying the PLA models described in Chapter 5 to the domain of the avionics navigation system. This case study represents an additional demonstration of the exercising of the PLA in a domain that is not related directly to that reported above in Section 6.2. The case study begins with a brief introduction to the navigation domain, a more detailed description is provided by [HE03b].

6.3.1 **Introduction to the Navigation Domain**

A typical avionics mission system encompasses a number of domains as illustrated in Table 3. One of the core domains is navigation; this domain is typically responsible for identifying the state vector of the vehicle. The generality of the navigation domain is fundamental to the operation of avionics mission systems and is sometimes referred to as the *avionics backplane*, it is also central to vehicles operating in a broad range of environments [CGK+93] (Figure 134).

![Figure 134 – Navigation Domain Applicability](image)

It is difficult to consider the Navigation domain in isolation; whilst it seeks to answer the fundamental question *'where are we?'*, it requires the services of lower-level domains to acquire the relevant source data, and it provides services to higher-level domains, such as Mission Management and Collision Avoidance. Certain aspects of variability exhibited by the Navigation domain may be considered obscure or esoteric for the purposes of a case study, such as the decision of whether to use Euler angles, Quaternions or Direction Cosines for determining attitude and heading (e.g. see [TW97b] for examples). Hence, in order to provide sufficient diversity in requirements at a user-visible level, we have elected to investigate the development of a thin vertical slice of functionality that will require the collaboration of a number of domains whilst retaining the focus largely within ‘Navigation’. The domains to be investigated are:

- Sensors: *environment interface*
- Navigation: *where are we?*
- Mission Management: *where do we want to go?*
- Man-Machine Interface: *we would like to render views of the system state*

The above domains are shown within the context of a domain model in Figure 135.
<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigation</td>
<td>Provides the air vehicle state vector and seeks to answer the question ‘where are we?’</td>
</tr>
<tr>
<td>Maintenance</td>
<td>System/sub-system health monitoring, covering the various forms of Built In Test and providing fault logs</td>
</tr>
<tr>
<td>Reconnaissance</td>
<td>Provides control of the various reconnaissance features of the air vehicle, including sensor management and navigational search patterns</td>
</tr>
<tr>
<td>Attack</td>
<td>Provides control of the air vehicle’s weapon suite</td>
</tr>
<tr>
<td>Mission Management</td>
<td>Provides the pilot with such services as route planning and management of the flight plan, may also provide control of the engine(s) and air vehicle; seeks to answer the question ‘where do we want to go?’ May also provide pre-planned manoeuvres (e.g. take-off, landing, go-around)</td>
</tr>
<tr>
<td>Defensive</td>
<td>Provides the pilot with access to counter measures and evasive manoeuvres</td>
</tr>
<tr>
<td>Communications</td>
<td>Communication links (both voice and data) for the air vehicle, its crew, and systems</td>
</tr>
<tr>
<td>Vehicle Control</td>
<td>Provides the aircraft with some level of autonomous control (e.g. autopilot and stability augmentation)</td>
</tr>
<tr>
<td>Sensor</td>
<td>Provides the air vehicle with standardised interfaces to common sub-systems (e.g. INS, GPS, ADC, etc.)</td>
</tr>
<tr>
<td>Situational Awareness</td>
<td>Provides the air vehicle and its crew with a view of both friendly and hostile forces in relation to present position and flight plan</td>
</tr>
<tr>
<td>Engine Control</td>
<td>Provides services such as throttle management, reverse thrust, afterburner, etc.</td>
</tr>
<tr>
<td>Collision Avoidance</td>
<td>Predicts impending collision with other air vehicles and/or the terrain and provides avoidance manoeuvre information</td>
</tr>
<tr>
<td>Weather Detection</td>
<td>Provides monitoring of the weather along the air vehicle’s intended track</td>
</tr>
<tr>
<td>Formation Manoeuvres</td>
<td>Provides steering information to allow the air vehicle to hold station relative to one or more other air vehicles</td>
</tr>
<tr>
<td>Man-Machine Interface</td>
<td>Provides the user interface</td>
</tr>
</tbody>
</table>

Table 3 – Avionics Mission System Domains
The ADAGE project placed much supporting information into the public domain on the modelling of navigation in the avionics domain; hence it is also believed to be one of the least contentious of the domains that could have been chosen.

6.3.2 Domain Engineering of the Navigation Function

The navigation domain determines the position and velocity of a vehicle, the general term for the position and velocity within a defined frame of reference is the state vector. Calculation of the state vector of an air-vehicle requires data about the environment in which the vehicle is operating, e.g. air temperature, wind speed and direction, etc.; such data is provided by components in the sensor domain, which is itself decomposed into a number of sub-domains (e.g. inertial sensors). Having established the state vector of the air vehicle from sensor data the navigation domain will need to do something with this information. Whilst it’s useful to know where the air vehicle is in space, it’s also useful to know the state vectors of other objects, such as waypoints, other vehicles, etc. Having established a view of the vehicle and the world around it, other domains will need to make use of this information to achieve some higher-level function, e.g. given that we know the state vector of an air vehicle and (say) that of an aircraft carrier we may wish to provide steering information to guide the air vehicle to the aircraft carrier. In this example, the mission management domain would use data provided by the navigation domain to calculate steering information, which, in turn, would
be used by the MMI domain to provide the pilot with steering cues in the cockpit (assuming it is a manned aircraft). Each of the domains involved in supplying data to, or consuming data from the navigation domain does itself exhibit variability, e.g. different types of sensor, various steering strategies, and display configurations. The analysis of the domains comprising the hypothetical Avionics Navigation System is described in Appendix 9; the analysis provides the scope for the product family and forms the basis for the development and modelling of the product line.

6.3.3 Feature Modelling of Navigation

For the purposes of this case study based on the navigation function it is necessary to consider the following domains in order to provide a representative slice of functionality:

- Sensors
- Navigation
- Mission Management
- Man-Machine Interface

The Mission System Domain Model illustrated in Figure 135 shows the top-level package hierarchy view of the components found in a typical mission system (note that the MMI layer will generally have visibility of all subordinate layers). Hence, in order to provide an example system based on the navigation domain, it is necessary to provide the navigation component with access to sensors to provide input data (present position, air speed, heading, etc.). In order to demonstrate a system in which the results of the navigation domain are used to determine a route to steer we also consider the mission management domain. Finally, we consider the MMI domain in order to allow the rendering of the steering information, thereby providing a thin vertical slice of functionality for an example mission system.

A feature model of an example avionic system supporting the navigation function may be composed from the feature models developed for each of the above domains, as illustrated in Figure 136. In this example we allocate features to domains as described in Table 4, and describe each of the four component feature models in greater detail in the sections that follow.
The root feature (Avionic System) is defined as having a single variation point relating it to each of the features illustrated in column 3 of Table 4, such that any valid configuration must comprise the features: Sensors, Earth Models, Atmosphere Model, Transient Objects Model, and Air Vehicle Model; the features Steering Model and Display Formats are left as optional.

In the sections that follow we will describe the decomposition of each of the top-level features as identified in Table 4, the complete listing of the Navigation feature model is provided at Appendix 11.

<table>
<thead>
<tr>
<th>No.</th>
<th>Domain</th>
<th>Top-Level Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Sensors</td>
<td>Sensors</td>
</tr>
<tr>
<td>2.</td>
<td>Navigation</td>
<td>Earth Models</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Atmosphere Models</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transient Object Models</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Air Vehicle Model</td>
</tr>
<tr>
<td>3.</td>
<td>Mission Management</td>
<td>Steering Model</td>
</tr>
<tr>
<td>4.</td>
<td>Man-Machine Interface</td>
<td>Display Formats</td>
</tr>
</tbody>
</table>

Table 4 – Domain to Feature Mapping
Figure 136 – Top-Level Feature Model of the Navigation Function
6.3.3.1 Feature Modelling of the Sensors Domain

The Sensors domain is partitioned following the guidance of Coglianese et al. [CGK+93], the resulting feature model is shown in the textual concrete syntax in Figure 137, and graphically in Figure 140. The textual description of the feature model may be parsed into the XMF tool and the instance validated against the constraints on the feature model description – the instance sensorsFM is valid.

```
<imports elided>
Root::sensorsFM :=
  @FeatureModel(FeatureModel_Sensors)
    Root Feature(Sensors) has a {AirData, Inertial, Database,
    ElectromagneticExternal, ElectromagneticInternal};

Mandatory Feature(AirData) is one of {AirResearch_CP1471A, MyADC};
Mandatory Feature(AirResearch_CP1471A) is leaf;
Mandatory Feature(MyADC) is leaf;

Mandatory Feature(Inertial) is one of {INS, AHRS};
Mandatory Feature(INS) is one of {FIN_1075, ASN_130, LN_100};
Mandatory Feature(FIN_1075) is leaf;
Mandatory Feature(ASN_130) is leaf;
Mandatory Feature(LN_100) is leaf;
Mandatory Feature(AHRS) is one of {AHRS_1, AHRS_2};
Mandatory Feature(AHRS_1) is leaf;
Mandatory Feature(AHRS_2) is leaf;

Optional Feature(ElectromagneticExternal) has a {TACAN, GPS};
Optional Feature(TACAN) is leaf;
Optional Feature(GPS) is one of {GPS_1, GPS_2};
Mandatory Feature(GPS_1) is leaf;
Mandatory Feature(GPS_2) is leaf;

Mandatory Feature(ElectromagneticInternal) has a {Radalt, DNS};
Mandatory Feature(Radalt) is leaf;
Optional Feature(DNS) is leaf;

Optional Feature(Database) has a {TERPROM};
Optional Feature(TERPROM) is leaf;

Feature TERPROM requires {Radalt, INS};
end;
```

Figure 137 – Sensors Feature Model (Concrete Syntax)

From this feature model we can see a one-to-one correspondence between the concrete application classes described in Figure 179 through Figure 185, however the component interdependencies are made more explicit; e.g. selection of the TERPROM feature requires selection of both the Radalt and INS features, and selection of the INS feature requires selection of any one of the set of alternative INS platforms, hence we couldn’t select the TERPROM feature in conjunction with an AHRS feature.
In the Sensors feature model we wish to mandate the selection of one Air Data Computer (either an Air Research CP1471A or a MyADC). We also wish to mandate selection of an Inertial feature, this feature is decomposed into two sets of alternatives, either an AHRS feature or an INS feature; in each case there are a set of alternative devices from which to choose. Selection of the External Electromagnetic equipment feature is optional, however if we choose to select the GPS feature then we must select either of the alternative GPS equipments available. Selection of the Internal Electromagnetic equipment feature is mandatory, as is selection of the Radalt sub-feature; however selection of the DNS feature is optional. Finally, selection of the TERPROM database sensor feature is optional and has been discussed above.

6.3.3.2 Feature Modelling of the Navigation Domain

The Navigation domain has been partitioned across the set of features as shown in Table 4 (Earth Models, Atmosphere Model, Transient Objects Model, and Air Vehicle Model). The Atmosphere feature is a mandatory leaf, its selection requires selection of both the Air Data and Inertial Sensor features (Figure 138).

```plaintext
// Atmosphere Model
Mandatory Feature(AtmosphereModel) is leaf;
// Feature Dependencies
Feature AtmosphereModel requires {AirData, Inertial};
end;
```

Figure 138 – Atmosphere Model Feature Description

The Air Vehicle Feature is similarly straight forward, it is mandatory and requires selection of the sub-feature Main Air Vehicle Components and allows the optional selection of the sub-feature Flight Control Surfaces (Figure 139).

```plaintext
// Air Vehicle Model
Mandatory Feature(AirVehicleModel) has a {MainAirVehicleComponents, FlightControlSurfaces};
Mandatory Feature(MainAirVehicleComponents) is leaf;
Optional Feature(FlightControlSurfaces) is leaf;
```

Figure 139 – Air Vehicle Model Feature Description

Whilst the feature Atmosphere is bound to the single application class AtmosphereModel (Figure 187), the features Main Air Vehicle Components and Flight Control Surfaces are bound to multiple model elements as illustrated in Figure 141.
Figure 140 – Sensors Feature Model
Figure 141 – Air Vehicle Feature to Model Element Mapping
The Earth Models feature is decomposed as illustrated in Figure 143. This feature diagram shows that selection of the Earth Models feature (which is mandatory) requires selection of the Ellipsoidal Models sub-feature. This sub-feature is interesting because it has two dimensions, one is a variation point linked to the mandatory sub-feature WGS 1984, the other is an optional inclusive-or relation linked to a set of ellipsoidal models. Hence, whilst selection of the WGS 1984 sub-feature is mandatory, selection of any number of the ellipsoidal models (International, Airy, Clarke, and Everest) is optional (Figure 142).

Mandatory Feature(EllipsoidalModels) has a {WGS_1984} and is any combination of {International_1924, Airy_1830, Clarke_1880, Everest_1830};

Mandatory Feature(WGS_1984) is leaf;
Optional Feature(International_1924) is leaf;
Optional Feature(Airy_1830) is leaf;
Optional Feature(Clarke_1880) is leaf;
Optional Feature(Everest_1830) is leaf;

Figure 142 – Feature Description of Ellipsoidal Models

Selection of the Local Geodetic Datum feature is optional, however it’s selection requires the corresponding selection of either one of the optional Datum Transformation features (Helmert Formula or Standard Molodenski), furthermore it’s selection also mandates the selection of at least one of the Local Geodetic Datum sub-features (Indian, ED50, and OSGB36).
Earth Models

- Datum Transformation
  - Standard Molodenski
  - Helmert Formula

- Local Geodetic Datum
  - Indian
  - ED50
  - OSGB36

- Ellipsoidal Models
  - International 1924
  - Airy 1830
  - Clarke 1880
  - Everest 1830
  - WGS 1984

Figure 143 – Earth Models Feature Model
The Transient Objects Model provides the features relating to the kinds of transient object relevant to the navigation domain (Waypoints, Targets, and Beacons), and also provides features relating to the management of collections of such objects. The feature Beacon is decomposed into the sub-features TACAN Station and VOR Ground Station; however the Beacon Database feature is assumed to support heterogeneous collections of Beacon objects. The selection of at least one transient object type is mandatory, hence we model the sub-features of Transient Object as being mandatory or-features, the decomposition of the sub-feature Beacon is structured similarly such that selection of the Beacon feature mandates the selection of either the TACAN or VOR sub-features (or both). Selection of the Transient Object Database features is optional, however, if chosen, at least one of the database sub-features must also be selected, as must the relevant Transient Object. The feature model for the Transient Objects Model is described textually in Figure 144.

```plaintext
<imports elided>
Root::transientObjectsModelFM :=
  @FeatureModel(FeatureModel_TransientObjectsModel)
  Root Feature(TransientObjectsModel) has a {TransientObject, TransientObjectDatabase};
  Mandatory Feature(TransientObject) is any combination of {Beacon, Waypoint, Target};
  Mandatory Feature(Beacon) is any combination of {TacanStation, VorGroundStation};
  Mandatory Feature(TacanStation) is leaf;
  Mandatory Feature(VorGroundStation) is leaf;
  Mandatory Feature(Waypoint) is leaf;
  Mandatory Feature(Target) is leaf;
  Optional Feature(TransientObjectDatabase) is any combination of {BeaconDatabase, WaypointDatabase, TargetDatabase};
  Mandatory Feature(BeaconDatabase) is leaf;
  Mandatory Feature(WaypointDatabase) is leaf;
  Mandatory Feature(TargetDatabase) is leaf;
  Feature BeaconDatabase requires {Beacon};
  Feature WaypointDatabase requires {Waypoint};
  Feature TargetDatabase requires {Target};
end;

Figure 144 – Feature Description of Transient Objects Model
```

The instance of the feature model created via the concrete syntax (above) is valid with respect to the constraints imposed on the feature model and is illustrated graphically in Figure 145.
6.3.3.3 Feature Modelling of the Mission Management Domain

The Mission Management domain is populated by only one feature `SteeringModel`, this optional feature provides access to the various steering missions we wish to support in the navigation PLA. The steering missions are partitioned into three mandatory inclusive-or sub-features, such that selection of the optional feature `SteeringModel` requires that at least one of the sub-features is also selected. The sub-feature `FlightAids` is decomposed further via the optional sub-feature `HoverHold`; we separate out this service as a sub-feature as it will not be appropriate for platforms that are unable to hover. Each of the top-level steering models requires the services of features from the Transient Objects Model; these are required in order to provide the steering missions with an appropriate state vector, and, in the case of the features Search Patterns and Take-Off and Landing a POI that is to provide the objective for the mission, e.g. the POI for a search pattern will be a Target and the POI for the Take-Off and Landing steering model will be a Waypoint. Figure 146 describes the feature model for the steering models.
Optional Feature(SteeringModel)
is any combination of {FlightAids, SearchPatterns, TakeOffAndLanding};

Mandatory Feature(FlightAids) has a {HoverHold};
Optional Feature(HoverHold) is leaf;
Mandatory Feature(SearchPatterns) is leaf;
Mandatory Feature(TakeOffAndLanding) is leaf;
Feature FlightAids requires {AirVehicleModel};
Feature SearchPatterns requires {AirVehicleModel, TargetDatabase};
Feature TakeOffAndLanding requires {AirVehicleModel, WaypointDatabase};

Figure 146 – Feature Description of Mission Management (Steering Model)

The feature model for the steering models to be supported is illustrated graphically in Figure 147.
Figure 148 – Mapping Flight Aids Features to Model Elements

This figure shows that the mandatory feature Flight Aids is mapped to the classes FlightAid and Cruise, whilst the feature Hover Hold is mapped to the class HoverHold. Since the feature Hover Hold is a variation of the feature Flight Aids, the superclass FlightAid will always be present in any valid configuration of the feature model.

Figure 149 – Mapping Search Patterns to Model Elements

This figure shows that the feature Search Patterns is mapped to the collection of classes (model elements) providing the search patterns supported by the domain model (ExpandingSquare, CreepingLine, SectorSearch, and the superclass SearchPattern).
Figure 150 – Mapping Take-Off and Landing to Model Elements

This figure shows that the feature Take-Off and Landing is mapped to the set of classes relating to take-off and landing steering missions in the domain model.

6.3.3.4 Feature Modelling of the MMI Domain

The MMI domain is populated by only one feature `DisplayFormats`, this optional feature provides access to the various navigation-related display formats we wish to support in the PLA. The Display Formats feature is optional and comprises a small set of features (one feature per display format) grouped as a set of mandatory inclusive-or sub-features, such that selection of the optional Display Format feature mandates the selection of at least one display format sub-feature. The feature model for display formats is described below in Figure 151 and rendered graphically in Figure 152.

```
<elided>
Optional Feature(DisplayFormats)
  is any combination of
    {AtmosphereDisplay, SteeringDisplay,
     BeaconDisplay, WaypointDisplay, TargetDisplay};

Mandatory Feature(AtmosphereDisplay) is leaf;
Mandatory Feature(SteeringDisplay) is leaf;
Mandatory Feature(BeaconDisplay) is leaf;
Mandatory Feature(WaypointDisplay) is leaf;
Mandatory Feature(TargetDisplay) is leaf;
<elided>

Feature AtmosphereDisplay requires {AtmosphereModel};
Feature SteeringDisplay requires {SteeringModel};
Feature BeaconDisplay requires {BeaconDatabase};
Feature WaypointDisplay requires {WaypointDatabase};
Feature TargetDisplay requires {TargetDatabase};
<elided>
```

Figure 151 – Feature Description of Display Formats
Figure 152 – Display Formats Feature Model
Each of the display format sub-features requires the services of a lower-level feature to provide the necessary source information to be rendered to the user, e.g. selection of the Waypoint Format feature requires that the Waypoint Database feature is also selected, etc.

We have now completed the description of a relatively large and complex feature model describing the components required to support a family of avionics navigation systems that is declared valid against the constraints of the feature metamodel (Section 5.1). The following sections will describe the configuration of a particular family member and then generate the relevant model elements in the form of a specified architecture.

6.3.4 Configuration Modelling of Navigation

The feature model for a family of avionics navigation systems may now be used as the basis from which we may configure a specific family member providing a defined suite of functionality. As an example we will configure an avionics navigation system with the following features:

- **MMI:**
  - Steering Display
  - Waypoint Display
- **Steering Model:**
  - Flight Aids
  - Take-Off and Landing
- **Air Vehicle Model**
  - Main Air Vehicle Components
- **Transient Objects Model:**
  - Waypoint
  - Waypoint Database
- **Atmosphere Model**
- **Earth Models:**
  - WGS84
  - Standard Molodenski LGD to WGS84 transformation
  - ED50 Local Geodetic Datum
  - International 1924 Ellipsoidal Model
- **Sensors:**
  - Air Research CP1471A Air Data Computer
  - FIN 1075 Inertial Navigation System
  - Radar Altimeter
The desired configuration is expressed via the textual concrete syntax in Figure 153, and validated against the constraints defined for the Configuration metamodel (Section 5.2.4.3).

```plaintext
<imports elided>
Root::avionicsNavigationSystemCM :=
@ConfigurationModel(ConfigurationModel_AvionicsNavigationSystem)
  From navigationFM select
   { AvionicSystem,
     DisplayFormats,
     SteeringDisplay,
     WaypointDisplay,
     SteeringModel,
     FlightAids,
     TakeOffAndLanding,
     AirVehicleModel,
     MainAirVehicleComponents,
     TransientObjectsModel,
     TransientObject,
     Waypoint,
     TransientObjectDatabase,
     WaypointDatabase,
     AtmosphereModel,
     EarthModels,
     DatumTransformation,
     StandardMolodenski,
     LocalGeodeticDatum,
     ED50,
     EllipsoidalModels,
     WGS_1984,
     International_1924,
     Sensors,
     AirData,
     AirResearch_CP1471A,
     Inertial,
     INS,
     FIN_1075,
     Electromagnetic_Internal,
     Radalt
   };
end;
```

**Figure 153 – Example Avionics Navigation System Configuration**

The configuration model is shown graphically in Figure 156, as a result of the size of the feature model we illustrate only those features selected by the above configuration description. We could specify an invalid configuration of an avionics navigation system in the form shown in Figure 154, and then exercise the model constraints to verify that the erroneous instance configuration model is identified against the composition rules described by the feature model described in Section 6.3.3.

Figure 157 illustrates the erroneous features included in the configuration shaded in red:

- INS sub-features FIN 1075 & ASN 130 are selected, however they are defined as alternatives
- feature International 1924 is not selected, however it is required by feature ED50
- feature Display Formats is selected, however none of its mandatory inclusive-or sub-features are selected
Figure 154 – Avionics Navigation System Configuration (Error)

Exercising the above configuration against the constraints of the feature relationships described by the feature model of the family of Avionics Navigation Systems provides the constraint report shown in Figure 155. The constraint report correctly identifies the three categories of error introduced by the configuration model as:

1. MandatoryInclusiveOrSubFeatureSelectionCardinality – the violation of this constraint is caused by the failure to select at least one of the Display Formats sub-features, such as the feature SteeringDisplay.

2. MandatoryExclusiveOrSubFeatureSelectionCardinality – the violation of this constraint is caused by the selection of more than one INS sub-feature, one, and only one, sub-feature should be selected.

3. AllDependenciesSatisfied – the violation of this constraint is caused by the failure to select the ellipsoidal model feature International 1924; this feature is required by the local geodetic datum feature ED50 (which has been selected).
Figure 155 – Avionics Navigation System Configuration Error (Constraint Report)
Figure 156 – Avionics Navigation System Configuration
Figure 157 – Erroneous Configuration Model for Avionics Navigation System
6.3.5 Specification Modelling of Navigation

The Specification model for the Avionics Navigation System comprises the configuration model specified in Figure 153 and illustrated graphically in Figure 156, with a target architectural model. We derive a four layered architectural model from the five-layer Mission System Domain model illustrated in Figure 135 (which is itself derived from the ADAGE architecture, e.g. see [BCG+95]). This architectural model is to be the target structure into which we wish to generate the specified Avionic Navigation System components. In this case study we do not require layer 4 of the Mission System Domain model, hence the target architecture comprises the four main layers shown in Figure 158. This target architecture is an instance of the layered architecture metamodel described in Section 5.3.2.

![Figure 158 – Example Avionics Navigation System Architecture](image)

The size of the model for the Avionics Navigation System is somewhat larger than that of the previous case study (Section 6.2), hence it is prudent for the architectural model to be decomposed into nested sub-layers; each of the main layers is decomposed as illustrated in Figure 159.

![Figure 159 – Partitioning of Avionics Navigation System Architecture](image)
6.3.6 Product Generation of Navigation

The product generation stage of the process takes the Avionics Navigation System specification model described above and merges the configuration with the specified target architecture based on given correspondences. In this example we use a mapping from the source Avionics Components packages (the kit of reusable parts) shown in Figure 160 to the target architecture shown in Figure 159.

![Diagram of Avionics Components Source Model Elements]

Figure 160 – Avionics Components Source Model Elements

The mapping from source model elements to target architecture is shown in Figure 161; in this example we are repartitioning the contents of the source model packages (which may themselves be nested) into nested layers within the target architecture, Figure 162 and Figure 163 provides a more detailed view of the transformations applied to the source model package SteeringModels. In this example the feature SearchPatterns is not selected in the
configuration model, hence it is not propagated to the generated model component. Furthermore, the optional flight aids component HoverHold is not selected, so this too is not propagated to the generated model component package MissionManagement (Figure 163).
Figure 161 – Avionics Components Model Element to Architecture Mapping
Figure 162 – Source Component Package Steering Models
Figure 163 – Generated Component Mission Management
The model generated as a result of the merging of the selected source model elements with the target architectural blueprint is fairly large, the outline package structure is shown in Figure 164; the relevant class diagrams for the complete model are contained at Appendix 12.

---

**Figure 164 – Generated Avionics Navigation System**

This case study has demonstrated the successful generation of a custom model of an Avionics Navigation System in a specified (and model-based) architectural style. The model has been generated from a feature model describing the variability on the domain that is linked to a kit of parts in the form of a domain model of Navigation. The configuration model identified a coherent set of features that were to be used, and the specification model combined this configuration model
with the specified architectural blueprint, thereby raising reuse to the model level and
demonstrating that a model-based PLA of the navigation domain is feasible.

6.3.7 Related Work

The models described above have been developed to validate and demonstrate the application of
the model-based PLA to the navigation domain. Applying a PLA approach to avionics and the
navigation domain is not new; some of the domain analysis and architectural styles presented above
have been derived from existing work in the literature, most notably that of the ADAGE project
[CGK+93], although such material rarely delves deeper into the system than a description of
relatively abstract models and definitions of component families expressed in the form of (e.g.)
GenVoca type equations. However, the layered architectural style resonates well with work
published on other embedded mission systems [Hol01a] and gives confidence of its general
applicability. The work presented above is novel because it raises the PLA of an avionics
navigation system to the model space, and divorces the implicit link between implementation
architecture and the components of the product family to provide an additional dimension of
flexibility and is consistent with recent model-driven approaches to system development (see
Section 2.3). The models describing the avionics navigation system PLA are underpinned by
metamodels (described in Chapter 5) that may be implemented in tools (such as XMF-Mosaic) to
support well-formedness constraint checks and provide model generation capabilities to facilitate
the fabrication of custom models from a kit of parts.

6.4 Evaluation of Case Studies

This chapter has described the application of the SPLA as described in Chapter 5 to case studies
derived from the domain of Tactical Data Links and Avionics Navigation Systems. The first case
study relating to the TDL domain comprised two examples, one investigated the application of the
PLA to the definition of a family of TDL Units which represents a fairly obvious candidate
problem area, whilst the other (larger) example investigated the application of the PLA to the
configuration and generation of models of the TDL functions and their related message catalogue
components. There does not appear to be any comparable work available of which the author is
aware relating to the application of the PLA to the TDL domain. The second case study described
the application of the PLA metamodels to the development of a model of a family of configurable
avionics navigation systems, family-based modelling approaches have been used in this domain
before and are reported elsewhere, e.g. see [BCG+95], however these approaches were not derived
from a feature oriented view of the domain, and, more importantly, were not based on an explicit
model of the PLA and did not support the generation of custom models from an architecture
agnostic kit of parts.
The success criteria for the application of the meta-modelled approach to the definition of the SPLA was that the technology should be *domain agnostic*, such that it should be applicable to both domains chosen. The case study describing the application of the SPLA to the domain of Navigation systems is not without precedent, and much has been derived from the literature; especially work relating to the ADAGE programme. It should be noted that whilst the ADAGE programme developed a domain model (which we have adapted), it did not discuss the identification of valid configurations, being based on type compatibility rather than semantic compatibility. The case study relating to the application of the SPLA to the domain of the Link 16 TDL standard appears to be without precedent.

Both case studies proved useful aids to the debugging and validation of the underlying metamodels and demonstrate the successful application of the SPLA within the respective domains, leading to the generation of custom models based on the definition of configuration models and specified target architectural styles. Whilst each of the two cases studies demonstrated the generation of custom models implemented in different target architectures, both case studies use the (metamodel-based) layered architectural profile described in Section 5.3.2. The differences in target architectures is restricted to the number and nesting of layers used, it would be beneficial for future work to apply the same SPLAs to alternative (metamodelled) target architectures, e.g. IMS or DDS.

Product generation from the Configuration model of a particular member (or profile) of the family of Link 16 TDL standards is likely to result in the generation of a model corresponding to the same schema as the model to which the candidate features are linked. Hence, the product generation mappings are likely to be somewhat more straight-forward – i.e. the result of the product generation for a particular profile would be a restriction in the variability admitted by the input standard, although such a restriction of variability would be guaranteed to be valid with respect to the constraints of the feature model. An additional requirement derived from this example is the need to capture the notion of a *product profile* explicitly; this is similar to the description of a line provided by Czarnecki & Eisenecker [CE99], although not expressed as C++ templates.

The TDL standard [DoD04] advocates the definition of the implementation requirements of a TDL platform via a number of refinements (Figure 165), of which the standard itself is the most abstract. National requirements would (say) provide a UK-specific set of TDL requirements, whilst the Service requirements would refine this to (say) a Naval view of the TDL requirements, before, finally, refining these down to the actual platform requirements. This goal resonates well with that of the Staged Configuration approach described by Czarnecki et al. [CHE05], to support such an approach to the definition of the PLA we would need to extend our metamodel of features to provide traceability between each successive refinement of the feature model and allow the configuration model to define a refinement of the feature model that retains variability. It is noted
that whilst the TDL standard advocates a staged configuration-like approach to the definition of platform requirements we are not aware of any project that follows this practice explicitly, projects generally refine the standard into the platform requirements in one step.

![Figure 165 – Refinement of TDL Platform Requirements](image)

The application of the feature model to the representation of the minimum implementation requirements of the Link 16 functions revealed a requirement to support a more complex form of feature dependency, such that a feature $f_1$ could depend on any one (or more) of a set of features elsewhere in the model $f_m \ldots f_n$, such that a valid configuration comprising $f_1$ must also comprise some subset of the set of dependent features $f_m \ldots f_n$. There is some resonance with the approach taken by Hein, et al. in the provision of cross-links [HSV00], allowing a sub-feature to play the role of the root feature in a secondary feature tree.

Another issue apparent on the modelling of the Link 16 TDL Unit family is the concise nature of the feature model in the form of a table. Whilst the table-based view provided by the standard and augmented at Table 6 lacks some structure (some features could be grouped for simplicity and to aid readability) it is clearly an appropriate medium in this instance, hence it may be appropriate to consider the rendering of the feature model both in graphical form (e.g. as per Czarnecki & Eisenecker [CE00]) and also tabular form – this could be achieved via the transformation of the feature model into a document model and then into XML for further transformation by XSLT into HTML and rendering by a standard web-browser (Figure 18 provides an illustration of such a transformational approach within a different context).

When recapturing the function to J-Message minimum implementation requirements, it became apparent that there is a need for an executable constraint language to express conditions under
which certain features are to be selected (or excluded). Whilst one could define separate feature
models for specific platform types (e.g. C² and non-C² platforms), it would (in some areas) be more
natural to annotate certain features as being applicable to one or other of the types. This could be
achieved via the expression of constraints directly in XOCL, or via a bespoke (modelled) constraint
language that could be executed in an interpretive style\textsuperscript{41}; section 2.2.4 provides an example of a
model of a fragment of such a language.

Whilst developing example models for both case studies a situation was encountered where the
feature/sub-feature relationship was of the type mandatory inclusive-or (e.g. see the Surveillance
Message family in Figure 202), this relationship means that if we choose to include the feature
Surveillance Message, then we must also include at least one of the set of inclusive-or sub-features.
However, if there happens to be only one sub-feature, as in the case of the Antisubmarine Warfare
Message family (see Appendix 10 Figure 204) then this has the semantics such that if we choose to
select the feature Antisubmarine Warfare Message, then we must also select its one and only sub-
feature Acoustic Range And Bearing Message; thus the semantics of the mandatory inclusive-or
sub-feature is equivalent to that of a variation point if there is only one mandatory sub-feature
present (Figure 166); in fact the semantics are equivalent even if the sub-feature is optional.

\textsuperscript{41}Use of XOCL as an interpreter isn’t mandatory, but it is a relatively simple solution.
Figure 166 – Inclusive-Or versus Variation Point Semantics
Following the approach of Czarnecki & Eisenecker (e.g. [CE00]), there would be no such semantic equivalence because the authors attach the semantics of programming language constructs to the feature/sub-feature relationships, e.g. an inclusive-or would be taken to mean an inheritance relation between feature and sub-feature, and a variation point would be taken to mean an association relation – the authors do this because their goal is code generation. In the case described above, the semantics are equivalent because we have divested the feature model with the responsibility of modelling the relationship between model elements, such relationships remain in the domain model to which the feature model links, and the semantics of the feature/sub-feature relationship is restricted to describing how one feature is related to another, rather than describing how it will be implemented, hence this is a degenerate case. Choice of whether to use a variation point or an inclusive-or relation should be driven by the feature modelling process, e.g. there is currently only one antisubmarine message (J5.4), however if another message were to be added to the catalogue then any redundancy in the semantics would be resolved.

It can be seen from the second case study (avionics navigation system) that the feature models can become large, especially when expressed using the graphical concrete syntax. It would be beneficial if support for the structuring of feature models were to be provided; which may or may not affect the abstract syntax. The metamodel for features has been demonstrated to be able to support large feature models, hence one could simply conclude that this is an issue for the rendering of the feature model, however metamodelled support for modular feature models would be beneficial for at least two reasons: (1) larger feature models are more difficult to understand and debug, and (2) monolithic feature models are more likely to encounter name clashes, hence management of the namespace would be a useful extension. However, feature models tend to exhibit global dependencies between clusters of features, and this is likely to complicate the metamodel somewhat.

Finally, the simple textual concrete syntax developed to provide a concise mechanism for specifying feature models (see Appendix 3) does not provide for the linkage between features and model elements, although the abstract syntax does support such relationships. Hence, in order to demonstrate the model generation capabilities fully, it has been necessary to develop the feature models in the abstract syntax which is somewhat more verbose than the equivalent specification in the concrete syntax. Hence, the concrete syntax should be extended to allow statements of the form:

```
Feature (name:String) is leaf
traces to {Set: (ModelElementName:String)}
```

Whilst not particularly difficult, this remains to be implemented in the feature model CS parser.
7 Conclusions and Future Work

This section provides a summary of the work undertaken and seeks to address the three fundamental issues relating to any thesis; (1) in what way is the approach novel and how does it contribute to the understanding of the domain, (2) if this work were to undertaken again what would (or should) be done differently, and, (3) given the outputs of the work, what are the opportunities for further work that may be envisaged.

7.1 Evaluation of Hypotheses

The hypotheses investigated by this thesis are described in section 1.3.2. This section summarises each hypothesis and then describes how it has been satisfied by the work presented in chapters 4, 5, and 6.

7.1.1 A Metamodelled Foundation for the SPLA

The first hypothesis proposed at section 1.3.2 is restated below:

It is asserted that the notion of the SPLA is not well-founded in the literature. In particular, the feature model appears to perform multiple roles, defining the relationship between the rather abstract concepts represented by features whilst also implicitly defining the relationship between domain elements. This is seen as blurring the distinction between the problem space (the feature model) and the solution space (the domain model). The thesis will investigate the applicability of a metamodelled description of the SPLA following the modular design illustrated by the package diagram at Figure 4. The SPLA metamodel will comprise a number of related components with the intention of providing a modular and coherent description, such that the overloaded purpose we see reported in the literature regarding the feature model is removed, resulting in an SPLA metamodel exhibiting a strong separation of concerns. The hypothesis will be tested by the development of metamodels elaborating the packages illustrated in Figure 4 (see chapters 4 and 5), and the subsequent application within two case studies drawn from different domains (see chapter 6).

A metamodel for the SPLA comprising a number of coherent sub-components has been described to increasing degrees of rigour in chapters 4 and 5. The top-level view of the SPLA metamodel is provided in Figure 4.

The SPLA metamodel hierarchy confirms the hypothesis that a metamodelling approach supports the separation of concerns required to tease apart the overloaded role of the feature diagram by providing direct linkage from the feature model to artefacts from the domain model via the association components between the classes Feature and Element (see Figure 54).
Successful linkage of the feature model to one or more domain concepts has been demonstrated in both the case studies described in chapter 6, a fragment of which is presented in Figure 141.

The model fragment illustrated by Figure 141 shows that the mandatory feature AirVehicle is related to the domain concept (component) AirVehicle, the mandatory sub-feature Main Air Vehicle Components is related to the collection of domain concepts: Fuselage, Undercarriage, Engine, and FuelTank, whilst the optional sub-feature Flight Control Surfaces is related to the collection of domain concepts: FlightControlSurfaces, Aileron, Flap, and Rudder. Hence, it can be seen that a feature may be related to zero or more components in the domain model. The feature model says nothing about the relationship between elements in the domain model, this is an implementation detail. Instead the feature model simply allows the user to specify the relationship between features in the form of the following: VariationPoint, InclusiveOr, Alternative, and identify any dependent features; the user may also specify the optionality of a feature (optional or mandatory); these are the only concepts that have been found to be required to describe a feature model. Furthermore, the metamodel of the feature model (Figure 66) is restricted to providing only the elements required to instantiate and validate a feature model; it says nothing about how the feature model should be used when configuring an individual family member from the space of systems admitted by the feature model; hence the need for a strong separation of concerns has been demonstrated by the feature metamodel.

The instance of the configuration metamodel (Figure 84) provides the semantic domain for the feature model. It provides the necessary elements required to describe the configuration of one particular member of the family of systems admitted by the feature model. The configuration model provides a number of constraints that are required to determine the validity of any given configuration with respect to the feature model from which it is derived.

The configuration model takes a set of features (choices) selected for a particular system configuration and then validates the chosen collection of features against the constraints of the relationships between the features, e.g. whilst the feature model illustrated in Figure 167 is valid, a configuration comprising the features shaded in blue is invalid for two reasons:

1. The coupe feature requires that both a manual gearbox feature and a petrol engine feature be selected; however an electric engine has been selected.
2. The gearbox sub-features manual and automatic are alternatives; however both sub-features have been selected.
Figure 167 – Erroneous Configuration Model

Features selected for the example configuration shaded in blue.
The configuration model is not responsible for validating the feature model, the feature model does this itself; the configuration model is only responsible for validating the configuration derived from the feature model. Furthermore, the configuration model does not have any knowledge of how the chosen system configuration is to be assembled; hence the strong separation of concerns has been demonstrated by the configuration metamodel.

The specification metamodel is very simple; it binds the system configuration to a user-defined architectural model (Figure 98). This approach avoids coupling the architectural blueprint to the configuration model, thereby allowing the configuration to be described and validated in advance of the specification of the target architectural model, and, possibly, even reused across different architectural models.

The product generator provides the mechanism by which a valid instance of a specification model is turned into a custom model. The separation of concerns has been enforced less strongly here as the implementation described in section 5.4 has explicit knowledge of the mapping from source domain elements to target domain architecture. However, it may be possible to alleviate this via a more generic approach using (e.g.) colouring of the domain model. Hence, whilst the product generator does not have explicit knowledge of the configuration, it is polluted to some degree with the specific details of the mapping to be used; clearly, this could be improved in future work.

In summary, it is concluded that the metamodels presented in chapter 5, and supported by the implementation-independent specification of chapter 4 have largely satisfied the hypothesis, with the only remaining issue being the opportunity to enforce further separation between the product generator and the source/target domain models such that a generic model generator could be developed for each style of target architecture.

7.1.2 A Semantics for Features

The second hypothesis proposed at section 1.3.2 is restated below:

Given the overloaded role played by the feature model it is reasonable to question that the use of three relationships (aggregation/decomposition, generalisation/specialisation and parameterisation) on the feature diagram will be adequate for all possible situations. In particular, use of the aggregation and specialisation relationships may be viewed as introducing design decisions at an arbitrarily early stage in the description of the SPLA; a more suitable semantics should be described that is independent of the typical programming constructs. Hence, it is asserted that the relationships between features are simply that [relationships between features], what this means at the level of the application program components may vary depending on the context and even
upon the features supported by individual programming languages. It is asserted that the application of meta-modelling techniques following the style popularised by Language-Driven Development (LDD) and Language Oriented Programming (LOP) approaches will help to make explicit the semantics of the SPLA and clarify the transformation of the SPLA into a valid system configuration. Furthermore, support for model transformation via appropriate mappings facilitates the generation of views of the SPLA appropriate to many domains, e.g. generation of code, documentation, validation constraints, etc., adapted for specific customers/disciplines (publications, development, safety, testing). This hypothesis will be tested via the development of two case studies in chapter 6.

It has been demonstrated in section 7.1.1 that the overloaded role played by the feature model has been resolved via the use of the metamodelled architecture of the SPLA as illustrated in Figure 4, underpinned with an appropriate set of constraints over the models. The metamodel for the feature model illustrated at Figure 66 provides the relationships necessary to support the construction of valid feature models: VariationPoint, InclusiveOr, and Alternative, it also supports the identification of dependencies between features and provides the ability for an extensible feature to possess a number of extension points. The relationship between features says nothing about the implementation of the features, this responsibility being delegated to the domain model (in the solution space). The case study described in section 6.3 illustrates the need for the separation of feature relationships from domain model relationships. In Figure 54 the feature Air Vehicle Model and its sub-features are associated with a number of model elements from the domain model, the relationship between the features of the feature model is illustrated in Figure 168; the feature/sub-feature relationship is provided via a variation point attached to one mandatory and one optional sub-feature.

![Figure 168 – Feature Model Fragment (Air Vehicle Model)](image)

The relationship of these features to the domain model is illustrated in Figure 54, from which it can be seen that the relationship from the AirVehicle class (to which the feature Air Vehicle Model traces) to the subordinate classes (to which the sub-features trace) is provided via a directed
association relation, e.g. the association \texttt{AirVehicle.myFuselage} relating an \texttt{AirVehicle} class to a \texttt{Fuselage} class (see Figure 169).

![Figure 169 – Air Vehicle Domain Model Components](image)

In contrast to this, the \texttt{Steering Model} feature is modelled having a mandatory sub-feature \texttt{Flight Aids} related via a variation point, the sub-feature \texttt{Flight Aids} has an optional sub-feature \texttt{Hover Hold}, also related via a variation point (see Figure 170).

![Figure 170 – Feature Model Fragment (Steering Model)](image)

In this case the domain model to which these features are mapped is illustrated in Figure 148. Here it can be seen that the feature \texttt{Steering Model} is not related to any domain model elements, whilst the mandatory sub-feature \texttt{Flight Aids} is related to the class hierarchy \texttt{FlightAid} and \texttt{Cruise}. 

241
whilst the feature Hover Hold is related to the sub-class HoverHold. It can be seen in this example that the relationship between the domain classes FlightAid and HoverHold is one of inheritance.

It can be seen from the examples illustrated above that it is necessary for relationships of the same type in the feature model to trace to components in the domain model that make use of differing relationships, in other words there is not a one-to-one mapping from feature relationships to domain model relationships.

In summary, it is concluded that the metamodels presented in chapter 5, supported by the examples presented in the case studies in chapter 6 have satisfied the hypothesis.

7.1.3 Is the Metamodelled SPLA Domain Agnostic?

The third hypothesis proposed at section 1.3.2 is restated below:

Although there is evidence in the literature of the application of SPLA-based software development programs in diverse domains, e.g. engine controllers (Cumins) [CE02] and avionics mission systems (Boeing) [Sha00], however there is no supporting evidence to suggest use of a common underpinning SPLA metamodel, or even a common understanding of the concept of the SPLA. Hence, it is asserted that the metamodelled SPLA components should be domain agnostic, this will be tested via the application of the metamodelled SPLA on two case studies addressing model reuse in significantly different domains.

The case studies described in chapter 6 make use of the same underpinning metamodel of the SPLA, the implementation of which is described in chapter 5. The case studies presented are cast in two different domains, that of a family of Tactical Data Link and an Avionics Navigation System.

Two issues were exposed by the SPLA modelling of the TDL domain.

1. It was found that the TDL feature model contained a number of situations where a feature was extended with a mandatory sub-feature that could be modelled via either of the relationships: VariationPoint or InclusiveOr, with no impact on the resulting semantic.

2. It was found that the modelling of the minimum implementation requirements appealed for that a more complex form of feature dependency to capture a dependency group with an arbitrary cardinality constraint.

In the case of the first issue, this is not seen to be something that would be peculiar to the TDL domain; in fact one might see this as a natural consequence of the evolution of the product line, where a placeholder is provided to support future extension. If the product line was being structured to admit future extension of certain features, then it’s quite likely that the developer would know which feature relation was most appropriate; in the case of the example illustrated in
Figure 171, it is clear from the structure of similar message families that the InclusiveOr relationship is most appropriate, the decision would ultimately be driven out when the time came to extend the set of sub-features beyond a single feature.

In the case of the second issue, it was found that the minimum implementation requirements for the TDL required an extended for of feature dependency such that the feature model could express the semantic that feature $f_1$ depended on a subset of a collection of features $f_m .. f_n$. Although it is possible to work around the lack of the necessary expressiveness of the feature model by restructuring the feature hierarchy, further work is required to extend the metamodel for the feature model to fully address the issue. Furthermore, one can speculate that the need to represent a feature dependency relationship of the n-from-m type is not going to be restricted to the TDL domain, e.g. this is often the manner in which some restaurants organise set menus.

In summary, it is concluded that the metamodels presented in chapter 5, supported by the examples presented in the case studies in chapter 6 have satisfied the hypothesis set out above, although it is to be noted that the need for refinement of at least one aspect of the metamodels was identified.

7.2 Contributions of This Work

The contributions of this work are threefold, firstly the concepts underpinning the feature model as introduced by Kang et al. [KCH+90] and subsequently refined by Czarnecki and Eisenecker [CE99] have been captured in the form of an implementation-independent specification in Z [Spi92] and a metamodel expressed in a language based heavily on the MOF that forms the backbone to a number of languages, including the industry standard modelling language (the UML). Therefore the work is open to the object modelling community via the XMF-based models, and accessible to a wider audience via the Z specifications, such that it could be implemented in a variety of technologies, e.g. an Eclipse plug-in. The metamodels developed are executable and the results of the implementation executing on a commercial tool (XMF-Mosaic) have been presented. Previous work in the area of feature modelling, whilst important, has been founded upon informal
textual descriptions of the metamodel underpinned by examples, see [KCH+90], [CE99], [CE00], [Gom04]. In some cases metamodels have been described graphically in the literature, using lightweight extension mechanisms of the UML [HSV00], whilst others have themselves been expressed in the form of a feature model [Vra04]. However, in all of these cases the feature model rests only on a lightweight metamodel with (in some cases) a questionable semantics.

The use of lightweight extensions to the metamodel is pragmatic; most UML tools will support this feature; however the resulting semantics are questionable as they are dictated by the meta-class to which the stereotype is applied. The use of a lightweight profile is also pragmatic, as the user needs only to learn one language (the feature modelling language) to understand the semantics. However, in this case one must question the underlying metamodel (meta-meta-features) used to support the metamodel; there does not appear to be the necessary grounding required to define the meta-metamodel. In contrast, the approach presented in this thesis describes the metamodels in a form amenable to checking by a tool. The separation of the feature model and configuration model also serves to provide rigorous constraint checking of each of the models, such that we can validate that a feature model is valid against the underlying metamodel, and subsequently confirm that a configuration model is valid, both against the constraints of the feature model and the well-formedness rules of the metamodel underpinning the configuration model.

The second contribution made by this work relates to the definition of the SPLA in the form of a set of related (meta)models; although much work has been published in this domain (e.g. [Bos00] and [GS04]) the authors tend to take a broad view of the SPLA and its artefacts, whilst the work reported in this thesis has established the artefacts and the inter-relationships in the form shown in Figure 4.

Building the metamodels from the common foundation of MOF\textsuperscript{42} ensures that the models are tailored to a particular set of requirements, exporting concepts that may be unified across the set of models; this is a flexible and open approach as other models (derived from the same foundation language) may be integrated at a later date, such as a document model to support the rendering of the models in some other domain-specific language, such as a table-based representation.

The final contribution of this work relates to the demonstration of the linkage between models to provide an integrated solution to the modelling of the features to be supported by family members, and the relationship to a set of core components that are to be merged into a generated product. The decomposition of the essence of the SPLA across a number of related metamodels has been demonstrated to provide a strong separation of concerns, with each model having a single,

\footnote{As exported via the XMF package XCore.}
well-focussed objective. This aspect of the work leverages from the concepts of the OMG’s MDD initiative, using the XMap transformation language provided by XMF-Mosaic, enforcing the view that models rather than, say, code, are of primary importance, becoming the currency of reuse. The decoupling of the concepts identified in the SPLA across a number of coherent components provides scope for easier maintenance, such as amending the semantics of feature relationships, and also provides openness to allow the models to be integrated into additional views.

7.3 The Benefit of Hindsight

Following a simple model validation exercise drawing snapshots in XMF-Mosaic, the initial development of the models described herein was undertaken by writing instances directly in the abstract syntax; although this is a very flexible approach (in that one has complete control over the construction of the instance) it tends to result in verbose code that quickly becomes a maintenance issue as the metamodels evolve. The intention was to produce a graphical concrete syntax for creating model instances; however this was precluded as a result of a limitation with the graphical language support provided by XMF-Mosaic (XTools), although it should be possible to provide a solution via an alternative technology such as Eclipse GME/GMF. The definition of a textual concrete syntax and associated parser is not a trivial undertaking, and is also subject to the same maintenance issues if the underlying metamodel changes. However, once developed the concrete syntax was found to provide a significant improvement in productivity, making creation of example models a simple matter, this, in turn, served to support the testing and validation of constraints on the models.

It has been noticed that the textual feature model descriptions can become quite long (although this is still somewhat more concise than writing in the abstract syntax), and there is clearly a need to structure the models such that a feature model $FM_1$ containing a leaf feature $f_1$ may be decomposed in a separate feature model $FM_2$ or a referenced module with $f_1$ appearing as the root feature. Although the feature model decomposition could be modelled fairly easily, thought would need to be given to the use of feature dependencies as these tend to crosscut the model. The current feature model does not provide support for namespaces, clearly, as instances of the feature models become larger there is a greater possibility of name clashes, hence support for namespaces should be considered for future work.

Demonstration of the product generation is based on a number of relatively small models, clearly this would be more compelling if it were to be supported by a more significant example. The mappings provided to demonstrate the model merging capability are potentially very complex, further thought needs to be given to the handling of namespaces and attribute relationships. The model merging mappings have signatures of the general form $\text{Package} \rightarrow \text{Package}$, placing all
knowledge of the actual form of the transformation in the mapping object, adding to its complexity. This situation could be alleviated by the use of a component model-specific profile such that the major elements have a more descriptive stereotype\(^43\) (e.g. following the style of the layered architectural profile described in Section 5.3.2); one might envisage mapping of the form `EarthModel \rightarrow FoundationLayer`. The mappings would then be more in keeping with the more general pattern-matching style, and be of a finer level of granularity; this would simplify the transformations and make them more amenable to reuse.

The product generator currently operates on model elements identified by the specification model, using these to populate the target model (see Figure 106), i.e. model elements are taken from the source model and moved to the target model.

This approach has the drawback that execution of the product generator affects the state of source model; this is considered to be an undesirable side-effect. The issue could be addressed by populating the target model with model elements cloned from the source model; this approach would also be more amenable to the traversal of the source element and pruning of unwanted features (e.g. attributes that are not required for some particular configuration).

Although relatively simple, the case studies described in Section 6 provided a valuable source of realistic test data. The approach taken to development of the SPLA metamodels was primarily via a thin vertical slice; on reflection it may have been more appropriate to take a broader approach, as the aim was for a *domain-agnostic* profile. Such an approach would have highlighted the importance of a constraint language, model interrelationships, and also the possibility of rendering the models in other graphical formats, e.g. tables (see Figure 172).

---

\(^{43}\) This is sometimes referred to as *colouring* the model.
Similarly, a breadth-first approach would also have identified the benefit of developing a richer semantics for cross-feature dependencies, of the form:

\[
\text{Feature } \langle X \rangle \text{ requires any } \langle n \rangle \text{ from } \{\langle \text{requiredFeatureList} \rangle \}
\]

In addition to the current form:

\[
\text{Feature } \langle X \rangle \text{ requires } \{\langle \text{requiredFeatureList} \rangle \}
\]

Although a constraint was added to class \texttt{Feature} to ensure that a feature does not have a sub-feature with a binding time earlier than that of itself (see Equation 33) and tested to ensure correct operation, it was found that the concept of binding time had little importance in the feature models developed, since these models were primarily related to the generation of application models. Therefore, of the enumeration literal values provided by the existing \texttt{BindingTimeType} class in package \texttt{FeatureModels} (Generate, Compile, Link, Run), the default is invariably Generate, with the actual component’s binding time attributes delegated to the models generated, hence there is no remaining variability in the Configuration model; if the models were to be used to represent application code, then this may not be a desirable feature, although one may also envisage the provision of a specific run-time loading feature (e.g.).

\[
\text{BindingTimeEqualToOrLaterThanSuperFeature}
\]

context Feature

\[
\text{self.superFeature() ->forall(parent | self.bindingTimeEqualToOrLaterThan(parent))}
\]

\textbf{Equation 33 – Feature/Sub-Feature Binding Time Constraint}

Finally, the complexity of dealing with component namespaces and attribute relations has been discussed within the context of the model merging approach to product generation, however there is a further issue relating to reconciling the semantics of the feature model against those of the component model. The feature model provides a view of how the components to which the features are bound may be composed to generate a valid product (model). However, it is possible that a component relationship may not be consistent with the view of the feature model, e.g. component X may have an association with a component Y that is part of another feature; in this case feature X should be shown to require feature Y in the feature model. A more difficult (contradictory) situation arises if features X and Y are described as being mutually exclusive. To prevent such situations from occurring it would be necessary to develop cross-checks between the feature model and the component model, or to allow the component model to inform the feature model of component (feature) dependencies.
7.4 Opportunities Envisaged

The work presented by this thesis has demonstrated a novel approach to the specification of the SPLA and the generation of specific family members via well-defined set of metamodels. A majority of the work proceeded well and has provided the expected results; however there is scope for improvement of the work undertaken thus far, and also extension to support further functionality. Corrective maintenance activities that should be considered are described above in Section 7.3, whilst candidate perfective changes are described below\(^{44}\).

The suggested extension of the Features metamodel to support a modular approach to the construction of large feature models has an associated issue relating to the allowed scope of feature dependencies. Furthermore, it is suggested that, as the feature models become more fine-grained, one might begin to see recurring feature composition patterns in similar manner to that seen in computer programs. Hence, it is hypothesised that, if such patterns are found to recur, it may be possible to characterise these in the form of reusable feature templates, or common refactoring strategies to provide accelerated feature descriptions.

When specifying a system from the feature model it is likely that there will be situations where the choice of features is not obvious, or a number of configurations might provide the required functionality. Under such cases it would be useful if tool support was to be provided to guide selection of the most appropriate set of features. Selection criteria may be based on the number or complexity of the component classes to be configured, or perhaps closeness of fit into the target architecture that is to be generated. It is noted that Batory’s P3 generator and design wizard provides similar functionality; although this is related to the generation of efficient Java container classes from the P3 programs [BCR+00]. Further work is required to establish a suitable set of discriminating attributes that could be used as the basis for tool-aided feature selection, however the XMF-Mosaic environment provides the language XRules (analogous to Prolog), hence it should be possible to implement a powerful optimising algorithm.

Referring back to the TDL-related case study described in Section 6.2, a useful extension to the configuration model support would be the ability to define a partial configuration and have tool support to guide the user through the remaining choices until all variability has been removed. A tool could infer the selection of some features automatically, e.g. automatic resolution of all required features, and selection of all ancestors back to the root; the default approach might be to present all areas of the feature model where the user needs to make a choice from a small set of options. There is some resonance with the Contextual diagrams described by Felfernig et al. [FJZ00].

\(^{44}\) Corrective maintenance = bug fixes, perfective maintenance = new functionality [Pre92]
As reported in section 5.5.6, there is some debate in the literature regarding the decoration of feature relationships with cardinalities. The view taken by work presented in this thesis is that feature cardinalities are not relevant, and that this is an implementation detail that can be delegated to the domain model. However, there may be situations where it may be required to restrict the cardinality of some association between components in the domain model, e.g. configuring an engine with four cylinders from a PLA supporting engines with a range of cylinders, say from 3 to 8. It is possible that this could realised quite simply by extending the family of features available by providing a parameterised feature in the metamodel for the feature model.

It has been noted elsewhere (e.g. [KCH+90], [CE99] and [CE00]) that a product family often contains a number of product ranges; these are partial configurations that retain some degree of variability, e.g. the Sport variant of some car vendors product line that might allow the customer to choose between a number of engines of differing power output, or choose between a manual or a semi-automatic gearbox, etc. On the one hand this relieves the user from the burden of specifying a fully enumerated configuration, whilst also constraining the, otherwise possibly enormous, number of potential product incarnations. The metamodel of the feature model could be extended to support the explicit modelling of a product range; this might be achieved quite easily via extension of the class FeatureModel, such that a feature model may contain both features and product ranges.

Stevenson identifies an inconsistency between existing domain-specific engineering practices and the, largely decoupled, ongoing research focussing on the development of feature model in the UML (such as FodaCOM). The problem identified is that the UML development work does not seek to support existing domain-specific tools, and Stevenson concludes that it would be prudent to unify any metamodel descriptions with the UML metamodel [Ste02]. The approach followed is founded on the MOF that provides the common backbone to a number of the OMGs modelling languages (including the UML), which would appear to be aligned with Stevenson’s concluding remarks. Furthermore, integration of differing languages based on metamodelling and model transformation has been demonstrated successfully elsewhere [DSB+08], hence unification of the metamodelled SPLA and existing domain-specific tools should be feasible.

The items identified as possible candidates for future work are summarised in Table 5

---

45 Clearly, such constraints would be beneficial in a manufacturing environment.
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Develop further examples of the SPLA (feature model) to investigate the appearance of common patterns and refactoring/simplification strategies.</td>
</tr>
<tr>
<td>2.</td>
<td>Introduce tool support to guide selection of optional features based on model heuristics, and identify the relevant tell-tale heuristics.</td>
</tr>
<tr>
<td>3.</td>
<td>Provide support in the metamodels (and tools) for the auto-completion of configuration choices, e.g. by automatically selecting all ancestors of a selected feature to the root.</td>
</tr>
<tr>
<td>4.</td>
<td>Investigate the use of m-from-n feature dependencies and extend the metamodel of the feature model to provide the necessary model and tool support.</td>
</tr>
<tr>
<td>5.</td>
<td>Use the structural dependencies of the domain model to augment and/or validate the structure of the feature model.</td>
</tr>
<tr>
<td>6.</td>
<td>Investigate the introduction of cardinality constraints in either the feature model of configuration model (or both) to support restriction of the cardinality of associations in the domain model.</td>
</tr>
<tr>
<td>7.</td>
<td>Extend the metamodel for the feature model to allow for the description of product ranges to help reduce the size of the system configuration descriptions.</td>
</tr>
<tr>
<td>8.</td>
<td>Reduce coupling between the product generator and the target architecture via the application of some form of colouring approach to the domain model.</td>
</tr>
<tr>
<td>9.</td>
<td>Investigate the unification of the metamodelled SPLA presented in this thesis with existing domain tools using a metamodelling and transformational approach.</td>
</tr>
</tbody>
</table>

Table 5 – Summary of Opportunities for Future Work
## 8 Definitions

**Alternative**
An exclusive-or relationship between a feature and a set of sub-features such that at most one alternative feature must be selected for any valid configuration.

**Choice**
A feature chosen for inclusion in a configuration.

**Configuration**
A coherent collection of related features.

**Configuration Model**
A model containing a coherent collection of features and a reference to the originating feature model.

**Dimension**
A relationship between a feature and a set of sub-features where each of the sub-features is an alternative.

**Domain Architecture**
The collection of models and mappings needed to transform a domain model into an application.

**Feature**
A property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances (taken from [CE00]).

**Feature Model**
A model describing the features relating to some PLA and showing the configuration constraints.

**Inclusive-Or**
A relationship between a feature and a set of sub-features such that any combination of sub-features may be selected for any valid configuration.

**Mandatory**
A feature that must be selected, subject to selection of its super-feature and the relationship between itself and its super-feature.

**Optional**
A feature that may be selected, subject to selection of its super-feature and the relationship between itself and its super-feature.

**Product Line**
A group of product exhibiting sufficient commonality as to make it advantageous to study the common properties of the group before analysing individual members (paraphrased from [Par76]).

**Product Line Architecture**
A description of the structural properties for building a group of related systems (Product Line), typically the components and their relationships. The inherent guidelines about the use of components must capture the means for handling required variability among the systems (taken from [CN02]).
| **Profile** | The mechanism used to tailor existing metamodels towards specific platforms or domains. A profile must be based on a metamodel such as the UML that it extends, and is not very useful standalone [OMG07b] |
| **Profile (heavyweight)** | A customisation mechanism for creating languages and language dialects tailored to particular platforms or (more likely) particular domains. The heavyweight profile approach is characterised by the ability to introduce new model elements. The heavyweight profile is characterised by being applied at the level of the MOF (i.e. applied at the m3 layer of the UML 4-layer metamodel architecture). The application of a heavyweight profile by a tool requires that the tool provide explicit visibility of the underlying metamodel (MOF). |
| **Profile (lightweight)** | A customisation mechanism for creating language dialects tailored to particular platforms [Wat08]. The lightweight profile mechanism is restricted to reusing existing model elements, possibly adding attributes and/or constraints. The lightweight profile mechanism is characterised by being specified and applied at the m2 layer of the UML 4-layer metamodel architecture. |
| **Software Product Line Architecture** | A set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way (taken from [CN02]) |
| **Specification Model** | A model relating a Configuration Model to an Architectural Model (prototype) |
| **Variation Point** | A relationship between a feature and a set of sub-features such that all mandatory sub-features must be selected for any valid configuration and also any (or no) optional sub-features |
9 Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACVC</td>
<td>Ada Compiler Validation Capability</td>
</tr>
<tr>
<td>ADAGE</td>
<td>Avionics Domain Application Generation Environment</td>
</tr>
<tr>
<td>ADC</td>
<td>Air Data Computer</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>AHRS</td>
<td>Attitude Heading Reference System</td>
</tr>
<tr>
<td>AoA</td>
<td>Angle of Attack</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASADAL</td>
<td>A System Analysis And Design Aid tool</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
</tr>
<tr>
<td>BSP</td>
<td>Board Support Package</td>
</tr>
<tr>
<td>C²</td>
<td>Command and Control</td>
</tr>
<tr>
<td>CDT</td>
<td>Customisation Decision Tree</td>
</tr>
<tr>
<td>CEM</td>
<td>Concurrently Executable Module</td>
</tr>
<tr>
<td>COM</td>
<td>Common Object Model</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off The Shelf</td>
</tr>
<tr>
<td>CWM</td>
<td>Common Warehouse Metamodel</td>
</tr>
<tr>
<td>DDS</td>
<td>Data-Distribution Service (for real-time systems)</td>
</tr>
<tr>
<td>DFI</td>
<td>Data Field Identifier</td>
</tr>
<tr>
<td>DI</td>
<td>Data Item</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library</td>
</tr>
<tr>
<td>DNS</td>
<td>Doppler Navigation System</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>DUI</td>
<td>Data Use Identifier</td>
</tr>
<tr>
<td>EAI</td>
<td>Enterprise Application Integration</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>Epsilon</td>
<td>Extensible Platform for Specification of Integrated Languages for mOdel maNagement</td>
</tr>
<tr>
<td>ERD</td>
<td>Entity Relationship Diagram</td>
</tr>
<tr>
<td>FARS</td>
<td>Functional Area Requirements Specification</td>
</tr>
<tr>
<td>FODA</td>
<td>Feature-Oriented Domain Analysis</td>
</tr>
<tr>
<td>FORM</td>
<td>Feature-Oriented Reuse Method</td>
</tr>
<tr>
<td>Acronym</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>GME</td>
<td>Generic Modeling Environment</td>
</tr>
<tr>
<td>GMF</td>
<td>Graphical Modelling Framework</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HCI</td>
<td>Human-Computer Interface</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Mark-up Language</td>
</tr>
<tr>
<td>IAS</td>
<td>Indicated Air Speed</td>
</tr>
<tr>
<td>ICCL</td>
<td>Implementation Components Configuration Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Description Language</td>
</tr>
<tr>
<td>IMS</td>
<td>Integrated Modular Systems</td>
</tr>
<tr>
<td>INCOSE</td>
<td>International Council in Systems Engineering</td>
</tr>
<tr>
<td>INS</td>
<td>Inertial Navigation System</td>
</tr>
<tr>
<td>J2EE</td>
<td>Java 2 Platform Enterprise Edition</td>
</tr>
<tr>
<td>JTIDS</td>
<td>Joint Tactical Information Distribution System</td>
</tr>
<tr>
<td>JU</td>
<td>JTIDS Unit</td>
</tr>
<tr>
<td>LDD</td>
<td>Language-Driven Development</td>
</tr>
<tr>
<td>LGD</td>
<td>Local Geodetic Datum</td>
</tr>
<tr>
<td>LOP</td>
<td>Language Oriented Programming</td>
</tr>
<tr>
<td>MARTE</td>
<td>Modeling and Analysis of Real-Time and Embedded Systems</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
</tr>
<tr>
<td>MDSD</td>
<td>Model-Driven Software Development</td>
</tr>
<tr>
<td>MFC</td>
<td>Microsoft Foundation Classes</td>
</tr>
<tr>
<td>MIL</td>
<td>Module Interconnection Language</td>
</tr>
<tr>
<td>MIN IMP</td>
<td>Minimum Implementation (rules)</td>
</tr>
<tr>
<td>MMI</td>
<td>Man-Machine Interface (synonymous with GUI/HCI)</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>MPS</td>
<td>Meta-Programming System</td>
</tr>
<tr>
<td>NEC</td>
<td>Network Enabled Capability</td>
</tr>
<tr>
<td>OAT</td>
<td>Outside Air Temperature</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform-Independent Model</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>PLA</td>
<td>Product Line Architecture</td>
</tr>
<tr>
<td>PLE</td>
<td>Product Line Engineering</td>
</tr>
<tr>
<td>POI</td>
<td>Point Of Interest</td>
</tr>
<tr>
<td>PP</td>
<td>Preset Position</td>
</tr>
<tr>
<td>PPLI</td>
<td>Precise Participant Location and Identification</td>
</tr>
<tr>
<td>PRS</td>
<td>Platform Requirement Specification</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform-Specific Model</td>
</tr>
<tr>
<td>QSDL</td>
<td>Queuing Specification and Description Language</td>
</tr>
<tr>
<td>QVT</td>
<td>Queries, Views and Transformations</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>Rx</td>
<td>Receive</td>
</tr>
<tr>
<td>SIENM</td>
<td>System Information Exchange &amp; Network Management</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integrity Level</td>
</tr>
<tr>
<td>SPL</td>
<td>Software Product Line</td>
</tr>
<tr>
<td>SPLA</td>
<td>Software PLA</td>
</tr>
<tr>
<td>SysML</td>
<td>Systems Modeling Language</td>
</tr>
<tr>
<td>TACAN</td>
<td>Tactical Air Navigation</td>
</tr>
<tr>
<td>TADIL</td>
<td>Tactical Digital Information Link</td>
</tr>
<tr>
<td>TAS</td>
<td>True Air Speed</td>
</tr>
<tr>
<td>TDL</td>
<td>Tactical Data Link</td>
</tr>
<tr>
<td>TERPROM</td>
<td>Terrain Profile Matching</td>
</tr>
<tr>
<td>Tx</td>
<td>Transmit</td>
</tr>
<tr>
<td>UK</td>
<td>United Kingdom</td>
</tr>
<tr>
<td>US</td>
<td>United States (of America)</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VHF</td>
<td>Very High Frequency</td>
</tr>
<tr>
<td>VMF</td>
<td>Variable Message Format</td>
</tr>
<tr>
<td>VOR</td>
<td>VHF Omni-directional Range</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Mark-up Language</td>
</tr>
<tr>
<td>XOCL</td>
<td>eXecutable OCL</td>
</tr>
<tr>
<td>XSLT</td>
<td>eXtensible Stylesheet Language for Transformations</td>
</tr>
</tbody>
</table>
10 List of References


[GHJ+95] Gamma, E., Helm, R., Johnson, R., Vlisides, J., ‘Design Patterns: Elements of Reusable Object-Oriented Software’, Addison Wesley, 1995


[Hol01a] Holmes, C., ‘Methods of attacking the requirements backlog within the avionics domain – Results of a literature survey’, University of York, Department of Computer Science, YCS 335 (2001)


260


Spivey, M., ‘The Z Notation’, Prentice-Hall, 1992


263
1 Parsing of GenVoca Models and Specifications

The concrete (textual) syntax of GenVoca models is parsed into the XMF model using the following grammar:

```
parserImport XQCL;
parserImport Parser::BNF;
import Root::ArchitectureDescriptionLanguages::GenVoca::Models;
context Model
@Grammar extends OCL::OCL.grammar
  Model ::= '(' n = Name ')' realms = Realm* { let model = realms->
           iterate(realm m = [ | Model[name = <n.lift()>] ] )
           in [ [ <model>.initialise() ] ]
           end } .
  Realm ::= n = Name ':' layers = Layer* ';' { let realm = layers->
         iterate(layer r = [ | Realm[name = <n.lift()>] ] )
         in [ [ <realm> ] ]
         end } .
  Layer ::= TerminalLayer | ParameterisedLayer .
  TerminalLayer ::= n = Name { [ | Layer[name = <n.lift()>] ] } |
                  n = Name ':' { [ | Layer[name = <n.lift()>] ] } .
  ParameterisedLayer ::= n = Name '([' ds = DependencyList '])' { let layer = ds->iterate(d l = [ | Layer[name = <n.lift()>] ] )
                    in [ [ <layer> ] ]
                    end } |
                  n = Name '([' ds = DependencyList '])' '{
                  let layer = ds->iterate(d l = [ | Layer[name = <n.lift()>] ] )
                  in [ [ <layer> ] ]
                  end } .
  DependencyList ::= formalName = Name ':' type = Name { Seq([ | <type.lift()> ] ) } |
                  formalName = Name ':' type = Name ',' tail = DependencyList { Seq([ | <type.lift()> ] ) + tail } .
```
The concrete (textual) syntax of the specification of a system from a GenVoca model is parsed into the XMF model using the following grammar:

```plaintext
parserImport XQCL;
parserImport Parser::BNF;

import Root::ArchitectureDescriptionLanguages::GenVoca::Specifications;

context Specification

@Grammar extends OCL::OCL.grammar

Specification ::= '(' n = Name ',' m = Exp ')' te = TypeEquation ';'
{ let spec = [] Specification[name = <n.lift()>,
    model = <m>,
    typeEquation = <te>]
    in [] <spec>.initialise() []
    end } .

TypeEquation ::= CompositeTypeEquation | AtomicTypeEquation .

CompositeTypeEquation ::= n = Name '{' params = ParameterList '}'
{ let component = params->
    iterate(param te = [] TypeEquation[name = <n.lift()>] []
      in [] <component> []
      end }

AtomicTypeEquation ::= n = Name
{ [] TypeEquation[name = <n.lift()] [] } .

ParameterList ::= te = TypeEquation
{ Seq{[] <te> []} | te = TypeEquation ',' tail = ParameterList
    Seq{[] <te> []} + tail } .

end
```
2 Abstract Specification of the Product Line Architecture

The following specification has been prepared using the CADiZ\textsuperscript{46} tool.

\begin{verbatim}
section Reusable parents standard_toolkit

-- Reusable specifications --

REPORT ::= ok | error

A report comprises either ok or error.

\begin{tabular}{|l|}
\hline
Success \\
\hline
\hline
repl : REPORT \\
\hline
repl = ok
\hline
\end{tabular}

Report success schema.

\begin{tabular}{|l|}
\hline
Fail \\
\hline
\hline
repl : REPORT \\
\hline
repl = error
\hline
\end{tabular}

Report fail schema.

\begin{verbatim}
section FeatureModelling parents Reusable

-- Feature Model --

The set of all features.

\[ FEATURE \]

The set of all model elements.

\[ MODELELEMENT \]

The set of all extension points.

\[ EXTENSIONPOINT \]

The optionality type comprises optional and mandatory. Note: free types are like enumerations.

OPTIONALITY ::= optional | mandatory

\[ open, closed : \mathbb{P} FEATURE \]

\[ \langle open, closed \rangle \text{ partition } FEATURE \]

Divide the set of all features into two types: open and closed. Every feature is either open or closed, but no feature is both open and closed.
\end{verbatim}

\textsuperscript{46} Available at: \url{http://www.cs.york.ac.uk/hise/cadiz/}
\[ \text{variationPoint, alternative, inclusiveOr : } \mathcal{P} \text{ EXTENSIONPOINT} \]
\[
\langle \text{variationPoint, alternative, inclusiveOr} \rangle \\
\text{partition EXTENSIONPOINT}
\]

Divide the set of extension points into three types: variationPoint, alternative, and inclusiveOr. Every extension point is either a variationPoint, alternative, or inclusiveOr, but no extension point is more than one.

\[ \text{optionality : FEATURE } \rightarrow \text{OPTIONALITY} \]

optionality is a function from a feature to its optionality type, this function allows us to infer if a feature is optional or mandatory.

\[ \text{superfeature, ancestor : FEATURE } \rightarrow \text{FEATURE} \]
\[ \text{ancestor = superfeature}^+ \]

superFeature is a function from a feature to its parent feature, the relationship is acyclic (a feature can't be a super-feature of itself). A feature may have a maximum of one super-feature - hence we model it as a function. The relation ancestor is the transitive closure of the superfeature relation.

\[ \text{extensionpoint : (FEATURE } \leftrightarrow \text{open} \rangle \leftrightarrow \text{EXTENSIONPOINT} \]

extensionpoint is a relation from an open (extensible) feature to an set of extension points, i.e. closed features do not have extension points.

\[ \text{subfeature : EXTENSIONPOINT } \leftrightarrow \mathcal{P} \text{ FEATURE} \]

subfeature is a relation from an extension point to a set of features.

\[ \text{child } \leftarrow \text{extensionpoint } \circ \text{subfeature} \]

the relation child is the composition of the extensionpoint and subfeature relations

\[ \text{vp : (variationPoint } \leftarrow \text{EXTENSIONPOINT} \rangle \leftrightarrow \text{FEATURE} \]

vp is a relation from a variation point to a set of features

\[ \text{alt : (alternative } \leftarrow \text{EXTENSIONPOINT} \rangle \leftrightarrow \text{FEATURE} \]
\[ \forall f1, f2 : \text{FEATURE } \bullet \text{optionality}(f1) = \text{optionality}(f2) \]

alt is a relation from an alternative extension point to a set of features with the constraint that all features mapped to have the same optionality type (i.e. either all optional or all mandatory)

\[ \text{incor : (inclusiveOr } \leftarrow \text{EXTENSIONPOINT} \rangle \leftrightarrow \text{FEATURE} \]
\[ \forall f1, f2 : \text{FEATURE } \bullet \text{optionality}(f1) = \text{optionality}(f2) \]
incor is a relation from an alternative extension point to a set of features with
the constraint that all features mapped to have the same optionality type (i.e.
either all optional or all mandatory)

\[ \text{dependency} : \text{FEATURE} \rightarrow \mathbb{P} \text{FEATURE} \]

dependency is a relation from a feature to a set of features upon which the
feature depends.

\[ \dim : \text{FEATURE} \rightarrow \mathbb{N} \]

\[ \forall f : \text{FEATURE} \Rightarrow \dim(f) = \# \text{ran extensionpoint} \]

The dimension of a feature is the cardinality of its extension points.

\[ \text{allparents} : \text{FEATURE} \rightarrow \text{FEATURE} \]

\[ \text{allparents} = \text{ancestor} \]

The allparents relation is the set of all the ancestors of a feature up to the root.

\[ \text{descendant} : \text{FEATURE} \rightarrow \text{FEATURE} \]

The relation descendant is between a feature and its descendant features

\[
\begin{array}{c}
\text{FeatureModelDB} \\
\text{features} : \mathbb{P} \text{FEATURE} \\
\text{superfeature} : \text{FEATURE} \rightarrow \text{FEATURE} \\
\text{links} : \text{FEATURE} \rightarrow \text{MODELELEMENT} \\
\text{dependency} : \text{FEATURE} \rightarrow \text{FEATURE} \\
\end{array}
\]

\[ \text{dom links} \subseteq \text{features} \]

\[ \text{dom dependency} \subseteq \text{features} \]

\[ \# \text{features} > 0 \Rightarrow \exists 1 f : \text{FEATURE} \mid f \in \text{features} \cdot \#	ext{ran superfeature} = 0 \]

FeatureModelDB schema:

1. The links relation is a subset of the set of features in the feature model,
i.e. features don’t have to be mapped to model elements, but model elements
have to be mapped to by a feature.
2. The dependsOn relation is defined similarly.
3. If the feature model contains any features then one must be the root.

\[
\begin{array}{c}
\text{InitFeatureModelDB} \\
\Delta \text{FeatureModelDB} \\
\text{features}' = \emptyset \\
\text{links}' = \emptyset \\
\text{dependency}' = \emptyset \\
\end{array}
\]

268
the initial feature model is empty: no features, no links, and no feature dependencies

section ConfigurationModelling parents FeatureModelling

--- Configuration Model ---
The set of all choices.

\[
\text{[CHOICE]}
\]

\[
\begin{align*}
\text{AllParentsIncluded} \\
\text{selection : CHOICE} & \rightarrow \text{FEATURE} \\
\forall f : \text{FEATURE} \mid f & \in \text{ran selection} \bullet \\
\text{allparents}(f) & \subseteq \text{ran selection}
\end{align*}
\]

\[
\text{VariationPoint}
\]

\[
\begin{align*}
\text{feature : FEATURE} & \rightarrow \text{open} \\
\text{vpSelection : CHOICE} & \rightarrow \text{FEATURE} \\
\exists sf : \text{P FEATURE} & \mid \\
\forall f : \text{FEATURE} \mid f & \in sf \land \text{optionality}(f) = \text{mandatory} \bullet \\
\text{sf} & \subseteq \text{child(feature)} \land \\
\text{extensionpoint(feature)} & \in \text{variationPoint} \bullet \\
\text{sf} & \subseteq \text{ran vpSelection}
\end{align*}
\]

all mandatory sub-features attached to a variation point must be included in the set of selected features

\[
\text{MandatoryAlternatives}
\]

\[
\begin{align*}
\text{feature : FEATURE} & \rightarrow \text{open} \\
\text{mandatoryAlternative : CHOICE} & \rightarrow \text{FEATURE} \\
\exists sf : \text{P FEATURE} & \mid \\
\forall f : \text{FEATURE} \mid f & \in sf \land \text{optionality}(f) = \text{mandatory} \bullet \\
\text{sf} & \subseteq \text{child(feature)} \land \\
\text{extensionpoint(feature)} & \in \text{alternative} \bullet \\
\text{sf} & \subseteq \text{ran mandatoryAlternative} \land \#sf = 1
\end{align*}
\]

exactly one sub-feature from a set of mandatory alternative sub-features must be selected if the parent is selected

269
Optional Alternatives

```
feature : FEATURE ⇝ open
optionalAlternative : CHOICE ⇝ FEATURE

∃ sf : ∃ FEATURE |
∀ f : FEATURE | f ∈ sf ∧ optionality(f) = optional •
.sf ⊆ child(feature) ∧
.extensionPoint(feature) ∈ alternative •
.sf ⊆ ran optionalAlternative ∧ #sf ≤ 1
```

A maximum of one sub-feature from a set of optional alternative sub-features must be selected if the parent is selected.

Mandatory Inclusive-Or

```
feature : FEATURE ⇝ open
mandatoryInclusiveOr : CHOICE ⇝ FEATURE

∃ sf : ∃ FEATURE |
∀ f : FEATURE | f ∈ sf ∧ optionality(f) = mandatory •
.sf ⊆ child(feature) ∧
.extensionPoint(feature) ∈ inclusiveOr •
.sf ⊆ ran mandatoryInclusiveOr ∧ #sf ≥ 1
```

A minimum of one sub-feature from a set of mandatory inclusive-or sub-features must be selected if the parent is selected.

Optional Inclusive-Or

```
feature : FEATURE ⇝ open
mandatoryInclusiveOr : CHOICE ⇝ FEATURE

∃ sf : ∃ FEATURE |
∀ f : FEATURE | f ∈ sf ∧ optionality(f) = optional •
.sf ⊆ child(feature) ∧
.extensionPoint(feature) ∈ inclusiveOr •
.sf ⊆ ran mandatoryInclusiveOr ∧ #sf ≥ 0
```

Any number of sub-features from a set of optional inclusive-or sub-features must be selected if the parent is selected.

Dependent

```
feature : FEATURE
dependent : CHOICE ✈ FEATURE

ran dependent = dependency(feature)
```

All dependant features attached to a feature must be included in the set of selected features.
ConfigurationModelDB schema

(1) the relation selection maps a choice to a feature in the feature model
(2) the binary relation selection maps a choice to a feature

InitConfigurationModelDB

ΔConfigurationModelDB

choices' = ∅
selection' = ∅

the initial configuration model is empty

section ArchitecturalModelling parents standard_toolkit

Layered Architectural Model
The set of all Architectural models.

[ARCHITECTURE]
The set of all Layers.

[LAYER]
The set of all Components.

[COMPONENT]
The set of all Interfaces.

[INTERFACE]
The set of all Controllers.

[CONTROLLER]

\[ foundation, layer : \mathbb{P} \text{LAYER} \]

(foundation, layer) partition LAYER

Divide the set of all architectural layers into two types: foundation and layer. Every architectural layer is either a layer or a foundation layer, but no architectural layer is both a layer and a foundation layer.
contains : ARCHITECTURE \sqsubseteq LAYER

A layered architecture contains a set of layers.

classes : LAYER \sqsubseteq COMPONENT

A layer comprises a set of components.

sublayer : LAYER \sqsubseteq LAYER

Layers may be nested.

realises : INTERFACE \sqsubseteq LAYER

A layer realises a set of interfaces.

dependsOn : (LAYER \sqsubseteq \text{layer}) \sqsubseteq INTERFACE

A layer depends on a set of interfaces. Hence a foundation layer does not depend on a set of interfaces.

client == dependsOn \circ realises

A layer is a client of another layer if it depends on the interface realised by the other layer.

\[
\begin{align*}
\text{parent, superordinate} & : LAYER \sqsubseteq LAYER \\
\text{superordinate} & = \text{parent}^+
\end{align*}
\]

parent is a function form a layer to its parent layer, the relationship is acyclic (a parent can’t be a parent of itself). A layer may have a number of parents - hence we model it as a relation. The relation superordinate is the transitive closure of the parent relation.

controls : CONTROLLER \rightarrow LAYER

A controller controls a single layer (total function from controller to layer).

\[
\forall \text{layer : LAYER}; \text{controller : CONTROLLER} | \\
\text{layer} \subseteq \text{dom} \text{controlledBy} \land \text{controller} \subseteq \text{ran} \text{controlledBy} \bullet \\
\text{controls} (\text{controller}) = \text{layer}
\]

A layer is controller by a single controller (total function from layer to controller).
ArchitectureModelDB

layers : P LAYER
interfaces : P INTERFACE
controllers : P CONTROLLER
components : P COMPONENT
architecture : ARCHITECTURE ⇔ LAYER

tax architecture = layers
tax controls = layers
tax dependsOn = interfaces
tax realises = layers
dom realises = interfaces
dom classes = layers
tax classes = components

InitArchitectureDB

\[ \Delta \text{ArchitectureModelDB} \]

layers' = ∅
interfaces' = ∅
controllers' = ∅
architecture' = ∅

The initial architectural model is empty.

section SpecificationModelling parents ConfigurationModelling, ArchitecturalModelling

-- Specification Model --

SpecificationModelDB ⊑ ConfigurationModelDB
 SpecificationModelDB ⊑ ArchitectureModelDB

section ProductGeneratorModelling parents SpecificationModelling

-- Product Generator Model --

ProductGeneratorModelDB

\[ \bowtie \text{ConfigurationModelDB} \]
 \( \bowtie \text{SpecificationModelDB} \]
 \( \bowtie \text{ArchitectureModelDB} \]
components : P COMPONENT
projection : FEATURE ⇔ COMPONENT

dom projection = ran selection
ran projection = components
A projection describes the mapping from a feature to a component.

\[
\begin{align*}
\text{GenerateModel} & \\
\Delta \text{ProductGeneratorModelDB} & \\
\text{transformation} & : \text{MODELELEMENT} \leftrightarrow \text{COMPONENT} \\
\forall f : \text{FEATURE}; c : \text{COMPONENT} & | \\
& f \in \text{dom projection} \land c \in \text{ran projection} \land \\
& \text{links}(f) \in \text{dom transformation} \land \\
& c \in \text{ran transformation} \\
\end{align*}
\]

A model is generated by transforming MODELELEMENT objects into COMPONENT objects.

\[
\begin{align*}
\text{DoGenerateModel} & \\
\text{GenerateModel} \land \text{Success} \lor \text{Fail} & \\
\end{align*}
\]
3 Parsing of Feature Models

The concrete (textual) syntax of the FeatureModel is parsed into the XMF model using the following grammar:

```
parserImport XOCL;
parserrImport Parser::BNF;
import Root::ProductFamilies::FeatureModels;
import OCL;

class FeatureModel

@Grammar extends OCL::OCL.grammar

FeatureModel ::= 
  '(' n = Name ')' features = Feature* { 
    let featureModel = features->iterate(f fm = [| FeatureModel[name = n.lift()] |] | 
      [| fm.addToFeatures(f) |]) 
    in [| featureModel.initialize(); featureModel |] } | 
  '(' n = Name ')' features = Feature* dependencies = Dependency* { 
    let featureModel = features->iterate(f fm = [| FeatureModel[name = n.lift()] |] | 
      [| fm.addToFeatures(f) |]) then 
      featureModel = dependencies->iterate(d fm = [| featureModel |] | 
        [| fm.addToDependencies(d) |]) 
    in [| featureModel.initialize(); featureModel |] } .

Feature ::= 
  (AtomicFeature | CompositeFeature) ';' .

AtomicFeature ::= 
  ft = FeatureType 'Feature' '(' n = Name ')' 'is' 'leaf' { 
    [| Feature[name = n.lift()], selection = ft.lift(), bindingTime = ProductFamilies::FeatureModels::BindingTimeType::Compile |] } .

CompositeFeature ::= 
  ft = FeatureType 'Feature' '(' n = Name ')' r = Relation { 
    let feature = 
      [| ExtensibleFeature[ 
        name = n.lift(), selection = ft.lift(), bindingTime = ProductFamilies::FeatureModels::BindingTimeType::Compile |] | 
    in [| feature.addToExtensionPoints(r) |] } | 
  ft = FeatureType 'Feature' '(' n = Name ')' relation = Relation 'and' 
    tail = Relation* { 
    let feature = 
      [| ExtensibleFeature[ 
        name = n.lift(), selection = ft.lift(), bindingTime = ProductFamilies::FeatureModels::BindingTimeType::Compile |] | 
    relations = Seq{relation} + tail 
    in relations->iterate( r f = feature | [| f.addToExtensionPoints(r) |] ) } .

Relation ::= 
  Meronym | Hypernym | Composition .

Meronym ::= 
  'has' 'a' '{' fs = FeatureList '}'
  { 
    let node = fs->iterate(f n = [| VariationPoint[name = "Meronym"] |] | 
      [| n.addToSubFeatureNames(f) |] ) 
    in [| node |] } .
```

275
Composition ::= 'is' 'any' 'combination' 'of' '{' fs = FeatureList '}'
\{ let node = fs->iterate(f \n = [ | InclusiveOr [name = "Hypernym"] | ] |
\[ | <n>.addToSubFeatureNames(<f>) | ]
\} in [ | <node> | ] end }.

Hypernym ::= 'is' 'one' 'of' '{' fs = FeatureList '}'
\{ let node = fs->iterate(f \n = [ | Alternative [name = "Hypernym"] | ] |
\[ | <n>.addToSubFeatureNames(<f>) | ]
\} in [ | <node> | ] end }.

FeatureList ::= n = Name [ Seq\{ | <n.lift()> | ] | n = Name ',' tail = FeatureList [ Seq\{ | <n.lift()> | ] + tail | n = Name '|' tail = FeatureList [ Seq\{ | <n.lift()> | ] + tail |.

FeatureType ::= 'Optional' { ProductFamilies::FeatureModels::SelectionType::Optional | 'Mandatory' { ProductFamilies::FeatureModels::SelectionType::Mandatory | 'Root' { ProductFamilies::FeatureModels::SelectionType::Mandatory |.

Dependency ::= FeatureDependency ';'.

FeatureDependency ::= 'Feature' n = Name 'requires' '{' fs = FeatureList '}'
\{ let requires = fs->iterate(f req = [ | Dependency[name = <n.lift()>, sourceFeatureName = <n.lift()>] | ] |
\[ | <req>.addToTargetFeatureNames(<f>) | ]
\} in [ | <requires> | ] end | | end }.

end
4 Parsing of Configuration Models

The concrete (textual) syntax of the ConfigurationModel is parsed into the XMF model using the following grammar:

```plaintext
parserImport XOCL;
parsimport Parser::BNF;
import ProductFamilies::ConfigurationModels;
import ProductFamilies::FeatureModels;
import OCL;

class ConfigurationModel

@Grammar extends OCL::OCL.grammar

ConfigurationModel ::= 'n = Name ' 'From' fm = Exp 'select' 'cs = Choices ' ';
{ cs->iterate(c cm = [ConfigurationModel|name = <n.lift()>,
             featureModel = <fm>] ||
             [<cm>.addToChoices(<fm>.getFeatureByName(<c>)) [] ] ||
             [<cm>.addChoice(<fm>.getFeatureByName(<c>)) [] ] ||
             [<cm>.addChoice(<fm>.getFeatureByName(<c>)) [] ] )
   Choices ::= n = Name
{ Seq[|<n.lift()>|] ||
   n = Name ',' tail = Choices
{ Seq[|<n.lift()>|] + tail |
end
```
5 Parsing of Specification Models

The concrete (textual) syntax of the SpecificationModel is parsed into the XMF model using the following grammar:

```plaintext
parserImport X0CL;
parserImport Parser::BNF;
import ProductFamilies::SpecificationModels;
import ProductFamilies::ConfigurationModels;
import OCL;

class SpecificationModel
  @Grammar extends OCL::OCL.grammar
  SpecificationModel ::= '(' n = Name ')' 'Bind' cm = Exp 'to' abp = Exp ';'
  { [] SpecificationModel[name = <n.lift()>,
    configurationModel = <cm>,
    architecturalBlueprint = <abp>] [] } .
end
```
6 Parsing of Layered Architectural Models

The concrete (textual) syntax of the Layered Architectural Model is parsed into the XMF model using the following grammar. It is to be noted that the parser automatically creates both a Controller and Interface class for each layer. However, this is a relatively naïve implementation and would require extension to support nested architectures (however, nested architectures are supported by the underlying profile).

```plaintext
parserImport XOCL;
parserImport Parser::BNF;
import ProductFamilies::ReferenceArchitecturalComponents::ArchitecturalProfiles;
import OCL;

// helper operation to initialise the architecture created by the parser
context Architecture
@Operation initialise
(): Architecture
  // create the interfaces for each layer
  self.contains->
  collect(layer | layer.interfaceList->
  collect(n | layer.addToRealises(Interface(n))));
  // bind layers to dependencies (not required for foundation layers)
  self.contains->
  select(layer | layer.isKindOf(Layer))->
  collect(d | self.classes->select(c | c.name.toString() = d)->
  collect(i | i.addToDependsOn(d(i))));
  // create one controller per layer
  self.contains->
  collect(layer | layer.setControlledBy(Controller(layer.name + "Controller")));
  self
end

context Architecture
@Grammar extends OCL::OCL.grammar

Architecture ::= '
  (\n  n = Name )' 'layers = Layer*'
  { let a = layers->iterate[layer a = \[
    (| Architecture[name = <n.lift()]> |)]
  in \[
    <a>.initialise()
  ]
end}.

Layer ::= 'Layer' (\n  n = Name )' 'realises' (\n  'interfaces = InterfaceList |') |';
  { interfaces->iterate[interface fi = \[
    (| FoundationLayer[name = <n.lift()]> |)]
  in \[
    <fi>.addToInterfaceList(<interface>)
  ]
end}.

Foundation ::= 'Layer' (\n  n = Name )' 'depends' (\n  'dependencies = InterfaceList |') |';
  { let layer = interfaces->iterate[interface l = \[
    (| Layer[name = <n.lift()]> |)]
  in dependencies->iterate[dependency dl = \[
    <l>.addToDependencyList(<dependency>)
  ]
end}.

Parameterised ::= 'Layer' (\n  n = Name )' 
  'depends' (\n  'dependencies = InterfaceList |') |';
  { let layer = interfaces->iterate[interface l = \[
    (| Layer[name = <n.lift()]> |)]
  in dependencies->iterate[dependency dl = \[
    <l>.addToDependencyList(<dependency>)
  ]
end}.

InterfaceList ::= n = Name \[
  Seq{| <n.lift()> |}] |
  n = Name , tail = InterfaceList
  { Seq{| <n.lift()> |} | tail }.
end
```

279
7 Domain Engineering of TDL Platforms

Although the Link 16 standard [DoD04] makes reference to specific platform types (or capabilities), it does not seek to provide an explicit definition of how such platforms are related, this must be inferred from the text. An initial analysis suggests the taxonomy shown in Figure 173.

The taxonomy suggests that there are three primary features required:

1. Network Participation
2. Unit Type
3. Data Forwarding

7.1 Network Participation

This feature provides the TDL unit with the ability to connect with and participate on the net. All TDL units must implement this feature; hence it is declared mandatory. The network participation feature is decomposed into the Data Source and Subscriber sub-features, these are essentially roles that may be played by the TDL Unit, and a unit may play either or both roles; but it must play at least one role. Therefore, the sub-features Data Source and Subscriber are modelled as mandatory sub-features of the type Inclusive-Or. The Data Source may optionally provide a Position Reference feature to clients of the net. The Subscriber feature is decomposed into the sub-features Primary User and Secondary User, from which one (and only) ay be chosen, hence these sub-features are modelled as mandatory alternatives.

7.2 Unit Type

This feature allows the definition of the actual type of the TDL unit. All TDL units will incorporate this feature; hence, it is declared mandatory. A number of unit types are supported on Link 16:

- Participating Unit (PU)
- Reporting Unit (RU)
- JTIDS/MIDS Unit (JU)

A Participating Unit is defined to be a unit that is participating directly on Link 11 [DoD04], hence the unit is acting as a data forwarding unit with a Link 11 net. Therefore, the PU feature is modelled with a requires relationship with the Link 11 feature such that selection of the PU feature requires selection of the Link 11 feature. A Reporting Unit is a unit that is communicating on a point-to-point link that can be identified as a data source, this feature is modelled as having a requires relationship on both the Point-to-Point feature and the Data Source feature. Modelling the RU with a requires relationship on a Point-to-Point feature allows for future extension of the feature with other forms of Point-to-Point link, e.g. dialects of the Link 4 TDL [HJ05b]. A JU is
simply a JTIDS/MIDS unit that is communicating on Link 16, the JU may provide any one (or none) of the following capabilities:

- **C² JU** – a JU with Command and Control capability, additionally a C² JU may also provide capability to initiate real-time tracks (Surveillance System)
- **IEJU** – a JU that transmits the Initial Entry message
- **Network Manager** – a JU provided with the ability to manage the Link 16 network

Therefore, the Unit Type is modelled as having a group of mandatory alternative sub-features (JU, PU, RU) from which one (and only one) sub-feature must be selected. The JU sub-feature is modelled as having a group of optional alternative sub-features (C²JU, IEJU, Network Manager) from which a maximum of one sub-feature must be selected.

### 7.3 Data Forwarding

This feature identifies the TDL unit’s ability to communicate concurrently on other links, not all TDL units require this feature; hence it is declared *optional*. Data Forwarding is simply the transmission of data received from one TDL on another (different) TDL using the appropriate data format. Data Forwarding is provided via Point-to-Point or Broadcast. The model describes only Link 11 and Link 11B Point-to-Point links although others exist (see [HJ05a]), and we assume that a minimum of one Point-to-Point link must be selected; hence this is modelled as a group of mandatory inclusive-or features. The Broadcast feature is provided via a Generic Link (this is any non-Link 16 TDL, except Link 11/11B), or Link 16 (message repromulgation), again, if Data Forwarding is selected, a minimum of one Broadcast sub-feature must be selected; hence the sub-features Link 16 and Generic Link are modelled as a mandatory inclusive-or group.
Figure 173 – Taxonomy for TDL Units
8 Domain Engineering of TDL Function and Message Structure

The Link 16 TDL features a message-based interface; the message catalogue is defined in terms of a collection of fixed format functionally-oriented messages. Each message family comprises a set of sub-labels, and each sub-label comprises a set of J-Words, and each J-Word comprises a set of fields where the type of each field is identified by a unique code (the DFI/DUI). Hence, messages are tree-structured (as shown in Figure 174), whilst the data dictionary is defined in terms of unique DFI/DUI pairs. In this example the DFI relates to a generic concept (e.g.) the DFI no. 283 relates to the generic concept of Quality, and the DUI no. 002 within this DFI relates to the concept of Altitude Quality, such that the unique DFI/DUI 283/002 is the identifier for Altitude Quality, and is represented by the discrete values (or ranges of values) identified by the associated Data Items.

Well-formedness rules (described in prose) define structures comprising valid J-Messages, e.g. a valid J-Message must contain one Initial Word, a minimum of zero and a maximum of two Extension Words, and a minimum of zero and a maximum of 32 Continuation Words. As part of a separate body of work a metamodel of the TDL domain has been developed and is described elsewhere (e.g. see [HJR07a]), whilst the TDL metamodel is outside the scope of this body of work; a brief introduction is provided only to provide the context to this case study.

Figure 174 – Message Structure Metamodel

47 Such well-formedness rules have been transcribed as OCL constraints in our models.
8.1 Defining the Minimum Implementation Requirements

The minimum implementation (MIN IMP) requirements for participation on the Link 16 network are expressed in an hierarchical style as shown in Figure 175, such that a function may depend on the implementation of a number of related functions and the implementation of particular J-Messages, which, in turn depend on the implementation of particular J-Words, etc. (the requirements then descend through the tree structure to the leaves representing the actual values to be supported by specific fields). It is intended that the informal model of the MIN IMP requirements be read top-down, i.e. first one identifies the functions to be implemented, these then reveal the related functions required, J-Messages, etc. A TDL platform is free to implement features in addition to those identified by the MIN IMP rules, however such additional features will incur further MIN IMP rules that must be satisfied.

The minimum implementation requirements are partitioned (coarsely) into C2 and non-C2 requirements48, and are described in a single appendix of the standard comprising nearly 500 pages of information. Although the implementation of some TDL functions is mandatory, many are optional, and some are interdependent. The selection of optional features of the specification then brings into play all mandatory features stemming from the selected feature; this is a view that resonates well with the notion of a feature model/PLA.

The MIN IMP requirements are expressed in a mixture of prose and tabular form (primarily the latter), however the form is not sufficiently rigorous to provide simple tool support to facilitate the checking of correctness of either the specification itself, or implementations derived from the specification. The cells of the tables are text; there are no hyperlinks between elements within or across tables, and there are no links back to either the message catalogue or the data dictionary. Hence one may conclude that there is considerable scope for inconsistency; this is known to be true, we have encountered maintenance issues where (e.g.) the message catalogue has been amended without propagating the necessary change through to the minimum implementation requirements. This does not necessarily appeal for the use of feature modelling or the SPLA, rather the use of a model-based approach in general.

48 Provision is made for a finer grained partitioning via the FARS, but this is left undefined by the standard.
Figure 175 – Minimum Implementation Hierarchy

As an example, the top-level minimum implementation requirement for a C² platform is taken from [DoD04] and presented (in a slightly enhanced form) in Table 6 in the form of an N2 table. From this table we can see that each function is considered to be available in the form of an output from (Tx) and/or an input to (Rx) the platform, and that the dependency relationship is really at the level of granularity of the underlying J-Messages. The example shown in Table 6 has been refined to make explicit the Tx/Rx dependency, and augmented with the use of colour to improve readability; however one of the main issues with this is that Function MIN IMP requirements have no formal relationship with the tables that define the next level of decomposition of the dependency relationships, the message MIN IMP requirements. Furthermore, once we begin to investigate the message level MIN IMP requirements, we begin to encounter more complex constraints based on (e.g.) terminal type; such as if X is a JU that is to be a forwarding unit then Y must hold. We also begin to see the emergence of vague requirements, e.g. the applicable PPLI messages for the type of unit shall be implemented; where the definition of the applicable message set is either vague or even undefined. Although this may be considered to be a concise notation, tool support for the enumeration of such statements would be beneficial and support a more formal validation of the TDL platform specification. This is also the point at which a clear taxonomy becomes beneficial, as an example see [HJ05b].
### Table 6 – C2 Platform Minimum Implementation Requirements

<table>
<thead>
<tr>
<th>Function</th>
<th>SIENM</th>
<th>PPLI</th>
<th>Air Surv</th>
<th>Surface Surv</th>
<th>Sub Surface Surv</th>
<th>Land Surv</th>
<th>Space Surv</th>
<th>Elec Surv</th>
<th>EW/Intel</th>
<th>Mission Mgmt</th>
<th>Wpns Coord &amp; Mgmt</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIENM</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PPLI</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>Rx</td>
<td>Rx</td>
<td>Rx</td>
<td>Rx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Air Surv</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>Tx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Surface Surv</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Sub Surface Surv</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Land Surv</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Space Surv</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Elec Surv</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>EW/Intel</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Mission Mgmt</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Wpns Coord &amp; Mgmt</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Control</td>
<td></td>
<td></td>
<td>Tx</td>
<td>Rx</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Key:**
- Mandatory
- Optional
- Choose one or more

---

Table 6 – C2 Platform Minimum Implementation Requirements
The next level of decomposition of minimum implementation requirements is at the level of the J-Message, once again the requirements are specified in tabular form. However, in this instance, they take the form of a set of functionally oriented mandatory requirements bearing little or no relationship to the parent functional requirements shown at Table 6, and are followed by a set of tables detailing the message implementation requirements for the functions Air Surveillance to Control, as identified in Table 6. Whilst we can assume that the six initial tables relate to the implementation of the SIENM and PPLI functions, there is no direct traceability as is illustrated in Figure 176. Hence, we must assume that the Net Control Station Message feature is some composite component of (e.g.) SIENM – but this relationship is not made explicit and is clearly open to misinterpretation.

![Figure 176 – Minimum Implementation Decomposition (function to message)](image-url)
The minimum implementation requirements for each J-Message are then mapped to the minimum implementation requirements of the constituent words, and these are, in turn, mapped down onto the minimum implementation requirements for the data elements (DFI/DUIs) that comprise the fields of each word. Finally, the minimum implementation requirements are defined for each data element (DFI/DUI) in terms of the values (DIs) that must be supported. There are essentially two forms of data element, those that support the notion of a No Statement\textsuperscript{49} value and those that don’t. Hence, for any given top-level function, the implementation requirements should be clear, right down to the level of the values to be supported by each field within each J-Word.

In addition to the lack of precision and traceability evident in the specification of the network minimum implementation requirements, one must wonder about the maintenance overhead involved in retaining consistency between so many complex tables manually, and the implication of the occurrence of errors. Clearly, if there were to be a model containing the semantics of the minimum implementation rules, it would be a very small step to then be able to evaluate a TDL platform specification against this model to determine its validity and identify areas of non-compliance, more semantically aware models might then be able to provide insight into the effect on platform capability caused by such non-compliances.

Casting the TDL function hierarchy in the form of a PLA provides a concise and checkable form of MIN IMP definition and also provides the ability to validate any configuration against the feature model of the TDL functions. Furthermore, the generative PLA approach provides the additional capability of supporting the generation of components of the PRS from the standard; this would otherwise be a challenging manual task.

\textsuperscript{49} ‘No Statement’ is used to imply that no data is available for a particular data element.
9 Domain Engineering of Navigation Function

The Navigation domain seeks to establish the vehicle’s state vector and provide an answer to the fundamental question: *where are we?* This provides a view of the vehicle’s location in space and the forces acting upon it. For the navigation domain to provide this view of the vehicle it requires inputs from relevant sensors (in the Sensor Domain) and must report the vehicle’s state vector within the context of some reference frame such as WGS84 [DMA91], this is referred to (generically) as an Earth Model. The vehicle’s state vector may be used by higher level components to (e.g.) provide steering information via a steering model, and this information may be rendered to the user via one or more display formats. Each of the domains of relevance to the demonstration of a PLA based avionics system implementing a navigation function is described below with the associated models, from these models and the description of the PLA we will demonstrate the generation of custom models from a kit of parts.

9.1 The Sensor Domain

The Sensor domain is responsible for monitoring aspects of the external environment within which the system is operating; this is the domain in which we find the boundary objects that provide the interface between the system and its environment. Components within the sensor domain relevant to navigation are partitioned in accordance with Figure 177, as reported by Coglianese et al. in [CGK+93]. There may be an additional Effectors domain [Lea94] containing all boundary objects required to control or provide output to components that may change the state of the vehicle, some equipments may provide both sensors and effectors, e.g. a Stores Management Computer will report the state of the ordinance, and also support jettison and release commands. For the purposes of this case study we restrict our view of the boundary objects to those of the sensor domain only.

Coglianese et al. decompose the Sensors domain further into *Realms* according to their logical type, e.g. an Inertial Navigation System (INS), and have their physical (vendor specific) interfaces

![Figure 177 – Sensor Families (Hierarchy)](image-url)
shielded from the application via a logical interface; an interface all members of the realm are expected to export. Figure 178 provides an example of the realms expressed in the form of GenVoca type equations.\footnote{For more information on GenVoca see [Hol01a]}

Air Data:

\[
\text{ADC} = \text{AiResearch\_CP-1471A}, \ldots ;
\]

Inertial:

\[
\begin{align*}
\text{IN} & = \text{FIN\_1075, ASN\_130, LN\_100,} \ldots ; \\
\text{AHRS} & = \text{FIN\_1075, ASN\_130, LN\_100,} \ldots ;
\end{align*}
\]

Electromagnetic (External):

\[
\text{GPS} = \text{LN\_100,} \ldots ;
\]

Electromagnetic (Internal):

\[
\begin{align*}
\text{RADALT} & = \text{TBD,} \ldots ; \\
\text{TACAN} & = \text{TBD,} \ldots ;
\end{align*}
\]

Database:

\[
\text{TERPROM} = \text{TBD,} \ldots ;
\]

Figure 178 – Sensor Families (GenVoca Type Equations)

In this example it can be seen that product line variability is being mixed with architecture, whilst variability may be expressed more clearly in the form of a feature model of Sensors (see section 6.3.3).

The realms comprising the Sensor domain are summarised below to illustrate the role of each realm with respect to the navigation domain:

- **Air Data Realm**: The Air Data sensor family encompasses sensors providing data about the airmass through which the vehicle is travelling; this includes (but is not limited to) such items as:
  - Indicated Airspeed (IAS)
  - True Airspeed (TAS)
  - Angle of Attack (AoA)
  - Outside Air Temperature (OAT)
  - Air Density
  - Barometric Altitude

- **Inertial Sensor Realm**: The Inertial sensor family provides state vector information derived from (e.g.) gyroscopic sensors and accelerometers installed on the air vehicle, i.e. the
classical inertial platform or modern strapdown inertial navigation systems. The Inertial
sensor family covers sensors of the type Inertial Navigation System (INS) and Attitude
Heading Reference System (AHRS). Inertial data includes (but is not limited to) such items
as:

- Present Position (PP)
- Heading
- Velocity (in air vehicle axes)
- Euler angles
- Roll Rate (not provided by AHRS)
- Pitch Rate (not provided by AHRS)

• Electromagnetic Sensor Realm: The Electromagnetic sensor family provides state vector
information. This family is subdivided into the Internal and External families depending on
the level of autonomy of the sensor.
  - External Electromagnetic Sensor Realm: The External Electromagnetic sensor
    family provides state vector information via information provided by some other
    external equipment, e.g. Global Positioning System (GPS) requires data from a
    constellation of satellites, a TACAN receiver requires a signal transmitted by a
    TACAN station, etc.
  - Internal Electromagnetic Sensor Realm: The Internal Electromagnetic sensor
    family provides state vector information via one or more on-board electromagnetic
    sensors, e.g. Radar Altitude (Rad Alt), Doppler Navigation System (DNS), etc.

• Database Sensor Realm: The Database sensor family provides state vector information by
  comparing measured terrain against a digitised chart using complex algorithms, e.g.
  TERPROM; it is generally the case that, in order for this sensor to function correctly,
  outputs from other sensors are required, such as height above ground, provided by (e.g.) a
  Rad Alt, and an estimate of position, provided by (e.g.) an INS.

The model of the Sensors domain follows the structure identified by Coglianese et al. [CGK+93],
see Figure 179, and each component is populated with a number of representative classes.

![Figure 179 – Top-Level View of Sensors Model](image-url)
The package `Sensors::AirData` comprises an abstract root class encapsulating aspects common to all Air Data Computers, and also provides two simple concrete classes representing specific devices (Figure 180).

The package `Sensors::Inertial` comprises an abstract root class encapsulating aspects common to all Attitude Heading Reference Systems; from this we drive two simple concrete AHRS classes. The abstract class `AttitudeHeadingReferenceSystem` is extended by the abstract class `InertialNavigationSystem` to provide a class from which individual INS components may be derived; two example three example INS platforms are provided (Figure 181).
Figure 181 – Example Model of AHRS and INS Components

The package Sensors::Electromagnetic is partitioned to differentiate between internal and external electromagnetic sensors (Figure 182).

Figure 182 – Component Packages of Electromagnetic Sensors

Package Sensors::Electromagnetic::Internal comprises classes representing example electromagnetic sensors that may be wholly contained by the vehicle, i.e. electromagnetic sensors that are able to operate without the aid of any external equipment (Figure 183).
Figure 183 – Example Model of Internal Electromagnetic Sensors

Package Sensors::Electromagnetic::External comprises classes representing example electromagnetic sensors the operation of which requires equipment external to the vehicle. An abstract root class GlobalPositioningSystem provides a foundation from which two example GPS equipments are derived, and the class Tacan captures the general characteristics of a TACAN receiver (Figure 184).

Figure 184 – Example Model of External Electromagnetic Sensors

The package Sensors::Databases provides an example class Terprom that requires data from a Radar Altimeter and an Inertial Navigation System to estimate the vehicles position (Figure 185).
We have described a rich collection of models appropriate to the Sensors domain. These models will provide the foundation upon which the subsequent domains will build in the sections that follow.

### 9.2 The Navigation Domain

The Navigation domain is responsible for establishing the vehicle’s state vector relative to some coordinate frame such as WGS84; this provides a view of where the vehicle is in space and the forces acting upon it. The Navigation domain is also responsible for maintaining a view of the state vector of any Points of Interest (POIs), such as Waypoints, TACAN stations, targets, etc.

Navigation data is usually provided to components in other domains to support higher level functions, such as guidance.

The vehicle needs to know where it is in space at any particular time. The data elements comprising the vehicle’s state vector will be derived from the relevant sensors (see Section 9.1) and comprises such items as:

- Positional information (latitude/longitude/height)
- Heading information
- Attitude information
- Motion through the air-mass

The Navigation domain may be decomposed into the sub-domains Atmosphere Model, Earth Models, Transient Object Models, and Air Vehicle Models as illustrated in Figure 186.
The Atmosphere Model provides a view of the atmospheric conditions affecting the air vehicle (e.g. cross-wind), deriving it’s calculations from the INS and ADC sensors (Figure 187).

The Earth Models component is responsible for providing a common frame of reference for all positional information, and also provides the means of transformation between positions specified in different frames of reference, e.g. OSGB 1936 to WGS84 (or vice versa). With the exception of WGS84, all other positions would be specified in the form of a Local Geodetic Datum that is based on an Ellipsoidal Model (e.g. OSGB 1936 is based on the Airy ellipsoidal model); this leads to the package structure shown in Figure 188.
Figure 188 – Package Structure of Earth Models Components

Package EllipsoidalModels contains an abstract root class from which a number of representative ellipsoidal models are derived, a factory class is also provided (not shown). Ellipsoidal models generally differ only in the values assigned to their attributes defining the shape of the ellipsoid approximating to the shape of the geoid (Figure 189).

Figure 189 – Example Ellipsoidal Model Components

The Local Geodetic Datum (LGD) provides an adjustment to the ellipsoidal model in order to provide a closer fit to the geoid within some specific geographical region, hence the LGD has an
association relation to a reference ellipsoid and an origin about which the LGD is defined (Figure 190).

Finally, package **DatumTransformations** provides the algorithms required to support transformation of coordinates between the frames provided by the Local Geodetic Datum, typically this is from any LGD into (and out of) WGS84. The example transformations are provided via two placeholder classes: **HelmertFormula** and **StandardMolodenski** (not shown).

The package **TransientObjectModels** provides information on real-world objects that may be used and discarded as the air vehicle performs its mission. The type of objects supported by such models includes waypoints, targets, and navigation beacons (e.g. TACAN, VOR, etc.). The Transient Object Model is described in the form of an inheritance hierarchy of Transient Objects and a corresponding hierarchy of Transient Object databases to provide access to collections of Transient Objects (see Figure 191).
Figure 191 – Example Transient Object Model Components

The package **AirVehicleModels** is responsible for calculating the state vector of the air vehicle and may also provide information about the major aerodynamic components, such as the flight control surfaces, engines, etc. The **AirVehicle** is modelled in the form of a whole-part pattern and is illustrated in Figure 192.

Figure 192 – Example Air Vehicle Model Components

At this point we have described a rich collection of models appropriate to the Navigation domain partitioned across a number of sub-domains. These models build on the services provided by the Sensors domain (section 9.1).
9.3 The Mission Management Domain

The Mission Management domain is responsible for providing the steering information (and commands) necessary to guide the vehicle from point A to point B at the required time and following the appropriate route. The vehicle needs to know its state vector and that of the intended destination (POI). The information defining the POI is provided by a database of items in the Transient Object Models and is supplied in an appropriate reference frame. The air vehicle provides its state vector, and the Transient Object Model provides the state vector of the POI, hence the mission management domain provides the steering models required to guide the air vehicle to the POI. Navigation steering information to the POI provides scope for variability, e.g. the vehicle may be required to adopt a number of strategies to plan the route to the target, e.g. shortest, fastest, most economical, threat avoidance, airways steering, or following specific search patterns.

The Mission Management domain is supported by a single sub-domain comprising the steering models to be supported, each of which is modelled in its own package within SteeringModels (Figure 193).

![Figure 193 – Example Steering Model Components](image)

The components TakeOffAndLanding, FlightAids, and SearchPatterns provide steering models for specific phases of flight, not all of which may be appropriate for all air vehicles. Package TakeOffAndLanding provides steering strategies for take-off, landing, and missed approaches (Figure 194).
Figure 194 – Example Take-Off and Landing Steering Model Components

Package FlightAids provides components to support general purpose in-flight steering missions, such as Cruise and HoverHold (Figure 195).

Figure 195 – Example Flight Aids Steering Model Components

Finally, package SearchPatterns provides the steering missions that might be required to support Search and Rescue missions (Figure 196). Typically, such missions require the steering patterns Expanding Square, Creeping Line, and Sector Search, see [GC92] for examples.
At this point we have described a rich collection of models appropriate to the Mission Management domain in support of a number of steering models. These models build on the services provided by the Navigation domain (section 9.2).

## 9.4 The MMI Domain

The MMI domain is responsible for rendering views of the system state and receiving inputs from the user. For the purposes of this case study we restrict the components of the MMI domain to a small number of simplified classes required to render views relevant to avionics navigation; all of the classes are contained in the Displays package (Figure 197).
Figure 197 – Modelling of MMI Domain
This concludes the description of the domain engineering activities relating to the description of a rich set of models appropriate to the avionics navigation domain. Clearly, not all avionics navigation systems would require all of the functionality described above, in the sections that follow we will describe the allocation of model elements to features and the structure of a feature model for the avionics navigation system such that we can demonstrate the application of a PLA approach using the metamodels described in Chapter 5.
10 TDL Functions – Example Feature Models

The following listing provides an example definition of the minimum implementation requirements for the Link 16 TDL Functions for a C2JU with associated J-Message minimum implementation requirements.

```ruby
import Root::ProductFamilies::FeatureModels::FeatureModel;
import Root::ProductFamilies::FeatureModels::Feature;
import Root::ProductFamilies::FeatureModels::ExtensibleFeature;
import Root::ProductFamilies::FeatureModels::VariationPoint;
import Root::ProductFamilies::FeatureModels::InclusiveOr;
import Root::ProductFamilies::FeatureModels::Alternative;
import Root::ProductFamilies::FeatureModels::Dependency;

Root::tdlFunctionFM := @FeatureModel(TdlFunctionFeatureModel)
    Root Feature(Function) has a {SIENM, PPLI, Support, MessageFamily};
    Mandatory Feature(SIENM) has a {SIENM_Tx, SIENM_Rx};
    Mandatory Feature(SIENM_Tx) is leaf;
    Mandatory Feature(SIENM_Rx) is leaf;
    Mandatory Feature(PPLI) has a {PPLI_Tx, PPLI_Rx};
    Mandatory Feature(PPLI_Tx) is leaf;
    Mandatory Feature(PPLI_Rx) is leaf;
    Optional Feature(Support) has a {Surveillance, EwIntelligence, Control, Management};
    Optional Feature(Surveillance) is any combination of {Air, Surface, SubSurface, Land, Space, Electronic};
    Mandatory Feature(Air) is any combination of {Air_Tx, Air_Rx};
    Mandatory Feature(Air_Tx) is leaf;
    Mandatory Feature(Air_Rx) is leaf;
    Mandatory Feature(Surface) is any combination of {Surface_Tx, Surface_Rx};
    Mandatory Feature(Surface_Tx) is leaf;
    Mandatory Feature(Surface_Rx) is leaf;
    Mandatory Feature(SubSurface) is any combination of {SubSurface_Tx, SubSurface_Rx};
    Mandatory Feature(SubSurface_Tx) is leaf;
    Mandatory Feature(SubSurface_Rx) is leaf;
    Mandatory Feature(Land) is any combination of {Land_Tx, Land_Rx};
    Mandatory Feature(Land_Tx) is leaf;
    Mandatory Feature(Land_Rx) is leaf;
    Mandatory Feature(Space) is any combination of {Space_Tx, Space_Rx};
    Mandatory Feature(Space_Tx) is leaf;
    Mandatory Feature(Space_Rx) is leaf;
    Mandatory Feature(Electronic) is any combination of {Electronic_Tx, Electronic_Rx};
    Mandatory Feature(Electronic_Tx) is leaf;
    Mandatory Feature(Electronic_Rx) is leaf;
    Optional Feature(EwIntelligence) is any combination of {EwIntelligence_Tx, EwIntelligence_Rx};
    Mandatory Feature(EwIntelligence_Tx) is leaf;
    Mandatory Feature(EwIntelligence_Rx) is leaf;
    Optional Feature(Control) is any combination of {Control_Tx, Control_Rx};
    Mandatory Feature(Control_Tx) is leaf;
    Mandatory Feature(Control_Rx) is leaf;
    Optional Feature(Management) is any combination of {Mission, WeaponsCoordination};
    Mandatory Feature(Mission) is any combination of {Mission_Tx, Mission_Rx};
    Mandatory Feature(Mission_Tx) is leaf;
    Mandatory Feature(Mission_Rx) is leaf;
```

305
Mandatory Feature(WeaponsCoordination) is any combination of {WeaponsCoordination_Tx, WeaponsCoordination_Rx};
Mandatory Feature(WeaponsCoordination_Tx) is leaf;
Mandatory Feature(WeaponsCoordination_Rx) is leaf;
Mandatory Feature(MessageFamily) is any combination of {J3, J5, J7};
Mandatory Feature(J3) is any combination of {J3_0, J3_1, J3_2, J3_3, J3_4, J3_5, J3_6, J3_7};
Mandatory Feature(J3_0) is leaf;
Mandatory Feature(J3_1) is leaf;
Mandatory Feature(J3_2) is leaf;
Mandatory Feature(J3_3) is leaf;
Mandatory Feature(J3_4) is leaf;
Mandatory Feature(J3_5) is leaf;
Mandatory Feature(J3_6) is leaf;
Mandatory Feature(J3_7) is leaf;
Mandatory Feature(J5) is any combination of {J5_4};
Mandatory Feature(J5_4) is leaf;
Mandatory Feature(J7) is any combination of {J7_0, J7_1, J7_2, J7_3, J7_4, J7_5, J7_6, J7_7};
Mandatory Feature(J7_0) is leaf;
Mandatory Feature(J7_1) is leaf;
Mandatory Feature(J7_2) is leaf;
Mandatory Feature(J7_3) is leaf;
Mandatory Feature(J7_4) is leaf;
Mandatory Feature(J7_5) is leaf;
Mandatory Feature(J7_6) is leaf;
Mandatory Feature(J7_7) is leaf;
Feature Mission requires {WeaponsCoordination_Tx, WeaponsCoordination_Rx};
Feature Control requires {Air_Tx, Air_Rx, WeaponsCoordination_Tx, WeaponsCoordination_Rx};
Feature EwIntelligence requires {Surveillance};
Feature Air_Tx requires {J3_2, J7_0, J7_5};
Feature Air_Rx requires {J3_1, J3_2, J7_0, J7_5};
end;

The above feature model description is rendered using the graphical concrete syntax described in [CE00] in Figure 198 through Figure 206.
Figure 198 – Top-Level TDL Function Feature Model

Decomposed on Figure 199

Decomposed on Figure 202

Decomposed on Figure 204

Decomposed on Figure 203
Figure 199 – TDL Support Feature Decomposition
Figure 200 – Management Feature Decomposition
Figure 201 – Surveillance Feature Decomposition
Figure 202 – Surveillance Message Feature Decomposition
Figure 203 – Information Management Message Feature Decomposition
Figure 204 – Antisubmarine Warfare Message Feature Decomposition
Figure 205 – TDL Function Interdependencies
Figure 206 – Air Surveillance Feature Message Dependencies
11 Avionics Navigation System – Example Feature Model

The following listing provides the complete definition of the Avionics Navigation System feature model expressed in the textual concrete syntax admitted by the grammar described in Appendix 3.

```plaintext
parserImport Root::ProductFamilies::FeatureModels;
import Root::ProductFamilies::FeatureModels::FeatureModel;
import Root::ProductFamilies::FeatureModels::Feature;
import Root::ProductFamilies::FeatureModels::ExtensibleFeature;
import Root::ProductFamilies::FeatureModels::VariationPoint;
import Root::ProductFamilies::FeatureModels::InclusiveOr;
import Root::ProductFamilies::FeatureModels::Alternative;
import Root::ProductFamilies::FeatureModels::Dependency;

Root::navigationFM := @FeatureModel(FeatureModel_Navigation)
  Root Feature(AvionicSystem) has a {Sensors, EarthModels, AtmosphereModel, TransientObjectsModel, AirVehicleModel, SteeringModel, DisplayFormats};

  // Display Formats
  Optional Feature(DisplayFormats)
    is any combination of {AtmosphereDisplay, SteeringDisplay, BeaconDisplay, WaypointDisplay, TargetDisplay};

  Mandatory Feature(AtmosphereDisplay) is leaf;
  Mandatory Feature(SteeringDisplay) is leaf;
  Mandatory Feature(BeaconDisplay) is leaf;
  Mandatory Feature(WaypointDisplay) is leaf;
  Mandatory Feature(TargetDisplay) is leaf;

  Optional Feature(SteeringModel)
    is any combination of {FlightAids, SearchPatterns, TakeOffAndLanding};

  Mandatory Feature(FlightAids) has a {HoverHold};
  Optional Feature(HoverHold) is leaf;
  Mandatory Feature(SearchPatterns) is leaf;
  Mandatory Feature(TakeOffAndLanding) is leaf;

  // Air Vehicle Model
  Mandatory Feature(AirVehicleModel) has a {MainAirVehicleComponents, FlightControlSurfaces};

  Mandatory Feature(MainAirVehicleComponents) is leaf;
  Optional Feature(FlightControlSurfaces) is leaf;

  // Transient Objects Model
  Mandatory Feature(TransientObjectsModel) has a {TransientObject, TransientObjectDatabase};

  Mandatory Feature(TransientObject)
    is any combination of {Beacon, Waypoint, Target};
  Mandatory Feature(Beacon) is any combination of {TacanStation, VorGroundStation};

  Mandatory Feature(TacanStation) is leaf;
  Mandatory Feature(VorGroundStation) is leaf;
  Mandatory Feature(Waypoint) is leaf;
  Mandatory Feature(Target) is leaf;

  Optional Feature(TransientObjectDatabase)
    is any combination of {BeaconDatabase, WaypointDatabase, TargetDatabase};

  Mandatory Feature(BeaconDatabase) is leaf;
  Mandatory Feature(WaypointDatabase) is leaf;
  Mandatory Feature(TargetDatabase) is leaf;

  // Atmosphere Model
  Mandatory Feature(AtmosphereModel) is leaf;
```

316
// Earth Models
Mandatory Feature(EarthModels) has a [EllipsoidalModels,
   LocalGeodeticDatum,
   DatumTransformation];
Optional Feature(DatumTransformation) is one of [StandardMolodenski, HelmertFormula];
Mandatory Feature(StandardMolodenski) is leaf;
Mandatory Feature(HelmertFormula) is leaf;
Optional Feature(LocalGeodeticDatum) is any combination of [Indian, ED50, OSGB36];
Mandatory Feature(Indian) is leaf;
Mandatory Feature(ED50) is leaf;
Mandatory Feature(OSGB36) is leaf;
Mandatory Feature(EllipsoidalModels) has a [WGS_1984] and
   is any combination of [International_1924,
   Airy_1830,
   Clarke_1880,
   Everest_1830];
Mandatory Feature(WGS_1984) is leaf;
Optional Feature(International_1924) is leaf;
Optional Feature(Airy_1830) is leaf;
Optional Feature(Clarke_1880) is leaf;
Optional Feature(Everest_1830) is leaf;
Mandatory Feature(EllipsoidalModels) has a [AirData, Inertial, Database,
   Electromagnetic_External,
   Electromagnetic_Internal];
Mandatory Feature(AirData) is one of [AirResearch_CP1471A, MyADC];
Mandatory Feature(AirResearch_CP1471A) is leaf;
Mandatory Feature(MyADC) is leaf;
Mandatory Feature(Inertial) is one of [INS, AHRS];
Mandatory Feature(INS) is one of [FIN_1075, ASN_130, LN_100];
Mandatory Feature(FIN_1075) is leaf;
Mandatory Feature(ASN_130) is leaf;
Mandatory Feature(LN_100) is leaf;
Mandatory Feature(AHRS) is one of [AHRS_1, AHRS_2];
Mandatory Feature(AHRS_1) is leaf;
Mandatory Feature(AHRS_2) is leaf;
Optional Feature(Electromagnetic_External) has a [TACAN, GPS];
Optional Feature(TACAN) is leaf;
Optional Feature(GPS) is one of [GPS_1, GPS_2];
Mandatory Feature(GPS_1) is leaf;
Mandatory Feature(GPS_2) is leaf;
Mandatory Feature(Electromagnetic_Internal) has a [Radalt, DNS];
Mandatory Feature(Radalt) is leaf;
Optional Feature(DNS) is leaf;
Optional Feature(Database) has a [TERPROM];
Optional Feature(TERPROM) is leaf;
// Feature Dependencies
// Display Formats
Feature AtmosphereDisplay requires [AtmosphereModel];
Feature SteeringDisplay requires [SteeringModel];
Feature BeaconDisplay requires [BeaconDatabase];
Feature WaypointDisplay requires [WaypointDatabase];
Feature TargetDisplay requires [TargetDatabase];
// Steering Model
Feature FlightAids requires [AirVehicleModel];
Feature SearchPatterns requires [AirVehicleModel, TargetDatabase];
Feature TakeOffAndLanding requires [AirVehicleModel, WaypointDatabase];
// Transient Objects Model
Feature BeaconDatabase requires [Beacon];
Feature WaypointDatabase requires [Waypoint];
Feature TargetDatabase requires [Target];
// Atmosphere Model
Feature AtmosphereModel requires [AirData, Inertial];
// Earth Models
Feature LocalGeodeticDatum requires {DatumTransformation};
Feature Indian requires {Everest_1830};
Feature ED50 requires {International_1924};
Feature OSGB36 requires {Airy_1830};
// Sensors
Feature TERPROM requires {Radalt, INS};
end;
12 Generated Avionics Navigation System

Figure 207 – Top-Level Navigation System Architecture

Figure 208 – Generated SubSystem Interfaces Layer
Figure 209 – Generated Sensors Components

Figure 210 – Generated Domain Model Component
Figure 211 – Generated Atmosphere Model

Figure 212 – Generated Earth Model
Figure 213 – Generated Transient Objects Model

Figure 214 – Generated Air Vehicle Model
Figure 215 – Generated Domain Services Layer

Figure 216 – Generated Mission Management
Figure 217 – Generated MMI Layer

Figure 218 – Generated Display Formats
13 Index

A
abstract syntax 18, 21, 24, 32, 36, 37, 38, 39, 40, 41, 42, 51, 119, 126, 134, 137, 139, 141, 150, 156, 169, 170, 235, 245
architecture description language (ADL) .................................................................70, 73, 76, 77, 253
avionics domain application generation environment (ADAGE)........ 94, 95, 172, 201, 223, 230, 231, 253
C
common warehouse model (CWM) ..........................................................................23, 35, 253
component-based development (CBD) ........................................................................55, 56
components
baroque ..................................................................................................................17, 58, 101
specialisation of ......................................................................................................56
concrete syntax ....18, 24, 32, 36, 37, 38, 39, 40, 41, 42, 47, 51, 67, 82, 84, 104, 108, 119, 130, 131, 134, 137, 139, 148, 150, 154, 157, 158, 159, 166, 167, 170, 172, 175, 176, 184, 198, 205, 211, 218, 235, 245, 306, 316
configuration model .............................................................................................103, 108, 109, 112, 142, 143, 148, 150, 161, 251, 252
costs .....18, 19, 22, 24, 27, 34, 35, 37, 55, 64, 65, 73, 77, 79, 80, 81, 84, 85, 90, 94, 101, 102, 108, 109, 116, 118, 119, 121, 122, 135, 136, 138, 140, 141, 142, 143, 144, 145, 146, 150, 152, 153, 154, 156, 166, 168, 169, 178, 180, 188, 190, 192, 205, 211, 217, 218, 219, 231, 233, 237, 240, 244, 245, 249, 250, 251, 252, 283, 285
D
domain-specific language (DSL)..............................................................................18, 21, 22, 35, 36, 52, 67, 68, 253
E
eXecutable OCL (XOCL) .........................................................................................47, 84, 119, 169, 233, 255
F
feature diagram ...........................................................................................................17, 27, 58, 59, 60, 63, 67, 98, 104, 112, 118, 168, 209, 236, 239
feature modelling ..................................................18, 24, 25, 57, 58, 60, 67, 84, 85, 97, 98, 99, 119, 137, 198, 235, 243, 244, 284
feature oriented domain analysis (FODA) ..............................................................26, 56, 59, 97, 98, 118, 168, 253
features
alternative ...........................................61, 62, 98, 124, 126, 127, 128, 130, 137, 144, 145, 146, 151, 161, 237, 240, 251
homogenous ..................................................................................................................130, 166
inclusive-or .................................................................................................................128, 30, 153, 237, 240, 242, 243
variation point ...........................................................................................................69, 118, 125, 128, 144, 145, 153, 237, 240, 242, 252
frame technology .......................................................................................................17, 25, 50, 66, 67, 85, 86, 118, 201, 289, 295, 296, 300
frameworks ................................................................................................................16, 50, 52, 56, 69, 73, 117

G

generative programming .........................................................................................17, 25, 56, 66, 101, 102

L

language-driven development (LDD) ......................................................................27, 45, 46, 47, 50, 51, 240, 254

M

meta object facility (MOF) .......................................................................................18, 21, 22, 23, 24, 25, 29, 33, 35, 36, 64, 77, 84, 86, 103, 119, 243, 244, 249, 252, 254
metamodelling ........................................................................................................18, 21, 24, 25, 27, 29, 32, 33, 34, 36, 37, 42, 45, 52, 83, 84, 85, 98, 119, 157, 165, 174, 240
meta-programming system (MPS) ...........................................................................50, 51, 84, 254
mixin-inheritance ....................................................................................................74
mixin-layers ...............................................................................................................17
model driven architecture (MDA) .........................................................................17, 21, 22, 25, 31, 34, 44, 45, 50, 51, 86, 93, 98, 102, 113, 116, 167, 254
model-driven development (MDD) ........................................................................103, 165, 245, 254
model-driven software development (MDSD) .......................................................37, 50, 51, 57, 254

O

object constraint language (OCL) ............................................................................29, 77, 254, 255, 283

P

patterns .......................................................................................................................33, 48, 51, 56, 67, 68, 69, 70, 73, 74, 75, 82, 114, 167, 246, 248, 250, 299, 301
platform-independent model (PIM) .........................................................................44, 45, 93, 113, 167, 254
platform-specific model (PSM) ...............................................................................44, 93, 113, 167, 255
profile .......................................................................................................................17, 18, 22, 24, 28, 29, 31, 33, 36, 41, 53, 61, 64, 65, 77, 97, 116, 158, 159, 160, 198, 231, 244, 246, 252, 255

Q

queries views and transformations (QVT) ...............................................................18, 45, 255

R

reuse ..........................................................................................................................16, 25, 28, 29, 41, 44, 45, 50, 52, 54, 55, 56, 58, 59, 70, 75, 77, 88, 90, 101, 102, 113, 169, 230, 242, 245, 246, 253
class-based ...............................................................................................................77
kit of parts .............................................................................................................. 17, 25, 134, 229, 230, 289

S

semantics .... 17, 18, 21, 22, 23, 24, 27, 29, 31, 32, 33, 34, 36, 37, 40, 41, 45, 51, 52, 54, 57, 60, 61, 62, 64, 65,
66, 67, 68, 69, 70, 73, 76, 77, 84, 89, 92, 95, 97, 99, 101, 102, 103, 105, 106, 109, 112, 113, 118, 119, 124,
126, 127, 128, 129, 130, 142, 143, 144, 145, 146, 147, 150, 151, 152, 153, 166, 167, 169, 171, 198, 231,
233, 235, 237, 239, 242, 243, 244, 245, 288

software factories ...................................................................................................... 33, 67, 68, 69, 108, 118

software product line architecture (SPLA).16, 17, 18, 22, 24, 25, 26, 27, 29, 56, 57, 60, 64, 67, 68, 69, 85, 87,
169, 170, 171, 175, 187, 198, 230, 231, 236, 239, 240, 242, 244, 246, 248, 249, 250, 255, 284

specification model ...................................................................................................... 103, 161, 252

SysML .......................................................................................................................... 29, 35, 53, 54, 119, 255

T

tactical data link

  link 16 ................................................................. 46, 171, 174, 175, 183, 193, 195, 198, 231, 232, 280, 281, 283, 284, 305
tactical data link (TDL) 46, 48, 94, 171, 172, 174, 175, 178, 182, 183, 184, 187, 190, 192, 193, 194, 198, 230,

U

unified modeling language (UML).17, 18, 21, 22, 23, 25, 28, 29, 31, 32, 33, 34, 35, 36, 41, 44, 46, 52, 53, 54,
61, 64, 66, 68, 69, 76, 77, 83, 86, 97, 99, 101, 119, 243, 244, 249, 252, 255

X

XMF

Mosaic...22, 24, 25, 32, 36, 39, 47, 51, 78, 83, 84, 97, 98, 118, 119, 134, 140, 150, 171, 172, 230, 243, 245, 248

XCore .......................................................................................................................... 33, 51, 120, 244

XRules ...................................................................................................................... 248

XTools ....................................................................................................................... 51, 172, 245