Trusted Product Lines

Stuart Graham Hutchesson

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

February 2013

Abstract

This thesis describes research undertaken into the application of software product line approaches to the development of high-integrity, embedded real-time software systems that are subject to regulatory approval/certification. The motivation for the research arose from a real business need to reduce cost and lead time of aerospace software development projects.

The thesis hypothesis can be summarised as follows:

It is **feasible** to construct product line models that allow the specification of required behaviour within a reference architecture that can be transformed into an effective product implementation, whilst **enabling** suitable supporting **evidence** for certification to be produced.

The research concentrates on the following four main areas:

1. Construction of an argument framework in which the application of product line techniques to high-integrity software development can be assessed and critically reviewed.

2. Definition of a product-line reference architecture that can host components containing variation.

3. Design of model transformations that can automatically instantiate products from a set of components hosted within the reference architecture.

4. Identification of verification approaches that may provide evidence that the transformations designed in step 3 above preserve properties of interest from the product line model into the product instantiations.

Together, these areas form the basis of an approach we term “Trusted Product Lines”. The approach has been evaluated and validated by deployment on a real aerospace project; the approach has been used to produce DO-178B/ED-12B Level A applications of over 300 KSLOC in size. The effect of this approach on the software development process has been critically evaluated in this thesis, both quantitatively (in terms of cost and relative size of process phases) and qualitatively (in terms of software quality).

The “Trusted Product Lines” approach, as described within the thesis, shows how product line approaches can be applied to high-integrity software development, and how certification evidence created and arguments constructed for products instantiated from the product line. To the best of our knowledge, the development and effective application of product line techniques in a certification environment is novel and unique.

List of Contents

List of Figures

List of Tables

Acknowledgements

This thesis, and the work it documents, would not have been possible without the support and encouragement from my colleagues at Aero Engine Controls and Rolls-Royce. In particular, I would like to thank Dean Armstrong, Duncan Brown and Stuart Jobbins for their understanding that a part-time PhD is a long commitment that can sometimes distract one from the immediate job in hand. I would also like to thank Phil Elliott, Ian Hopkins, Andy Nolan and Francis Thom for helping me crystallise and refine the concepts described herein such that they may actually be useful.

Many thanks must be given to my supervisor, John McDermid, who was always generous with his scarce time and abundant wisdom. John skilfully guided the research and thesis production down a path that had a fighting chance of delivering something tangible at the end. In addition, I would like to thank Richard Paige for his incisive comments during the research.

Finally, none of this would have been possible without the love and support of my lovely wife Beverley, who could not have envisaged the grief involved at the start of this process (and for actually reading the thing at the end!)

This work is for her.

Author's declaration

Some of the material presented within this thesis has previously been published in the following papers:

1. Stuart Hutchesson and John McDermid, “Development of High-Integrity Software Product Lines Using Model Transformation”, SAFECOMP 2010, LNCS

2. Stuart Hutchesson and John McDermid, “Towards Cost-Effective High-Assurance Product Lines”, Proceedings of the 15th Software Product Lines Conference - SPLC 2011, IEEE

3. Stuart Hutchesson and John McDermid, ”Trusted Product Lines”, Information and Software Technology Journal, Elsevier, In Press 2012, http://dx.doi.org/10.1016/j.infsof.2012.06.005

In addition, the following paper has been submitted for consideration:

Stuart Hutchesson and John McDermid,” Certifiable Engine Control System Product Lines: Principles and Practice”, Special Edition on Variability in Software Architecture, Journal of Systems and Software, Elsevier

Except where stated, all of the work contained within this thesis represents the original contribution of the author.

1 Introduction

Trusted (1) – "To have or place confidence in"

Trusted (2) – "To place into the care of another"

(www.answers.com)

The decision to develop a set of software products as a product line is first and foremost a business decision. Even if this decision is made on technical grounds, the development of a product line is committing the business to a significant change in the way products are developed, managed, supported and funded (if the product line initiative is to succeed in the long term). When developing high–integrity systems, this business context also includes the ability to approve or certify a product developed from the product line, and manage the product in-service, where service life can be measured in decades for certain classes of system. The recognition of these additional challenges has led us to introduce the concept of a "Trusted Product Line".

A Trusted Product Line has two subtle but important interpretations; firstly, the product line must be capable of creating a product that can be trusted within a defined operational environment – it must supply the evidence that can be used to satisfy all stakeholders that a particular instantiated product is fit for purpose. That, primarily, is a technical challenge: to understand how evidence that traditionally has been created on a single system instance can be produced when designing for a set of systems. The second interpretation is more of a business challenge - "To place in the care of another". Experience has shown that successful product line approaches are associated with significant organisational change [1], including the separation of the development of the core product line assets from the development of any one particular product (sometimes described as Domain Engineering and Application Engineering [2]). This naturally shifts resource, budget and management oversight from the traditional product delivery teams into the core-asset development team. The management responsible for delivering products to end customers have now "to place in the care of another" the development of substantial parts of their product and this can lead to a perceived (or actual) loss of control, power, prestige (...) and a concern that they can no longer be agile in response to customer problems and demands. The Trusted Product Line concept, therefore, has to deliver significant business advantage, as well as technical advantage, over single product development models to become accepted and eventually institutionalised.

1.1 Product Line Engineering

The SEI define a Software Product Line as follows:

Software Product Line : "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"[3]

The primary focus of product line research over the past decade has been to enable productivity gains in the commercial software development industry. It was recognised that "software reuse" as a concept was not providing the benefits that should be gained from designing a product once and using many times.

"*The efforts to achieve software reuse have led to the general acceptance of two important lessons. The first is opportunistic reuse ... is not effective in practise; a successful reuse programme within an organization must be a planned and proactive effort. Secondly bottom-up reuse ... does not function in practise; successful reuse programmes are required to adopt a top-down approach."[1]*

Before adopting a software reuse programme, most organisations have a legacy base of many products and applications. It is very tempting to try to "harvest" these assets in the name of productivity and efficient code reuse. However it is almost always the case that these assets were not designed with reusability in mind and, more importantly, were not developed to fit into a common architectural framework.

This realisation has led to the study and adoption of "Product Lines" as opposed to reuse as a mechanism for the effective realisation of a "design-once-use-many" approach to software development. Product Lines are not a small-scale adoption of a reuse library concept, but a fundamental, organisational-wide change to the way systems are designed and developed.

"Product lines do not appear accidentally, but require a conscious and explicit effort from the organisation interested in using the product line approach."[1]

Product line approaches are not confined to software, although the software development industry has pioneered the research and application of the ideas. Indeed Bosch [1] makes the point that organisations need to look at the product line implications of all aspects of system development if there are significant non-software components in the product:

"If systems, in addition to software, contain considerable pieces of mechanics and hardware, the product-line approach needs to be synchronised to all three aspects"[1]

The work described in this thesis has been undertaken in the context of an embedded, electro-mechanical system deployed within an encompassing gas-turbine machine. Product line initiatives have been applied to all these aspects at various times, but not necessarily synchronised as recommended by Bosch. This thesis concentrates on a particular contribution to the development of software product lines within this context.

1.2 High-Integrity Software Development

The development of software for deployment in a safety-critical or safety-related system provides a set of significant engineering challenges. Such systems are widely used in the aerospace, defence, transport, power generation and medical domains, for example, and are characterised by their potential to do harm if the systems fail in a hazardous manner. We will use the term "High-integrity system" generically to describe this class of system within this thesis, and the term "High-integrity software development" to refer to the development, verification and approval of the software used in such systems.

The assessment of the hazards posed by a high-integrity system and the analysis and justification that a system, when deployed in its operating context, is acceptably safe is performed by systems and safety engineers. They use a variety of techniques, both quantitative and qualitative, to determine that all potential hazards are identified and mitigated, and that the system failure rate is acceptable. The evidence required to support this analysis and associated safety claims varies by industry sector. Many of the industries and sectors into which such systems are deployed are governed by safety legislation and are regulated by government bodies. We will use the terms "Regulated domain", "Regulators" and "Regulating body" in this thesis to refer to such bodies and their activities. The regulator's involvement with the development and deployment of specific systems varies by industry sector, but most have to approve systems before they can be used "in service", and this approval process typically takes the form of audit, review and inspection of the safety evidence (sometimes called a "safety case").

High-integrity systems have become reliant on software to operate efficiently and effectively and perform tasks of increasing complexity. However, providing the evidence that the software failure rate is acceptable as a component part of an overall system is a very difficult task. Again, the approach taken differs across industry sectors, but most require a set of evidence based upon the process used to develop, verify, manage and control the software. Industry sectors vary in their approach to quantification of software failure rate, and it is not uncommon to find a qualitative treatment of software issues in what is otherwise a quantitative safety case.

Within this thesis, we are going to concentrate on the development of high-integrity software for the civil aerospace domain. This is regulated by government airworthiness agencies, such as the European Aviation Safety Agency (EASA) (supported by the Civil Aviation Authority (CAA) in the UK) and the Federal Aviation Administration (FAA) in the United States. The software developed for this domain has to be approved by a regulator as part of the certification of an aircraft or engine. This "certification" process takes the form of ensuring that the software has been developed in accordance with the objectives set out in DO-178B/ED-12B "Software Considerations in Airborne Systems and Equipment Certification" [4]. This provides guidance on the development and approval of avionics software and contains 66 objectives; the applicability and rigour of which depend on the "assurance level" of the software. There are 5 identified development assurance levels (DALs) within DO-178B/ED-12B, Levels A to E, where Level A applies to software that can contribute to "catastrophic" failure of the system, and Level E applies to software that has no safety impact (and therefore the guidance of DO-178B/ED-12B does not apply). In this context, no attempt is made to quantify the software failure rate, so we will not discuss software reliability models and estimation in this thesis.

1.3 Challenges of Trusted Product Lines

We have briefly discussed the technical and business challenges of producing a trusted product line and instantiating a usable product earlier in the introduction to this thesis. Here we describe the challenges in more detail and propose an approach to ensuring such an endeavour would be technically successful.

The overall goal of such a project can be summarised as follows:

A Software Product Line approach is used to develop and approve high-integrity software systems, which yields a significant improvement in development cost and lead-time over single-system developments, without compromising the system integrity or ability to certify.

Figure 2 below shows a Goal Structuring Notation view of the key goals and strategies that need to be argued for such a development to be credible.

Firstly we set the development within a specific context – that of civil avionics software development in accordance with the guidance of DO-178B/ED-12B. Specifically the development needs to provide assurance evidence that is sufficient to demonstrate compliance of a product to Level A (the highest development assurance level recognised in DO-178B/ED-12B).

Secondly, we recognise the business realities of product line development – that development cost savings are only made after a number of products have been produced. Industry data indicates that product lines typically break even (or even show a slight profit) after 3 products have been instantiated and deployed [5] (see Figure 1). This is framed as an assumption within Figure 2. We will revisit this assumption within the evaluation section of this thesis to determine if it still holds, given the constraints of a safety-critical development environment with a formal certification process, when following the approach advocated in this thesis.

Finally, we recognise that existing product-based techniques for software development, verification and management may be ineffective, inadequate and/or inappropriate when applied to product-line development. This is due in part to the additional complexity of designing solutions for a class of systems, and the problems inherent in verification of implementations that contain variability. To address this, we research novel techniques in this thesis (particularly for verification of developments that include transformation) and we validate the use of existing product-line techniques within the context of high-integrity software development.

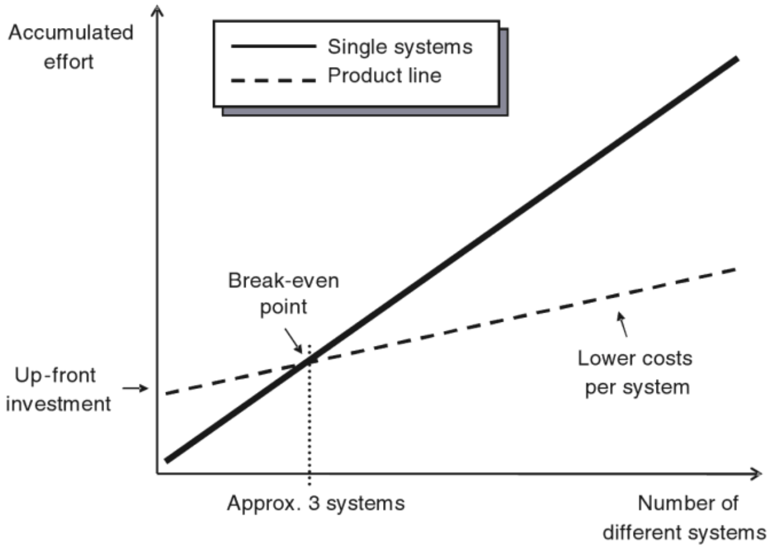


FIGURE 1 SOFTWARE PRODUCT LINE ECONOMICS [5]

Given this goal, within the context of civil avionics and the assumption of a 3-project payback period, we have identified five main technical challenges to address credibly the overall goal.

1.3.1 PL Scoping is Possible

This challenge relates essentially to the ability to clearly identify a product line scope; i.e. to be able to define robustly when a specific product is a member of the product line (and equally when a product is **not** a member). In addition, it must be possible to identify common parts of the product line (those aspects that are present in all members) and the variable parts (aspects included/excluded by selection).

This is primarily an engineering challenge; whilst it is a difficult activity to do well (requiring domain experts with a significant depth of knowledge) the literature describes a number of techniques to help perform the scoping exercise [6, 7]. There are also a number of well-defined techniques for structuring the information relating to the common and variable parts of the product line [8-10].

A well-defined scope is a necessary pre-requisite to producing a credible product line and we will discuss novel approaches to capturing and managing this information when developing a significant high-integrity product-line, especially in the context of providing traceability for downstream development activities (see chapter 4 section 4.8). However, it is not the main focus of the research described in this thesis.

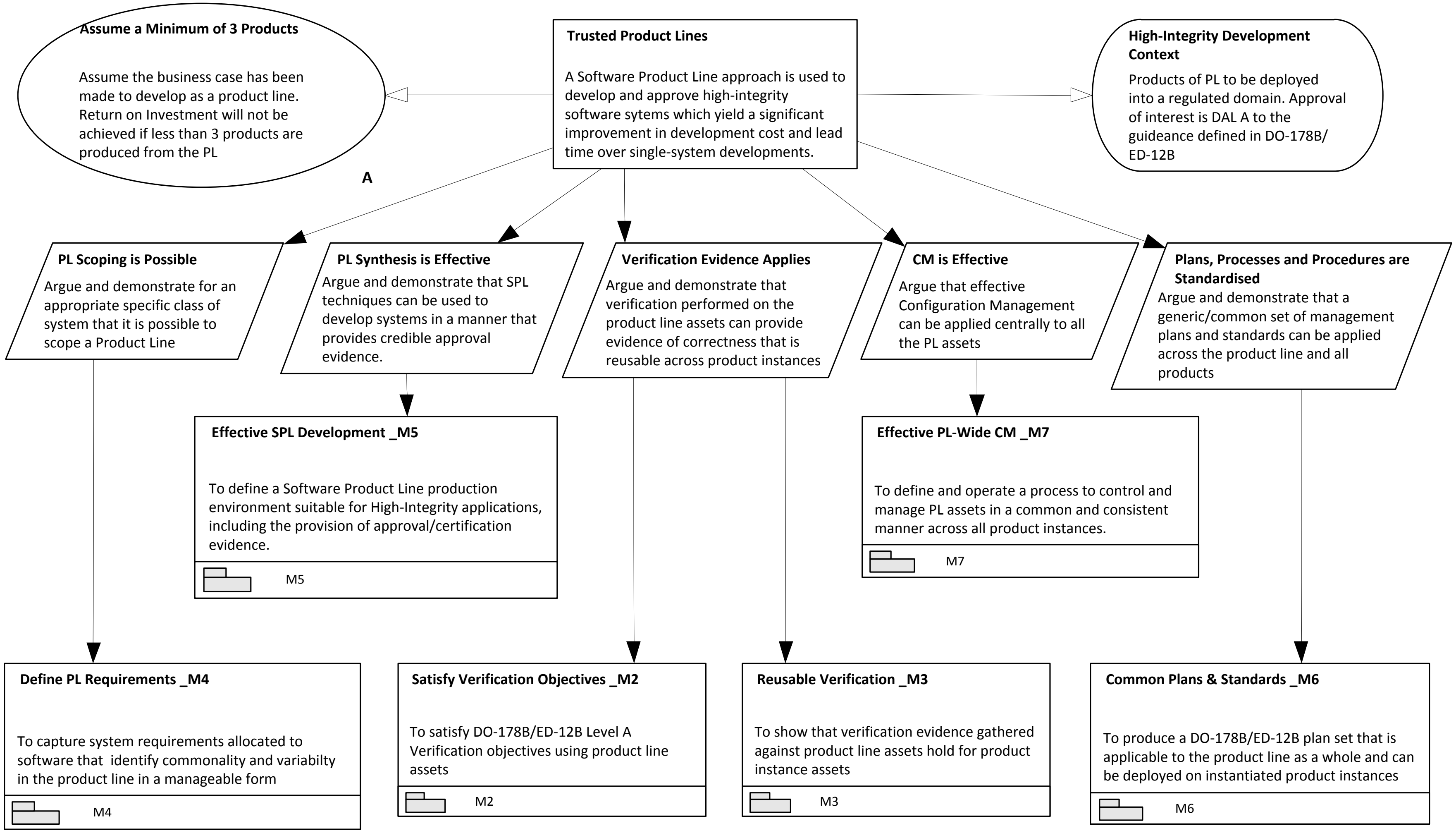

Assume a Minimum of 3 Products
Assume the business case has been made to develop as a product line. Return on Investment will not be achieved if less than 3 products are produced from the PL
A
Trusted Product Lines
A Software Product Line approach is used to develop and approve high-integrity software sytems which yield a significant improvement in development cost and lead time over single-system developments.
High-Integrity Development Context
Products of PL to be deployed into a regulated domain. Approval of interest is DAL A to the guideance defined in DO-178B/ ED-12B
PL Scoping is Possible
Argue and demonstrate for an appropriate specific class of system that it is possible to scope a Product Line
PL Synthesis is Effective
Argue and demonstrate that SPL techniques can be used to develop systems in a manner that provides credible approval evidence.
Verification Evidence Applies
Argue and demonstrate that verification performed on the product line assets can provide evidence of correctness that is reusable across product instances
CM is Effective
Argue that effective Configuration Management can be applied centrally to all the PL assets
Plans, Processes and Procedures are Standardised
Argue and demonstrate that a generic/common set of management plans and standards can be applied across the product line and all products
Effective SPL Development _M5
To define a Software Product Line production environment suitable for High-Integrity applications, including the provision of approval/certification evidence.
M5
Effective PL-Wide CM _M7
To define and operate a process to control and manage PL assets in a common and consistent manner across all product instances.
M7
Define PL Requirements _M4
To capture system requirements allocated to software that identify commonality and variabilty in the product line in a manageable form
M4
Satisfy Verification Objectives _M2
To satisfy DO-178B/ED-12B Level A Verification objectives using product line assets
M2
Reusable Verification _M3
To show that verification evidence gathered against product line assets hold for product instance assets
M3
Common Plans & Standards _M6
To produce a DO-178B/ED-12B plan set that is applicable to the product line as a whole and can be deployed on instantiated product instances
M6

Figure 2 GSN Form of the Overall Argument for Development of High-Integrity Software product Lines

1.3.2 PL Synthesis is Effective

This challenge relates to the ability to apply product line synthesis techniques to the creation of product instances from artefacts developed for the product line. Product line synthesis has its origins in the generative programming work of Czarnecki and Eisenecker [9] and has been commercialised with tools such as pure::variants and Gears. Support for the types of transformations required to perform product instantiation is increasingly being included in modelling environments such as the Eclipse Modeling Project [11] and UML [12] tools.

However, there remain a number of significant challenges to apply this type of technology to a high-integrity development domain. The approval and certification regulations currently have no concept of product lines; the evidence requirements are stated from a single system viewpoint. Therefore, it must be possible to create artefacts and evidence as if the product had been developed in isolation. Therefore, the transformations must not only apply to the production of product source code but also to the development artefacts: design descriptions, requirements etc. Traceability must also be maintained in a credible manner to demonstrate the relationships between the development artefacts, in particular to demonstrate requirements satisfaction.

There are enough significant challenges to the application of this type of development to high-integrity product lines to make this a worthwhile area for research. The literature has little to say on the application of generative programming and transformations to high-integrity applications, and this has not significantly changed for the duration of this research project (see chapter 2 section 2.6).

1.3.3 Verification Evidence Applies

Verification is the major source of non-recurring cost in the development of high-integrity systems. Industry data shows that the verification effort can account for at least 50% of the development costs for traditional high-integrity lifecycles [3]. Therefore, any benefits a software product line approach can bring to reducing the verification burden or spreading the cost of verification across multiple product instances would provide business benefit. Conversely, any product line practice that hinders verification or requires additional verification activities to be performed must be outweighed by cost and schedule savings elsewhere. McGregor [4] observed that the ability to reuse test assets could be as significant as the savings from the reuse of development assets within a software product line. For high-integrity systems, the ability to minimise the cost of verification across multiple projects may actually provide greater savings than from the reuse of development assets.

There is a fundamental issue regarding how to optimise the verification processes for high-integrity product lines. Key to this is the extent to which verification should be carried out on the product line assets, as opposed to on the final instantiated product. It is intuitively attractive to carry out verification on the product line assets because any use is then "verified by construction". However there may be many tens or even hundreds of possible configurations of even a modest-sized component and, further, transformational process may add to the code, e.g. providing interfaces, as well as making selections, so it is not clear how representative asset-level testing evidence will be of the end product. Product line practices that enable the product-specific verification to be simply regression testing/analysis (and ensuring the results are still valid and positive) would appear to be the most attractive in this regard. However, this may not be achievable if approaches such as compositional development are adopted, and the transformations used are not property preserving (see Figure 3).

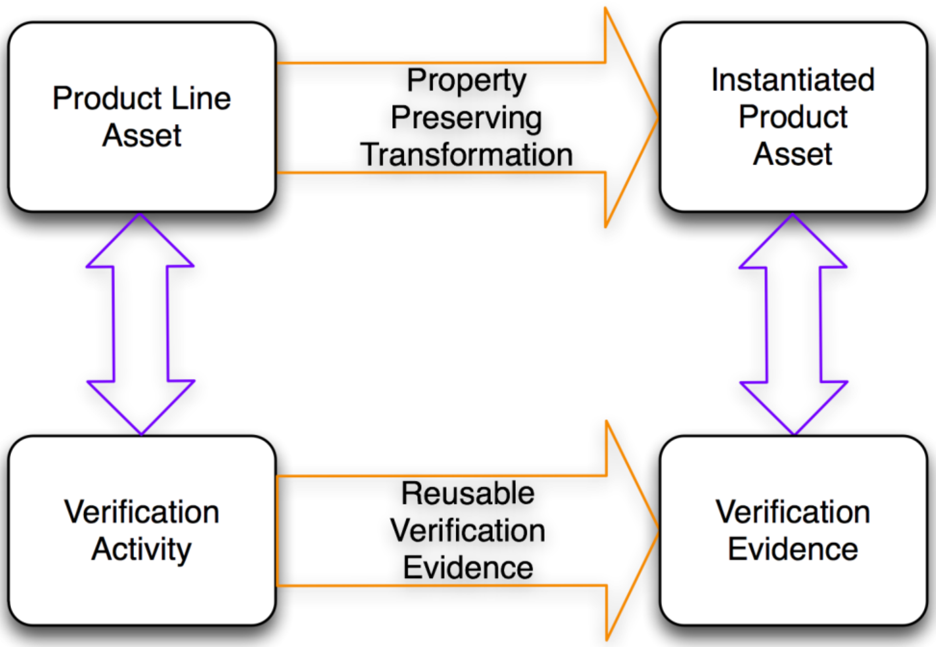

Product Line Asset
Property Preserving Transformation
Instantiated Product Asset
Verification Activity
Reusable Verification Evidence
Verification Evidence

FIGURE 3 THE ROLE OF PROPERTY PRESERVING TRANSFORMATIONS IN ENSURING APPLICABILITY OF VERIFICATION

Here, we are interested in approaches that do not destroy or call into question the value or applicability of process evidence that has been gathered during the development of the product line asset. At the detailed level this may include formal mathematical guarantees that a given set of model or program semantics have been preserved over a transformation, however this is not the focus of this research. We wish to construct arguments that a product instantiated from the product line is fit for purpose whilst minimising the economic cost of producing that product. Those arguments have to convince developers, regulators and users that the following hold :

- Applicability – the requested product has been instantiated.
- Conformance – all artefacts conform to the required and declared standards.
- Compliance – all artefacts demonstrably comply to their requirements, specifications and architectual constraints.

As much of this evidence is gathered today via manual inspection and analysis, and is expensive and time consuming to collect, its value has to be preserved throughout the product lifecycle.

The verification challenge for high-integrity product lines is therefore one of process optimisation to give the best return on investment (RoI) over the life of the product line. One potential approach is to demonstrate that it is efficient to perform the required verification on the product line assets (i.e. prior to transformation) and then to show (hopefully automatically) that the verification evidence still holds for the product instance assets (i.e. post transformation). There is little work in the literature that addresses these problems and this is a significant research challenge (see chapter 2 section 2.5).

1.3.4 CM is Effective

One of the most challenging practical problems for trusted product lines is to implement effective configuration management and change control. This encompasses both the technical challenge of managing component configurations and their changes for the product line AND the product instantiations, plus the business/political challenge of persuading project managers to accept the changes and updates made on the product line into their individual products.

The credibility of the assets and their development process evidence is key to a successful product approval or certification. This credibility is provided by a well-designed and well-managed CM process/system and the resultant audit trail provided for each configured asset. However, this is primarily an engineering and organisational challenge, and therefore is not considered further in this thesis.

1.3.5 Plans, Processes and Procedures are Standardised

The development of any software system in accordance with the guidelines of DO-178B/ED-12B is governed by a set of plans, a shorthand for which is the “DO-178B Plan Set”. The planning documents and their contents are proscribed within the guidance; they consist of an overarching “Plan for Software Aspects of Certification” (PSAC), and supporting plans that cover development, verification, quality assurance, configuration management, and the “qualification” of tools used in the development environment. These are used both to manage the development of the software system, and as the primary means of describing the system and development process to the regulating authority. As mentioned earlier, current regulatory guidance does not recognise product line development practice and therefore the plan set needs to be carefully structured to be acceptable to the regulator but also be applicable to the product line development. This is further complicated by the fact that the regulating authority may differ from product instance to product instance (e.g. European Aviation Safety Agency (EASA) approval may be sought for product A whereas the US Federal Aviation Administration (FAA) approval is sought for product B). Regulating authorities (and indeed individual regulators) can have slightly different interpretations of the guidance and also have particular interest in certain parts of the development lifecycle that they want emphasising in the plan set. This potential for variation in the planning documentation needs to be considered early in the product line development process to ensure that it does not result in unnecessary cost and rework when a product is submitted for approval.

This is a significant engineering challenge and one that could benefit from research into the efficient and effective management of the process information. This is not directly addressed in this thesis.

1.4 Thesis Hypothesis

In the previous section, we identified the major technical challenges in adopting a trusted product line approach to system development. Of the five challenges identified, three contained areas of potentially significant or novel research, and we consider two in detail within this thesis; these are the synthesis of a high-integrity product and supporting assets, and assuring the reusability and applicability of verification evidence.

It would not be credible or achievable for a single PhD thesis to fully address these issues with complete satisfaction. However, it should be noted that this research has been conducted alongside the practical application of product line techniques to a substantial industrial product line, and therefore a number of these issues have been dealt with pragmatically on the project.

The areas we wish to focus on specifically in this research are as follows:

- Support for variability in models used to represent software architecture and design
- The generation of product instances from such models, typically via model transformation
- Supporting certification evidence and artefacts with such models
- Supporting reusable verification using such models

We focus on these because our experience (including refinement of our approaches) shows these to be pivotal to the successful application of product lines.

The research hypothesis is therefore:

*It is **feasible** to construct product line models which*

a) allow the specification of required behaviour (including the identification of common and variable aspects in a product-line)

b) allow the definition of a reference implementation architecture which can be transformed into an effective, efficient and analysable product implementation

*and **enable** suitable supporting **evidence** for certification to be produced, including **effective** verification.*

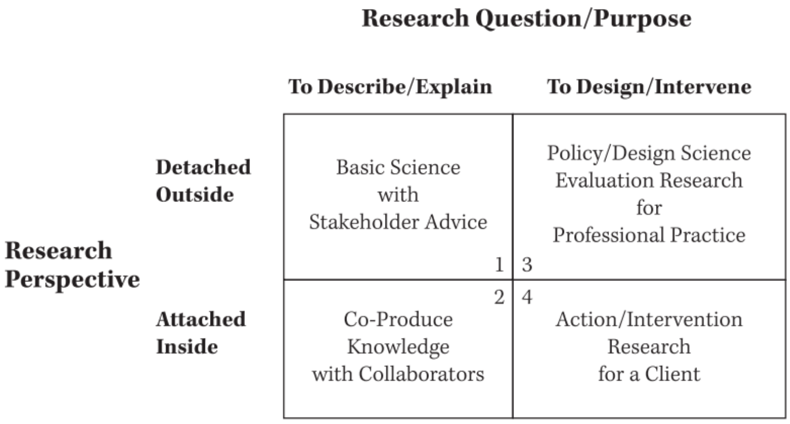
1.4.1 Research Value and Relevance

Throughout the thesis, a number of technical options and decisions are discussed and critically evaluated, and significant/novel research results and conclusions are described in detail. Whilst the engineering motivation for the research described in this thesis is clear, there is also original scientific value to the work undertaken. In particular, the objective to investigate mechanisms for reuse in a high-integrity system engineering context, whlist making use of higher abstraction engineering artefacts (e.g. via models). Achieving systematic reuse (and abstraction) while simultaneously sustaining confidence in the quality of the derived products is both scientifically interesting and a significant challenge.

We discuss at the start of Chapter 5 a set of “Essential” and “Accidental” challenges and constraints. The essential challenges describe the problems associated with this research that are of general interest and applicability, and these provide the scientific basis for the research direction and decisions made in the thesis. The accidental challenges frame the work within the real industrial context in which it was performed, and addressing these constraints were the reason the work was successful in realising actual, sizable product and enabled the quantitative and quantitative analysis of the research to take place.

1.5 Mode of Research

We have conducted the research described in this thesis from the perspective of an “interested participant”. The purpose of the research has not been to describe/explain phenomena as a detached observer, rather it has been to demonstrate and validate the efficacy of an approach within a domain of interest. Van de Ven [13] describes this type of research as *Action/Intervention*, where a researcher engages and intervenes in a particular domain. Figure 4 illustrates how this approach differs from other forms of engaged scholarship.



Research Question/Purpose

Research Perspective		To Describe/Explain	To Design/Intervene
	Detached Outside	Basic Science with Stakeholder Advice 1	3 Policy/Design Science Evaluation Research for Professional Practice
	Attached Inside	2 Co-Produce Knowledge with Collaborators	4 Action/Intervention Research for a Client

FIGURE 4 ALTERNATIVE FORMS OF ENGAGED SCHOLARSHIP (FROM [13])

Region 4 describes most aptly the approach taken in this this thesis, where the researcher utilises available knowledge to understand the problem at hand. However, existing knowledge “may require substantial adaptation to fit the ill-structured or context-specific nature of the client’s problem” [13].

Van de Ven notes that this type of research often consists of limited ("N-of-1") studies where comparative evidence can be difficult to gather, and may consist of trial-and-error studies over time. It can be argued that the only way to understand such systems is to change through deliberate intervention and diagnose responses to the intervention [13]. The research evaluation described in chapter 6 should be read in this light.

1.6 Thesis Model & Structure

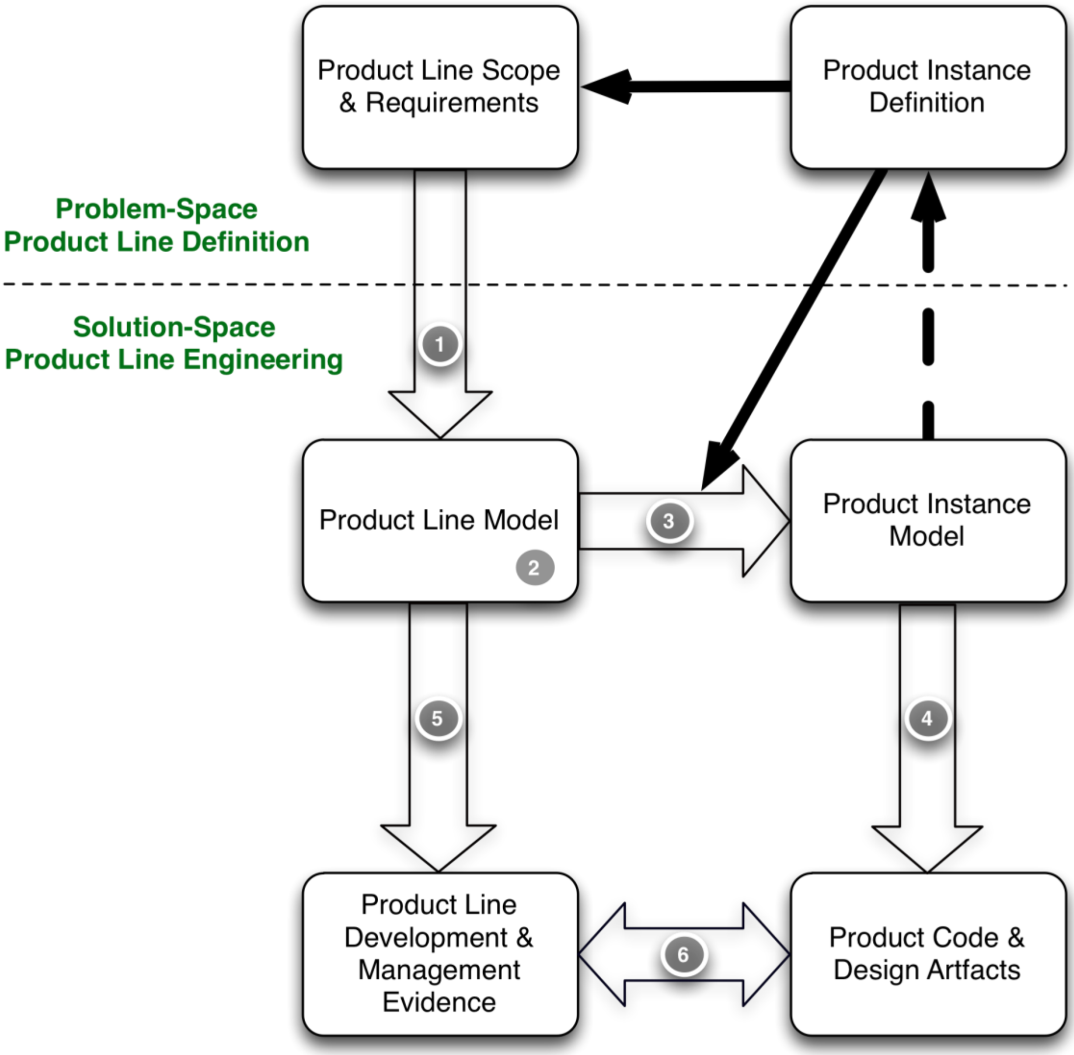

Product Line Scope & Requirements
Product Instance Definition
Problem-Space
Product Line Definition
Solution-Space
Product Line Engineering
1
Product Line Model
2
3
Product Instance Model
5
4
Product Line Development & Management Evidence
6
Product Code & Design Artfacts

FIGURE 5 ANNOTATED MODEL FOR TRUSTED PRODUCT LINES RESEARCH

The diagram provided in Figure 5 shows a conceptual process model for product line development. The annotations enumerate the areas of research contribution described in this thesis, and these are described below. It can be seen from the annotations that, in general, the particular form of each of the artefacts (rounded rectangles) is not the focus of the research; the areas of interest are in the transformations performed and relationships between these artefacts.

1. The overall scope and form of the product line definition is not of direct interest to this research, however the relationship between the product line definition and design model produced in response to this definition is covered in Chapter 4.
2. A major focus of this research is the form of product line design model. In particular we investigate and propose meta-models to capture product architectures and component designs suitable for use in a high-integrity context. This is the major focus of Chapter 4 of this thesis.
3. Mechanisms for transforming a product-line model into a product-instance model are a central topic of this research. Chapter 5 describes the research undertaken in this area and Appendix B describes the transformations produced in support of this research in detail.
4. The automatic creation of the product artefacts (not just the source code) from the instantiated product model is key to the trusted product line research. This is discussed in general in Chapter 5 and some of the problems with applying this in practice are discussed and addressed in Chapters 6 and 7.
5. Generation and management of compelling evidence for the use of product lines in a high-integrity context is necessary for this approach to be successful in practice. A framework for the creation, management and analysis of this information was provided earlier in this chapter, and is critically revisited in Chapter 6. Chapter 6 also provides qualitative and quantitative evidence on the success of the approach in an industrial context.
6. Successfully arguing the applicability of evidence collected against the product line in support of the product instance is key to achieving the economic benefits of product line engineering. The problems with this in a high-integrity context are explored in Chapter 6, and potential solutions to these problems are provided in Chapter 7.

The overall model in Figure 5 illustrates how the trusted product lines research has considered and addressed the full scope of solution-space product line engineering. It is instructive to note that the major focus of this research is to address the solution-space issues of engineering a high-integrity product line; this is analogous to "building a software factory". Whilst the problem-space product line issues, such as scoping, capturing feature requirements, analysing feature interaction etc. are interesting and valuable to to study, they are outside the scope of the research described in this thesis.

The thesis is organised as follows:

- **Chapter 2** provides a critical review of the literature on software product line development, with particular regard to the use of product lines within high-integrity or related domains, and the use of model-based approaches.

- **Chapter 3** discusses the challenges of software product line development within the context of a specific class of high-integrity systems: Full Authority Digital Engine Control (FADEC) systems for civil aviation applications, under the regulatory guidance of DO-178B/ED-12B. The chapter outlines the motivation for the research work, including the business challenges that make development as a product line attractive, and the resultant technical, engineering and academic challenges that are a consequence of this business strategy.

- **Chapter 4** describes an approach to architectural modelling of a FADEC software system that enables the development of components in a product line manner. It defines architectural and component meta-models.

- **Chapter 5** describes an approach to the instantiation of product instances from product line models using model transformation techniques.

- **Chapter 6** evaluates the approach described in the previous chapters using data obtained from industrial use of the technique. The data provides quantitative information on the cost-effectiveness of the approach and qualitative information on the ability of the process to provide product approval evidence.

- **Chapter 7** discusses improvements to the approach following the critical evaluation provided in the previous chapter.

- **Chapter 8** provides a summary of the work described in the thesis, including overall conclusions of the research, and identifies areas of potential further investigation.

- **Appendix A** provides background information on the development of SPARK [14] programs, including an approach to modelling SPARK using UML [12].

- **Appendix B** describes in detail the design of the model transformations used to implement the product instantiation.

- **Appendix C** contains a case study demonstrating the approach on a number of example components.

2 Literature Review

This chapter presents a review of the literature relevant to the development and verification of trusted product lines using models. It concentrates on model-based approaches to product line development, with an emphasis on material that is directly or indirectly relevant to high-integrity system development. The most interesting and fruitful areas for study are those that lie in intersections between the domains. Figure 6 illustrates these intersections and contains annotations that guide the reader through the contents of this chapter.

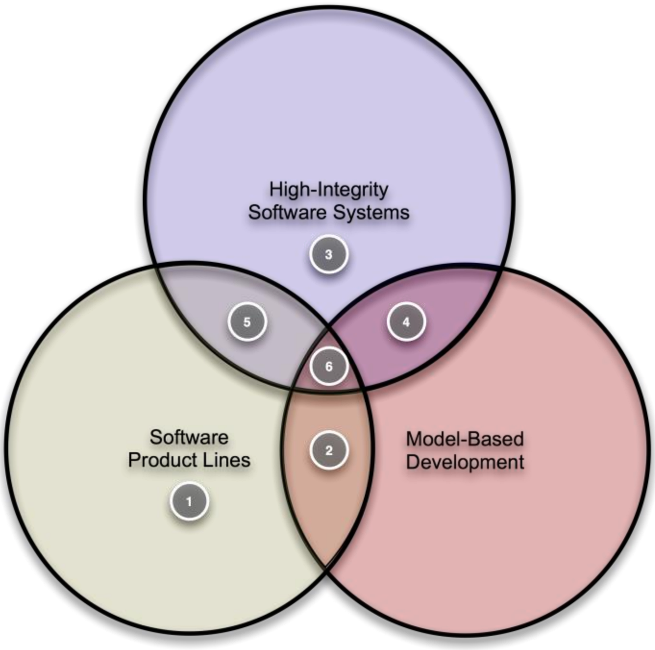


Figure 6 Venn Diagram Denoting Domains of Interest and their Intersections

The structure of this chapter is as follows:

2.1 Provides a brief overview of product line theory (Region 1)

2.2 Discusses the development of product lines using model-based techniques, including a critical review of UML and component-based approaches to product line design (Region 2)

2.3 Provides an overview of the development and approval/certification of high-integrity software systems in regulated domains (in particular civil aerospace) (Region 3)

2.4 Discusses how model-based approaches have been used within the development of high-integrity software systems (Region 4)

2.5 Provides a critical review of the literature regarding software product line approaches to high-integrity software development in general; this includes a review of product line verification, with a particular emphasis on the verification requirements for regulated domains (Region 5)

2.6 Provides a critical review of the literature regarding model-based software product line approaches to high-integrity software development (Region 6)

(Note: The general topic of model-based development is huge, so this review only covers the bounded area of application to product-lines and high-integrity domains.)

2.1 Software Product Line Development

Software Product Lines have been applied and studied as a recognised discipline for a number of years. The concept of studying "a family of programs" can be traced back to Parnas [15] in the mid-1970s (the terms "Software Families" and "Software Product Lines" being regarded today as essentially synonymous.) Interest in a product line approach to software development increased in the mid-1990s when it became clear that simple, bottom-up "reuse" of software was not delivering the cost and schedule benefits that might initially be expected [1]. Software Product Lines as a concept is distinguishable from simple software reuse primarily by its focus on the development of a family of products as a managed activity, rather than the fortuitous reuse of previously developed software components **[3].**

2.2 BAPO

Van der Linden et al. [16] identified four independent concerns that are important when adopting a product line approach:

- *Business - how to make profit from your products*
- *Architecture - technical means to build the system*
- *Process - roles, responsibilities and relationships within system development*
- *Organisation - actual mapping of roles and responsibilities to organisational structures*

These concerns, termed the BAPO model, are represented diagrammatically in [16] as follows:

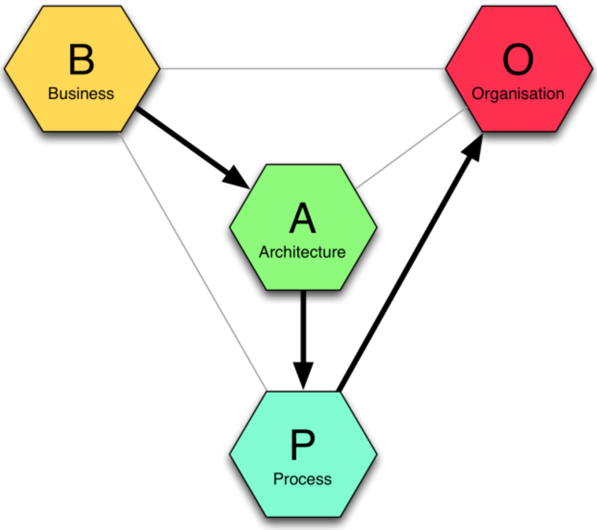


FIGURE 7 THE BAPO CONCERNS [16]

"Arrows denote a natural order to traverse the concerns, giving an order to the acronym as well. The Business is the most influential factor. This has to be set up right in the first place. The Architecture reflects the business concerns in ... structure and rules. The Process is set up to be able to build the products determined by the architecture. Finally, the Organisation should host the process." [16]

The importance of architectural design to the success of a product line initiative is made clear in the BAPO model and is an aspect we will return to later.

2.2.1 Product Line Processes

Product line developments distinguish between the concepts of domain engineering and application engineering [2]. The domain engineering task is to create a set of assets or artefacts (commonly termed **core assets** [3] or **family assets** [17]) that can be used by the application engineers to construct useful products [18]. The domain engineers undertake a development lifecycle similar to that used for single product developments, but are focussed on the development of assets for the product line as a whole (see Figure 8). In addition to producing reusable artefacts, the domain engineering task has to understand how products within the product line vary between each other and then encode this variability into the assets produced. This information can be captured in a **variability model** [2]

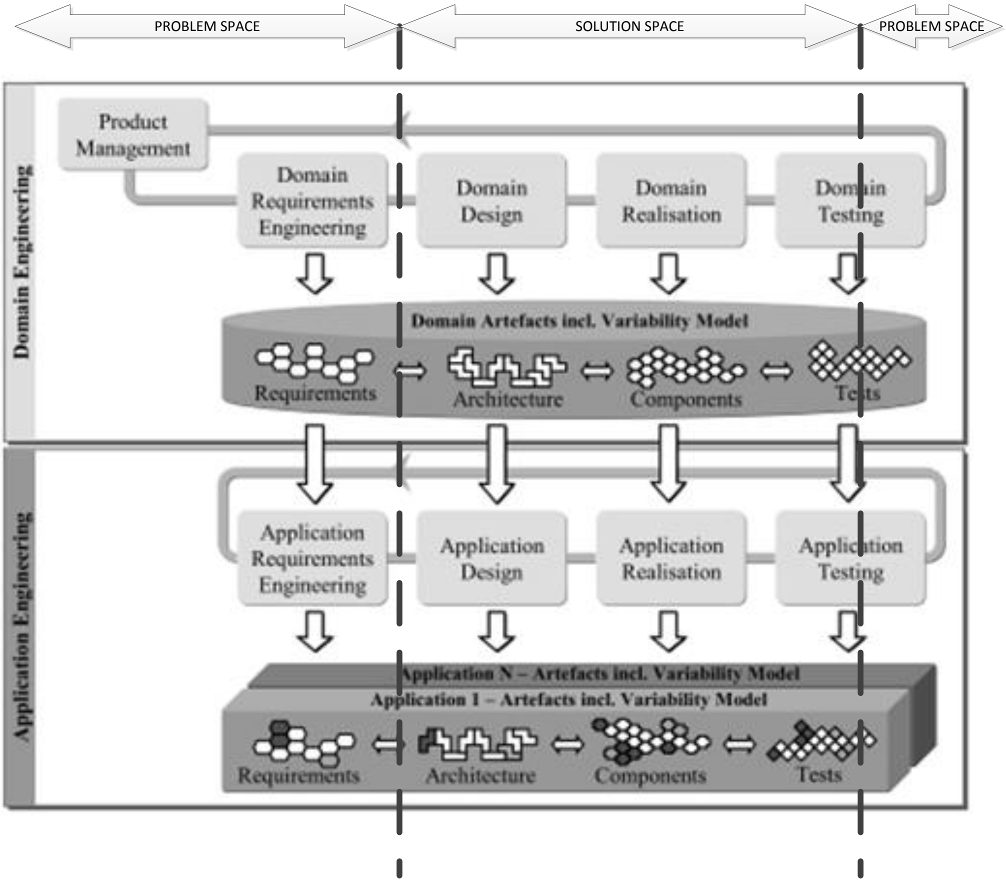


FIGURE 8 SOFTWARE PRODUCT LINE PROCESSES (FROM [2] WITH ADDED ANNOTATION)

2.2.2 Commonality and Variability

The identification of commonality and variability between members of the product line is one of the main distinguishing aspects of Product Line development. Much of the research into Product Line development has been concerned with the identification and management of commonality and variability [19, 20], both in product line requirements (problem space) and product implementation (solution space):

- Problem space variability is concerned with the scoping of the product line and differentiating the products in terms of common and variable features.[21]
- Solution space variability is concerned with the artefacts that compose the system itself and how these can be varied to deliver the required product.[21]

The concepts of problem space and solution space are orthogonal to domain and application engineering and need to be taken into account in both - see the annotation to Figure 8. Essentially, this shows that the problem space activities take into account the development of the requirements for the product line plus a proportion of the verification that the right product has been built (sometimes termed validation).

The most widely adopted and studied approach to the management of commonality and variability in the problem-space is **Feature Modelling** [8, 9].

2.2.3 Features & Feature Modelling

A number of definitions of “feature” exist in the literature; the most useful for our purposes is the following, paraphrased from [1] :

Feature – A set of related requirements that represent a logical unit of functionality for the user of the product.

The concept of Feature Modelling was introduced as part of the feature-oriented domain analysis (FODA) method [8]. A Feature Model describes a tree of features, where variabilities (alternative features) are indicated using *and/or* nodes. FODA introduced a graphical syntax for such feature trees; this has been used extensively within the Product Lines literature, typically in its extended form as described in [9] and illustrated in Figure 9.

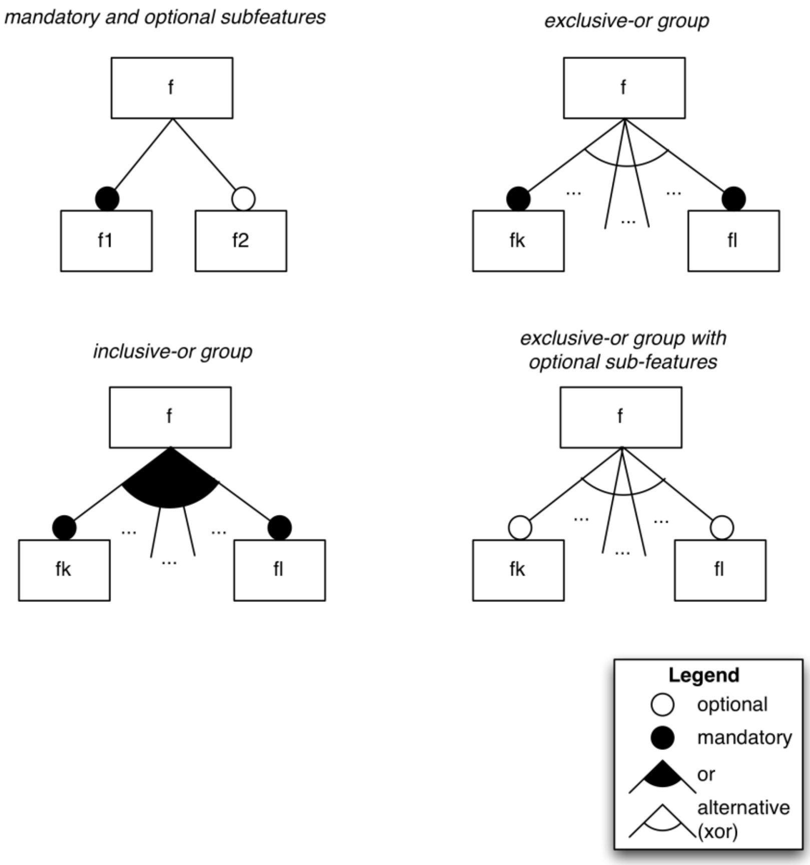


FIGURE 9 FODA NOTATION AS EXTENDED BY [9] (ADAPTED FROM [10])

The "Domain Requirements Engineering" phase of the Product Line processes shown in Figure 8 concerns itself with the identification of the common and variable aspects of the Product Line, and typically documenting these in a feature model. This process is also known as Product Line **Scoping** [6]. Identifying the set of features (and how these features vary across the products in the product line) provides the necessary requirements for the subsequent "Domain Design" process activities. In reality, there is a difference between true Product Line Scoping and the derivation of the feature model: scoping is primarily a business-driven activity, focusing on how to generate return on investment by deciding which products and product features should be in the product line scope. The feature modelling exercise is then part of a follow-on requirements engineering activity.

2.2.4 Commonality and Variability in the Solution-Space

Pragmatic approaches to solution-space variability dominated the early attempts at software product line development. For instance, a relatively easy and low-cost approach to providing variability in software was to make use of the existing conditional compilation techniques available using language pre-processors provided in the C language (for example the use of "#ifdef" statements). Many commercial product lines are deployed using pre-processor directives and conditional compilation to instantiate specific products from a code-base containing variability [22]. Initially this was the only credible alternative to deploying so-called "generic" products, where all alternative behaviour was available in the installed product and was enabled/disabled at run-time via configuration settings. For commercial applications such as automotive ECUs (Engine Control Units), the overhead of supporting all variants in a single image can be costly if it requires larger memory devices to hold the built program, which is one of the reasons why compositional approaches (even simple “#ifdef”s) are attractive.

Many of the software tools that support product line development augment development tools that are designed for single system development with the concept of variability. Tools such as pure::variants [23] and Gears [24] allow models of variability to be constructed within the tool, and are “aware” of the file formats of the development environments being used to develop the product line artefacts. In this way, they provide a more sophisticated and more manageable approach to essentially pre-processing product line artefacts.

There are issues of scalability and complexities of managing commonality and variability data when using this type of approach. One solution to this is to adopt the approach that computer science typically uses whenever faced with complexity problems, and that is to adopt useful abstractions; hence the interest in model-based approaches to product line development.

We build upon some of these fundamental concepts of product lines in our research; we concentrate on solution-space variability, as our motivation is primarily the practical realisation of a product line rather than the definition of its scope. Our approach to variability definition and management provides a well-defined structure to the product line representations and is more than just a source-code manipulation technique.

2.3 Model-Based Development of Product Lines

For many years, the implementation of product lines in real systems relied upon the pre-processing of artefacts to remove the parts of the product that were not required for that variant [23, 25]. The BAPO model shows us, however, that for a product line strategy to be fully effective, the business, architecture, process and organisation aspects all need to be mature with regard to the product line. This would suggest that any technical approach that does not treat commonality and variability as “first-class citizens” in the product design process (as opposed to being a “bolt-on” to traditional methods) would not be fully effective. The definition of a product-line architecture is one of the main lessons of BAPO; source-file composition approaches to product lines cannot make use of higher-level abstractions like “*logical architecture and design patterns*” [26].

Model-based approaches to solution-space product-line development can provide first-class modelling concepts to allow the expression of commonality and variability as an integrated part of the design process. This allows concepts of commonality and variability to be modelled alongside other design abstractions such as component dependency, architectural layering etc. (Note that approaches to problem-space modelling do exist, but they typically provide their own abstractions (e.g. FODA), or extend notations like UML class modelling to express problem-space concepts such as features – see section 2.3.2.)

2.3.1 Modelling Product Lines with UML

Many software product line modelling approaches target the Unified Modeling Language (UML) [12] as the modelling language of choice. UML is widely used in industry, and has built-in mechanisms for extending the language through the use of stereotypes and profiles. This extension mechanism can be used to provide support for product-line concepts such as variability that does not exist in the base UML specification. There are a number of published UML profiles that support the modelling of product line concepts [27-29], although none have yet been adopted by the Object Management Group (OMG) as a standardised extension to UML. (At the time of writing the Common Variability Language (CVL) [30] was in the process of being adopted as an OMG standard – CVL is discussed later in this chapter.)

2.3.2 Problem-Space Modelling with UML

Gomaa [28] provides a number of suggested UML extensions to support the modelling of product lines at various levels of abstraction (both in the problem and solution spaces). Figure 10 shows a set of suggested stereotypes to support feature modelling which are semantically equivalent to the extended FODA notation we discussed earlier.

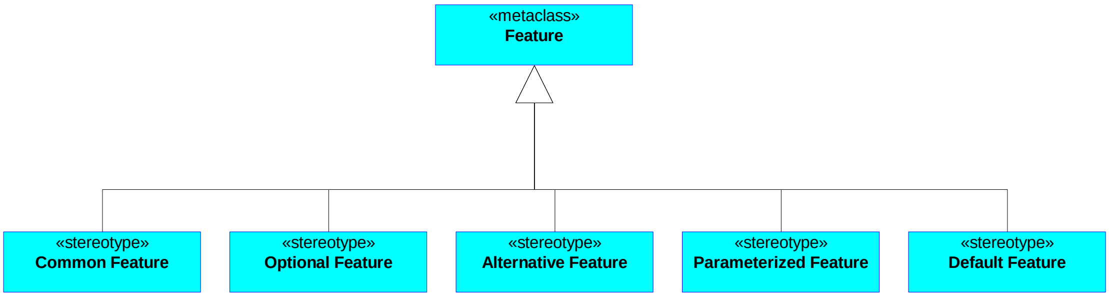


FIGURE 10 CLASSIFICATION OF FEATURES IN A FEATURE MODEL [28]

The types of feature that can be identified using this model are as follows:

- Common Feature
  Feature provided by every member of the product line.

- Optional Feature
  Features that need to be provided by only some members of the product line.

- Alternative Feature
  Two or more features may be alternatives. A constraint on the allowable choice of alternatives (e.g. mutual exclusion) may be given using the Feature Group and Dependencies mechanisms described later.

- Default Feature
  Within a group of alternative features, one may be selected as the default (i.e. the pre-selected alternative) .

- Parameterized Feature

> A feature whose behaviour varies dependent upon the value of a parameter. The parameter's value needs to be defined when configuring a member of the product line.

Related features can be grouped into Feature Groups which place a cardinality constraint on how the features are used by a given member of the product line. Figure 11 illustrates the classification of Feature Groups.

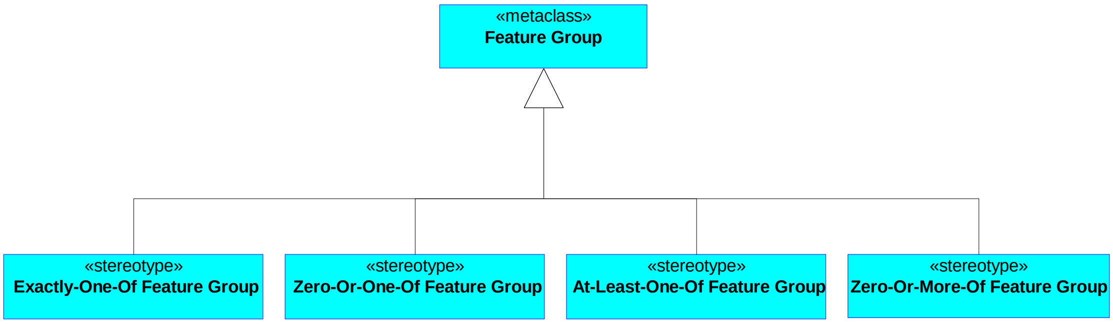

«metaclass»
Feature Group
«stereotype»
Exactly-One-Of Feature Group
«stereotype»
Zero-Or-One-Of Feature Group
«stereotype»
At-Least-One-Of Feature Group
«stereotype»
Zero-Or-More-Of Feature Group

FIGURE 11 CLASSIFICATION OF FEATURE GROUPS IN A FEATURE MODEL [28]

Features may have dependencies on other features – this can be modelled using stereotyped dependency relationships within the UML feature model. The classification of dependencies is shown in Figure 12.

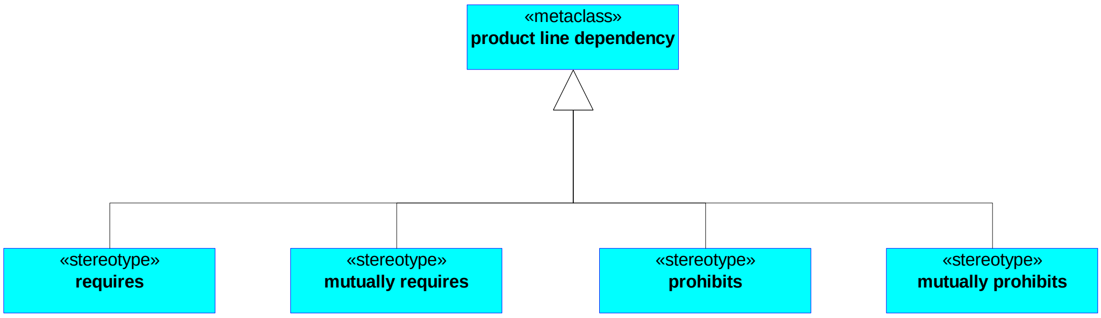


FIGURE 12 CLASSIFICATION OF PRODUCT LINE DEPENDENCIES

Note that these relationships are intended to model dependencies across features and feature groups; they are NOT intended as alternatives to Feature Groups (e.g. do not use "mutually prohibits" to model an "Exactly one of" feature group.)

Gomaa does not "formally" define a UML profile for feature modelling (the dependency classifications in Figure 12 are inferred from examples in the book), however there are enough text references and examples provided to construct a useful set of stereotypes if UML feature modelling were required. The advantage to this approach is that UML is in widespread use in industry, and therefore having a modelling environment that can capture this information that is available and known to practising engineers is of benefit. In addition, if the implementation is to be modelled in UML, having the feature model available in the same environment is advantageous. However, class modelling syntax is not as natural a paradigm as the extended FODA notation for capturing and conveying feature descriptions. It would be possible, however, to use the UML as the underlying repository for the information but render it in a more suitable form via transformation if this was required. It would be possible to automatically map to/from a FODA representation .

2.3.2.1 Solution-Space Modelling with UML

The BAPO model discussed earlier highlighted the importance of architecture in the development of Product Lines. Academic and industrial case studies into the successful introduction of product line development indicate that an early focus on architecture, including the development of a product line architecture (reference architecture), is crucial to the success of the initiative [1]

A Family Model [31, 32] provides the design response to the Feature Model and the commonality and variability identified therein. The Family Model encompasses the Product Line architecture and shows how products can be realised to achieve the requirements defined in the Feature Model. The central role of Family Models in bridging the gap between the feature understanding and the product realisation can be seen in Figure 13 which is taken from Polzer et al.[32].

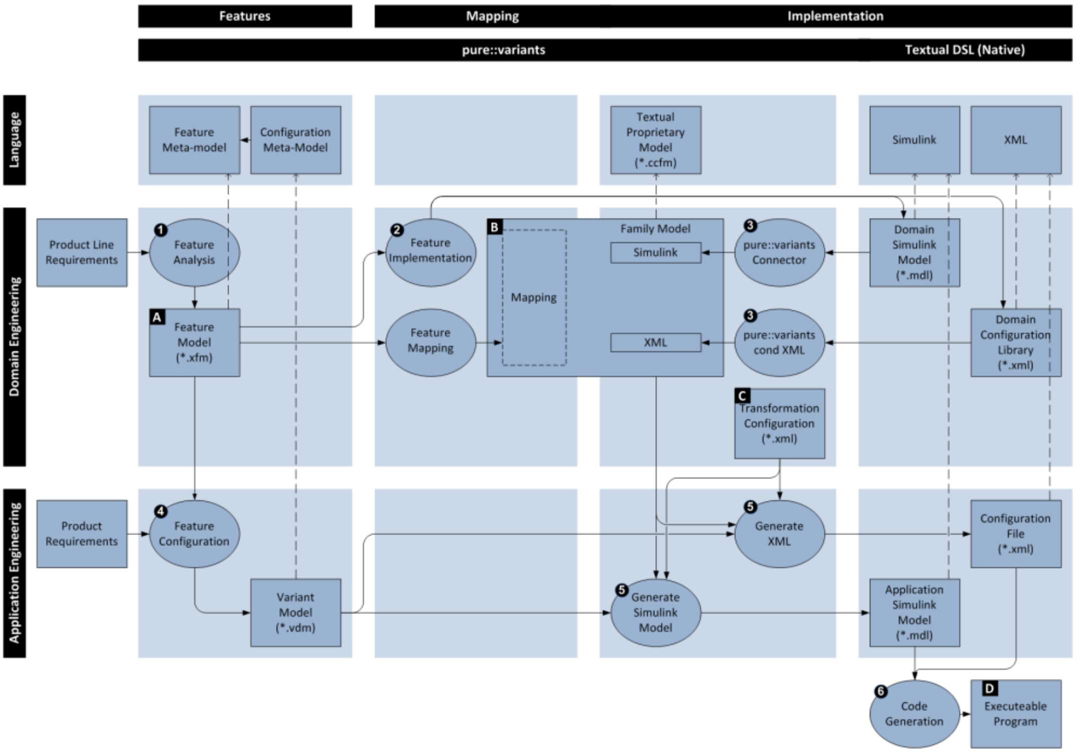


FIGURE 13 ROLE OF FEATURE & FAMILY MODELS IN POLZER ET AL. [32]

Here we can see that the Family Model (B) provides the mapping between the definition of features (held in a feature model) and the design/solution technologies (Simulink and XML).

Family Models often contain, or refer to, product line Reference Architectures. A Reference Architecture [33] provides a standard solution structure for a class of products.

All product instances should conform to the reference architecture and all product components (be they common assets or project-specific) should comply with the reference architecture guidelines for component construction and interfacing. Reference architectures can encompass system, software and hardware representations and can exist at a number of levels of abstraction. The FORM method [34] is one approach to software reuse that makes specific use of a reference architecture.

Figure 14 illustrates the FORM engineering process; this shows the central role played by the reference architecture in the scoping of the product and development of reusable components that are then made available to the application engineering process.

The Family Model & Reference Architecture approach to the decoupling of problem domain analysis from solution domain design appears credible and scalable, and is one that we make significant use of in our research.

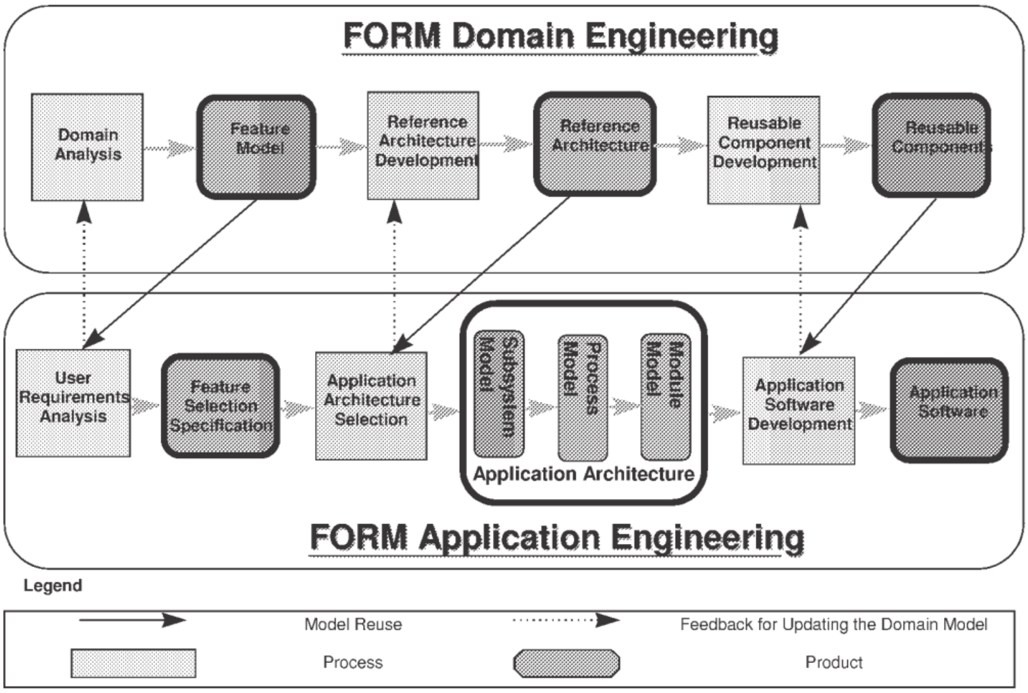

FORM Domain Engineering
Domain Analysis
Feature Model
Reference Architecture Development
Reference Architecture
Reusable Component Development
Reusable Components
User Requirements Analysis
Feature Selection Specification
Application Architecture Selection
Subsystem Model
Process Model
Module Model
Application Architecture
Application Software Development
Application Software
FORM Application Engineering
Legend
Model Reuse
Feedback for Updating the Domain Model
Process
Product

FIGURE 14 FORM ENGINEERING PROCESS (FROM [34])

Product variation in Family Models can be captured in a similar manner to feature variability in Feature Models. Gomaa [4] illustrated a means of capturing commonality and variability in UML class models using a specialised profile. This is summarised in a "component profile" in Figure 15.

Figure 15 identifies the following component classifications in the profile:

- Kernel Component
  A component provided by every member of the product line.

- Optional Component
  A component provided by some members of the product line, but not all.

- Variant Component
  One of a set of similar components that have some identical properties and some different properties. Different components are used by different members of the product line.

- Default Component
  The default (pre-selected) component amongst a set of variant components.

Components of all types can optionally introduce variation via parameterisation. To denote this, there are a set of "param-vp" versions of each of the component classifications given above. This indicates that the values of the configuration parameters need to be set by the individual product line members when using this component. These stereotypes are explicitly denoted "vp" because product line variability is introduced at this point.

In addition, components of type kernel, optional and variant can be denoted as pure "vp" components, indicating that product line variability is introduced via specialisation. The component itself defines the interface that all the specialised components must provide (as a minimum).

Gomaa's approach of defining a UML profile to categorise product line/reusable components and identify the types of variability and component usage is one we make use of in our approach to some extent. For our purposes, however, the detail of Gomma's approach tries to capture too many dimensions that should be kept orthogonal, especially where components may be reused across multiple product lines. In this case, the identification of, say, the default component may not hold true across all users of the component, and therefore this information may need to be held elsewhere in the product line model.

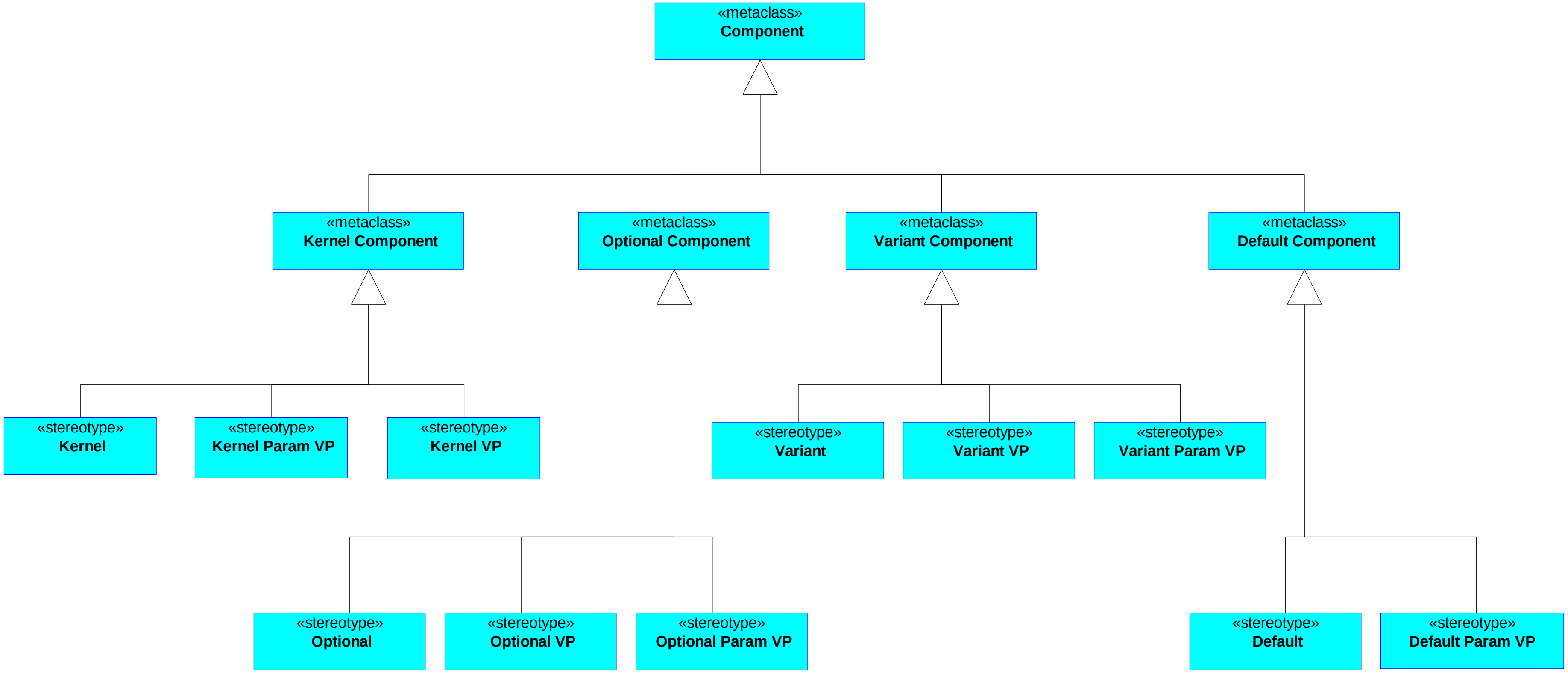


Figure 15 UML Profile for Classification of Components in a Family Model (Adapted from [4])

2.3.2.2 *Mapping Feature Models to Family Models*

If we regard a family model as a solution space response to a product line defined within a feature model, then for the family model to be truly useful in implementing that product line it must have a mapping back to the parent feature model. In this way the commonality and variability in the implementation is traceable back to the needs of the product line, and it provides necessary information for the automation of product derivation. Gomaa and Shin [35, 36] describe how the UML representations of feature and family models (as class models) described in the previous section can be mapped; the meta-model for this is shown in Figure 16.

Noticeable here is the one-to-many mapping of features to components; this would indicate that:

- Features are realised by sets of one or more components
- Components cannot contribute to more than one feature

It can be argued that this is an overly restrictive mapping between the problem and solution domains; components may certainly contribute to more than one feature (particularly "lower-level" components in embedded systems such as components implementing communications protocols). Also in this approach, variability is implemented at the component level (replacement, removal and/or inclusion of components to implement the selected feature set). Other approaches allow a more fine-grained implementation of variability that can be more useful in certain domains; this is discussed later in this chapter.

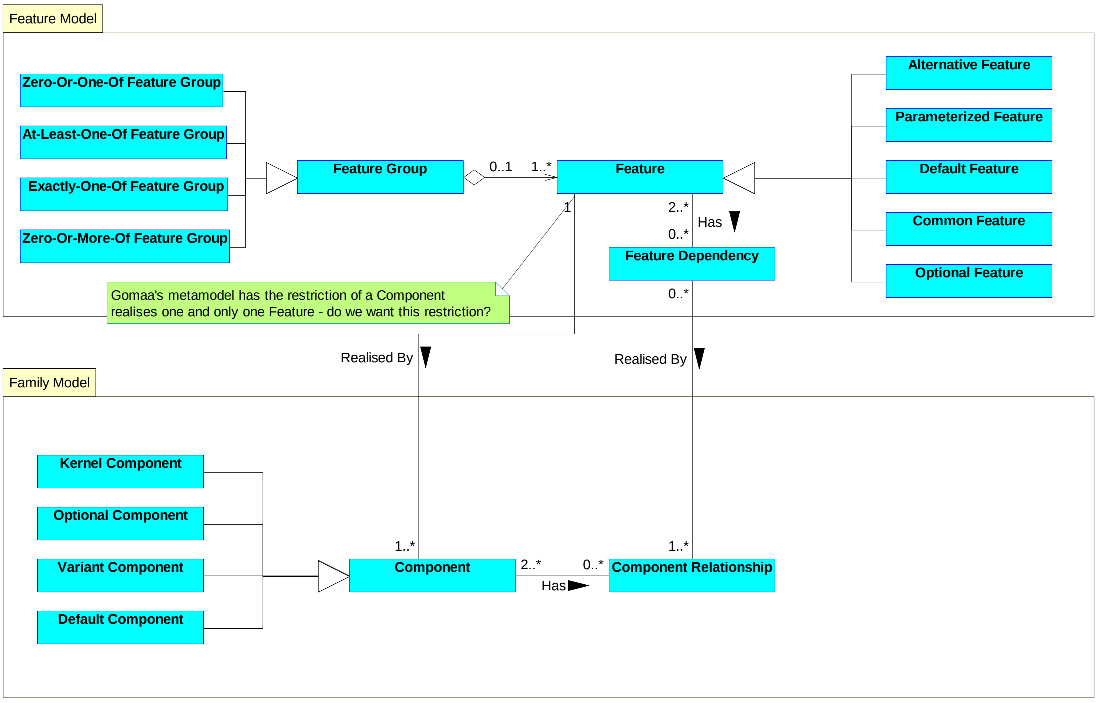


FIGURE 16 META-MODEL MAPPING FEATURE AND FAMILY MODELS (ADAPTED AND SIMPLIFIED FROM [36])

2.3.3 Common Variability Language

The Common Variability Language (CVL) [30, 37] has been proposed as a standardised mechanism to describe variability for MOF-compliant [38] languages (MOF stands for Meta Object Facility and is a meta-meta-modelling language defined by the Object Management Group (OMG)). (At the time of writing CVL is in the RFP stage of the OMG standardisation process.) The Request for Proposal (RFP) for a CVL [30] states that *"Product line modeling includes a base product line model, a (separate) variability specification that applies to the base model, and resolutions of variabilities in order to generate specific product models. The objective of product line modeling is that the derivation of specific models based upon resolutions of variabilities should be as automatic as possible"*. This is shown diagrammatically in Figure 17.

The RFP's statement that variability models are separate from the product line model is not necessarily true as a general case for product line modelling. (The RFP goes onto to state that "*the concrete syntax of the specification of the relationships between the variabilities and the base may appear as annotations to the base notation.*" This would suggest that the base mode would be "polluted" with a degree of variability information.) However, the objective of automatically deriving specific models based on the resolution of variabilities is the aim of most product line modelling approaches. The final resolved model is in a form that can be processed by "regular base language tools" i.e. it is of the same or similar form to a model of a single product. It is not clear if the RFP requires this model to be available for inspection, serialisation etc., or whether a transitory model would suffice.

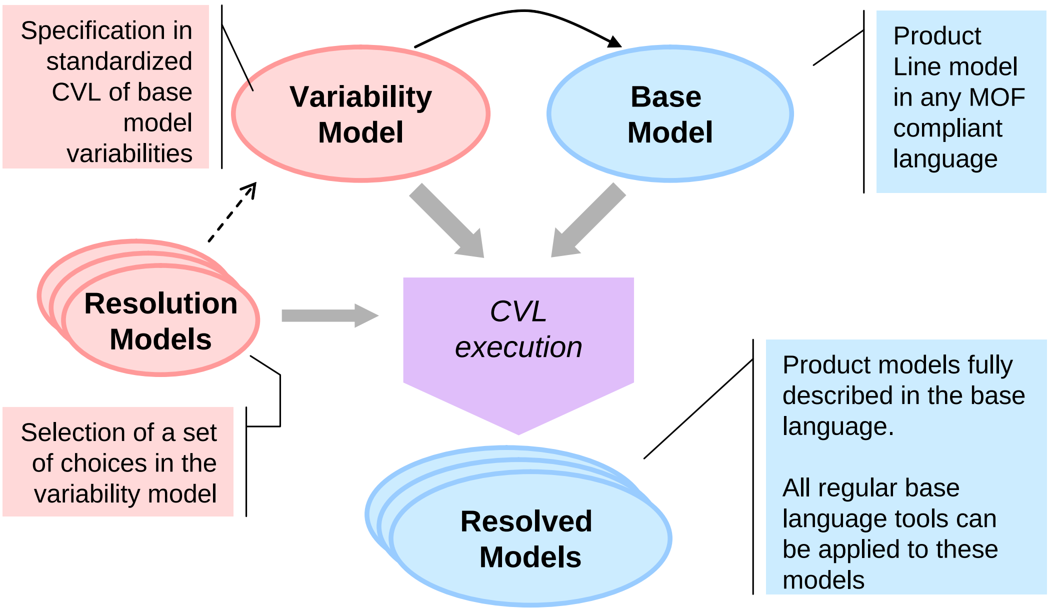

Specification in standardized CVL of base model variabilities
Variability Model
Base Model
Product Line model in any MOF compliant language
Resolution Models
CVL execution
Selection of a set of choices in the variability model
Resolved Models
Product models fully described in the base language.
All regular base language tools can be applied to these models

FIGURE 17 CVL ARCHITECTURE [30]

In their proposed implementation of CVL, Haugen et al. [37] provide a meta-model for specifying variability and a process for variability resolution when instantiating product models. They term their approach BaseVariationResolution or BVR (Figure 18).

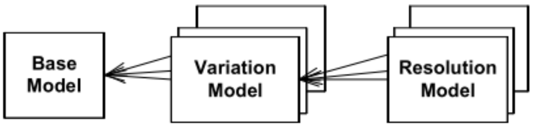


FIGURE 18 BASEVARIATIONRESOLUTION (BVR) APPROACH [37]

In BVR, a single base product-line model is specified whose variability is described in a separate, orthogonal Variation Model. Products are derived via the definition of a Resolution Model that specifies the set of variability selections for that specific product.

The BVR meta-model is shown in Figure 19. The definition and relationship of the Variation Model to the Base Model is clearly shown here, including the mapping of variability specifications to model elements in the base model.

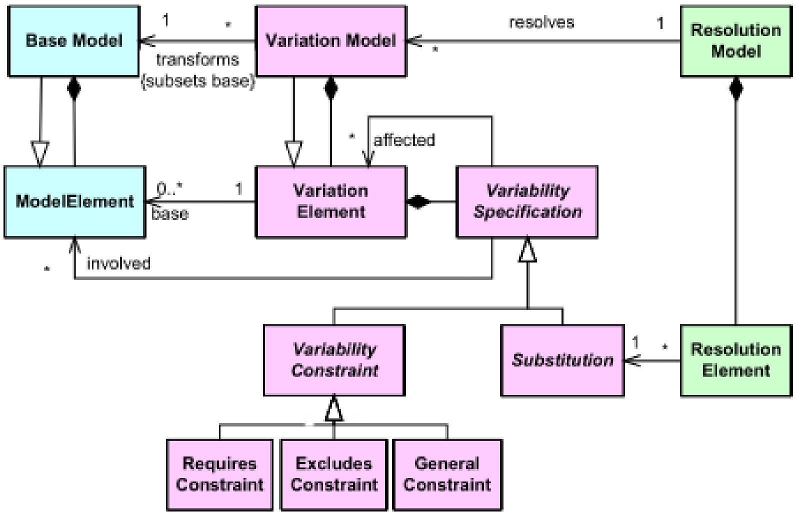


FIGURE 19 BASEVARIATIONRESOLUTION (BVR) APPROACH META MODEL [37]

More detail is provided regarding the variation via substitution in the meta-model shown in Figure 20. Here the variability of an attribute in the base model is described, using substitution of attribute value (Value Substitution), and also potential replacement of the attribute by another (Reference Substitution). In the reference substitution example, the variability specification identifies the target attribute in the base model (*original*), the set of potential alternatives (*alternatives*), and when applying a resolution model, the chosen attribute (*chosen*).

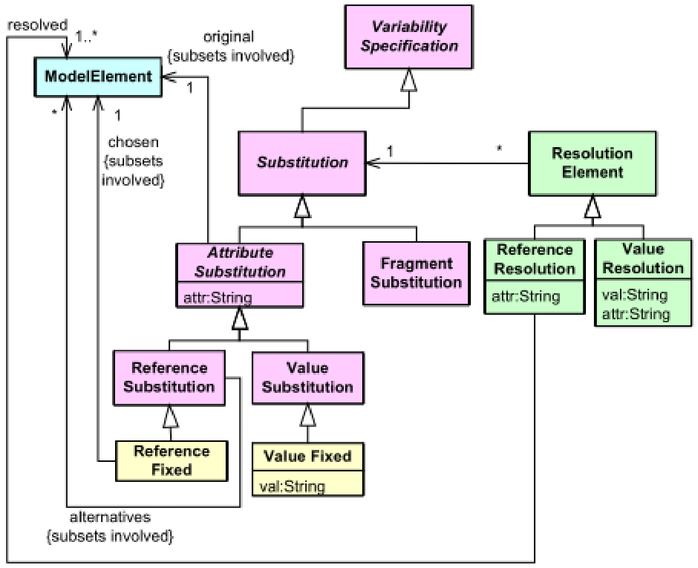


FIGURE 20 VARIABILITY SPECIFICATION (FROM [37])

The process for resolving a product model is shown in Figure 21.

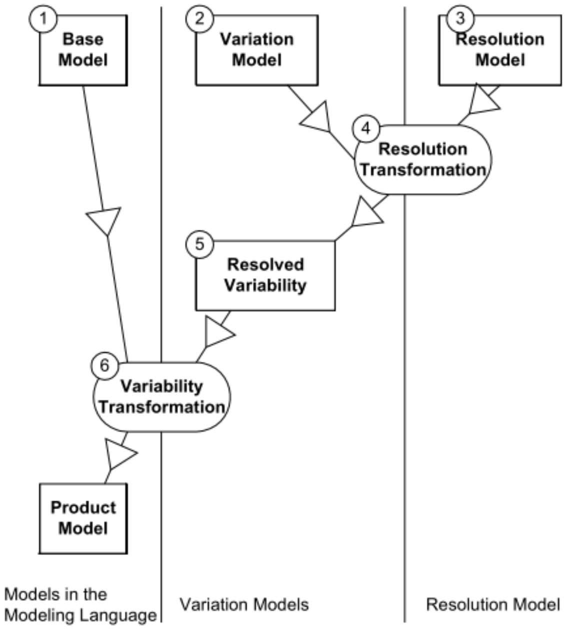


FIGURE 21 VARIABILITY MODEL RESOLUTION PROCESS (FROM[37])

Here, two transformations are described; the first resolves the variability set by combining the resolution model and variability model. The second takes this resolved variability and applies it to the base model to produce the final instantiated product model.

One of the major advantages claimed for the CVL/BVR approach is that it allows the separation of concerns. The Base model is orthogonal to the Variability model and they can be developed separately using the skills of product modelling experts (Base) and domain experts (Variability). However, it would appear that although the base model is free of variability information; it needs to contain sets of alternatives from which the variability model can denote choice (as in the Reference Substitution example in Figure 20). This is problematic in two ways; the developers of the base model need to have a "reason" or need to include the additional/alternative model elements – this is naturally driven from the need for variability (i.e. the variability model). Although the variability and base models may be semantically and syntactically orthogonal, from a **process** viewpoint they are not. In addition, it is difficult to verify and validate the base model without the variability model, as again it would be unclear to the reviewer the rationale for the structure of the base model without the identified need for variability. Again, the separation of concerns argument falls down here. CVL/BVR is an interesting approach, but is not built upon further in the research described in this thesis.

2.3.4 Component-Based Architecture and Variability

The literature contains a number of approaches to component-based software development [39] that do not use UML as the underlying modelling paradigm or provide extensions to the UML. Some of these propose graphical notations/concrete syntax and some provide purely textual representations. A number of these approaches have been extended to provide variability support [40, 41]. Component modelling approaches such as Koala [42] , KobrA [43], COPA [44] have been proposed for product line development; some designed to target specific domains (for example COPA was developed to target telecommunication infrastructure and medical domains [45])

2.3.4.1 MontiArc^{HV}

In their recent paper, Haber et al. [21] discussed an approach to component-based development that incorporates variability to enable product line instantiation. Many approaches to solution space variability propose a model that describes the variability across the whole system [27, 46]. These **variability models** [2] can be monolithic and are in many cases held separately from the system description model. This can result in problems with management and synchronisation between the separate models. It is argued in [21] that the development of a system product line using a component-based approach has to satisfy the following requirements:

1. Component variability and hierarchy need to be treated uniformly in one model
2. Variability must be specified locally to the components.
3. The variability model should allow focussing on the common architecture of all system variants, on the component variability and on the configuration of the used components.
4. Design/Configuration decisions at a high level map to variant selection on components at a low level of the hierarchy.

Point 2 in the above list is especially important if the product line development is to be undertaken by diverse, geographically separate teams [47].

Haber et al. [21] propose an approach called MontiArc^{HV} which is an extension of an architectural description language (ADL) MontiArc [48] to include the concept of hierarchical variability. The extension recognises variation points as first-class citizens in the modelling language, compliant with the meta-model shown in Figure 22.

The meta-model describes how variability is modelled as a first-class modelling element, where a VariationPoint is a type of architecture element (ArcElement) in the same manner as, say, a port or connector. Definitions of variants can be modelled using the MontiArch component syntax, and a selection made of a valid variant to augment common component behaviour at the point of product instantiation.

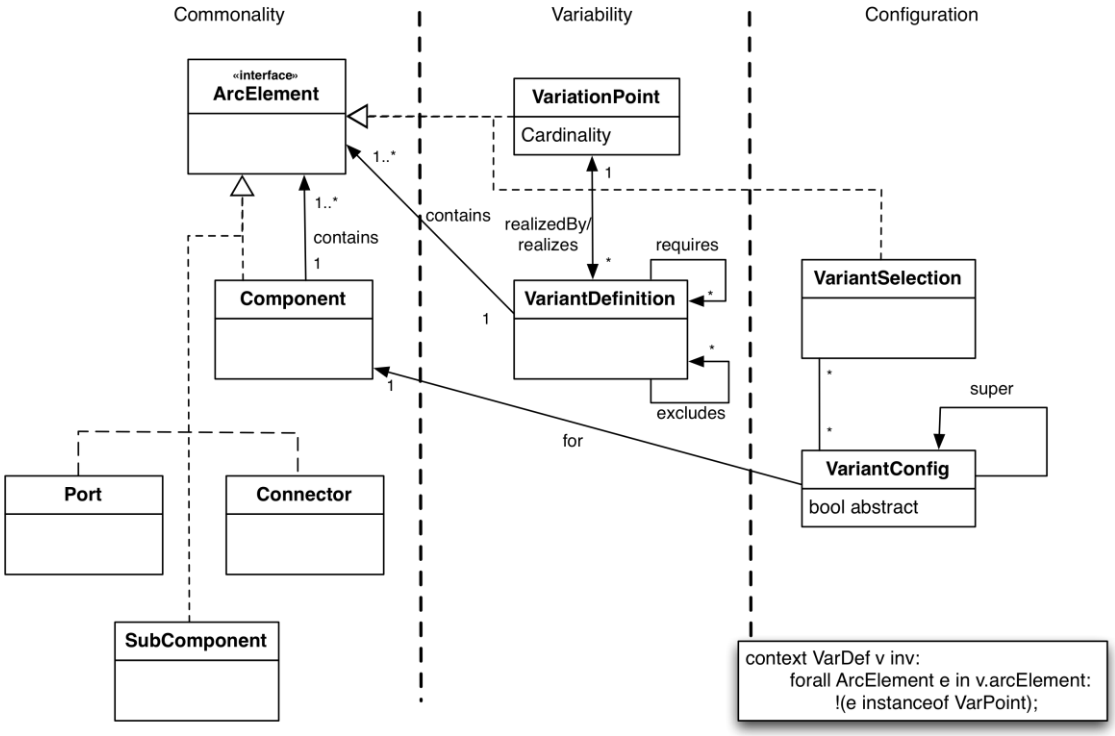


FIGURE 22 META MODEL FOR HIERARCHICAL VARIABILITY MODELLING (FROM [21])

Using this approach, component designs can be created that contain VariationPoints, enabling variation to be modelled and instantiated at the component level. An example of this approach is shown graphically in Figure 23 and Figure 24.

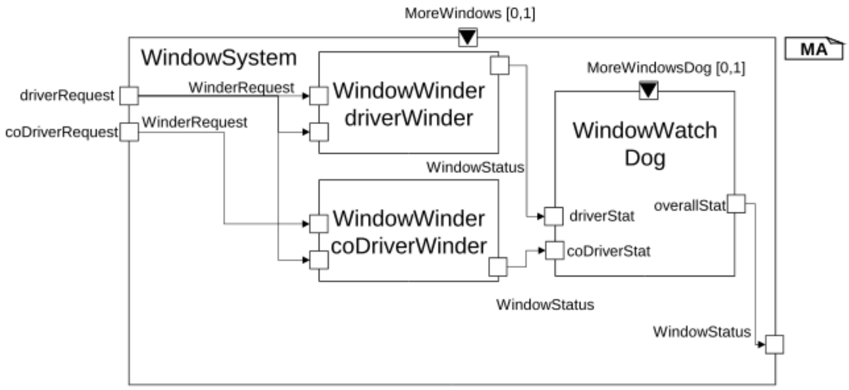


FIGURE 23 EXAMPLE MONTIARCH^{HV} COMPONENT FROM [21]

Here, a WindowSystem component for a vehicle is described which exposes a VariationPoint called MoreWindows, with a [0..1] cardinality. This allows for 0 or 1 variation extensions to be included when instantiating this component into a product. Figure 24 provides an example of a variant extension, showing how the component can be extended to provide rear window winder behaviour. The resulting instantiated component will provide a superset of the common component and variant extension.

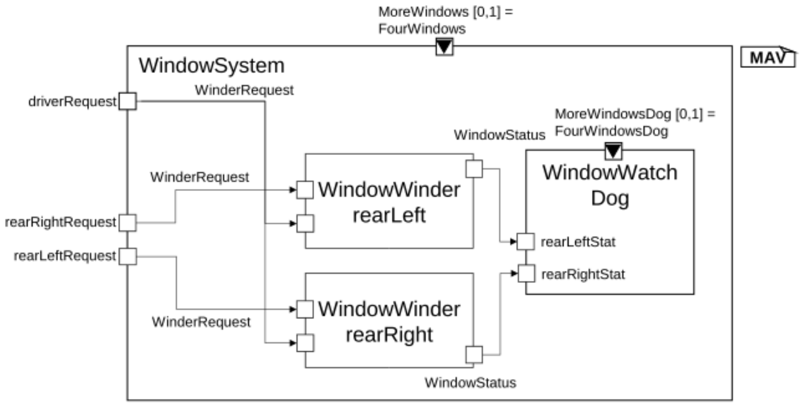


FIGURE 24 EXAMPLE VARIANT COMPONENT DESCRIBED USING MONTIARCH^{HV} FROM [21]

The strength of this approach is twofold: the recognition that the modelling of variability as a first-class citizen in the component design allows the variability to be clearly modelled at design time, and the component development can be distributed without the distribution of the complete variability model. It also enables the reuse of components across disparate product lines, as long as the component-specific variability is of use to the recipient product line.

The main weakness of the approach appears to be the inability to describe variability other than the result of additive composition of the modelled components and variants. Indeed the authors identify as future work the extension of the approach to include “more invasive composition techniques” [21, 49]. However this work is still probably the closest to our approach of “decision contracts” (described in [50] and later in this thesis) that exists in the literature today.

2.3.4.2 PlasticPartialComponents

Plastic Partial Components [51] provide an approach to component-based development that includes the ability to define internal variation of component behaviour using an invasive software composition [49] technique. This approach is termed Plastic as the component behaviour is easily adapted to each product of the SPL, and Partial as they only participate in the core product line definition with the behaviour that it is common to the family of products. The Partial Plastic Component meta-model is shown in Figure 25.

The approach defines a specialised component *PlasticPartialComponent* that aggregates a set of *VariabilityPoints.* A *VariabilityPoint is* characterised by three properties:

- Cardinality defining a “kind of variation”
- “Type” of variability (crosscutting or non-crosscutting)
- Weaving between variant and component

The approach uses an aspect-oriented programming (AOP) [52] approach to defining the weaving operator, using primitives such as *pointcuts, weavings* and *aspects.*

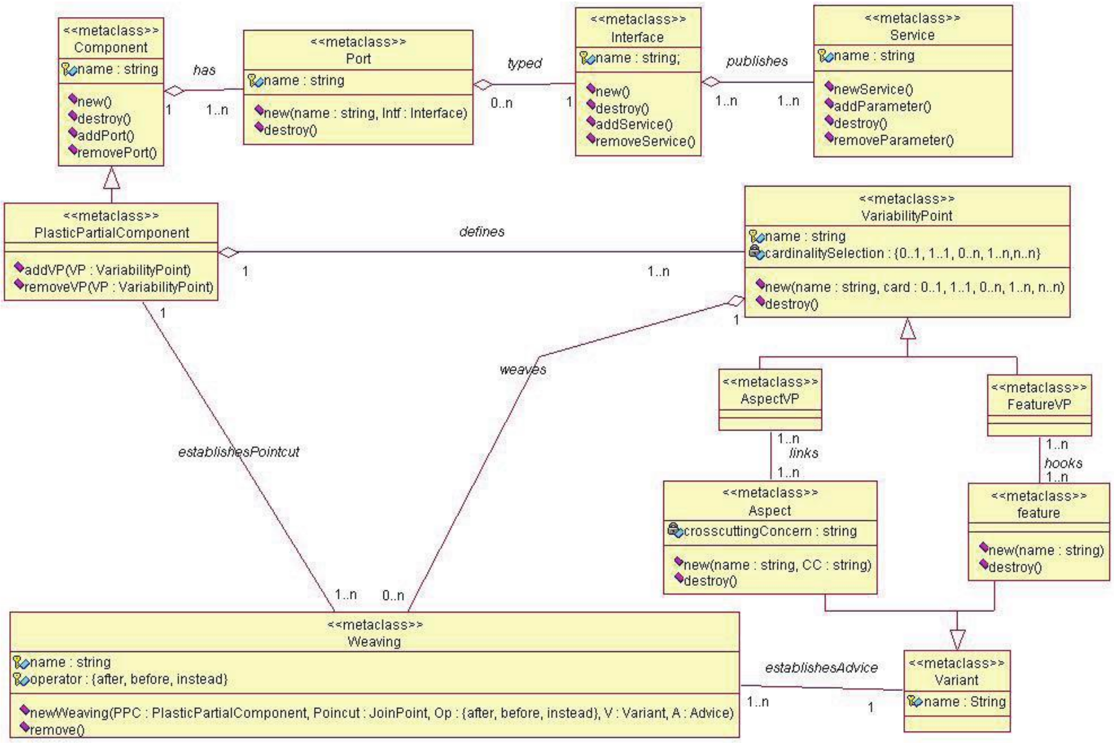


Figure 25 PartialPlasticComponents Meta-Model From [51]

The authors claim that the approach is designed to support the internal variation of architectural components; where that internal variation is an invasive composition of aspects and features. One of the weaknesses of this approach is that it appears to regard internal component variability to be behavioural alone; the “worked examples” given in [51] describe variability as the replacement of services provided by a component.

However, internal component variability for many embedded systems may manifest itself as, for example, variation of data, buffer and array sizes, multiplicity of data sources etc.

In addition, the binding time of variability proposed by the PlasticPartialComponent approach is unclear; the reliance on AOP would suggest a compile-time or run-time binding model. (Binding time refers to the point in the software development process at which the variability is resolved – i.e. the point in time where a specific product instance is defined. In some products, this can be as late as the execution of the software – so called "run-time binding".). The authors claim that one of the benefits of this approach is "*easily adoptability* (sic) *of the concept of Plastic Partial Component by any architectural model, that has a meta-model definition*" [51] However, as noted in [21], it would not be applicable to hierarchically modelled components , as variants cannot contain variant components.

2.3.5 Product Instantiation Using Variability & Transformation

Models can be descriptive (i.e. provide an abstract description of a system to aid understanding) or prescriptive (i.e. provide a plan, blueprint and/or process which guide the system's construction). (This classification is generally used to apply to architectural models [53] but can equally apply to behavioural models of a system.) The models most useful to the development of a software system have both prescriptive and descriptive views; they aid understanding of the system whilst simultaneously defining unambiguously how to build the system. Prescriptive product-line models that contain well-defined statements of commonality and variability can be used to automatically generate product instances, given a set of selections that resolve the variability for that product:

"The objective of product line modelling is that the derivation of specific models based upon resolutions of variabilities should be as automatic as possible" [30]

For trusted product lines it is also useful to use descriptive models (or the descriptive parts of models) to automatically produce descriptions and documentation of the instantiated product.

This process utilises a set of techniques known as **model transformation** [54] [55]. The following definition of model transformation is given by Kleppe et al. [56] :

*"A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition. A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."*

This is generalised by Mens and Van Gorp [55] in their taxonomy of model transformation to encompass the potential multiplicity of both source and/or target models. Their taxonomy also distinguishes the following characteristics of model transformation: **Endogenous transformations** transform between models compliant to the same underlying meta-model whereas **Exogenous transformations** transform between models compliant to differing meta-models. The taxonomy also distinguishes between transformations that span differing levels of abstraction (**vertical transformations**) and those that transform within the same level of abstraction (**horizontal transformations**). They also claim that these characteristics are orthogonal; code generation being an example of an exogenous vertical transformation and refactoring as an example of endogenous horizontal transformation. (Note that endogenous transformations are sometimes termed **model manipulation** rather than transformation as no change in underlying meta-model takes place.)

We can further classify transformations based on the form of their input or output. The output of a **Model-to-Model** transformation (commonly abbreviated to M2M) is a model compliant with a target meta-model. In contrast, transformations that result in a textual output are termed **Model-to-Text** transformations (M2T). The textual form of the output is not the only criteria for a transformation to be termed M2T; typically, these transformations do not attempt to understand or represent the meta-model of the output. M2T transformations are usually template based, mapping source meta-model elements to fragments of text.

2.3.5.1 Model Transformation Approaches for Product Line Instantiation

There are essentially two fundamental approaches for realising variability in product lines via model transformation: reductive and additive transformations (Figure 26).

Reductive Transformations

A reductive transformation (also known as negative variability [57]) removes information from an overall whole. The removed information is identified as not being required for the particular instance being instantiated.

Additive (Compositional) Transformations

An additive or compositional transformation adds information to a minimal core or base model where the optionally included information is identified as being required for the particular instance being instantiated via transformation [30]. This approach is also known as model injection [58] or positive variability [57]. Typically, we achieve this type of transformation by using a form of model weaving approach.

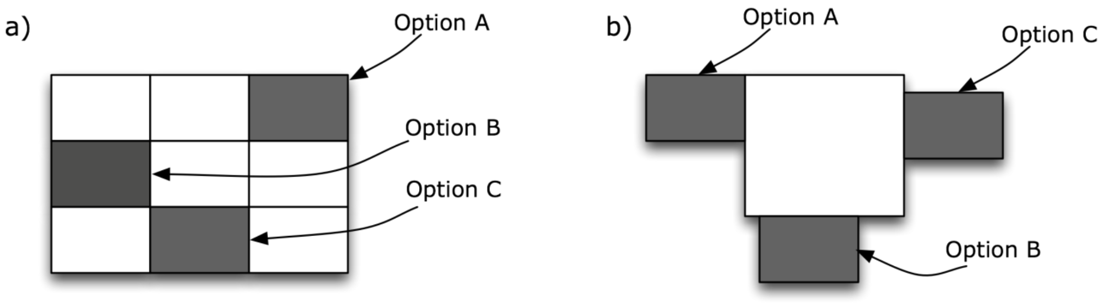


FIGURE 26 REDUCTIVE/NEGATIVE (A) AND ADDITIVE/POSITIVE (B) VARIABILITY [57]

As identified by Voelter [58], the main challenge with a reductive approach is that the product line model can become big and unwieldy. Voelter [58] also identifies the main challenge with the additive or model injection approach as being able to precisely identify the point of injection. It can be argued that there are other challenges with additive transformations. The "point of injection" issue can become more problematic when multiple injections are targeted at the same point. Order of application can then become an issue, especially when the model fragments being injected are related semantically. Another problem with injection is the identification of the minimal core model. Through a strict commonality and variability analysis, theoretically it should be possible for the minimal core model to be identified. However it is not inconceivable that increasing understanding of the product line scope and application may result in the migration of what were once common (core) features into points of variation and therefore migrated out of the core model. If these were themselves the targets for other variation (injection points) then this can become a non-trivial model management problem.

2.3.5.2 Architectural Transformations

Architectural transformations allow a product specific architecture to be automatically derived from a product line architectural description. Botterweck et al. [26] discuss an approach to derivation of a product specific architecture using a model driven approach. The derivation process proposed by [26] is illustrated in Figure 27:

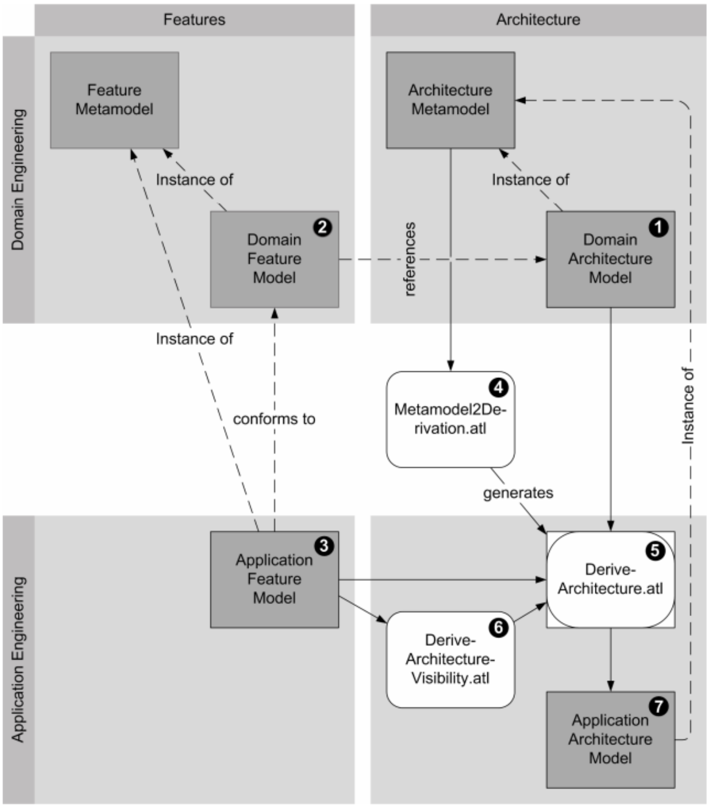


FIGURE 27 APPLICATION ARCHITECTURE DERIVATION PROCESS FROM[26]

This approach utilises a feature model mapped to an architectural model. The product specific architecture model is derived using an ATL transformation. This transformation selectively copies architectural components as required by the particular product feature selection. One of the limitations of this approach is that it assumes that the product-line architecture is structured to allow the inclusion or exclusion of features by the inclusion or exclusion of complete components. This precludes its use when features are implemented using internal variability of components (c.f. the PlasticPartialComponents approach [51]). However, the concept of transforming architectural and component models by the selective inclusion of architectural elements via transformation is powerful, and is one we will utilise later in this thesis (see chapter 5 section 5.3)

Botterweck et al. [26] conclude their paper by posing a number of research questions regarding the use of the generated application architecture model. One of these is how to use the derived application architecture model as a foundation for an implementation. This is a key theme to our research and will be addressed later in this thesis (see chapter 4 section 4.7 and chapter 5 section 5.4).

2.4 High-Integrity Software System Development

The domain of high-integrity software development is too wide for a full treatment in this literature survey. We are particularly interested in the development of software systems for use in civil aerospace applications, as this is the area of the author's expertise, and is the target domain for the systems used as a case study in this thesis. We therefore concentrate on the current and forthcoming regulatory requirements for the development and approval of software in civil avionics systems, and any associated literature in this domain.

2.4.1 DO-178B/ED-12B

Civil avionics is a typical example of a high-integrity regulated domain, in which software is developed to a set of industry guidelines and is subject to audit and approval by a regulatory authority or body (sometimes multiple authorities/bodies). Regulatory authorities are typically a governmental body who must approve products prior to their public use to ensure safety or security is not compromised. Prior to entry into service, civil avionics software is required to be approved by an airworthiness authority, a process more commonly known as "certification". This approval process typically takes the form of a set of audits designed to demonstrate the software has been developed in accordance with the guidance of DO-178B/ED-12B "Software Considerations in Airborne Systems and Equipment Certification" [4].

DO-178B/ED-12B provides guidance for software development in airborne systems and takes the form of a set of software development process objectives. In this context, a process objective describes some facet or attribute of the software development for which demonstrable compliance evidence needs to be supplied to support the approval of the software. The guidance recognizes five development assurance levels (DAL levels A to E), Level A being the most stringent. The software requirements and design process objectives remain relatively constant across the development assurance levels; however, the verification process objectives increase both in number and in the level of independence required as the development assurance levels become more stringent. An example of one of the software coding objectives is that “Source Code compiles with the Low Level Requirements” (Objective 1 Table A-5), an objective that needs to be demonstrated with independence for development assurance level A.

A number of the DO-178B/ED-12B objectives concern the use of tools within the software development process. Wherever a tool is used to automate part of the software development activity, and its output is not separately verified, then that tool requires **qualification**. Tool qualification provides evidence that a tool is operating as expected/required when used in support of DO-178B/ED-12B objectives. The requirements for tool qualification vary dependent upon whether the tool is a verification or development tool. Verification tools cannot introduce an error into the software product; they can only fail to detect an error. Therefore, the qualification requirements for verification tools are relatively straightforward, and take the form of a simple acceptance test of the tool against a set of operational requirements plus strict revision control. Development tools, however, produce output that forms part of the software product and therefore are capable of introducing an error into the product (for example automatic code generators producing source code). Development tools whose output is not separately verified are required to be developed to the same assurance level as the software product they are used to develop.

TABLE 1 OBJECTIVES VS LEVELS IN DO-178B/ED-12B

Level	Failure condition	Objectives	With Independence
A	**Catastrophic**	**66**	**25**
B	**Hazardous**	**65**	**14**
C	**Major**	**57**	**2**
D	**Minor**	**28**	**2**
E	**0**	**0**	

This causes significant problems for organisations wishing to develop and/or use qualified development tools, particularly for Level A projects. As DO-178B/ED-12B provides objectives for the software development process to follow, it is almost impossible to retrospectively provide qualification evidence for an existing tool. In addition, the safety critical software development tools market is so small that it is hardly ever commercially viable to develop a tool compliant with DO-178B/ED-12B Level A objectives. Currently the only commercially available development tool that is qualifiable to DO-178B Level A is the SCADE “pictures-to-code” environment produced by Esterel [59]; this is discussed later in this chapter.

2.4.2 DO-178C/ED-12C

DO-178B/ED-12B has been used in the approval of civil aerospace systems since it was ratified in 1992 (A Frequently Asked Questions/Clarifications document DO-248/ED-94 was released in 2001). In 2005, RTCA and EUROCAE (the industry bodies that publish the guidance material) decided to instigate a working group to revise the guidance in light of emerging technologies increasingly being used in the development of aerospace systems and to which the guidance was not being consistently applied. SC(Special Committee)-205/WG(Working Group)-71 was instigated with the terms of reference to produce a suite of guidance documents that included the following :

- DO-178C/ED-12C "Software Considerations in Airborne Systems and Equipment Certification" [60]
- DO-278A/ED-109A "Guidelines for Communications, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance" [61]
- DO-248C/ED-94C "Supporting Information for DO-178C and DO-278A" [62]
- Technology supplements covering the following :
    - DO-330/ED-215 Tool Qualification [63]
    - DO-331/ED-218 Model-Based Development and Verification [64]
    - DO-332/ED-217 Object-Oriented Technologies and Associated Techniques [65]
    - DO-333/ED-216 Formal Methods [66]

The terms of reference for the updates of the core documents were not to undertake a radical revision, but to include the errata that had been identified over the years and to reduce the need for FAQs and discussion papers. A key requirement was to preserve the 66 objectives from the previous guidance; however, some so called "hidden objectives" in DO-178B/ED-12B were clarified and made visible in the annexe tables that define how the objectives vary by DAL. The purpose of the technology supplements was to provide an agreed interpretation of the guidance for the development and approval of systems employing the technologies identified. This could include the definition of additional or alternate objectives if appropriate.

TABLE 2 OBJECTIVES VS LEVELS IN DO-178C/ED-12C

Level	Failure condition	Objectives	With Independence
A	**Catastrophic**	**71**	**33**
B	**Hazardous**	**69**	**21**
C	**Major**	**62**	**8**
D	**Minor**	**26**	**5**
E	**0**	**0**	

Figure 28 is taken from DO-333/ED-216, the Formal Methods supplement to DO-178C/ED-12C [66] – it illustrates the required verification processes for Level A software and the relationship of these to the design data required for DO-178C/ED-12C compliance. It provides a very useful overview of the set of verification objectives that should be met when approving a system as part of an aircraft or engine certification programme.

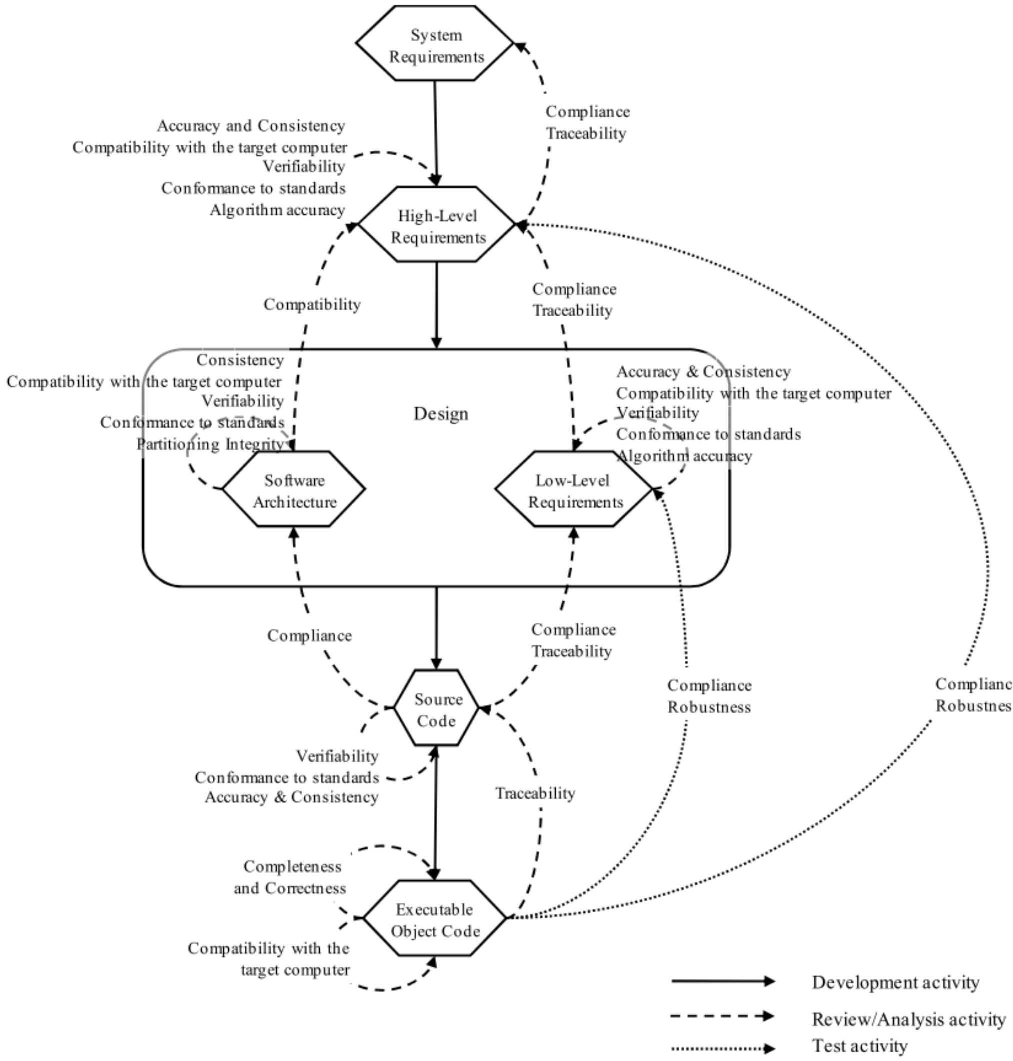


FIGURE 28 DO-178C/ED-12C LEVEL A SOFTWARE VERIFICATION PROCESSES [66]

The various verification objectives and methods (review, analysis, test) and their relationship to the development processes can be seen clearly in Figure 28. In the context of Trusted Product Lines, it must be borne in mind that this set of verification objectives needs to be satisfied **for an instantiated product**. We will take this as a framework against which to assess the effectiveness of the trusted product lines approach later in the thesis.

2.5 Model Based Development of High-Integrity Systems

We do not cover the subject of model-based development in general in this review, as the literature is vast and wide ranging on the subject. Instead, here we concentrate on the specific application of model-based techniques to the development of high-integrity software systems.

2.5.1 DO-331/ED-218 Model Based Development and Verification Supplement to DO-178C/ED-12C

Model-Based Development and Verification was one of the technology areas for which supplementary guidance was required as part of the DO-178C/ED-12C initiative. The interest in this technology supplement was very high; it was the largest sub-group in terms of attendees at the working group. This level of interest was shown from industry representatives, tool vendors and regulators, primarily because model-based development is now widely used for the development of avionics systems and the current regulatory guidance can be open to interpretation.

The supplement defines a model as:

An abstract representation of a given set of aspects of a system that is used for analysis, verification, simulation, code generation or any combination thereof. A model should be unambiguous, regardless of its level of abstraction.[64]

The main guidance provided by the supplement can be summarised as follows:

- Models need requirements
- Simulation is an acceptable means of satisfying certain verification objectives
  - Compliance of development (simulated) artefact to parent
  - Partially satisfy compliance of Executable Object Code to High Level Requirements (in specific circumstances)
- Traceability alone is not an acceptable means of identifying unintended functionality in design models

Primarily, however, most of the guidance is aimed at providing a regulatory framework around the use of **behavioural models**; this reflected the interests of most of the working group participants. The following sections discuss the use of this type of model in relation to avionics software development.

2.5.2 Model Environments

It is not the intention of this literature review to provide an in-depth study of the research surrounding the individual, technology-specific modelling approaches discussed here, but it provides overview of the most widely used techniques in avionics development. Much of the model-based development in avionics is centred on the construction of behavioural models of the system; this is prevalent in the development of embedded control systems, where the design and validation of the control logic is performed by control engineering specialists. The ubiquitous tool for this type of modelling is Matlab/Simulink [67].

2.5.2.1 Matlab, Simulink & Stateflow

MATLAB is a technical computing environment developed by Cleve Moler in the late 1970s, and was commercialised by The Mathworks company in 1984 [68]. Its primary users were control systems design engineers, but its usage has widened into other domains. The wide use of the tool in control systems designs led to the development of Simulink, a graphical-based environment for the modelling, simulating and analysis of dynamical systems. Simulink provides an interactive graphical environment and a customizable set of block libraries that allows the design and simulation of a variety of time-varying systems [69]. Stateflow is an extension to Simulink providing a design environment for developing state machines and flow charts [70].

Systems engineers in general, and control engineers in particular, find Matlab/Simulink useful as it enables the rapid development, simulation and analysis of algorithms and behavioural system designs. However, its relatively simplistic view of architecture (it supports a hierarchical functional decomposition) and the lack of formalism underpinning the semantics of the languages supported mean that it is flawed as a software design tool. One of the challenges for modelling tool developers and vendors is to provide environments that are understandable to the problem-domain experts (e.g. control engineers) but have a sufficiently strong theoretical basis to enable reasoned arguments to be made about the correctness of the designs and resulting software systems. This is one of the main reasons for the existence of the SCADE Suite from Esterel.

2.5.2.2 Esterel SCADE

The SCADE Suite[59] from Esterel Technologies has been developed to provide a "*correct-by construction"* [71] approach to the development of high-integrity software systems from model-based representations of software designs. The development of a software design in SCADE is based upon a graphical block-diagram notation similar to that used by Simulink, with a complementary "Safe State Machines" notation to describe state- or mode-oriented computations. (Where a "mode" refers to the specific behaviour of the software system in a particular run-time context, for example aircraft systems may have differing behaviour dependent upon whether the aircraft is on-ground or in-flight.) Both of these specification notations have precise semantics [71]

Esterel [72] identify the fundamental differences between Simulink and SCADE as:

- "*SCADE models time in discrete increments whereas Simulink models time continuously*"
- "*SCADE is completely modular, meaning that the behavior of a SCADE subsystem does not depend on its context, whereas the behavior of an equivalent Simulink 'subsystem' does."*

One of the most attractive features of the SCADE suite to the avionics development community is that it offers a DO-178B Level A qualified code generator. This means that the "Software Coding Process" is automated by a tool that has been developed to the requirements of DO178B Level A, and therefore the output of the process (source code) does not need to be verified. The removal of the need to perform the low-level verification of the source code is seen as a major development process cost saving. "*In particular, we have eliminated the very costly need for MC/DC coverage analysis of the Source Code*" [71]. However it should be noted that this has just moved the burden of collecting MC/DC coverage to the design model stage, where the equivalent of MC/DC coverage at the model level needs to be collected during simulation to satisfy the guidance (as clarified by the model-based supplement to DO-178C/ED-12C)

Esterel provide a model interchange tool "SCADE-Simulink Gateway" to enable the translation of models from Simulink into the SCADE environment for refinement to source code. This provides a behavioural mapping based upon the syntax and semantics of a restricted set of Simulink design "blocks" into SCADE – this cannot be an exact semantic translation as the underlying models of time differs between the two environments.

Esterel identify a model for software development as shown in Figure 29

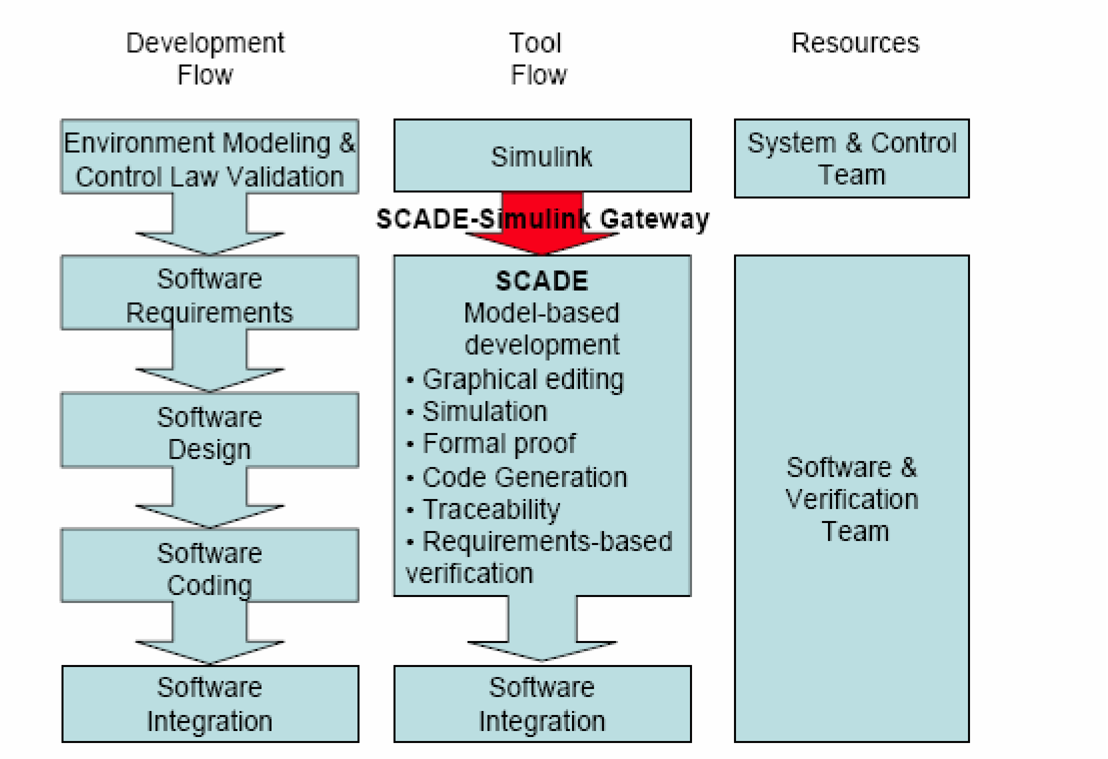


FIGURE 29 SCADE DEVELOPMENT MODEL INCLUDING SIMULINK GATEWAY (FROM [72])

If we compare the process outlined in Figure 29 with the guidance given in DO-178B, there is a distinct lack of recognition of the role of software architecture, and the implication appears to be that the role of the software requirements and design process is purely to formalise the algorithms allocated to software from the systems engineering process.

Dion [71] attempts to describe the role of SCADE within a DO-178B development process. Firstly, the development cycle for ARP4754/DO-178B is illustrated (as shown in Figure 30). ARP4754 [73] is the aerospace recommended practice (ARP) for the certification of complex aircraft systems, and is used in this context to identify the interface between the systems and software engineering processes.

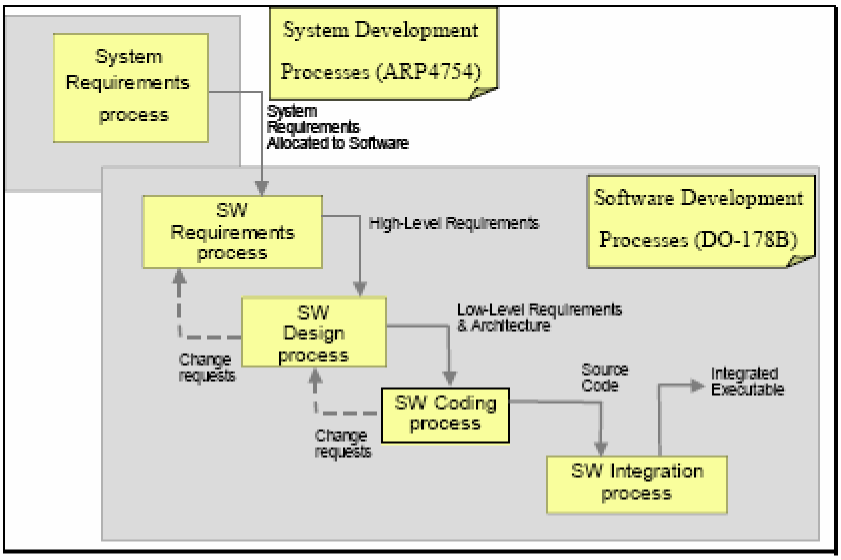

System Requirements process
System Development Processes (ARP4754)
System Requirements Allocated to Software
SW Requirements process
High-Level Requirements
Software Development Processes (DO-178B)
SW Design process
Low-Level Requirements & Architecture
Change requests
Source Code
Integrated Executable
SW Coding process
Change requests
SW Integration process

FIGURE 30 "THE ARP 4754/ DO-178B DEVELOPMENT CYCLE" FROM [71]

This correctly identifies the development processes and lifecycle data as required by DO-178B. Dion then describes how the verification objectives of DO-178B change as SCADE and qualified code generation is used (as shown in Figure 31)

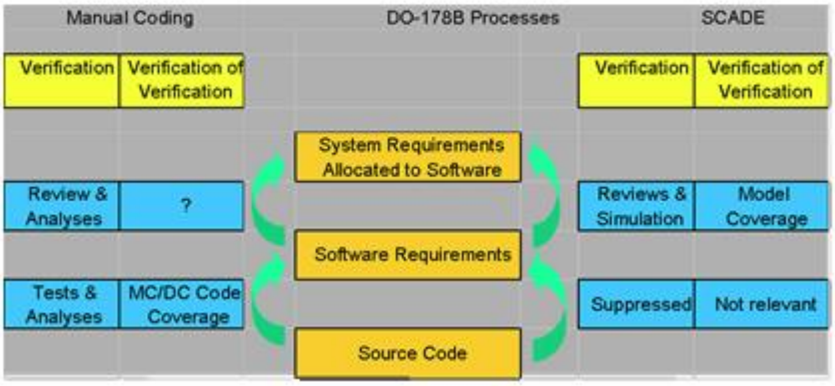


FIGURE 31 "THE USE OF SCADE VS THE USE OF MANUAL CODING" FROM [71]

Here we can see the comparison of the verification activities required for the Manual Coding and the SCADE-based processes. However, note how the "High Level Software Requirements" and "Low Level Software Requirements and Architecture" lifecycle data items have been collapsed into a single entity "Software Requirements". This is a naive view of software development for all but the simplest of systems. For systems of even moderate complexity, an engineered architectural decomposition of the software is crucial for effective management, maintenance and integration. SCADE is not a software architecture development or management tool and supports only the "box and line" functional decomposition of the software in a similar manner to Simulink. Similarly, for more complex systems, a single requirements/design layer between the System Requirements Allocated to Software and the Source Code may not be sufficient to represent the levels of requirements and design decomposition needed to fully specify and realise the software system.

Product line development adds extra complexity into the software process that requires a richer and more sophisticated architectural view than is embodied in tools such as SCADE and Simulink. Their simple approach to functional decomposition is inadequate to cope with the complexities of commonality and variability modelling and product instantiation required to support true product line development.

2.5.3 Model Analysis Techniques

The construction of a model of software system function and structure will only bring significant benefit if it enables the verification and validation of system properties prior to system realisation. This relies on the development of effective analysis techniques to demonstrate the presence (or absence) of defined properties in the model (and then ensuring those properties are preserved in any subsequent translations; this is discussed in detail later in this thesis). In this section, we review the relevant literature regarding model analysis techniques.

One approach to model-level analysis would be to utilise static analysis techniques currently employed on "traditional" programming languages, but raise the level of abstraction of the analysed artefact. One of the most effective applications of static analysis for high-integrity software development is in the SPARK language and associated toolset [14, 74, 75].

The following describes the fundamental requirements used when originally defining the SPARK language:

"*We are mainly concerned with software to perform system control functions. The integrity of the software is vital: it must be verifiable. We can assume that the programs are to be developed by professionals, supported by whatever tools are available, and that if necessary substantial resources will be expended in achieving high integrity of software prior to its application; but the problems involved in proving its fitness of purpose must be tractable, in practical terms.*"[14]

These principles could equally be applied to models:

- the integrity of the **model** is vital: it must be verifiable;
- if necessary substantial resources will be expended in achieving high-integrity of **model** prior to its *realisation* (although it should be an aim to minimise the cost of this)
- the problems involved in proving its fitness of purpose must be tractable

In the following section we look at the attempts to reconcile the SPARK approach with industry's desire to use the UML as a software development approach.

2.5.3.1 SPARK and UML

The UML summary describes the language as follows : "*The Unified Modeling Language is a language for specifying, constructing, visualizing, and documenting artifacts of a software-intensive system*"[76] Note that the use of the model for the analysis, validation or verification of a software-intensive system is not mentioned[1]. Amey and White [77] describe an approach which attempts to combine the benefits of UML to describe a software system architecture with SPARK as a principled implementation language for high-integrity systems. The approach taken is to essentially overlay UML with SPARK semantics; UML class diagrams become graphical representations of SPARK packages, and a UML SPARK profile has been defined to hold information required in SPARK that has no analogous concept in the UML (for example the information flow contract on a class operation).

"*The semantic precision of SPARK has a significant impact on both the construction of the UML model of the system to be developed and on the verification of the code generated from it*."[77]

The approach attempts to combine the INFORMED[78] approach to developing SPARK systems with the use of UML class diagrams to capture and illustrate the resultant design. There is no attempt to raise the analysis performed to the model level, however. A code generator is used to produce SPARK compliant code from the annotated UML class model; the code generated is a structurally sound, annotated SPARK program minus the behavioural code (i.e. the code between the *begin* and *end* statements). The SPARK Examiner toolset is then used to determine if the program structure is well formed with respect to the information flow contracts. As the program behaviour is developed, the conformance with the information flow (and pre and post-conditions if provided) is repeatedly checked via analysis. This strong mapping between UML model and code level semantics, and "early and often" use of the static analysis tools provides an effective method of ensuring the conformance of the modelled system with a set of predefined properties. However, this is only achieved by targeting a particular implementation language technology and reflecting its semantics onto the model. This approach does provide an indication that effective model analysis is possible, and it is easy to envisage an approach that verifies the model directly rather than via a code-generation step.

[1] It could be argued that visualization is a weak form of verification – "it doesn't look quite right"

A mapping of UML to SPARK has also been undertaken by Sautejeu, who claims that there is significant advantage to be gained by combination of the benefits of UML (e.g. visual expressiveness, ability to transform or generate from its associated models) with the qualities of SPARK (e.g. built-in static-analysis capabilities, cost-saving via early error detection) [79].

Sautejeu does not provide much greater technical insight over Amey and White (apart from demonstrating a mapping can be captured in iLogix Rhapsody (now IBM Rational Rhapsody) [80] in addition to the ARTiSAN Studio example of Amey and White). However the paper makes interesting comments regarding the "required" evolution of UML for high-integrity systems : *"...some evolutions of the UML are needed to integrate requirements derived from information-flow analysis", "…the high integrity aspects of systems on which SPARK focus should now be an essential part of UML models"*[79].

2.5.3.2 SPARK and SCADE

An interesting approach is outlined by Amey and Dion [81] who discuss the benefits that may be obtained in combining a SCADE and SPARK approach to high-integrity software development. They claim that the formal underpinnings of both tools make the approaches complementary, and that the combination can address the *"overall software development challenge".*

The paper recognises that complete software systems cannot be built using tools such as SCADE in isolation; real systems have to interface with hardware devices, whereas SCADE models and derived code exist within a boundary of hand written interfaces/drivers, whose integrity has to be established by means other than the SCADE-recommended process. The combination of a principled model-driven development tool such as SCADE with a target implementation language that is semantically well defined such as SPARK enables this "glue" code to be analysed to the same degree of rigour as the model-driven development. Also the problem of property-preserving transformation of the model can possibly be addressed by this technique: "*The generation of source code in an unambiguous notation reduces the possibility of the semantics of the model and the semantics of the generated code differing and therefore increases the value of any model-based verification that has been carried out".*

However the approach advocated in [81] is missing any recognition of architecture; due to the affiliations of the authors, the paper naturally focuses on SCADE as the MDD tool of choice and SPARK as the implementation language. We have already seen [82] that SPARK systems can be developed successfully from a profiled UML architectural design, and that this early focus on architecture is extremely useful in ensuring the effectiveness of static analysis. We have also seen that SCADE's computational model makes it difficult to integrate into a wider architectural design without regarding the SCADE design as a sub-system with well-defined input/output (IO) and run-time behaviour. To summarise, however, Amey and Dion have identified that the combination of a principled MDD development approach in conjunction with a well-defined target implementation language can be an extremely effective technology for high-integrity real-time systems; this provides a basis for further development, particularly in the integration of principled architectural design.

2.5.4 Using OCL for model analysis

There is a significant level of research regarding the specification of model constraints using the Object Constraint Language (OCL) [83] and the validation of UML models against these constraint sets. The predominant use of this technique appears to be in demonstrating that a particular model is *well-formed* against a given criteria set rather than demonstrating any form of correctness with respect to a higher level specification [84]. Whilst demonstrating that a model is well-formed is necessary, there are other model properties that should also be validated.

There has been work in using simple UML simulators for executing and "snapshotting" model states, then validating that the OCL constraints hold for the "snapshotted" instances (Richters [85]). However, this appears to be a very simplistic technique that has the same drawbacks as testing (i.e. verification coverage is only for the snapshot points not the general case).

The use of OCL to support "design by contract" approaches [86], where invariants, pre-conditions and post-conditions are defined for classes and operations in the model, has been of interest in the research community for a number of years.

"*Making certain that the invariants, pre-conditions, and post-conditions have been defined in the model almost always improves the software development effort dramatically."* [87] (Note the use of "*have been defined in the model*" rather than "*have been defined and shown to hold true in the model*"*.*)

Although providing operation contracts has been seen as a "good thing" , there has been little uptake of this technology within industry or by commercial tool vendors, as indicated by Amey and White [77] in their paper on the integration of SPARK and UML: "*The ability to express a contract for an operation at the UML level is limited, by most tools, to expressing a type and parameter signature for it. In principle we could strengthen the contract using the Object Constraint Language (OCL) but this is neither well supported by tools nor sufficiently well-defined for our purposes.* " [77]

The SPARK language supports a level of design-by contract, utilising the Ada specification and body separation mechanism to separate the contract (specification) from the implementation (body) and allowing contracts at various levels of rigour to be specified for the SPARK sub-programs (data flow, information flow, formal pre and post-conditions).

There has been work on extending OCL to enable the specification and analysis of model properties such as real-time constraints (Flake & Mueller [88]), particularly for state-oriented UML diagrams. This would appear to show that the concept is extensible to cover more of properties of interest in a high-integrity real-time domain.

The definition of an action language that is rich enough to provide domain-specific representations (and is semantically mapped to an underlying meta-model that also supports a constraint language rich enough to enable full design-by-contract assertions) would provide a sound basis for the application of model analysis (and possibly proof) at a higher level of abstraction than is possible using present "industrial strength" approaches (i.e. at language level e.g. SPARK).

2.6 Product Line Development of High-Integrity Systems

In this section, we review the literature regarding the development of product lines for high-integrity systems. Here we start to see smaller numbers of published papers in the literature as the domain becomes more specialised.

2.6.1 Are Reuse and Dependability Mutually Exclusive?

In their discussion of practical and safe software reuse, Leveson and Weiss [89] quote a number of high profile examples of where inappropriate reuse has resulted in mission failure or, in their words, "spectacular losses". The question posed is whether it is possible to get benefit from software reuse "without the drawbacks".

Leveson and Weiss [89] discuss a number of requirements for effective software reuse; these are summarised below (labelled for cross-reference purposes later in this thesis):

LW1. Documentation of design rationale.

LW2. Documentation of the assumptions about the operational environment implicit in the software.

LW3. Bi-Directional Traceability from high level system requirements through the design process to code.

LW4. Documentation of hazard analysis and safety information.

Essentially the message of the paper is that reuse needs to start at the requirements level; reuse of code is neither useful nor demonstrably safe.

We can compare this view of reuse from a dependability viewpoint (that inappropriate or badly managed reuse is positively dangerous) with Bosch's view [1] from a commercial viewpoint (that unplanned or opportunistic reuse is not economically justifiable). Bosch's solution to this problem was to apply product lines; a managed approach to planned reuse. We need to determine if this approach may be augmented with any lessons from badly-reused mission and high-integrity software to allow a Trusted Product Line approach to be defined

2.6.2 Regulatory Constraints & Reusable Software

In 2004, the Federal Aviation Administration (FAA) published an Advisory Circular AC20-148 on "Reusable Software Components (RSC)" [90]. The motivation for this AC was primarily that applicants were wishing to include third-party components in their software systems. Components such as Real-Time Operating Systems (RTOS) from third-party vendors, possibly including communication protocol components for example, were being included in products requiring regulatory approval. Although it was not produced as a response to a software product line initiative directly, the guidelines provided in the AC are a useful insight into the regulator's view on component reuse in general.

Essentially, the AC identifies two sets of guidelines: those applicable to the component developer and those applicable to the user of the component. This is analogous to the domain and application engineering distinctions [2] in product-line development. The main guidelines provided in AC20-148 are listed below (labelled for cross-reference purposes later in this thesis):

The guidelines for the RSC developer include:

AC-D1. Produce a Plan for Aspects of Software Certification (PSAC) for the RSC.

AC-D2. Address known issues with software reuse as identified in the AC.

AC-D3. List any information that is preliminary or unknown at the time of component development (e.g. anything that is target specific or system specific).

AC-D4. List any assumptions made on the use of the component (e.g. compiler settings).

AC-D5. Produce an analysis of any behaviour that could adversely affect the user's system (e.g. partitioning requirements).

AC-D6. Comply with the stated PSAC/Plans during component development.

AC-D7. Submit a configuration index (SCI) and accomplishment summary (SAS) for the component through the applicant.

In addition, the RSC developer must supply the following data to the RSC user:

AC-I1. Interface description data describing how to integrate the component both functionally and temporally.

AC-I2. Integration and/or installation procedures.

AC-I3. Data to support the user's completion of any partially satisfied/unsatisfied objectives.

AC-I4. Verification results, cases and procedures, particularly for those activities that need to be repeated on the integrated system installed on the target computer.

AC-I5. Identification of any verification data affected by configurable parts of the RSC ("settable parameters").

The guidelines for the RSC user include:

AC-U1. Integrate the RSC lifecycle data into that supporting the overall product (including plan-set, PSAC etc.).

AC-U2. Evaluate the impact of any issues listed in the RSC data on the overall system.

AC-U3. Propose risk mitigation to address any risks identified with the component.

AC-U4. Validate that any assumptions made in the RSC SAS hold in the integrated application.

AC-U5. Evaluate the common reuse issues for the integrated application.

AC-U6. Report in-service problems with the RSC to the RSC developer.

AC-U7. Investigate any in-service issues with the RSC (if the RSC has been used previously).

AC-U8. Establish a legal agreement with the RSC developer about continued airworthiness support.

The AC lists the following as areas in which "common software reuse issues" can manifest themselves:

AC-R1. Requirements Definition.

AC-R2. Re-verification.

AC-R3. Interface Issues.

AC-R4. Partitioning and Protection.

AC-R5. Data and Control Coupling.

AC-R6. Use of Qualified Tools.

AC-R7. Deactivated Code.

AC-R8. Traceability.

AC-R9. Robustness.

Although written from the perspective of a "pre-certified" software component from a third party vendor, many of these issues and guidelines are applicable to development of a product line, and it is likely that any regulatory audit of a product line development would use these guidelines as a checklist for regulatory compliance in the first instance.

Habli et al. [91] discussed the challenges of producing a product line for a civil avionics system that was subject to regulatory approval. The paper concentrates on the areas it claims are underestimated in the product line lifecycle – configuration management and certification. In their treatment of certification, the approach taken is very similar to that recommended by AC20-148, providing much of the plan set and lifecycle data for the product line components themselves, and only requiring the user projects to produce integration data. To achieve this, there have to be compromises in the software architecture; the design rule is that any part developed as part of the product line "should be composed of large-scale reusable artefacts i.e. not fine grained in order to reduce integration and testing effort."[91]. This is a practical example of an issue that is fundamental to any software product line that requires significant product verification evidence. There is a tension between the provision of highly variable components to enable a flexible product line that can instantiate a wide range of products, and the provision of pre-verified components to reduce the overall verification costs but restrict the range of products that can be instantiated. The approach we take is distinct, in that it provides for much finer-grained components.

Dordowsky et al. [92, 93] discuss the development of a software product line for military helicopter systems. They make many of the same observations as Habli et al., in that they dismiss SPL approaches that support source code modification based on feature selection, as this would require significant consequential verification effort, OR the tool performing the selection would need to be qualified. Their approach to variability is to implement features within separate code component, and they allow a small amount of run-time variability. This approach is viable in their particular instance as they have a very tightly scoped product line (i.e. known variants of the NH90 helicopter). They do not appear to have the need to instantiate "in-scope but unknown" product variants that would require finer-grained variability.

Boeing has long been interested in product lines, for example Sharp [94], but there are relatively few recent publications. Sharp [94] describes an approach to software component reuse, identifying the importance of a layered architecture to introduce abstraction and separation of concerns, and discusses a component model that enables late binding to target processor and hardware. However, this work appears to be at the conceptual level and there is only a passing mention in the paper's introduction of the flight test of a system developed using this approach. It is unclear whether a system has ever been approved/certified using this approach.

2.6.3 Verification of Software Product Lines

One of the most widely cited references on verification of software product lines is by McGregor [95]. This provides an overview of available testing techniques (mostly from single-system development processes) but provides little insight into the problem of balancing variability and verifiability. Indeed, at the start of a discussion on the testing of product line assets, the following observation is made: "*The number of variation points and possible values for each variation make the testing of all possible products that can be built from the product line impossible. This makes it imperative that products be composed of high-quality components*". We can only assume that the "high quality components" themselves do not contain variability (c.f. the observations of Habli et al. [91]). If we assume that components themselves are variable then the testing effort that contributes to the determination of "high quality" becomes commensurately more difficult and expensive.

It is intuitively attractive to carry out verification on the product line assets, because any use is then "verified by construction". However, as intimated by McGregor, there may be many tens or even hundreds of possible configurations of even a modest-sized component and, further, instantiation processes such as transformation, may add to the code, e.g. providing interfaces, as well as making selections, so it is not clear how representative asset-level testing evidence will be of the end product. In addition, for DO-178B/ED-12B Level A, Modified Condition/Decision Coverage (MC/DC) has to be achieved at object code level, so it is hard to escape the need to do (at least) coverage anaysis on the final product. Thus, there is a difficult trade to be made about the cost-effectiveness of pre-instantiation verification for high-integrity product lines.

Lutz [96] produced a survey of product-line verification and validation techniques for NASA (in 2007) as a deliverable that formed part of a research project on the "Product Line Verification of Safety-Critical Systems". Whilst this survey identified some useful techniques and references, it has an emphasis on verifying "conformance" to product-line requirements and architectures. This is made clear in the introduction where one of the questions posed is *"How should we verify that delivered software conforms to the product-line requirements and architecture levied on it and how do we document that conformance?*" This is further emphasised later where "*Verification that the software for each project satisfies its intended product-line constraints is essential*" as conformance *"will make or break the product-line approach"*.

The focus therefore appears to be demonstration that the product is indeed a valid member of the product line, rather than provide evidence that the product meets its specified requirements. This can be a problem when there is discontinuity between the product-line specification/architecture and the development of the product instance itself. For NASA this is a problem of ensuring that contractors provide systems that are compliant to a given product line specification. Here the specification of the product line is descriptive rather than prescriptive and therefore conformance has to be demonstrated rather than arguably being a natural consequence of the production process, as should be the case for prescriptive product-line architectures that make use of models and transformation to instantiate product. This should provide "conformance by construction" to the product line, and is an objective for the approach we propose in this thesis.

2.6.4 Formal Analysis of Product Lines

There has been recent work published (2011) on the use of formal techniques to establish given properties of product lines as distinct from single systems. These approaches attempt to adapt techniques such as model checking to analyse systems that contain variability, and address the resulting state explosion problem.

Classen et al. [97] propose a method for symbolic model checking, for example, temporal properties of product lines. Whilst their approach appears to make the problem tractable for the examples they provide, this was for a canonical elevator system with 9 independent feaures (yielding 2^9 enumerated products). Their approach requires the product line to be described using a language *fSMV* based on the input language of the model checker NuSMV. Whilst it is clear how simple features and changes can be modelled in fSMV, it is unclear how complex feature to solution interactions could be modelled. This approach looks promising, however, and as the work matures, it would be interesting to apply to variable components in the first instance. However the work to date is too immature and was published too late to be of direct relevance to our research.

Similarly, the work of Apel et al. [98] discusses how undesirable feature interactions can be detected via the use of model checking. This relies on a formal specification of the behaviour and constraints of each feature to be constructed. It also appears to impose architectural constraints on the construction of the program (“*we implement and specify features in separate and composable units*”[98]). As with Classen et al. the work is promising and may be a useful technology in the future; however it appears too immature for large, existing systems and imposes too many constraints in the form of the specification and the architectural decomposition of the solution to be directly relevant to our research.

2.7 Product Lines, Models and High-Integrity Systems

In this chapter, we have discussed both the development of software product lines and the development of high-integrity software, including a review of the use of model-based approaches in both domains. In this final section, we bring the two domains together and provide a review of the literature concerning the model-based development of product lines for high-integrity systems.

The current literature is very sparse on this specific set of topics. Trujillo et al. [99] attempted to “foster a discussion” on the issues faced in applying model-based product line development for dependable systems. The challenges identified in [99] that are supported elsewhere in the literature include :

1. Certification – Potential incompatibility between the requirements of regulation and Product Line approaches [91]
2. Modelling Safety Information – Providing a meta-model that allows safety analysis and assessment information to be held with the product line components [100, 101]
3. Model Transformation for Product Instantiation – How can product assurance arguments be supported when products have been instantiated using model transformation? [50, 102]
4. Verification and Test – This is not particularly well articulated in [99], however there is a significant challenge of providing cost-effective verification for high-integrity software product lines [103]

In addition to the challenges that are supported elsewhere in the literature, the authors of [99] identify additional challenges, based on their experience (these are listed below, using their terminology) :

5. Multi-disciplinary nature of the task
6. System complexity
7. Customisation across multiple domains
8. Reusability of elements within a system

9. “Explicitation” (sic) of the process – it would appear that this challenge is essentially the introduction of new and unfamiliar processes such as domain/application engineering, model transformation etc.
10. Distributed and collaborative teams
11. Increased rate of non-functional requirements
12. Impact of model-based product line engineering on the safety-oriented design

It can be argued that points 5, 6, 10 and 11 are not challenges that are specific to product line development; however, it is certainly true that the challenges are not reduced in any way when applying product line approaches. Challenges 1, 4, 7 and 8 are a consequence of the potential mismatch between the technical approaches advocated by the product-lines community and the regulatory requirements of high-integrity systems (as discussed in the previous section).

Only the remaining challenges (2, 3, 9 and 12) are a true result of the application of model-based techniques to high-integrity product lines. Of these, challenge 3 is of most interest in the context of this thesis; the fact that it is framed as an unsolved problem in a paper published in 2010 demonstrates that the problem is real and of concern. We articulate our own set of challenges later in this thesis.

2.8 Summary

The study of high-integrity software product lines includes a number of associated but different research areas. Our interest in the use of model-based techniques widens the area of study even further. However, whilst there is a wide body of literature in each of the related research domains, there is little published on the specific application of product line techniques to high-integrity system and software development, particularly for regulated domains that require the system to be certified or approved. In a recent (2012) paper, Braga et al. [104] recognise the issues regarding certification of product lines, and comment that approaches are “*beginning to emerge to support SPL certification*”. They then proceed to reference our work [50, 105].

Whilst the literature contains a number of examples of successful high-integrity product lines [91-93], they have all constrained the solution space, particularly with respect to variability. There appears to be little work (except our own) on the successful application of fine-grained variability to certified high-integrity software development.

Interestingly, 2012 saw the launch of the VARIES project under the Artemis framework, whose goal is to “*deliver a platform to help Embedded Systems developers to maximise the full potential of variability in safety critical embedded systems. The focus will be on the safety critical aspects, in particular the impact of reuse and composition on certification.*” [106]. The declared project duration is from May 2012 to April 2015, and includes a large consortium of tool vendors and academics (including pure::systems and Atego). It is clear, therefore, that the problem of using software product lines approaches for high-integrity systems is relevant, real and non-trivial.

3 Trusted Product Lines in Context

To gain a full understanding of the implications of developing a Trusted Product Line, the product context needs to be taken into consideration. This context can have a major effect on the product line approach used, in terms of development process and product realisation/instantiation, due to factors such as regulatory requirements, certification processes, development practices and customer expectations.

The research described in this thesis concentrates on the development and analysis of a software product line for Full Authority Digital Engine Control (FADEC) systems. Such systems are deployed on aircraft gas turbine engines; this particular research concentrates on FADEC systems aimed primarily at the large civil aerospace market. The chapter outlines the motivation for the research, including the business challenges that make FADEC development as a product line attractive, and the resultant technical, engineering and academic challenges that are a consequence of this business strategy.

The purpose of this chapter is to provide context for the research presented in this thesis. Whilst the author was involved in some of the work described here, this is not regarded as part of the thesis contribution. The information described in this section is based primarily on the background and experience of the author, who has worked for over 20 years developing software for FADEC systems.

3.1 Full Authority Digital Engine Control (FADEC) Systems

3.1.1 Role of a FADEC

The main purpose of a FADEC system is to control the gas turbine engine to provide a level of thrust as requested by the pilot and the aircraft systems. In addition, the FADEC controls the engine start and shutdown sequences and monitors engine performance to ensure it is operating efficiently and within safe limits. It also contains protection functions to shut-down the engine or reduce engine thrust when potentially hazardous conditions are detected, for example the mechanical failure of rotating shafts within the engine ("shaft break") causing the turbine stages to over-speed.

Figure 32 shows a typical FADEC system architecture in annotated block-diagram form. FADEC systems have control over a number of engine systems and parameters including the fuel-flow ❷, fuel shutoff, ignition system ❽, starting system ❼ and the variable parts of the engine airflow systems ❺. The FADEC is "full authority" in the sense that no backup or override systems are deployed for ensuring safe operation of the engine with respect to the controlled parameters. In addition, the FADEC can have partial control over the thrust reverser systems fitted to the aircraft.

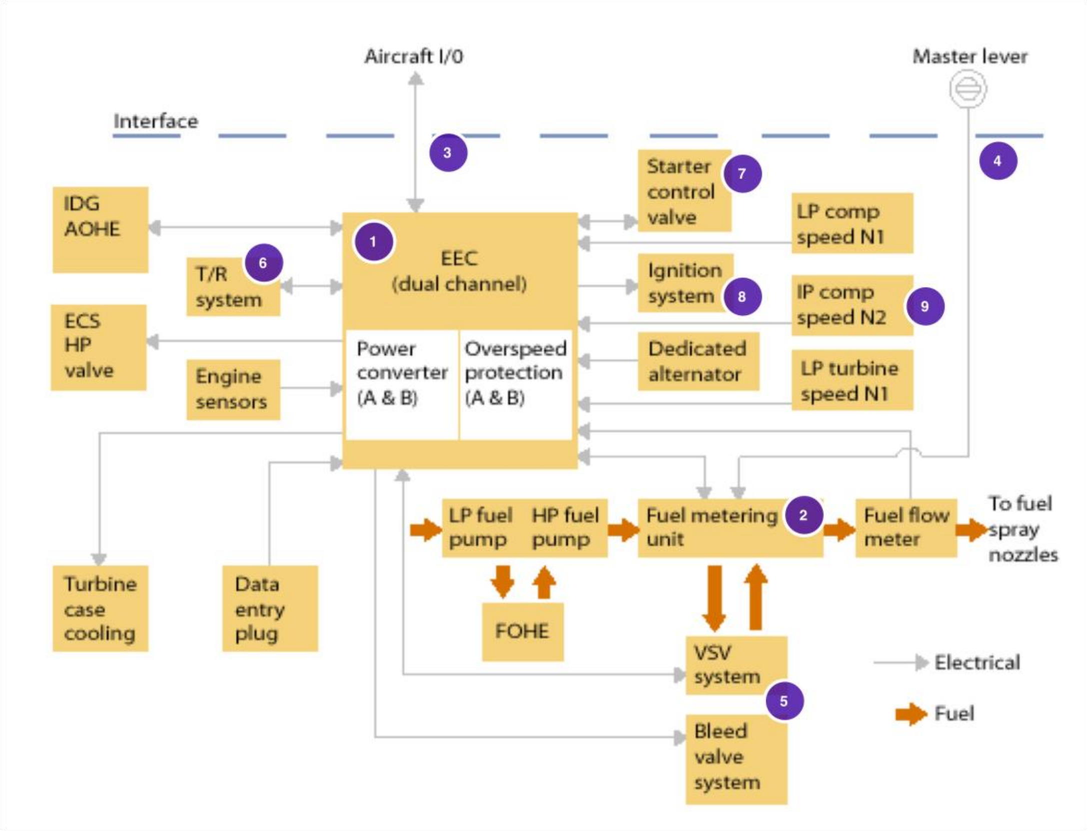


Figure 32 Generic FADEC Architecture (annotated from [107])

At the heart of the FADEC is the Engine Electronic Controller or EEC ➊. A typical EEC architecture consists of duplicate redundant channels, each capable of controlling the engine independently. In normal operation, the EEC is configured with one active channel and one standby channel. A channel change mechanism determines the "health" of the controlling channel and can instruct the standby channel to take control if required. The health of a channel may degrade due to failure of the internal components of the EEC, or loss of one or more of the sensors and actuators connected to that channel. Most engine parameters are measured using duplicated (duplex) sensors to provide independent measurements to each channel; some critical parameters may have three or more independent measurement sensors.

Each EEC channel contains one or more microprocessors. The increasing computation requirements of gas turbine control and the need to demonstrate separation between the protection functions and control functions for a number of the FADEC-related engine hazards is leading to multi-processor per channel architectures.

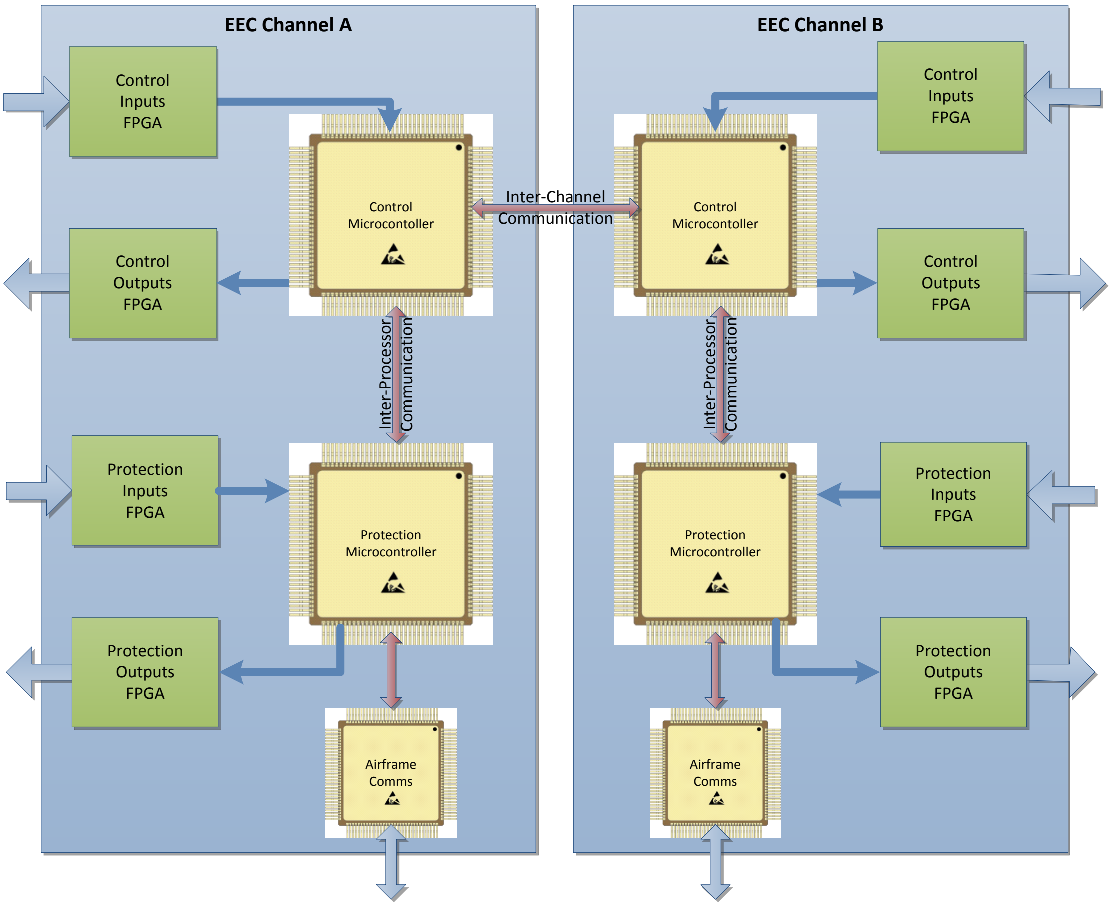


FIGURE 33 EEC INTERNALS BLOCK DIAGRAM (SIMPLIFIED)

Figure 33 illustrates the internal architecture of a modern EEC in simplified block diagram form. This architecture shows the separation of protection and control resources and the communication paths between processing resources and between channels of the EEC. This provides the hardware context into which the software is developed.

3.1.2 FADEC Software Development Programmes

Like many avionic systems, managing the development of FADEC systems and software is complicated by the need to produce a product (and associated evidence) that satisfies regulatory requirements whilst simultaneously producing interim development versions of the system to support the embedding system development, integration and verification to proceed. This is a particular challenge when the containing system is a complex machine such as an engine or aircraft where the primary verification and qualification mechanism is extensive and exhaustive development testing. Therefore, there is an over-riding customer need to deliver functional systems to allow engine and aircraft testing to proceed whilst the FADEC requirements themselves are immature.

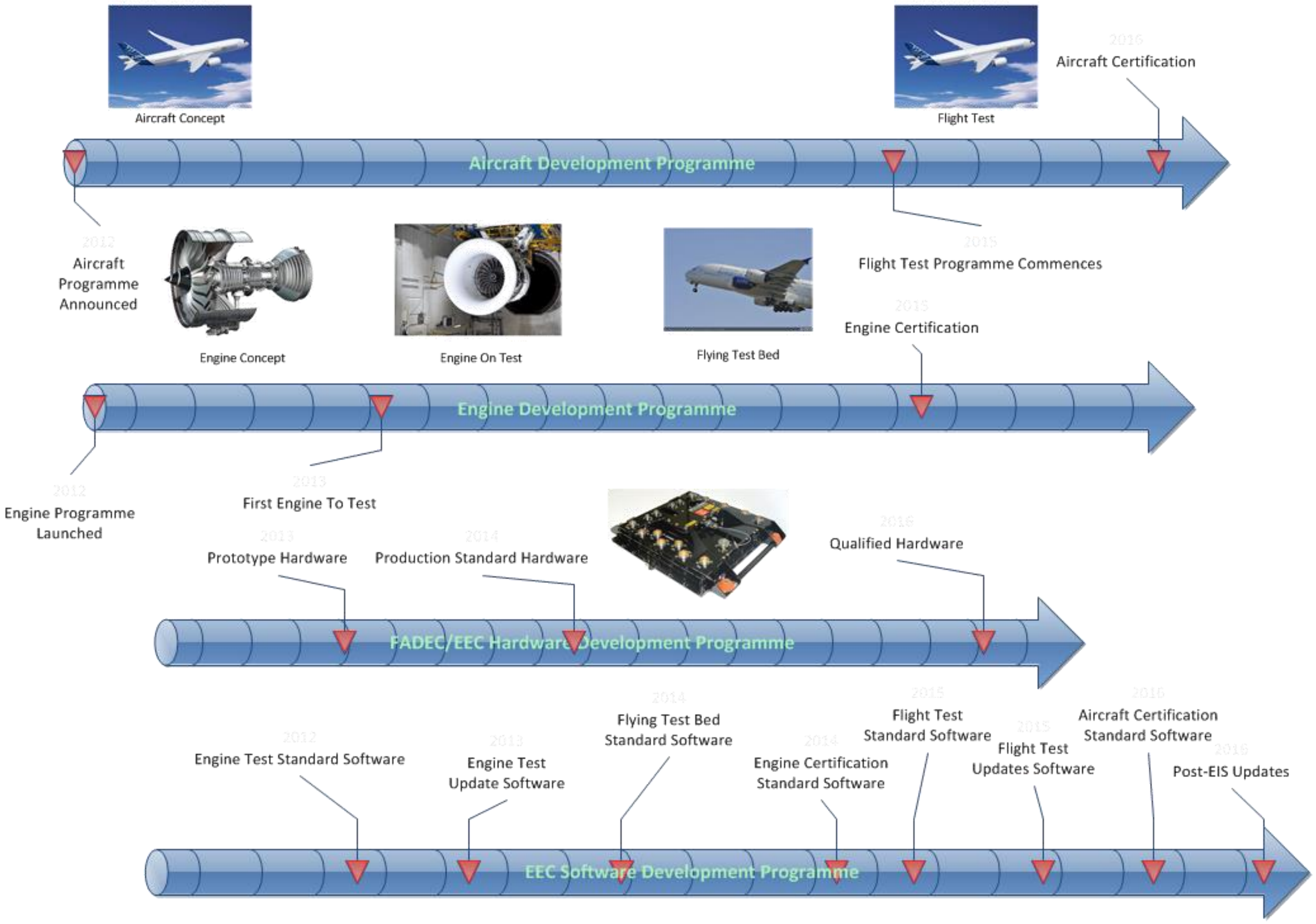

Aircraft Concept
Flight Test
2016
Aircraft Certification
Aircraft Development Programme
2012
Aircraft Programme Announced
2015
Flight Test Programme Commences
Engine Concept
Engine On Test
Flying Test Bed
2015
Engine Certification
Engine Development Programme
2012
Engine Programme Launched
2013
First Engine To Test
2013
Prototype Hardware
2014
Production Standard Hardware
2016
Qualified Hardware
FADEC/EEC Hardware Development Programme
2012
Engine Test Standard Software
2013
Engine Test Update Software
2014
Flying Test Bed Standard Software
2014
Engine Certification Standard Software
2015
Flight Test Standard Software
2015
Flight Test Updates Software
2016
Aircraft Certification Standard Software
2016
Post-EIS Updates
EEC Software Development Programme

Figure 34 Typical Phasing of Aircraft, Engine, FADEC and EEC Software Development Programmes

3.1.2.1 Engine & Aircraft Development

Figure 34 illustrates how the development programmes for the aircraft, engine and FADEC hardware influence the EEC software development programme and the software standards (deliveries) required to support the aircraft and engine system development and test programmes. Here we can see that the software development programme cannot be regarded as a simple "waterfall" of requirements elicitation, software development, verification and product delivery. Instead, between the launch of the software programme and the delivery of the approved, certified software product into service there are a number of interim software deliveries required to support the engine and aircraft development. Indeed the early software deliveries typically contain engine test features and "special functions" that allow special-to-test manoeuvres and operations to be performed on the engine that are not required in the delivered flight system.

There is an on-going process of identification of requirements for the software as the system evolves and the behaviour of the FADEC, engine and aircraft becomes known in more detail. The EEC software therefore can be regarded as both an enabler to the engine and aircraft development programme as well as a component part of the delivered system. Figure 34 also shows that the software development does not necessarily finish at aircraft certification. Typically Post-Entry Into Service (EIS) software builds are required to address issues found in service and provide additional features for particular aircraft operators (for example enhanced engine ratings for "hot" operation and routes including high-altitude airports.)

3.1.2.2 Software Approval for Certification – Planning Documentation

In addition to the planning of the software development schedule to deliver the full and interim software builds necessary to support the engine and aircraft development, a set of "planning documents" are required to define how the software programme will comply with the applicable regulatory requirements. DO-178B/ED-12B [4] requires the following set of plans to be produced in support of a software programme that is subject to approval and certification :

1. Plan for Software Aspects of Certification (PSAC)
2. Software Development Plan (SDP)
3. Software Verification Plan (SVP)
4. Software Quality Assurance Plan (SQAP)
5. Software Configuration Management Plan (SCMP)
6. Tool Qualification Plan (TQP)

Together, this set of planning documents provides the software project's intended means of compliance to the objectives of DO-178B/ED-12B. They are the main means of communication with the software development organisation ("the applicant") and the regulatory body/ aviation authority who approve the system and software for use ("the regulator"). They provide the definition of the software development, verification and management processes used on the project, and should identify the lower-level standards and procedures that govern the day-to-day activities undertaken by the development staff.

3.1.2.3 Software Approval for Certification - Stages of Involvement (SOI) Audits

The approval of software as part of an engine or aircraft certification involves the regulator conducting a series of “Stages of Involvement” (or SOI) audits [108]. The SOI audits allow the regulator to inspect the state of the software development programme (and the artefacts produced to date) to determine the robustness of the software product design and the compliance of the software programme with the objectives of DO-178B/ED-12B. Failure of an SOI audit can lead to significant levels of redesign and re-verification by the development organisation, with consequential programme timescale slip and cost overruns.

The set of SOI audits required during a software development programme are defined in [108], which contains the following summary table (Note that references to the FAA in the original have been replaced here by the term “the regulator”) :

TABLE 3 OVERVIEW OF REGULATOR STAGES OF INVOLVEMENT (FROM [108])

SOI	Description	Data Reviewed	Related DO-178B Table
1	**Planning Review** • Assure plans and standards meet DO-178B objectives and address other applicable software policy, guidance, and issue papers. • Assure that the processes described in the applicant’s plans meet the objectives of DO-178B and address other applicable software policy, guidance, and issue papers. • Obtain agreement between the regulator and applicant on the plans, standards, and proposed methods of compliance.	• Plan for Software Aspects of Certification (PSAC) • Software Verification Plan (SVP) •Software Development Plan (SDP) •Software Configuration Management Plan (SCMP) •Software Quality Assurance Plan (SQAP) •Software Development Standards (Requirements, Design, and Coding) • Safety assessment (preliminary system safety assessment (PSSA) or system safety assessment (SSA)) • Tool Qualification Plans, if applicable • Other applicable company	A-1, A-8, A-9, A-10

SOI	Description	Data Reviewed	Related DO-178B Table
		policy, procedures, and standards • System requirements (may be preliminary) and interface specifications • Description of any new technology or novel methods (typically contained in the plans)	
2	**Development Review** • Assess implementation of plans and standards for the software requirements, design, and code, and related verification, SQA, and SCM data. • Assess and agree to plans and standards changes. • Assess implementation of new technology and methods to ensure compliance to plans, standards, and agreements. • Assure life cycle data satisfies DO-178B objectives and other applicable software policy, guidance, and issue papers.	• Software Development Standards (Requirements, Design, and Coding) • Software Requirements Data • Design Description • Source Code • Software Verification Results (as applied to Tables A-2 to A-5) • Problem Reports • Software Configuration Management Records • Software Quality Assurance Records • Tool Qualification Data, if applicable • Resolution of previous review findings, if applicable	A-2, A-3, A-4, A-5, A-8, A-9, A-10
3	**Verification Review** • Assess implementation of verification and test plans	• Software Requirements Data	A-2, A-6, A-7, A-8, A-9, A-10

SOI	Description	Data Reviewed	Related DO-178B Table
	and procedures. • Assess completion and compliance of all associated SCM and SQA tasks. • Ensure software requirements are verified. • Ensure robustness testing is planned and is being performed. • Ensure analyses (including timing, memory, test coverage, structural coverage, and data and control coupling) are being performed, as required by DO-178B. • Ensure verification activities satisfy DO-178B objectives.	• Design Description • Source Code • Software Verification Cases and Procedures • Software Verification Results (including review results, analyses results, and test results) • Problem Reports • Software Configuration Management Records • Software Quality Assurance Records • Resolution of previous review(s) findings, if applicable	
4	**Final Review** • Assure final software product meets DO-178B objectives and is ready for certification. • Address any open items.	• Software Conformity Review Results • Software Life Cycle Environment Configuration Index • Software Verification Results (final test, analyses, and review results) • Software Configuration Index • Problem Reports • Software Accomplishment Summary • Final resolution of all	All

SOI	Description	Data Reviewed	Related DO-178B Table
		previous review findings and issues	

The need to support SOI audits with accurate and applicable information is a theme we will be returning to later in this thesis when we discuss the certification of products instantiated from a product line.

3.2 A History of Reuse in FADEC Systems

Over the past 25 years of engine control system and software development there have been many attempts at providing value to the business from reuse. Figure 37 below provides a “timeline” of reuse initiatives for engine control systems and software from the 1980s to the present day. This is elaborated on below:

3.2.1 Low Level Code Reuse

The early generations of software-based digital engine control systems, such as the FAFC (Full Authority Fuel Control) systems and the first FADEC systems, were developed using a technique called “threaded code”. This approach was introduced in the very first experimental FADEC systems, such as those trialled on Concorde in the early 1970s [109]. Languages such as LUCOL [110] were developed to use this approach, where programs are built from sequences of “modules” which call each other in turn and provide the flow of control through the program.

The component parts of threaded code languages (e.g. LUCOL “modules”) provide well-defined operations to perform specific tasks, and in the case of LUCOL the behaviour of these modules were formally proven [110] against their specification. This type of low-level code reuse has a number of parallels with modern approaches such as Domain Specific Languages (DSLs). LUCOL can be regarded as a simple DSL for engine control constructed from reusable, domain-aligned primitives.

3.2.2 Reuse Libraries

The early 1990’s saw a move away from specific engine control languages towards general purpose “Third Generation Languages” (3GLs) for the development of FADEC software. The size, complexity and functional breadth of engine control systems in particular and avionics software in general was increasing (on a trend that has not stopped to date – see Figure 35, Figure 36). This posed a problem for domain specific languages such as LUCOL in that the domain of interest was growing. FADEC systems were not now just targeted at closed-loop control of gas turbines - the complexity of the avionics/airframe interface was increasing, and an increasingly significant proportion of the FADEC functionality encompassed fault detection and accommodation. In addition, engine manufacturers started to recognise that the control system (in particular, the EEC software) was increasingly important for optimising the performance and function of the engine. Therefore, general-purpose languages such as Ada started to be used for FADEC software development, which provided a level of abstraction/decoupling from the proprietary hardware provided by EEC manufacturers.

The provision of "standard" function libraries was still regarded as a good design approach, however, and many of the functions that were encoded in LUCOL modules for previous generations of product were now provided as "utilities" or "reuse libraries". In addition, more complex utilities were provided to cater for what were regarded as "common" operations, such as the validation of simplex and duplex inputs for example.

This was regarded generally as a successful approach that utilized the flexibility of the 3GL but retained some of the LUCOL benefits of "standard" functions.

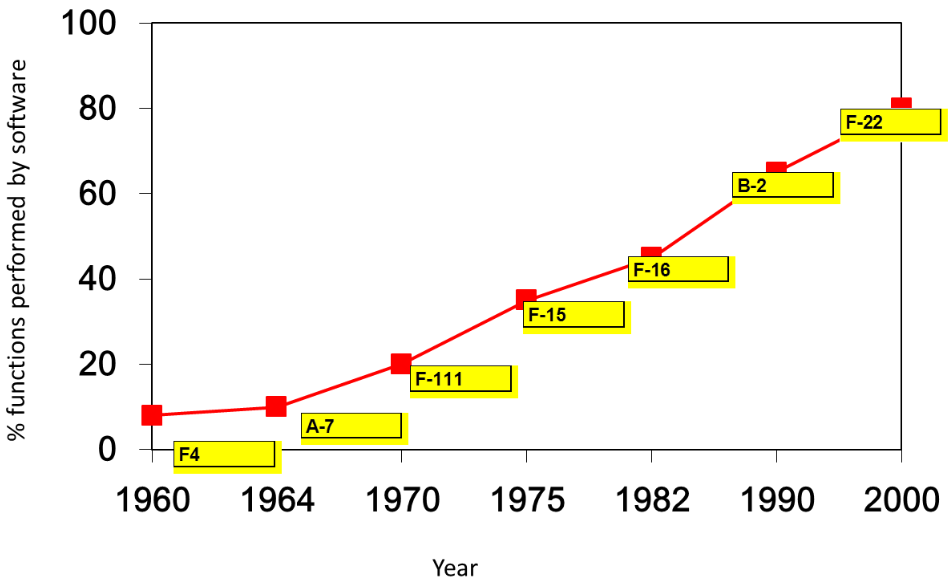


FIGURE 35 US AIRCRAFT SOFTWARE DEPENDENCE [111]

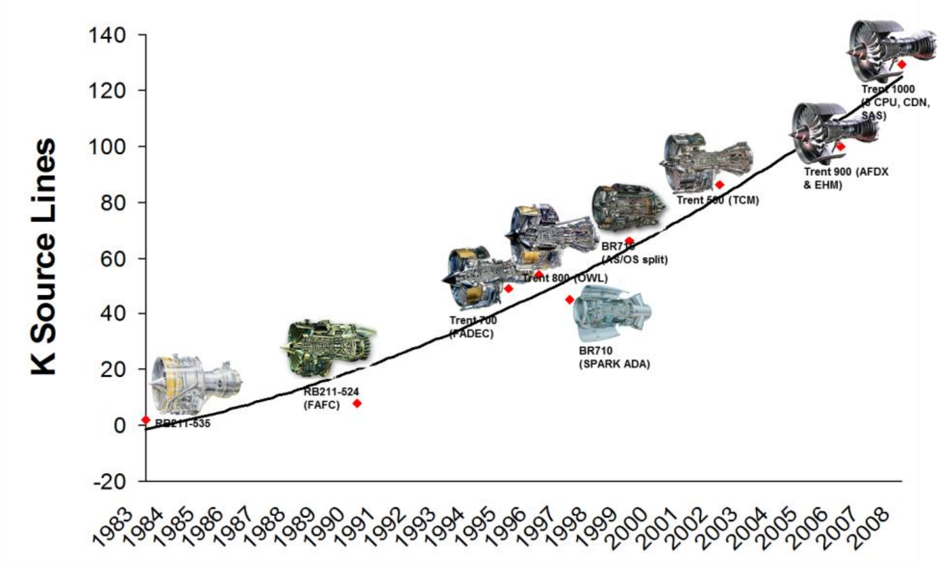


FIGURE 36 CODE SIZE GROWTH – CIVIL FADEC (INTERNAL COMPANY DATA)

3.2.3 Opportunistic Functional Reuse

By the mid-1990s, the number of FADEC development projects were increasing. This was to support new engine and aircraft development, plus the retrofitting of FADEC technology onto older engines to provide more efficient control and increased airframe avionics integration. This increasing level of system development required a corresponding increase in the development staff and resources. Management felt that there must be an approach to minimising the development resources required for these programmes by reusing software between the products so that the development time and effort was reduced.

As we have already discussed in Chapter 2, this opportunistic reuse approach did not provide the level of benefit originally envisaged, due to the problems noted by Bosch [1]. In this particular case, it was typical for the requirements of the donor project to diverge over time from the requirements of the recipient project. This led to a realisation that effective software reuse would only be possible when the requirements for the products were convergent.

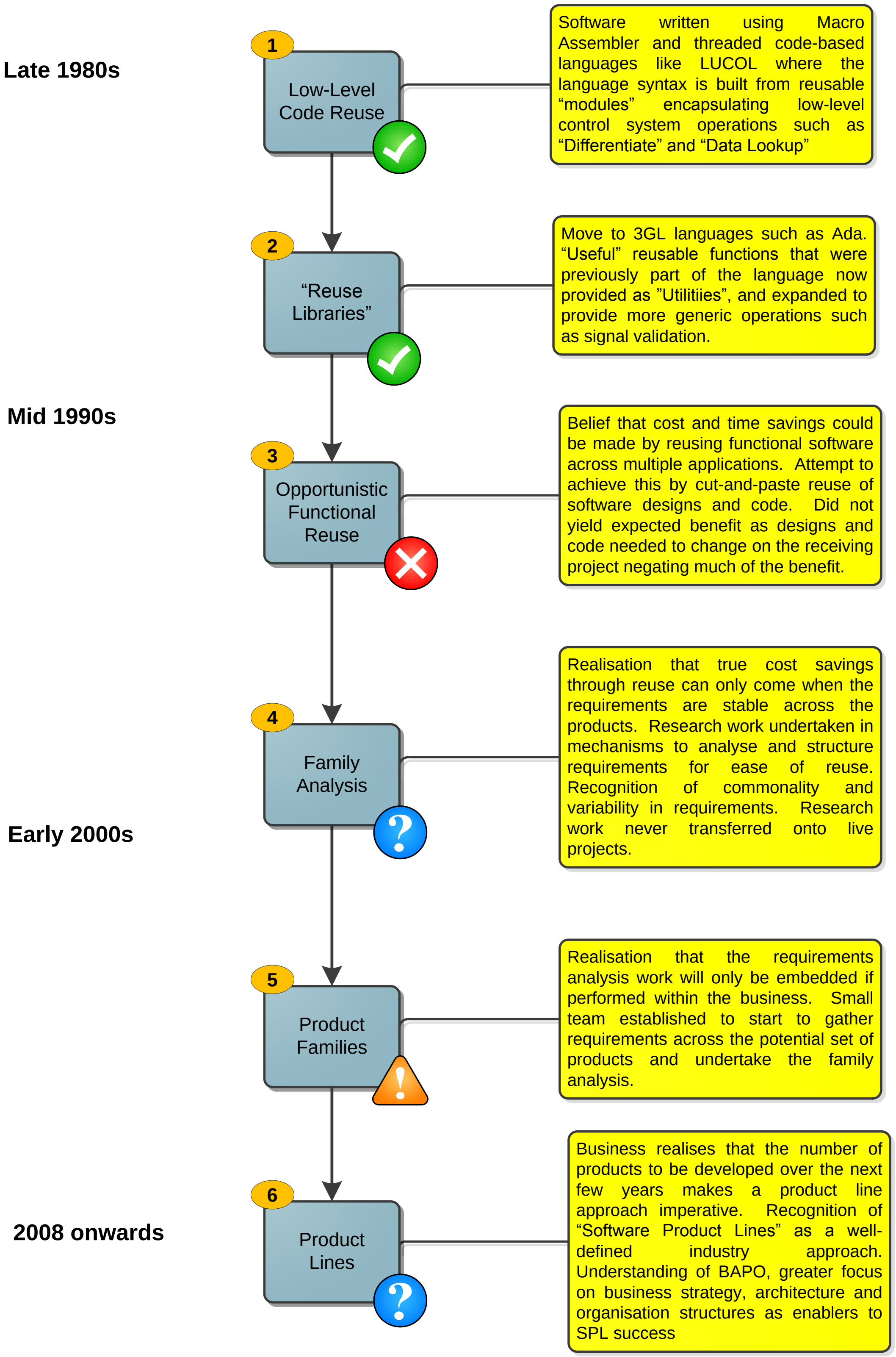
3.2.4 Family Analysis

The need to understand how requirements could be produced for a family of products led to a period of research activity in the late 1990s. Joint industrial/academic research on reusable requirements, requirements patterns and domain analysis for engine control systems was funded and progressed for a number of years. This work identified how to define family requirements through a systematic approach to requirements development and management [112, 113]. Whilst useful, this work was never adopted on live projects; problems of technology transfer from sponsored research into company working practice was a well-known issue at this time [114].

3.2.5 Product Families

The potential benefits of reuse in reducing the cost of system development were still attractive, and to this end, an internal “product families” team was established in the early 2000’s. The aim of this team was to perform the engine control domain analysis and gather family requirements in a similar manner to that recommended by the previous academic work documented in [112], but within the company to try and address the technology transfer issues.

Before this activity could deliver any meaningful results, it was overtaken by the Product Lines initiative described below.



Software written using Macro Assembler and threaded code-based languages like LUCOL where the language syntax is built from reusable "modules" encapsulating low-level control system operations such as "Differentiate" and "Data Lookup"

Move to 3GL languages such as Ada. "Useful" reusable functions that were previously part of the language now provided as "Utilitiies", and expanded to provide more generic operations such as signal validation.

Belief that cost and time savings could be made by reusing functional software across multiple applications. Attempt to achieve this by cut-and-paste reuse of software designs and code. Did not yield expected benefit as designs and code needed to change on the receiving project negating much of the benefit.

Realisation that true cost savings through reuse can only come when the requirements are stable across the products. Research work undertaken in mechanisms to analyse and structure requirements for ease of reuse. Recognition of commonality and variability in requirements. Research work never transferred onto live projects.

Realisation that the requirements analysis work will only be embedded if performed within the business. Small team established to start to gather requirements across the potential set of products and undertake the family analysis.

Business realises that the number of products to be developed over the next few years makes a product line approach imperative. Recognition of "Software Product Lines" as a well-defined industry approach. Understanding of BAPO, greater focus on business strategy, architecture and organisation structures as enablers to SPL success

Figure 37 Timeline of "Reuse" Initiatives for Engine Control Software from the 1980s onward

3.2.6 Product Lines

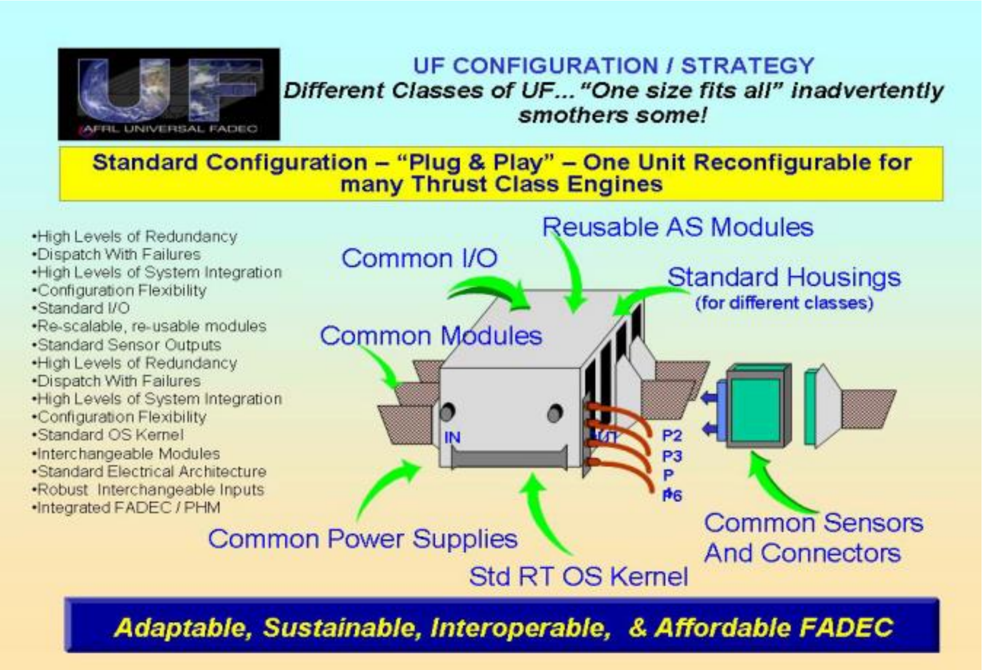
With the increasing demand for FADEC system developments in the late-2000s, it was realised that the business had to take reuse much more seriously if it was to be able to deliver the required systems on time and to budget. This realisation moved "product families" from a small-scale engineering initiative to a "Product Lines" business strategy. The characteristics of successful product line initiatives in other industries were studied, as was the more academic study of product lines as published by the SEI and others [3].

Organisation change to reflect domain engineering and application design activities was undertaken, and the company placed a greater emphasis on architecture as an enabler for product line delivery (as recommended by the BAPO model [16]).

3.3 Other FADECs & Reuse

The information described so far has been based on the author's own experience. There is some published material from other FADEC developers related to reuse programmes, but nothing of any great substance.

Behbahani [115] [116] discusses the need for a "Universal FADEC", and provides some ideas for how this may be achieved, but this is framed from a US Airforce customer viewpoint. Most of the discussion in [116] is posing challenges for the FADEC suppliers to meet, rather than provide any solutions to the technical challenges. Indeed, the problem that Behbahani discusses is that of FADEC obsolescence (primarily driven by electronic hardware), to which his solution is the provision of a generic "universal" FADEC that is applicable across engine and airframe types, with variability catered for via use of a modular "open architecture" (see Figure 38 and Figure 39).



UF CONFIGURATION / STRATEGY

Different Classes of UF… "One size fits all" inadvertently smothers some!

Standard Configuration – "Plug & Play" – One Unit Reconfigurable for many Thrust Class Engines

- High Levels of Redundancy
- Dispatch With Failures
- High Levels of System Integration
- Configuration Flexibility
- Standard I/O
- Re-scalable, re-usable modules
- Standard Sensor Outputs
- High Levels of Redundancy
- Dispatch With Failures
- High Levels of System Integration
- Configuration Flexibility
- Standard OS Kernel
- Interchangeable Modules
- Standard Electrical Architecture
- Robust Interchangeable Inputs
- Integrated FADEC / PHM

Reusable AS Modules

Common I/O

Standard Housings (for different classes)

Common Modules

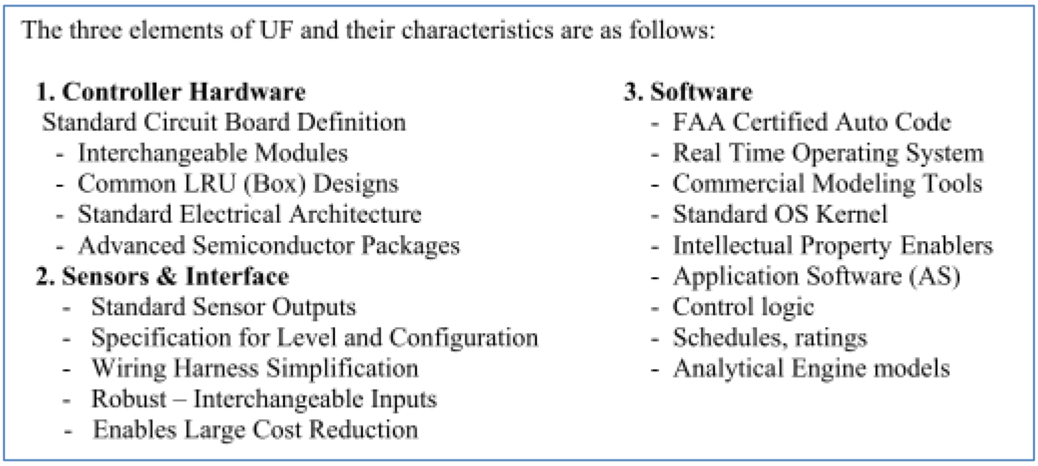
Common Power Supplies

Std RT OS Kernel

Common Sensors And Connectors

Adaptable, Sustainable, Interoperable, & Affordable FADEC

FIGURE 38 UNIVERSAL FADEC CONCEPT (FROM [116])



The three elements of UF and their characteristics are as follows:

1. Controller Hardware

Standard Circuit Board Definition

- Interchangeable Modules
- Common LRU (Box) Designs
- Standard Electrical Architecture
- Advanced Semiconductor Packages

2. Sensors & Interface

- Standard Sensor Outputs
- Specification for Level and Configuration
- Wiring Harness Simplification
- Robust – Interchangeable Inputs
- Enables Large Cost Reduction

3. Software

- FAA Certified Auto Code
- Real Time Operating System
- Commercial Modeling Tools
- Standard OS Kernel
- Intellectual Property Enablers
- Application Software (AS)
- Control logic
- Schedules, ratings
- Analytical Engine models

FIGURE 39 "THREE ELEMENTS OF UNIVERSAL FADEC (FROM[116])

3.4 Summary

The following points are key to the appreciation of the business and technical environment surrounding FADEC development:

- Products are developed within certification constraints, and are subject to scrutiny from regulatory authorities prior to deployment in service.
- The core engine control functionality of FADEC systems is relatively stable; however, additional functionality is causing growth of code size of approx. 7% per year.
- Any proposed advances in the development processes must address commercial as well as technical constraints
- Reuse is seen as a valid and desirable approach to reduce the cost and lead time of product development; however, the attempts to reuse software have met with limited success to date.

The research described within this thesis was both motivated by and undertaken in the business and technical context described above.

4 Defining a High-Integrity Product Line Model

We now have an understanding of product line theory and the constraints of high-integrity software development. In particular, we have an appreciation of the use of model-based software engineering approaches within high-integrity developments in general, and embedded real-time control systems in particular. We also recognise the current sparse state of the literature on high-integrity product line development. This chapter describes our approach to the definition of a product line model that allows the instantiation of products that can be approved to the guidance specified in DO-178B/ED-12B [4], as part of the certification of an airborne system (aircraft or engine).

4.1 Background

The author has experience of successfully using UML models as part of a DO-178B/ED-12B-compliant process [82] for single-system development; it was decided to adapt and augment this approach to cater for product-line development. This adaptation must not compromise the product and process attributes that contribute to the approval of the system; however, it must yield business benefit from the design and development of a set of systems rather than systems in isolation.

We show in this chapter how the single-system product architecture can be extended to become the reference architecture for a class of products. We define meta-models for describing product line architectures and components that are suitable for deployment in a high-integrity development. We describe how components that include variation can be hosted within, and products can be instantiated from the reference architecture framework.

4.2 From Single Systems to Product Lines

An approach of using a combination of UML class and structure models to architect single system applications was adopted successfully on a number of FADEC developments between 2004 and 2010. This approach used class models to describe the software structure, and employed a model-to-text transformation to generate a SPARK [75] implementation. A SPARK profile was used to extend the UML; this allowed the structure of the SPARK program to be fully described at the lowest modelled level of abstraction [77]. The UML modelling environment was used to define the architectural framework and the design details for the hosted components. Automatic report generation was used to produce design artefacts from the UML model that were used as configured design artefacts to support the software system approval (certification) process. This approach was successfully applied to a number of projects [82]. (Appendix A contains an overview of SPARK and details of the approach to modelling of SPARK programs in UML)

To respond to increasing demand for new products, the company decided to launch a software product line initiative and move the focus of the development process from single-products to the design of a range of products. As chief software architect, the author decided to take the previously successful architectural design approach and use this as the basis for the product line reference architecture. This had the advantage of enabling existing components to be donated ("harvested") into the product line with minimum rework/refactoring. (Note that the risk of the inappropriate reuse of these components was mitigated to a large extent by the adoption of the common architectural approach). It also minimised the learning curve for existing engineers that were used to using the UML/SPARK development processes.

4.3 Product Line Architectural Patterns and Reference Architecture

4.3.1 Reference Architecture Concept

A Reference Architecture provides a standard template architectural solution for a particular domain. Reference architectures are used as a basis for the development of particular software system solutions that fit within the target domain. They are especially useful as an enabler for a software product-line approach as they provide a framework within which product line assets can be developed. Assets that are compatible with the reference architecture will necessarily be compatible with a product instance derived from that reference architecture.

"*The reference architecture is capturing domain knowhow from the past and the vision of the future to guide architecting of future systems*" [33]

The purpose of the reference architecture for the gas turbine control system software product line is to provide standardised patterns, structure and framework for the application, enabling the hosting of components that contain variation. Our reference architecture is an evolution of the architectural concept used for the design of single-system solutions. The major changes were to address the shortfalls of this concept for the development of a product line; in particular the explicit support for variability. This was highlighted in an independent ATAM [117] assessment of the single-system architecture from the viewpoint of its suitability for use on a product line development [118]. In addition, we addressed lessons that emerged when adopting the architecture on a second system, primarily in the area of component interface identification and management.

The reference architecture contains three main facets:

1. Architectural Framework

The Architectural Framework consists of a definition of a **Platform** (framework aspects that exist at runtime) and an **Environment** (design, verification and management processes and tools to support the use of the platform). The framework identifies the standard software structure to be employed, defined in terms of software abstraction layers and communication interfaces. In this way, it structures the software system to solve a particular class of problems – here this is defined as software for high-integrity, real-time control, protection and monitoring systems. This very abstract, high-level software system scope definition actually allows the software architect to start to construct an appropriate architecture framework early in the project development cycle.

The framework provides the following as standard:

- Architectural Layers and Interfaces
- Computational Model (incl. Data Typing)
- Real Time Scheduling Support (including initialisation and modal support)
- Data Transport Infrastructure for distribution
- Monitoring for testing purposes
- Utilities for commonality of implementation

2. Components & Component Rules

A Component provides a set of cohesive functional software and associated provided and required interfaces. Each component has a well-defined purpose but may contain variation points to enable system variability to deliver a product line instance. The reference architecture does not necessarily identify the specific components for a particular application; however, the rules and constraints that candidate components must respect are defined as part of the reference architecture.

3. Deployment

The Deployment view shows an instance of the framework deployed on a particular microprocessor, with an allocated, instantiated set of components and bound set of interfaces. The Reference Architecture necessarily describes the process by which deployment is achieved; however, specific deployments are required for each microprocessor within each product instance.

4.3.2 Architectural Constraints

To enable the reference architecture to be defined, a set of constraints must be identified against which the architecture concept can be judged (usually qualitatively). Without a set of (preferably ordered) constraints, it is difficult to make decisions and trade-offs.

4.3.2.1 Product Line Constraints

The following set of constraints on the reference architecture were identified to aid the definition and management of the product line.

1. All software product line variation points shall be visible, identifiable and traceable in the product line architectural model (within the framework or within a component)

   Rationale: The intent is to deliver a BAPO Level 4 architectural solution. This specifies that *"…all products are developed based on the defined family architecture. In particular, it specifies how and when to configure variants"* [16]. The reference architecture is the primary vehicle to describe allowable solution-space variation in the product line.

2. All variation points identified in the architecture shall be traceable to identified product line stakeholder needs or domain configuration options (e.g. engine and airframe configuration).

Rationale: There needs to be a rationale for every variation point in the software. Unnecessary variation should be eliminated. If architecture and component variation is purely identified by analysing variation in previous project instances, then there is the danger that needless variation may be introduced.

3. All variation point choices that configure a product line instance shall be visible and traceable in the deployment model for that product instance.

   Rationale: There needs to be clarity in the choices made to configure a product instance – there needs to be an audit trail for each of the decisions made in selecting the specific product variants.

4.3.2.2 Architecture Design Constraints

A number of different software architectures may produce a solution that meets the functional requirements of a system; very few will meet both the functional requirements and the applicable technical and business constraints. The key, therefore, to a successful architecture and architecture-driven development process is a clear set of prioritised technical and business constraints.

These constraints can be modelled in the Artisan Studio UML tool using a "Constraints Diagram" as shown in Figure 40.

The blue curved-cornered rectangles describe constraint types, and the yellow rectangles describe instances of these types.

The diagram convention is that the constraints are shown in descending order of priority from left to right. The prioritisation of constraints enables the resolution of conflicting design approaches via trade-off analysis for example.

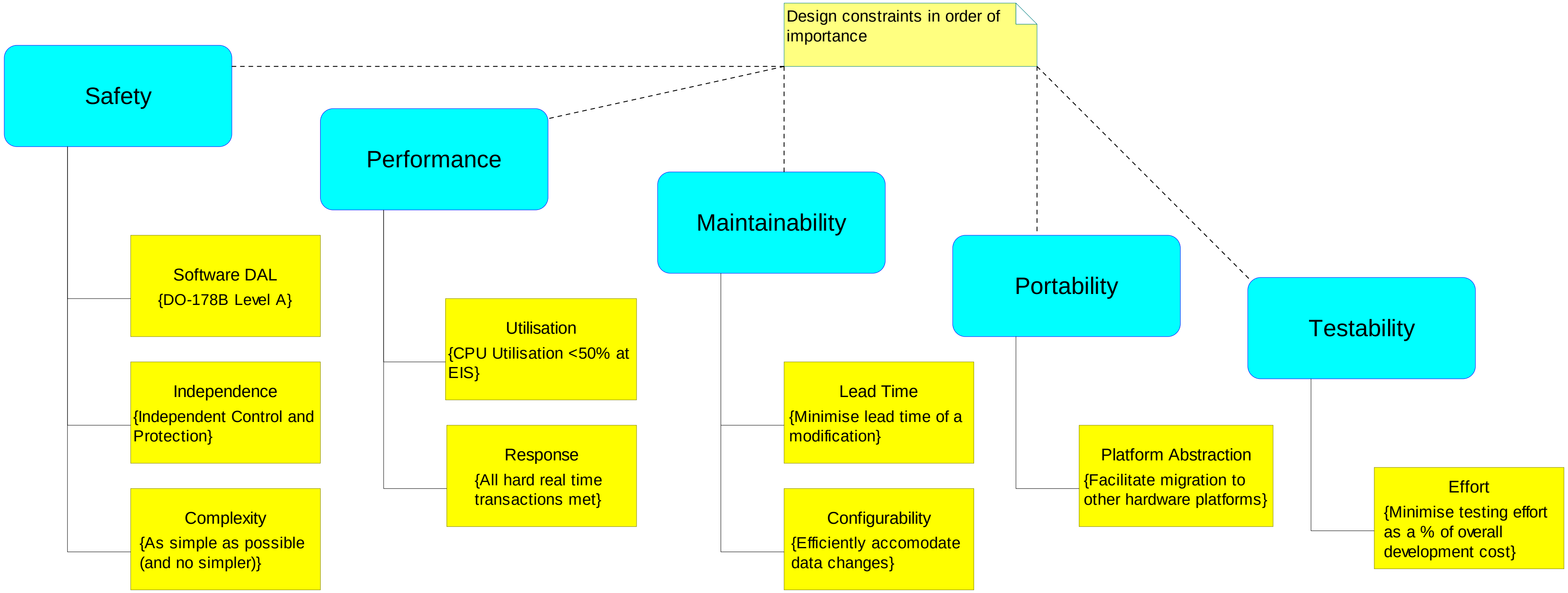

Design constraints in order of importance
Safety
Performance
Maintainability
Portability
Testability
Software DAL
{DO-178B Level A}
Independence
{Independent Control and Protection}
Complexity
{As simple as possible (and no simpler)}
Utilisation
{CPU Utilisation <50% at EIS}
Response
{All hard real time transactions met}
Lead Time
{Minimise lead time of a modification}
Configurability
{Efficiently accomodate data changes}
Platform Abstraction
{Facilitate migration to other hardware platforms}
Effort
{Minimise testing effort as a % of overall development cost}

FIGURE 40 PRIORITISED ARCHITECTURAL DESIGN CONSTRAINT DIAGRAM

We describe the constraint set shown in Figure 40 in more detail below:

1. Safety

An overall constraint called "Safety" covers the requirements to demonstrate the software, when integrated within the target system, meets the integrity and availability targets required for safe operation in service:

a) Software shall be developed to the requirements of DO-178B/ED-12B Level A
b) Control and Protection functions shall be independent
c) Software designs to be made as simple as possible (and no simpler)

2. Performance

The software is embedded within a real-time control system and has to meet hard real-time deadlines to comply fully with its operational requirements. The performance constraint also augments the real-time response requirements with resource utilisation targets:

a) Processor utilisation shall be < 50% at Entry Into Service (EIS)
b) All hard real-time transactions are met

3. Maintainability

The business has on-going targets to reduce lead-time for developing control systems and respond to customer problems in a timely manner. In software terms, these become targets for modifiability and maintainability of the software once the original development is completed:

a) Minimise lead time for a modification
b) Efficiently accommodate data changes

4. Portability

The business has on-going targets to reduce the cost of developing new control systems. The ability to port application software to other hardware platforms without incurring excessive redesign costs is important.

a) Facilitate migration to other hardware platforms with minimal effort

5. Testability

The testing cost of high-integrity software has been disproportionately high to date (~50% of development costs):

a) Minimise testing costs as a proportion of total development cost (Goal of 30% of total software costs)[2]

4.4 Product Line Architecture Framework

The baseline single-system software architecture was designed to satisfy the set of architectural design constraints described previously and was used successfully on two FADEC projects. We migrated this to take into account the product line constraints described in section 4.3.2.1.

We discussed in Chapter 2 the difference between problem space and solution space variability

- Problem space variability is concerned with the scoping of the product line and differentiating the products in terms of common and variable features.[21]
- Solution space variability is concerned with the artefacts that compose the system itself and how these can be varied to deliver the required product.[21]

The software reference architecture can be regarded as the first stage in the definition and modelling of the solution space variability (for software). We can model the relationship between the problem-space view in terms of "features" and the solution-space artefacts as shown in Figure 41.

[2] This is primarily based upon cost effectiveness of testing. The most expensive test vehicle (Low Level Test) is the one that finds the least number of errors. The business challenge is to reduce the cost of low-level testing to bring its "cost per error" rate in line with other testing techniques and thereby reduce the overall test cost as a proportion of total software development costs.

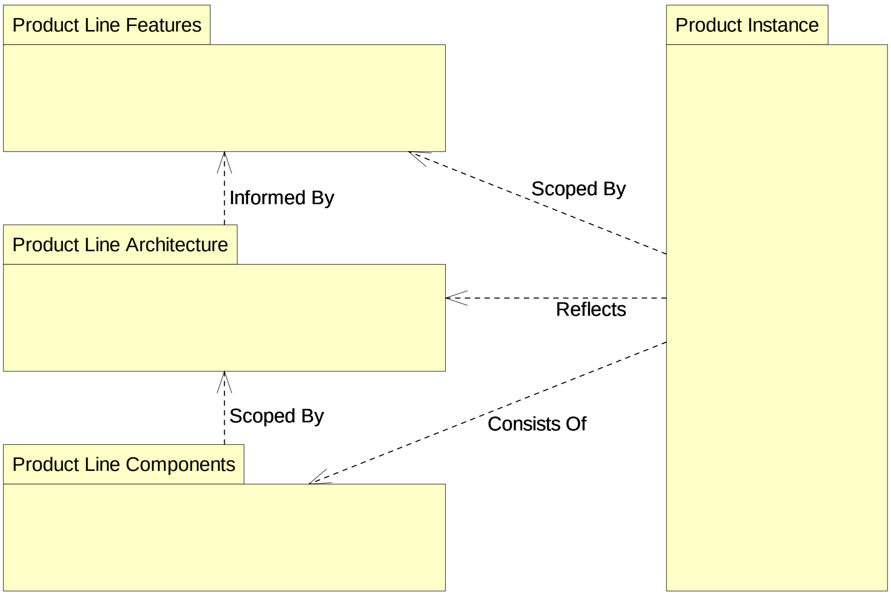


FIGURE 41 THE RELATIONSHIP BETWEEN PRODUCT LINE ARCHITECTURE, COMPONENTS, INSTANCES AND FEATURES

The package model shown in Figure 41 is described in more detail below:

- Product Line Features

  Here we use the term Features to describe system and software requirements and specifications defined for the common and variable parts of the Product Line. These could be captured in part by using feature-modelling techniques, or could be modelled using more traditional requirements capture and management tools. The important aspect is that they are problem-space representations of the product line features and have clearly identified the required commonality and variability. The precise notations and representations used to describe product-line features are out of scope of this thesis. (The initial approach taken to describe product line requirements used the PLUSS notation [119] to structure textual requirements and distinguish common/variable aspects. At the time of writing, work was being undertaken to augment this with a more formal feature model.) It is sufficient to note that the features should be described and decomposed to a level that allows traceability from the components in the solution-space that implement them.

- Product Line Architecture

  The Product Line Architecture is a framework into which components that may contain variability can be developed and deployed. As already discussed, this forms a **reference architecture** that defines the architectural concept for the product line, including rules for component construction and interfacing, identification of variation points and the definition and support for run-time behaviour, for example the temporal aspects of the software system such as sequencing and scheduling. The architecture is “informed” by the product line features, to the extent that it needs to be appropriate to the class of system being designed, and be able to support the most stringent requirements/constraints identified in the feature set.

- Product Line Components

  Product Line components are developed within the constraints of the architecture. These components implement the software requirements identified from the feature set and may contain variation points related to required variability. The components are "scoped by" the architecture which defines their provided/required interfaces and level of abstraction.

- Product Instance

  A Product Instance is scoped by an identified set of features (including the resolution of all allowable variability). It reflects the product line architecture as a blueprint for building the instantiated product. It consists of the set of components that reflect the selected features, with their variation points resolved appropriately.

4.4.1 Architectural Pattern - Layered Architecture

The baseline software architecture supporting the single-system developments was designed using a layered architecture pattern. The layered architectural pattern is commonly used where the following properties of the software product and development process are desired [120] :

- Need to localise changes to one part of the solution to minimise the impact on other parts, reducing the work involved in debugging and fixing bugs, easing application maintenance, and enhancing overall application flexibility
- Separation of concerns among components (for example, separating the control logic from the sensor validation) to increase flexibility, maintainability, and scalability
- Development of components that are reusable by multiple applications
- Independent teams need to work on parts of the solution with minimal dependencies on other teams and can develop against well-defined interfaces
- Cohesive individual components
- Loosely coupled unrelated components
- Various components of the solution need to be independently developed, maintained, and updated, on different time schedules.
- The need to deploy the application over multiple physical processors
- The solution needs to be verifiable (analysable and testable)

In a layered architecture the components in each layer are cohesive and at roughly the same level of abstraction. Each layer is loosely coupled to the layers underneath.

The key to the *Layers Architectural Pattern* is dependency management. Generally, components in one layer can depend on peers in the same level or components/interfaces from lower levels. Strict adherence to this principle eliminates or at least minimises inappropriate dependencies (and therefore maintenance cost). For large solutions involving many software components, it is common to have a number of components at the same level of abstraction that are not inter-dependent (i.e. they are purely dependent on the interfaces provided by the layer below).

Buschmann et al. [120] identifies the following benefits and liabilities of the layered architecture pattern:

Benefits:

- Reuse of Layers
- Support for Standardisation
- Localisation of Dependencies
- Exchangeability

Liabilities:

- Cascade of Changing Behaviour
- Lower Efficiency
- Unnecessary Work
- Difficult to Establish Correct Granularity

The liabilities need to be taken into account by the architect and mitigated if necessary, desirable and possible, given other constraints.

4.4.2 Generic Layered Architecture

Figure 42 below shows the abstract layered architectural pattern defined for the gas turbine control system software product line. A layered architectural concept was chosen that hosts components at various levels of abstraction. This was essentially unchanged from the abstract model developed by the author for the single system development approach.

The purpose of each of the layers is described in the following sections.

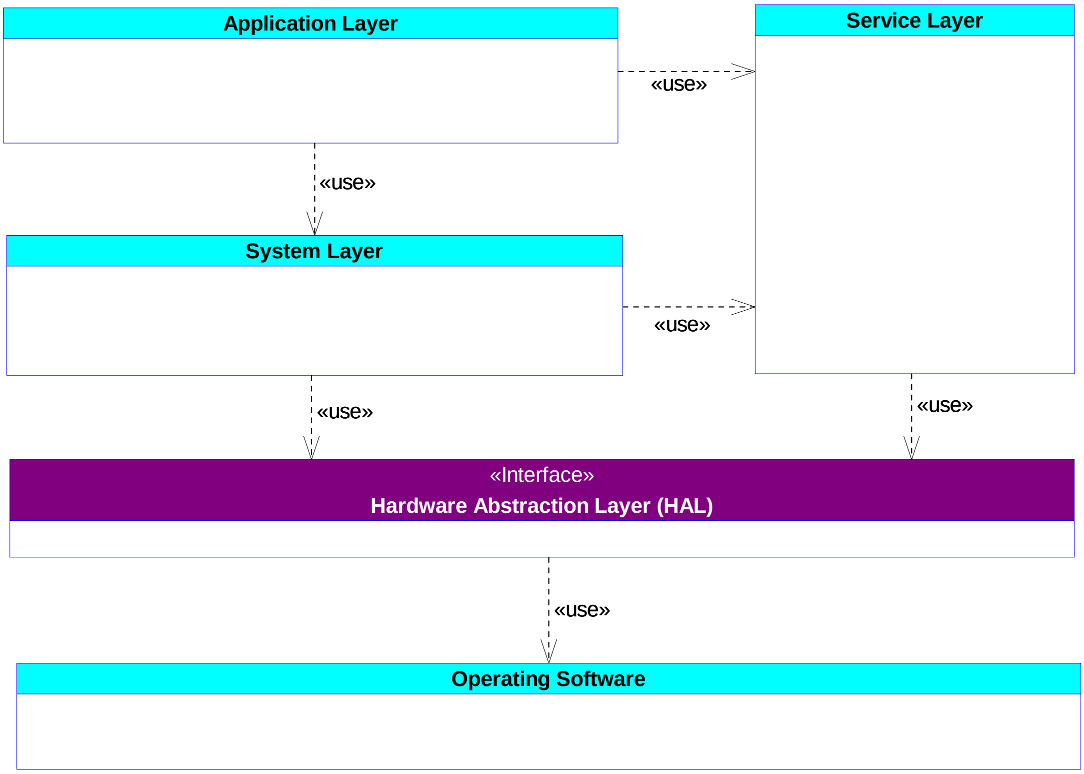


FIGURE 42 ENGINE CONTROL SYSTEM PRODUCT LINE – TOP LEVEL ABSTRACT ARCHITECTURAL PATTERN

The reference architecture defines standard abstraction layers in which the product line components are developed, and provides a run-time framework supporting the component execution. This includes a standardised scheduling/RTOS approach and a standardised data distribution mechanism to allow multi-processor deployment. The framework and support components are developed and managed by a central architecture team[3], who provide releases of the framework to the component development and product deployment teams.

The layers are defined as follows:

1. Application Layer

The Application Layer contains components that realise the end user's requirements for the system. Components located in the Application Layer should generally not be in support of other functions but should deliver behaviour that is recognisable externally to the system. A good test for an Application Layer component is to ask the question "If this component was all the system functionally delivered, would it still be a useful system?" i.e. does the

[3] The author was the Chief Software Architect for this team

external world have a use for this function? If the answer to the question is no then the location of the component in the application layer should be questioned.

Application Layer components operate in an idealised world with minimal knowledge of system configuration (e.g. duplex sensor configuration, processor allocation etc.)

2. System Layer

The System Layer contains components that ensure the continued operation of the system in the presence of faults (maximising the availability of the system), and to abstract the details of the system configuration away from the application layer. System Layer components translate between the ideal world of the application layer and the system device interfaces provided by the Hardware Abstraction Layer (HAL).

Typical system layer components will validate and select between multiple data sources, derive model parameter values from other available signals, take abstract demands from the application layer and convert them to device-specific commands to send to the HAL.

3. Service Layer

The Service Layer abstracts system services from the rest of the system. These services encapsulate access to generic system resources or collect/distribute non-cohesive data.

Components in the service layer typically provide abstractions for internal and external communications buses and non-volatile memory storage devices. In addition, the services are generally provided to multiple application/system components; removal of a single application/system component should not make a service component redundant.

4. Hardware Abstraction Layer (HAL)

The purpose of the Hardware Abstraction Layer (HAL) is to isolate the Application Software (AS) from the details of the underlying hardware platform. The layer implements a set of data classes and accessor operations that allow data transfer between the Operating Software (OS) and the AS. The layer completely isolates each side from the other, ensuring portability of the AS and minimising the impact of OS change on the AS.

The hardware abstraction layer provides a standardised interface to the device drivers provided the lower layers of the software system

5. Operating Software

The Operating Software (OS) provides the software interface to the hardware devices within the EEC. It converts between the engineering-unit domain of the HAL and the hardware-specific needs of the EEC electronics. The internal architecture and design of the OS is beyond the scope of this thesis.

4.4.3 Allocation of Components to Layers

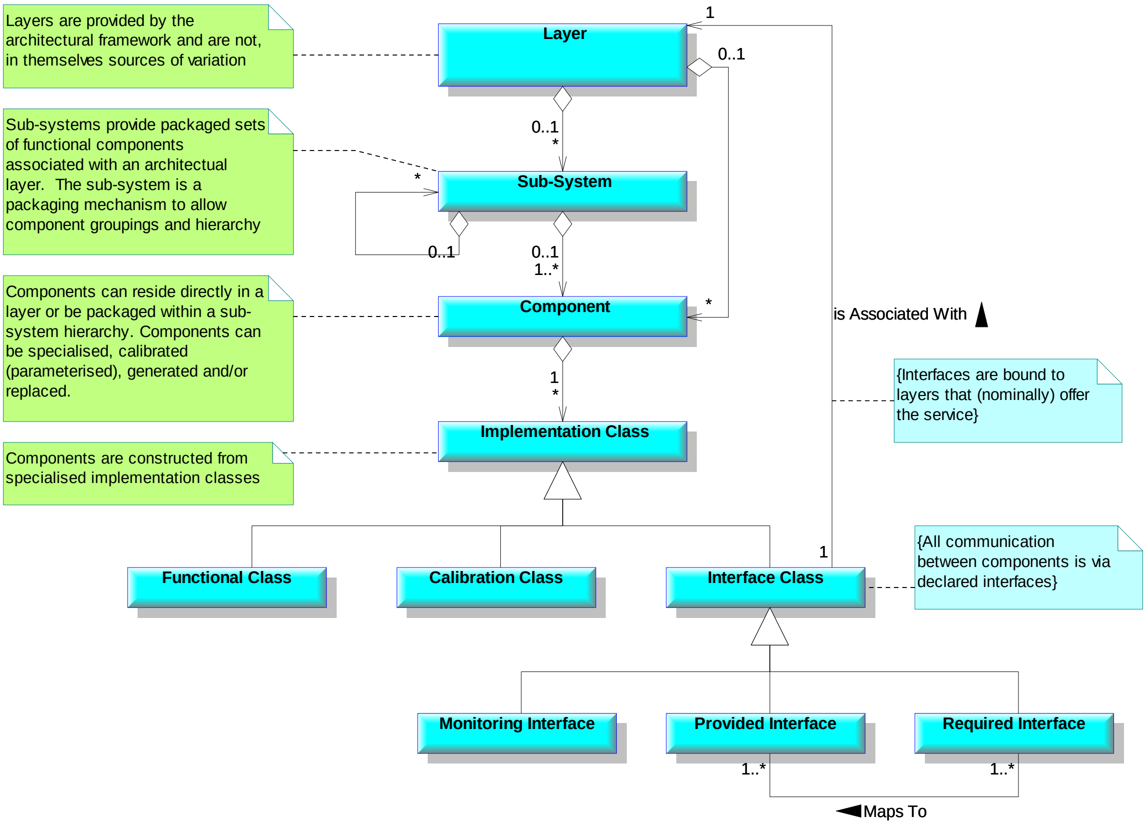

Layers are provided by the architectural framework and are not, in themselves sources of variation
Sub-systems provide packaged sets of functional components associated with an architectual layer. The sub-system is a packaging mechanism to allow component groupings and hierarchy
Components can reside directly in a layer or be packaged within a sub-system hierarchy. Components can be specialised, calibrated (parameterised), generated and/or replaced.
Components are constructed from specialised implementation classes
Layer
Sub-System
Component
Implementation Class
Functional Class
Calibration Class
Interface Class
Monitoring Interface
Provided Interface
Required Interface
1
0..1
0..1
*
*
0..1
0..1
1..*
*
1
*
1
is Associated With
{Interfaces are bound to layers that (nominally) offer the service}
{All communication between components is via declared interfaces}
1..*
1..*
Maps To

FIGURE 43 ARCHITECTURE STRUCTURE META-MODEL

Figure 43 illustrates the meta-model defining how components exist within the layers of the architecture described earlier. The parts of this model are described below.

1. Sub-Systems

Sub-systems provide a packaging mechanism to group related components within the component catalogue, and provide a convenient abstraction to describe cohesive functional groupings within high-level architectural descriptions. Sub-systems are the only means of providing hierarchical decomposition within the architectural description. Sub-systems are modelled as Packages within UML.

(Note the «subsystem» stereotype in the standard UML profile only applies to classes. Here we apply it also to UML Packages (Categories))

2. Components

Each layer of the software architecture contains a number of cohesive components that have a well-defined function or purpose. The set of components within a layer are nominally at the same level of abstraction, but are loosely coupled (if at all) to each other. Each component can have instances of four generic interface types as illustrated in Figure 44.

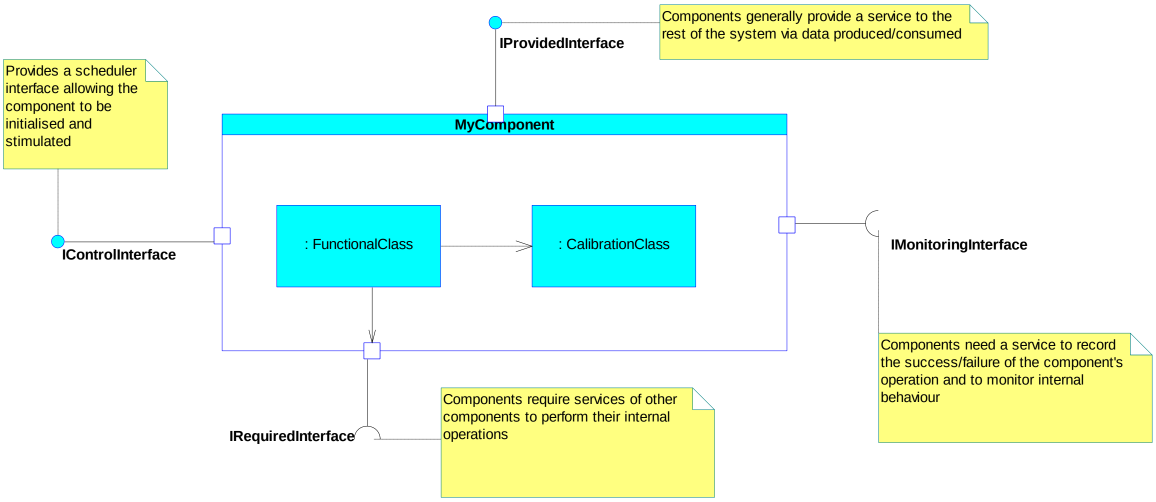

Components generally provide a service to the rest of the system via data produced/consumed
IProvidedInterface
Provides a scheduler interface allowing the component to be initialised and stimulated
MyComponent
: FunctionalClass
: CalibrationClass
IControlInterface
IMonitoringInterface
Components need a service to record the success/failure of the component's operation and to monitor internal behaviour
Components require services of other components to perform their internal operations
IRequiredInterface

FIGURE 44 GENERIC COMPONENT INTERFACES

The layers, components and interface concept allows the high level software architecture to be decomposed and allocated to a level from which code components can be identified and designed. In general, the software implementation details are captured in the lower level architecture in the UML model.

3. Implementation Classes

a) Functional Class

The functional class(es) provide the implementation to satisfy the requirements of the component. These classes use the interfaces to communicate between components in different subsystems.

The functional class exposes a control interface to enable the scheduler to execute the component's functionality. The scheduler can support both periodic (time-based) and sporadic (event-driven) operation. The required scheduling behaviour is defined as part of the interface specification.

Optionally, components may also present an initialisation interface. Passive components may present initialisation interfaces (for example, interface components that have no functional behaviour may need their default values initialising).

b) Calibration Class

Calibration classes are used to provide calibration data that is used by the functional classes. The data is in the form of Development Variables and Graphical look up tables that can be calibrated during testing, but are constants when the executable is delivered for production.

c) Interface Class

i. Provided Interface

Typically, components perform an activity and produce a set of results useful to other parts of the system. The result of a component's operation is presented to the rest of the system via the provided services interface. This generally forms part of the containing layer's interface.

Utility and library components can provide a callable interface that can be used from other components as part of their execution.

ii. Required Interface

Components require services of other components to perform their intended operation. The set of required services of a component should be provided by the available layer interfaces.

iii. Monitoring Interface

Components that need to record failure behaviour make use of a generic monitoring interface. This enables the centralised health and maintenance functions that analyse the faults to be loosely coupled to the source components.

This is a required interface i.e. it is provided elsewhere (note the "socket" notation). In general terms, this means that if this component were to be removed from the system the interface would still exist, as it is required by other components.

4.4.4 Compatibility with Previous Projects

The component design specified above is not dissimilar to those used on previous (non-product line) projects that employed a similar layered architecture and component breakdown. It is a design aim that the product line architecture can support components created for these previous projects with minimum change – this allows for the "harvesting" of existing components as required.

4.4.5 Deploying Architecture and Components

We have identified an architectural pattern for our FADEC software system, and defined the meta-model to allow components to be developed that comply with this pattern. We now map this model back onto our view of product line development to clarify how the reference architecture, product instance architecture and component set map onto the framework identified in Figure 41.

Figure 45 shows generically how the reference architecture layers define the product line architecture, and contain a set of product line components. These layers have equivalents within the product instance, which can host "bound" components, through a model allocation mechanism discussed later.

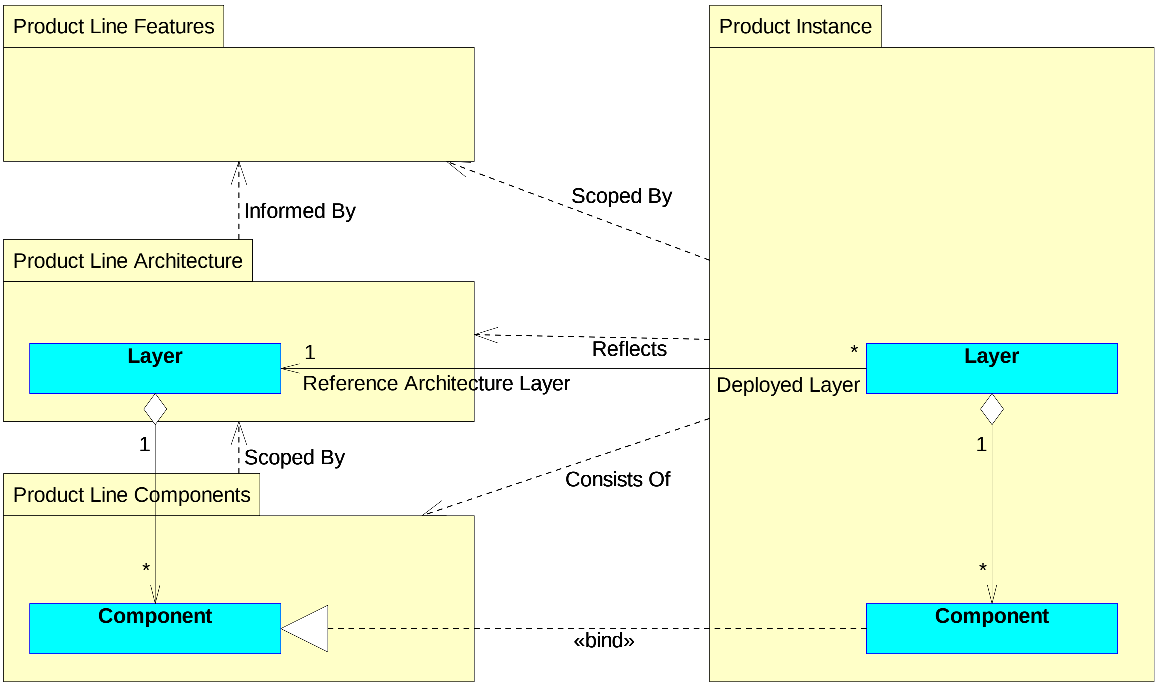


FIGURE 45 THE ROLE OF LAYERS AND COMPONENTS IN THE PRODUCT LINE CONCEPT FRAMEWORK

4.5 Designing Components

We define a component as a functionally cohesive collection of design, specification and implementation information, from which other representations can be generated via transformation. Source code implementations of the components can be generated using model transformations (as described in chapter 5.) The component is modelled using the UML class notation to describe its structure; this is augmented with algorithmic design detail defined using complementary UML notations (for example activity diagrams or state diagrams), or using functional modelling languages and tools such as Matlab/Simulink [69] or SCADE [59]. The problems of interoperability between modelling environments based on a functional/dataflow paradigm and those based on a structural/object paradigm is a significant issue in embedded system design [121]. Currently our approach uses the UML modelling environment as the master, and any design descriptions generated in other environments are imported into the master model as additional annotations (typically) on operations. Currently no syntactic or semantic integration is attempted between the modelling environments (this is discussed later in the context of future work in Chapter 8)

The UML definition of an operation is extended with SPARK as shown in Figure 46.

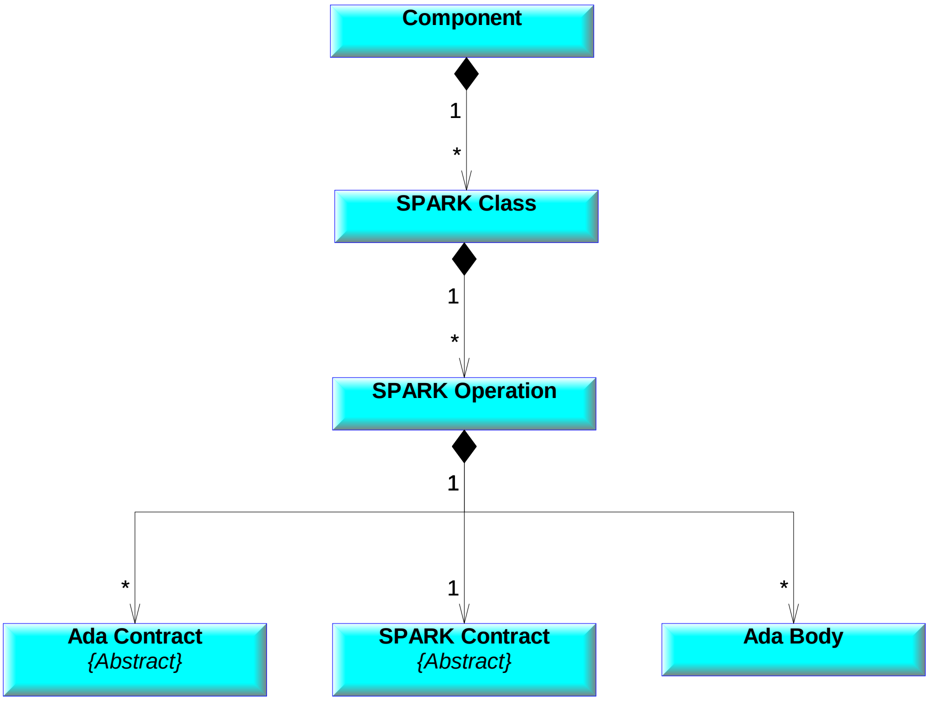


FIGURE 46 COMPONENTS BUILT FROM SPARK CLASSES

SPARK Operations introduce the concept of a SPARK Contract [77]. The SPARK mechanism allows a range of operation contract levels to be defined, from data-flow contracts to full pre and post conditions. Our approach uses information flow contracts that define the required input/output relationship of the operation. The implementation of the component can be verified against this information flow contract using the SPARK Examiner tool [75]. The ability to verify statically a component implementation against a contract is fundamental to our approach towards ensuring the correct generation of product line components containing variability. Figure 46 shows, in addition, that operations contain associated design descriptions and implementations, in the form of SPARK-compliant Ada code bodies.

4.6 Extending Component Contracts with Decisions

Feature model-based product line approaches often maintain a direct relationship between optional features and variation points with the product line assets. Our approach introduces a level of indirection into the variability model via the use of **decisions [122]**. Decisions provide a more granular means of describing variability, and these variability decisions are typically in the context of the implementation rather than the user-oriented view provided by the feature model. Significantly, this approach can be used to construct components before a complete understanding of the product line scope is available. These component decision points relate directly to **variation points** within the internals of the component. These variation points identify model elements that should be included or removed from the component when associated decisions are resolved.

Relationships can be established between features in a feature model and the component decisions, enabling feature-driven selection and traceability to be implemented. The provision of a level of indirection between a feature model and an implementation via a **decision model** has been demonstrated before [122] and is supported in prototype and commercial software product line environments. However, our approach is significantly different and novel in that it makes variability decisions first-class model elements and contains them within the components exhibiting the variability. In this way, variability decisions are prominent in the component designer's mind at the point of component design, and can be verified alongside the component design and implementation, for example via peer review. In addition, components containing variability can be shared between multiple product lines and the mapping between variation points and variability decisions is maintained.

The approach introduces the concept of **decision contracts**. The component contract is augmented by a model element termed a **decision**. The decision is a public attribute of the component contract. The decision attribute contains a set of possible resolutions to the decision, known as **options.** When a component is deployed, part of the action of deployment is to resolve each decision in the public decision contract. This involves choosing an available option for each of the published decisions. A meta-model defining this approach is shown in Figure 47. It clearly shows how the decision forms part of the component contract, and how decisions are related to modelling elements identified as variation points. In this way, the component contains and publishes the available variability in a concise manner, making the component reusable across product lines in a much more straightforward manner than would be the case if the variability were defined separately.

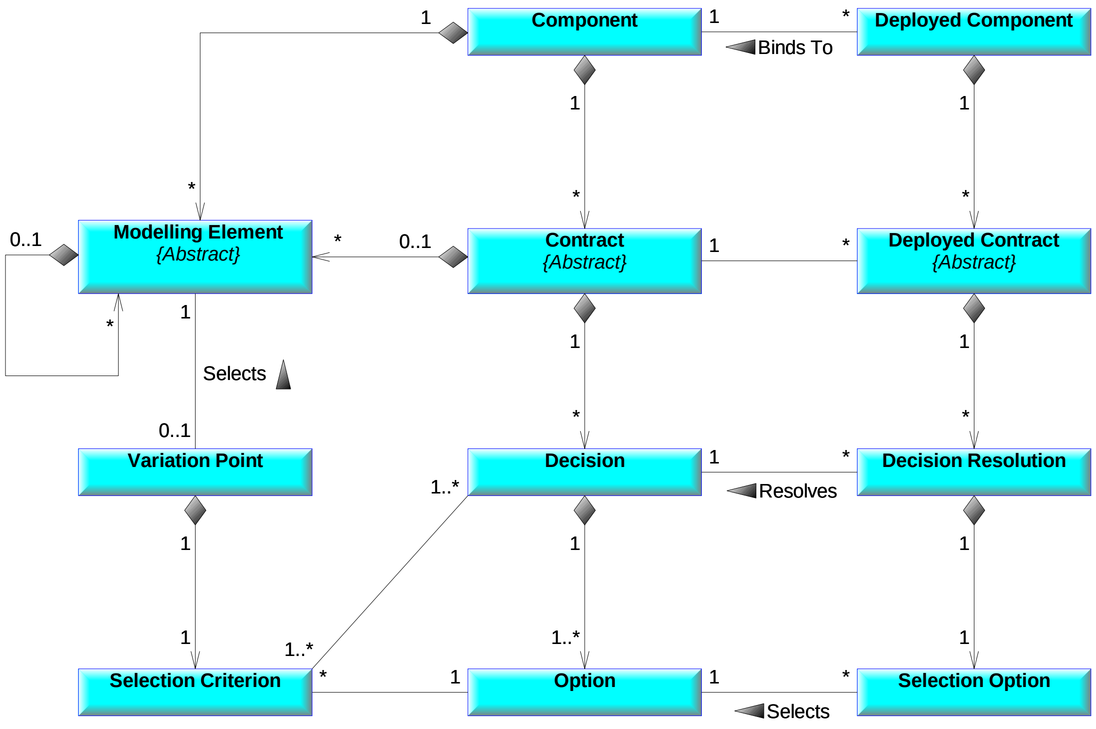


FIGURE 47 META-MODEL DESCRIBING COMPONENTS CONTAINING DECISION CONTRACTS

Component variability is realized via the use of «PL variation point» stereotypes within the component model. Variation point stereotypes can be applied to any relevant meta-model element. Each variation point stereotype contains a “select when” attribute; this attribute holds an expression in terms of component decisions. Evaluation of this expression determines whether the associated meta-model element is included in the product instance model. The set of model transformations that evaluate these expressions and produce a product instance are discussed in the next chapter.

4.6.1 Variability & Variation Points

Variation Points identify places in architecture and the set of components where product-to-product variability is allowed. In general, component variability can be realised using a number of different variation techniques, and the selection can be made at different stages in the development lifecycle (known as the “binding time”). In our approach, most variability is resolved at code generation time, where model-to-model and model-to–text transformations produce the instantiated product (as described in Chapter 5).

Our instantiation mechanism ensures that only the functionality required in the specific product is to be deployed. It is not advantageous to carry round additional functionality as:

- Functions not required but resident in the executable will need to be deactivated. Any deactivation mechanisms will need to be specified and verified as required by DO-178B/ED-12B.
- Product Line assets may contain data that is proprietary to specific customers, however the product line may be instantiating a product for that customer’s competitor organisations (e.g. for avionics applications the product line may be instantiating products for both Boeing and Airbus applications, and contain airframe-specific information). Given that customers may have a right of audit and scrutiny over the development processes and artefacts it should be possible to provide development assets that are free from competitor’s protected information.
- Embedded systems can be resource constrained (e.g. the amount of available PROM space for program storage), so it can be advantageous to remove unnecessary code.

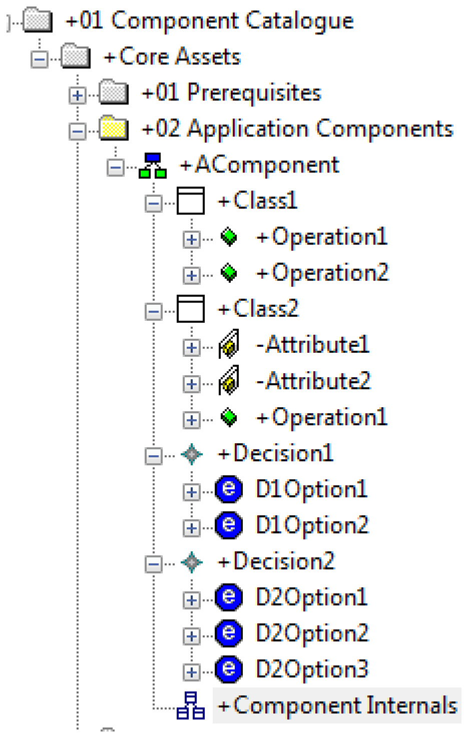
The following table lists the types of variability provided for each Meta-Model element of interest when modelling SPARK components:

Meta-Model Element	Variation Point?	Comment
Subsystem	Yes - Manually	Subsystems can form part of a hierarchy with only certain subsystems being required for particular deployments. The subsystems that are not required do not form part of the

		deployment set.
Class	Yes	Classes marked as variation points are to be removed from the deployment model automatically by the transformation if their selection criterion is not met.
Operation	Yes	Operations marked as variation points are to be removed from the deployment model automatically by the transformation if their selection criterion is not met.
Parameter	No	Operation parameters are not modelled as variation points. Operations that require varying signatures shall be modelled as alternate operations.
Attribute	Yes	Attributes marked as variation points are to be removed from the deployment model automatically by the transformation if their selection criterion is not met.
Types (Sequence, Record (Structure), Array, Enumeration)	Yes	Types marked as variation points are to be removed from the deployment model automatically by the transformation if their selection criterion is not met.
Record (Structure) Element	Yes	Record elements marked as variation points are to be removed from the deployment model automatically by the transformation if their selection criterion is not met.
Enumeration Literal	Yes	Enumeration literals marked as variation points are to be removed from the deployment model automatically by the transformation if their selection criterion is not met.
Associations	Yes	Associations marked as variation points are to be removed from the deployment model automatically by the transformation if their selection criterion is not met.

4.6.2 Encoding Variability

We have discussed the concept of decision contracts, and how they map onto variation points in the software architecture and the components. Here, we illustrate how this is realised in practice with an example component containing a decision contract and associated variation points modelled in UML. Figure 48 shows an expanded UML browser "tree" for a Product Line component named **AComponent**. The set of icons in the browser have been extended based upon the UML stereotypes used to implement the trusted product line meta-model. A Product Line component is modelled as a stereotyped UML package, and is indicated as 🖧 in the browser. A Decision is modelled as a stereotyped UML enumerated type, and is indicated as ✦ . In Figure 48, we see that **AComponent** publishes two decisions, with Decision1 having two possible options, and Decision2 having three possible options.



```
+01 Component Catalogue
  +Core Assets
    +01 Prerequisites
    +02 Application Components
      +AComponent
        +Class1
          +Operation1
          +Operation2
        +Class2
          -Attribute1
          -Attribute2
          +Operation1
        +Decision1
          D1Option1
          D1Option2
        +Decision2
          D2Option1
          D2Option2
          D2Option3
        +Component Internals
```

FIGURE 48 STRUCTURE OF A SIMPLE PL COMPONENT WITH CONTAINING A DECISION CONTRACT

Figure 49 shows a class diagram representation of the two classes contained in **AComponent.** Here we see that there is an association between the classes that model an Ada "With" clause. The class diagram also shows that particular operations and attributes within the classes are decorated with «PL variation point» stereotypes to indicate that those model elements are optional. Similarly, the association between the classes is denoted as optional, again via use of the «PL variation point» stereotype.

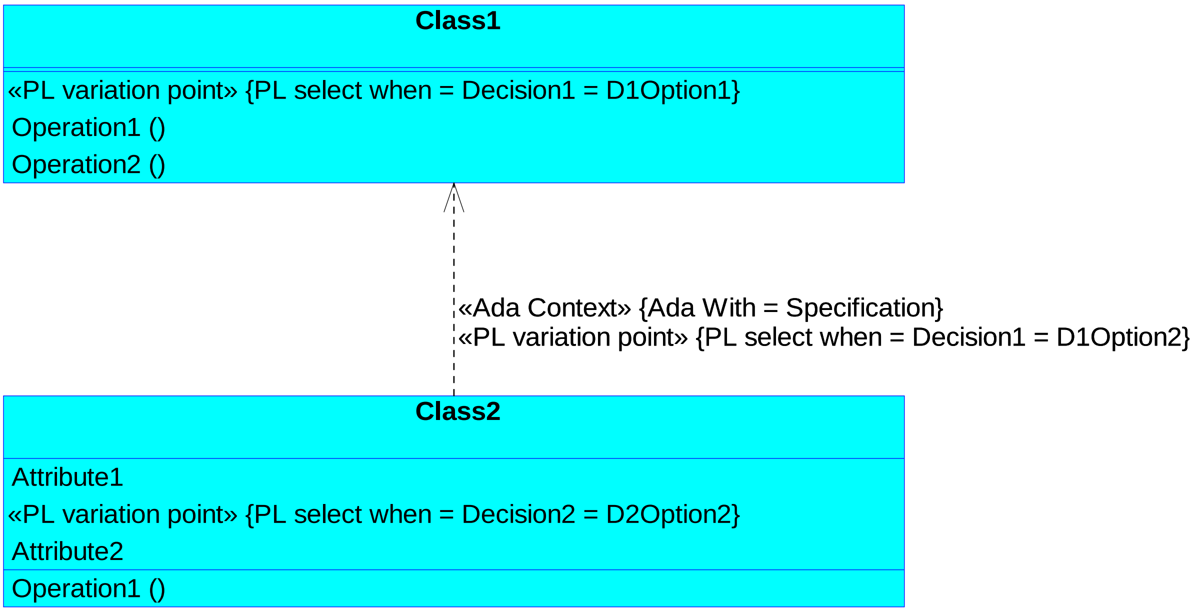

Class1
«PL variation point» {PL select when = Decision1 = D1Option1}
Operation1 ()
Operation2 ()
«Ada Context» {Ada With = Specification}
«PL variation point» {PL select when = Decision1 = D1Option2}
Class2
Attribute1
«PL variation point» {PL select when = Decision2 = D2Option2}
Attribute2
Operation1 ()

FIGURE 49 CLASS DIAGRAM ILLUSTRATING VARIATION POINTS

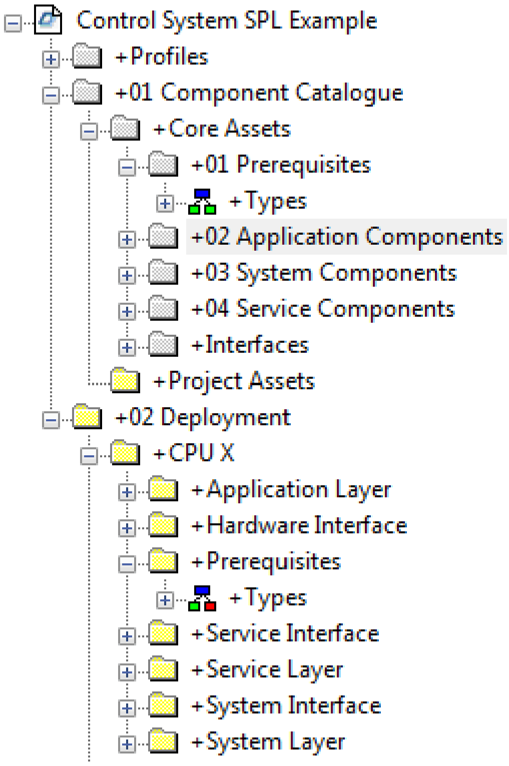
The conditions under which each variation point is selected are encoded in the "PL select when" expression. This is contained in a UML "tag" associated with the «PL variation point» stereotype. The "PL select when" expression is described in terms of the decisions and options published in the component's decision contract.

This approach results in a combined product design and variability model - there is no separate orthogonal variability model (as discussed with respect to the CVL approach in section 2.3.3). The single-model approach was chosen primarily for ease of verification-by-review of the product line models. It was felt that it was more straightforward to manually review the correctness and completeness of the variability mark-up if the complete set of information was presented in a single, coherent model. However, this is essentially a presentation issue (conceivably multiple models could be presented in a coherent combined view); the ease of verification of differing model forms and view could be the subject of future research. Also, as was discussed earlier in section 4.6, this approach allows variable components to be self-contained and therefore enables their reuse across product-line instances; this would be complicated by the use of separate orthogonal variability models.

4.7 Component Catalogue, Core Assets and Deployment

We have discussed so far the approach to modelling product line components within UML. We now look the model management and deployment strategy; i.e. how those components are stored and managed within a modelling environment and how component deployment is performed.

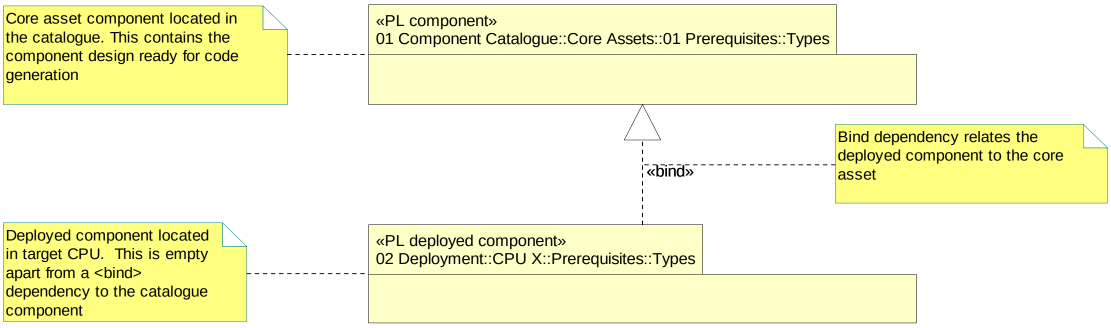
Figure 50 shows how components are stored in a "Component Catalogue", whose structure reflects the reference architecture layers. This component catalogue allows the storage and management of both product line components ("Core Assets"), and any components developed specifically for the project itself ("Project Assets"). Irrespective of their source or purpose, the components are constructed, managed and deployed in the same manner.



- Control System SPL Example
  - +Profiles
  - +01 Component Catalogue
    - +Core Assets
      - +01 Prerequisites
        - +Types
      - +02 Application Components
      - +03 System Components
      - +04 Service Components
      - +Interfaces
    - +Project Assets
  - +02 Deployment
    - +CPU X
      - +Application Layer
      - +Hardware Interface
      - +Prerequisites
        - +Types
      - +Service Interface
      - +Service Layer
      - +System Interface
      - +System Layer

FIGURE 50 MODEL HIERARCHY SHOWING CORE ASSET AND DEPLOYMENT TREE

Products are realised by deploying components onto CPUs; this can be seen in the tree structure shown in Figure 50, where a deployed version of the "Types" component appears in the pre-requisites folder for "CPU X" (note the slightly different icon colours for a deployed component). The actual deployment relationship is modelled in the class diagram shown in Figure 51.



Core asset component located in the catalogue. This contains the component design ready for code generation

Bind dependency relates the deployed component to the core asset

Deployed component located in target CPU. This is empty apart from a <bind> dependency to the catalogue component

«PL component»
01 Component Catalogue::Core Assets::01 Prerequisites::Types
«bind»
«PL deployed component»
02 Deployment::CPU X::Prerequisites::Types

FIGURE 51 BIND DIAGRAM SHOWING THE DEPLOYMENT OF THE TYPES COMPONENT

Here we see that the components themselves are modelled as stereotyped UML packages, with a "bind" dependency mapping the deployed component onto the core asset. This is an instance of the meta-model shown in Figure 47, which described the Component, and Deployed Component classes and the Bind To relationship between them.

4.8 Mapping to Requirements and Feature Models

We discussed the use of feature models in chapter 2, as a means of expressing commonality and variability in the problem domain. Here we discuss the role of requirements and specification in high-integrity developments, and the role played by traceability in justifying the correctness of products, including the absence of unintended function. We look at the role feature models may play in Trusted Product Line development, and we examine how traceability spans the problem and solution domains – including the effect that variability has on traceability.

4.8.1 Requirements & Traceability in DO-178B/ED-12B Developments

Let us revisit the diagram we first introduced in chapter 2 illustrating the objectives of DO-178B/C and their relationship to the development artefacts. Here (Figure 52), we can see the central role that traceability plays in the review/analysis of the product. DO-178C/ED-12C [60] provides the following definitions of Traceability and Trace Data:

<u>Traceability</u> – An association between items, such as between process outputs, between an output and its originating process, or between a requirement and its implementation.

<u>Trace data</u> – Data providing evidence of traceability of development and verification processes' software life cycle data without implying the production of any particular artifact. Trace data may show linkages, for example, through the use of naming conventions or through the use of references or pointers either embedded in or external to the software life cycle data.

Traceability is one of the primary mechanisms used in DO-178B/ED-12B and DO-178C/ED-12C to argue and justify that

a) Every part of the software has a reason/rationale for its existence

b) Every requirement placed on the product has been satisfied

i.e. the product does what it is required to do and no more. There is increasing focus by civil aerospace regulators that "unintended functionality" is identified and eliminated from software products [123]. This is a clear area of concern for product line approaches that include design-time variability - they have an inherent risk of inadvertent inclusion of unintended functionality.

The Trusted Product Lines approach must provide the means to define the Traceability associations between the lifecycle artefacts in the product line, and to provide the Trace Data to support the instantiated product.

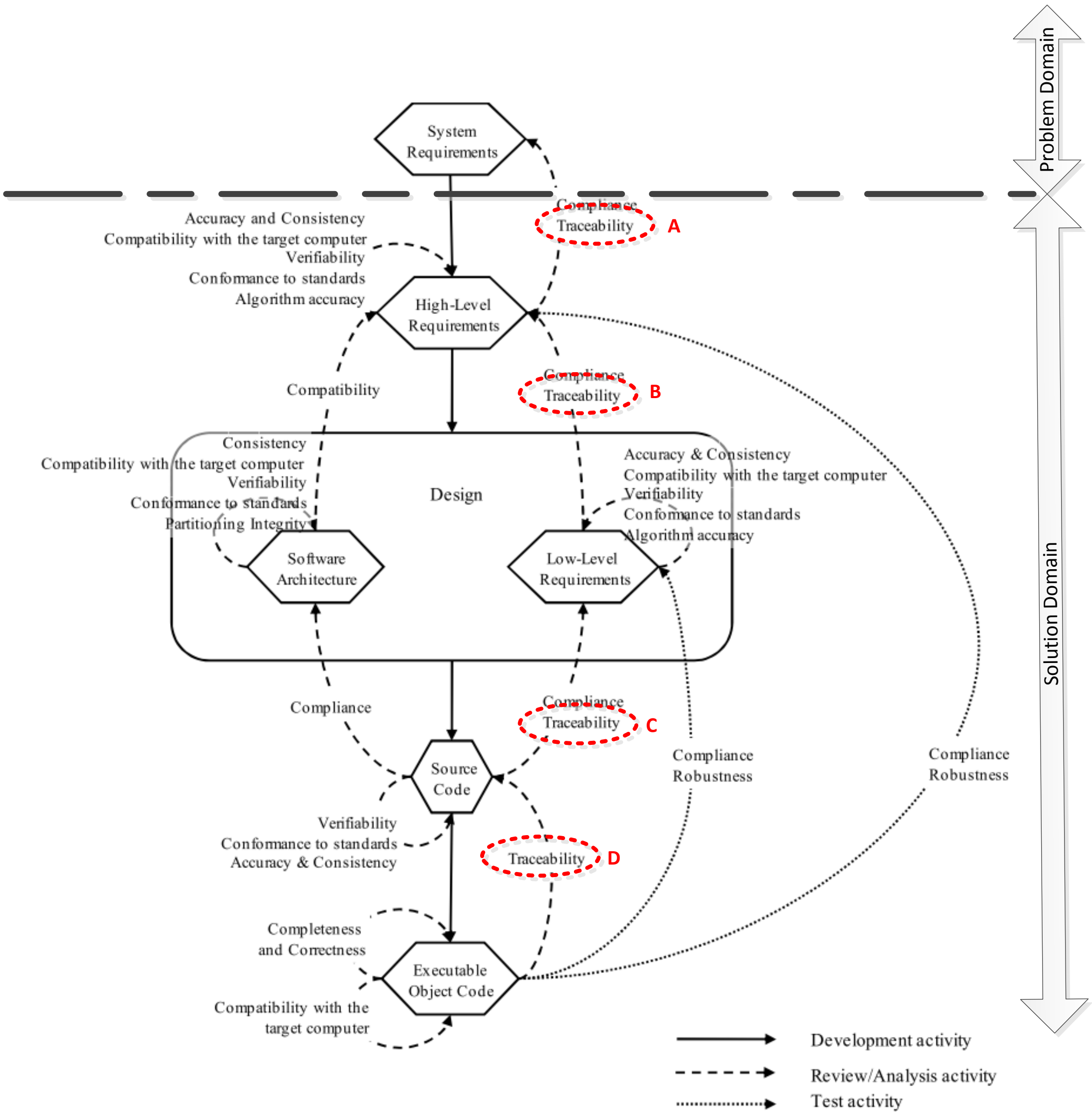


Figure 52 Annotated DO-178C/ED-12C Level A Software Verification Processes

4.8.2 Variant Traceability & Feature Linkage

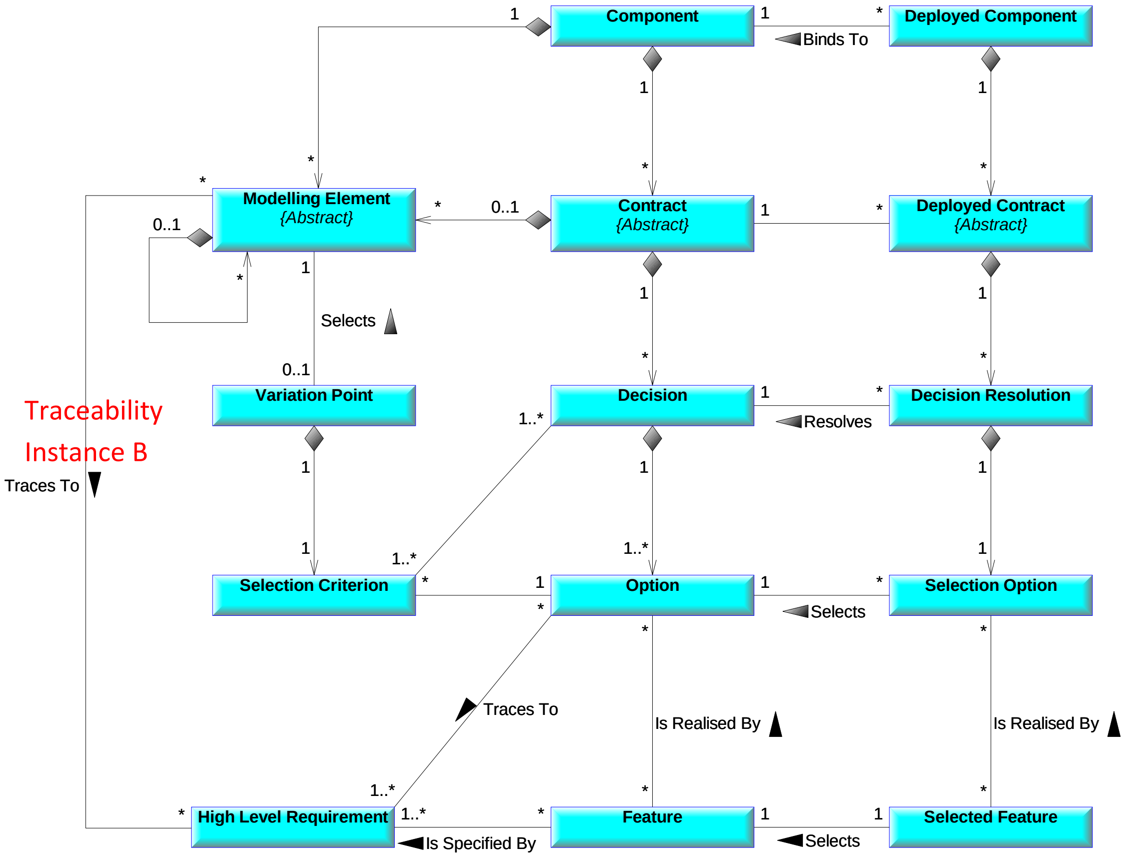


FIGURE 53 EXTENDING THE COMPONENT META-MODEL WITH TRACEABILITY AND FEATURES

Figure 53 shows how the component meta-model introduced in section 4.6 can be extended to include traceability associations and linkage to the problem-domain commonality and variability Feature Models. "Normal" DO-178C/ED-12C traceability associations (establishing traceability instance B from Figure 52) is established via the association between modelling elements and High Level Requirements. However, some of this trace data may include traceability to variant requirements. This can easily be included/removed from the product instance trace data if the modelling element is wholly included/removed during instantiation. However, there may be instances where a modelling element traces to both common and variable high-level requirements. Where the variable requirements are not included in a particular product instance, we need to remove those references from that product's trace data.

To address this, our extended component meta-model includes a "traces to" relationship between a decision **option**, and the high-level requirements satisfied when that option is selected. In this way, we can easily identify the set of variable requirements that are implemented by a product line instance, and produce the correct trace data for that instance via simple set operations, i.e.

{Product Instance Trace Data} = {Common Trace Data} ∪ {Selected Options Trace Data}

In practice, the determination of the product-specific model element's trace data is slightly more complicated than described above, but still straightforward. The set operations to determine a product-specific model element's trace data are as follows:

$$Mi = (Mt - Co) \cup (Mt \cap Po)$$

Where:

Mt is the set of trace data for a model element in the product line

Co is the set of variant trace data for the product line component

Po is the set of selected variant trace data for the specific product instance

Mi is the complete set of trace data for the model element in the product instance

The $(Mt - Co)$ term determines the set of common traces from the model element and the $(Mt \ \cap Po)$ term determines the selected set of variant traces. The union of the two sets gives the product specific trace data for the model element.

A complete worked example of this is provided in the Case Studies in Appendix C of this thesis.

This approach allows a definitive set of trace data to be created for an instance of a component automatically, given the resolution of a decision contract (i.e. a set of options are selected).

The extended meta-model in Figure 53 also defines how a problem domain model (e.g. a feature model) can be associated with a set of components that expose decision contracts. We discussed the difference between problem-domain and solution-domain models in Chapter 2. The Decision Contract concept provides the ability to identify and specify variability in the solution domain. This optionality may be related to feature selections in the problem domain via the "Is Realised By" associations modelled in Figure 53.

Here, we envisage that the realisation of a Feature is via the inclusion of one or more components in the software system, and the setting of particular options within the decision contracts of those components. This, then, decouples the design of components from the identification of the product line features. In this way, components can be re-used across multiple product lines, but still be mapped into the feature selection mechanisms for the product lines to which they contribute. The "Is Realised By" associations are established downwards from the features to the componentry chosen to realise those features – either by direct linkage or by an intermediate "mapping model". Again, this allows the re-use of components, as there is no hard-coded linkage to product features in the component.

The overall concept of problem-space feature models being mapped to solution-space variation points indirectly via the component decision contract is shown diagrammatically in Figure 54.

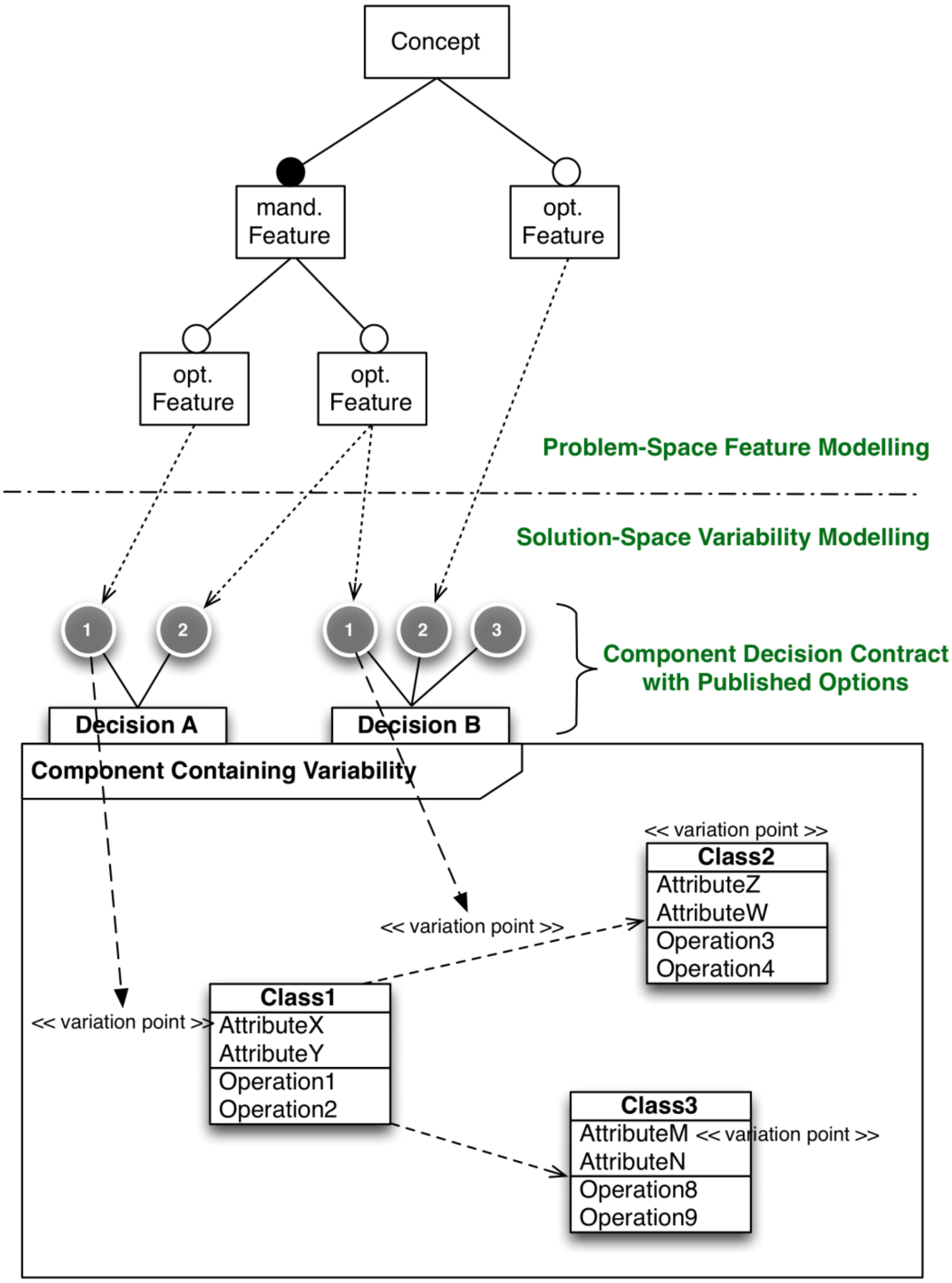


Figure 54 Diagrammatic View of Feature to Decision to Variation Mapping

Here, we can see clearly that we separate the concerns of the implementation variability from the definition of the product line features via the introduction of the abstraction "decision contract". This is a valuable and generally applicable abstraction that does not restrict or impose the problem-space representation, and allows the implementation components to be portable across multiple product lines. It also allows heterogeneous implementation technologies to be used inside the component as it abstracts away the peculiarities of the implementation from the user of the component.

4.9 Conclusions and Observations

We have defined a meta-model for component based product-line design that includes the novel concept of decision contracts. We have explained how the meta-model is used to capture, view and navigate the product-line architecture and component design. We understand how the meta-model can capture the relationship of the product line components to high-level requirements and product features, including the management of traceability and production of product trace data. We understand the use of architectural models and UML to structure product-line reference architectures, and understand the unique contribution of the decision contract approach.

This gives us a well-defined framework for designing and modelling product-line software solutions. We now need to understand in detail how product instances can be created from these models, largely automatically.

5 Instantiating Products using Model Transformation

We can regard many of the activities undertaken within software development as "purposeful transformations". For example, the transformation of requirements into designs, design to source code, source code to object code. One technology that is increasing becoming central to a number of software engineering approaches is that of Model Transformation. Put simply, this is the changing or modification of a model from one form to another. These transformations become "purposeful" when both the initial and transformed models have specific, useful purposes AND they preserve a given set of properties of the initial model in the transformed model. This becomes even more useful if you can **guarantee** that the properties of the initial model are held by the transformed model. There is little in the literature that addresses the problems of property-preserving model transformation; to our knowledge, no work has addressed this issue in the context of a formal certification process.

This chapter describes an approach to developing and deploying high-integrity product lines using a model transformation approach. We have demonstrated the techniques described by developing a model transformation based code generator, which has been deployed and used on a real high-integrity development project. Whilst there is pre-existing work on product lines and on model transformation (see chapter 2) none of the previous work has fully considered the challenges of high-integrity development (the work of Esterel and SCADE [59, 71] comes closest, but it does not address product line issues). This deployment has provided real-world data to demonstrate the applicability and scalability of the technology; this is evaluated later in the thesis.

Here we detail the design of model transformations to create particular product instances from a reference architecture model and product line component assets. This focuses on using components with **decision contracts** (which is a novel contribution as described in Chapter 4), resolving those decisions to reflect a particular required product, and using model-to-model and model-to-text transformations to instantiate the product-specific assets. Using a chained set of purposeful transformations, whilst not a novel concept per-se, we believe to be an original contribution for the instantiation of products from product line assets, particularly in a safety-critical environment.

It should be noted that while this approach has been designed to be applicable to the development of high-integrity systems, there is nothing inherent in the approach that prevents its more broad application. Our approach would be useful in any domain characterised by the need for reusable, variable components, strong architectural focus, clearly traceable design and credible verification (for example, the development of automotive and medical systems.)

5.1 Research Challenge

The main research challenge addressed in this chapter is to demonstrate successful achievement of the following goal:

To define a Software Product Line production environment suitable for High-Integrity applications, including the provision of approval/certification evidence.

This is not a hypothetical or abstract challenge; the Software Product Line production environment needs to be robust enough to be applicable on real industrial projects, be used by large (potentially geographically distributed) teams and be subject to the scrutiny of regulators. These considerations lead to a more detailed set of academically novel sub-challenges that can be categorised as “Essential” and “Accidental” (a useful philosophical distinction that is attributed to Aristotle).

5.1.1 Essential Challenges

An essential challenge is one that is a natural consequence of the overall goal; one that needs to be necessarily considered when undertaking and evaluating the research. The individual essential challenges arising from the problem of High-Integrity Software Product Line deployment are detailed as follows:

1. Scale & Size of Product

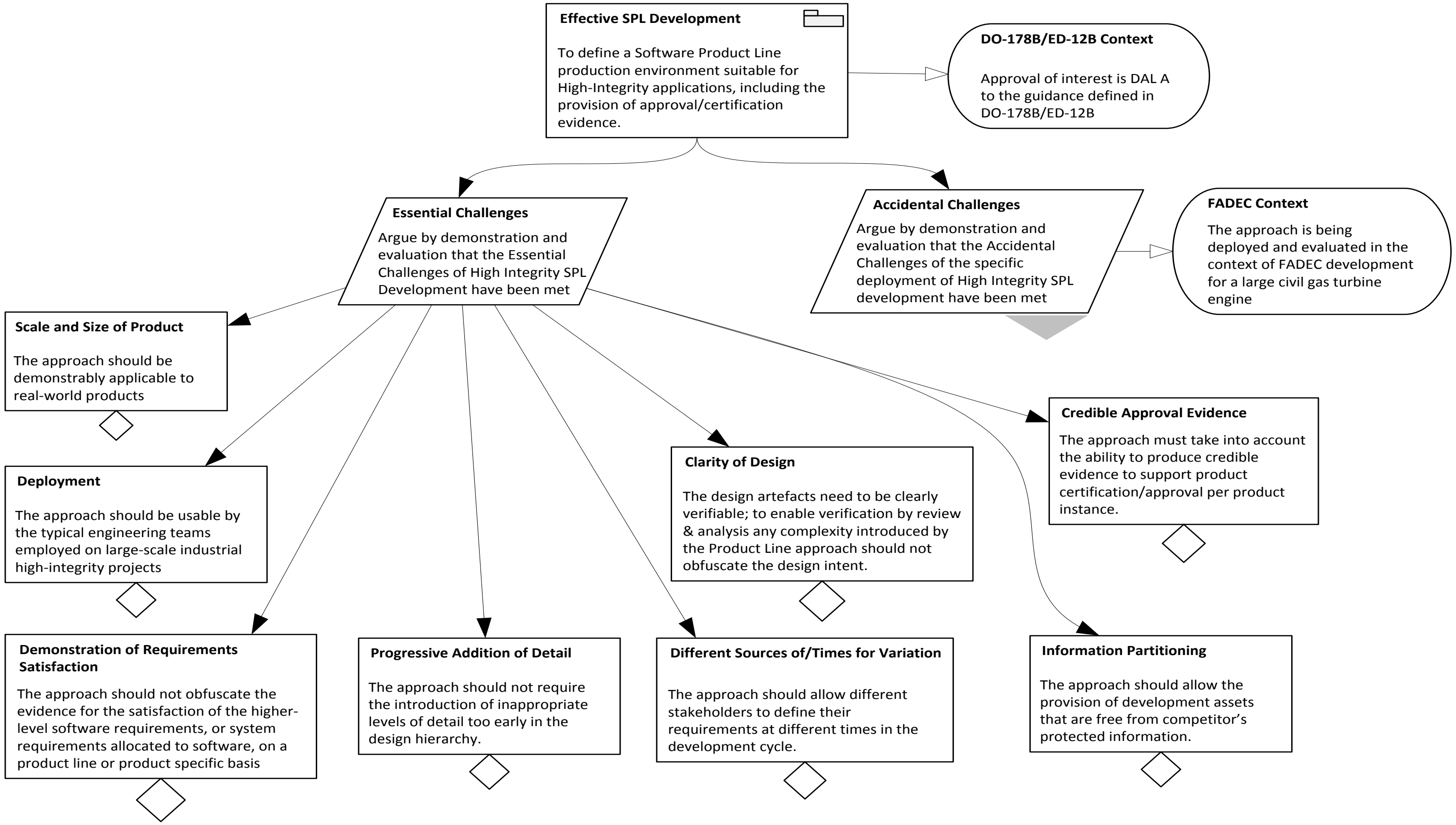
   The approach should be demonstrably applicable to real-world products (of typically > 100 kSLOC) and not be restricted to a small-scale prototype. (Although a prototype may be sufficient to demonstrate concepts, scalability can be difficult to argue; demonstration is preferable.)

2. Deployment into “typical” industrial teams

   The approach should be usable by the typical engineering teams employed on large-scale industrial high-integrity projects. Such projects can use a significant proportion of sub-contract and offshore labour, with a wide variety of experience and skills. Therefore usability needs to be argued and (preferably) demonstrated. We use an approach of appealing to previously successful methods/techniques and active demonstration and evaluation of the new technology to address this.

3. Enabling the demonstration of requirements satisfaction (validation, traceability, basic integrity)

   The approach should not obfuscate the evidence for the satisfaction of the higher-level software requirements, or system requirements allocated to software, on a product line or product specific basis. This is one of the crucial research objectives, and has not been demonstrated previously in the literature.



Effective SPL Development

To define a Software Product Line production environment suitable for High-Integrity applications, including the provision of approval/certification evidence.

DO-178B/ED-12B Context

Approval of interest is DAL A to the guidance defined in DO-178B/ED-12B

Essential Challenges

Argue by demonstration and evaluation that the Essential Challenges of High Integrity SPL Development have been met

Accidental Challenges

Argue by demonstration and evaluation that the Accidental Challenges of the specific deployment of High Integrity SPL development have been met

FADEC Context

The approach is being deployed and evaluated in the context of FADEC development for a large civil gas turbine engine

Scale and Size of Product

The approach should be demonstrably applicable to real-world products

Deployment

The approach should be usable by the typical engineering teams employed on large-scale industrial high-integrity projects

Credible Approval Evidence

The approach must take into account the ability to produce credible evidence to support product certification/approval per product instance.

Clarity of Design

The design artefacts need to be clearly verifiable; to enable verification by review & analysis any complexity introduced by the Product Line approach should not obfuscate the design intent.

Demonstration of Requirements Satisfaction

The approach should not obfuscate the evidence for the satisfaction of the higher-level software requirements, or system requirements allocated to software, on a product line or product specific basis

Progressive Addition of Detail

The approach should not require the introduction of inappropriate levels of detail too early in the design hierarchy.

Different Sources of/Times for Variation

The approach should allow different stakeholders to define their requirements at different times in the development cycle.

Information Partitioning

The approach should allow the provision of development assets that are free from competitor's protected information.

Figure 55 Essential Challenges of High-Integrity Software Product Line Development Practices

4. Progressive addition of detail

   The approach should not require the introduction of inappropriate levels of detail too early in the design hierarchy (c.f. MDA approaches that make use of Platform Independent Models (PIM) and Platform Specific Models (PSM) to introduce the details of the target implementation platform at the appropriate level in the design decomposition/hierarchy [124].) Again, there is novelty here; most Product Lines approaches do not include multiple levels of abstraction or hierarchy in the Feature Models/Variability Models (see Figure 56).

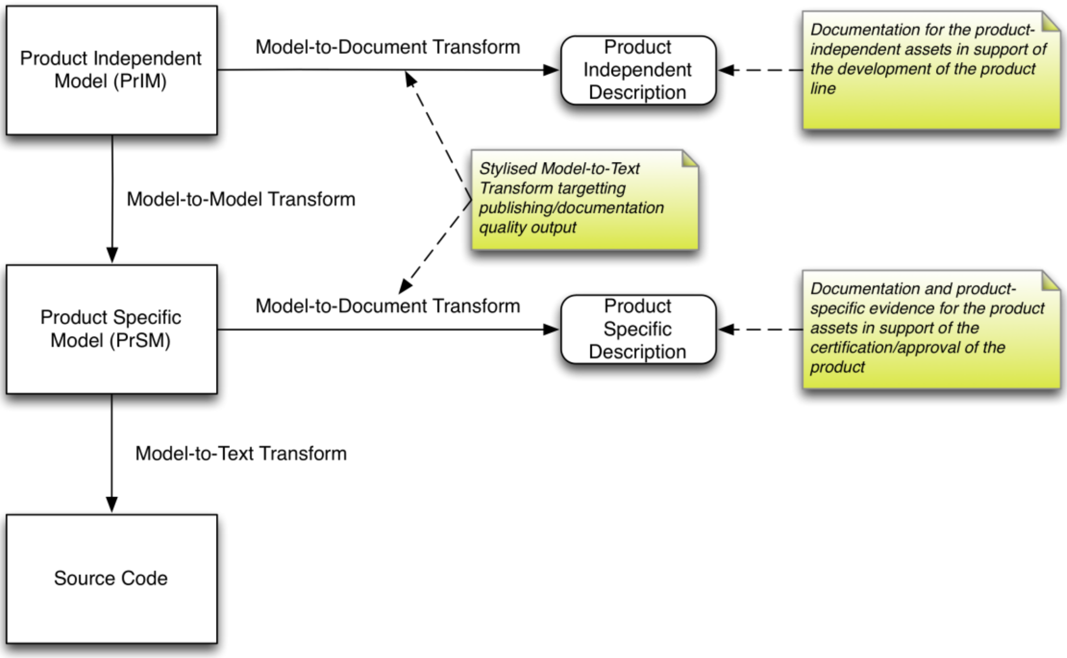

Product Independent Model (PrIM)
Model-to-Document Transform
Product Independent Description
Documentation for the product-independent assets in support of the development of the product line
Model-to-Model Transform
Stylised Model-to-Text Transform targetting publishing/documentation quality output
Product Specific Model (PrSM)
Model-to-Document Transform
Product Specific Description
Documentation and product-specific evidence for the product assets in support of the certification/approval of the product
Model-to-Text Transform
Source Code

FIGURE 56 "MDA-STYLE" ARTEFACTS AND TRANSFORMS SUPPORTING PRODUCT-INDEPENDENT AND PRODUCT-SPECIFIC ASSETS

5. Clarity of design

   The design artefacts need to be clearly verifiable; to enable verification by review & analysis any complexity introduced by the Product Line approach should not obfuscate the design intent. This is significantly impacted by the variability approach chosen. (As discussed in Chapter 2, many product line approaches use a “positive variability” approach (e.g. CVL[30]) which is analogous to aspect-oriented development. These rely on a base “common” artefact augmented with separate “advice” to provide the variable aspects.) There does not appear to be any literature containing a critical review of variability strategies with respect to their impact on verification by analysis, review or test. This is a novel aspect of the work described here.

6. Allowing different sources/drivers of variation at different times in the development

   Many product line approaches rely on the up-front development of feature models to direct the development of implementation assets. This is analogous to a "waterfall" approach that is not always possible or desirable when developing the classes of system of interest here. Typically feature requirements for such systems are sourced from many different stakeholders, who have different (and asynchronous) product development cycles (for example, engine manufacturers, airframe manufacturers, system design engineers, electronics design engineers – see Figure 57.) If this is not recognised in the software development processes then it can be a major source of requirements volatility and instability. We address this by providing levels of indirection and abstraction between system-level feature models and implementation components containing decision contracts. The recognition of this problem and the use of this abstraction mechanism to allow deferred requirements and design decisions is novel to the research described here.

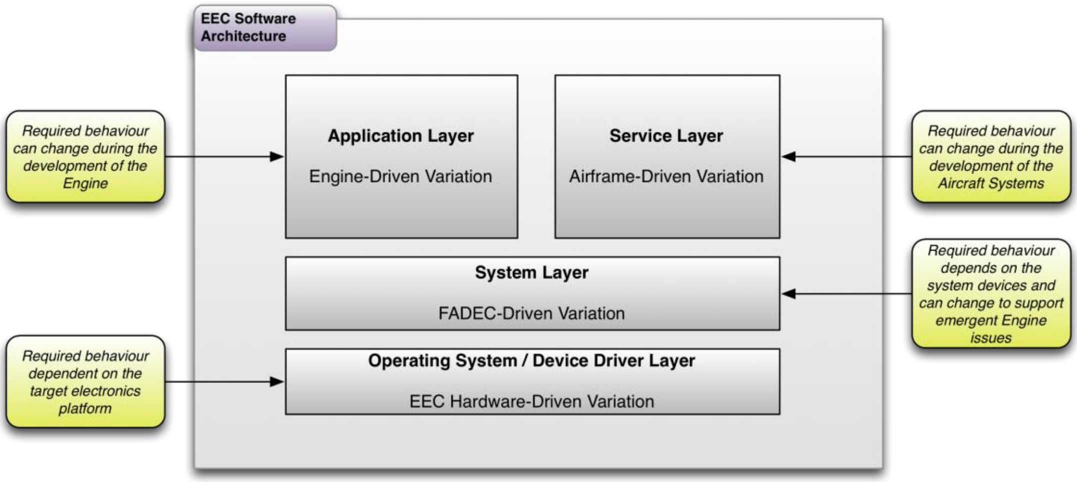

EEC Software Architecture
Application Layer
Engine-Driven Variation
Service Layer
Airframe-Driven Variation
System Layer
FADEC-Driven Variation
Operating System / Device Driver Layer
EEC Hardware-Driven Variation
Required behaviour can change during the development of the Engine
Required behaviour dependent on the target electronics platform
Required behaviour can change during the development of the Aircraft Systems
Required behaviour depends on the system devices and can change to support emergent Engine issues

FIGURE 57 EEC SOFTWARE LAYERED ARCHITECTURE AND SOURCES OF CHANGE/VARIATION

7. Credible Certification/Approval

   To make an economic argument for product lines, the chosen approach must take into account the ability to produce credible evidence to support product certification/approval. In addition, current regulations do not recognise product line approaches. The detail of the certification guidance is defined for a single system being subject to certification approval. Any product line evidence has to be shown to be applicable and credible from the viewpoint of the product instance being approved.

8. Information Partitioning/“Chinese Walls”

   Protection of intellectual property is increasingly important in industry, particularly when dealing with customer-sourced or export-controlled data. This was discussed in detail in section 4.6.1. Note the military helicopter product line described by Dordowsky et al. [93] had a restriction on the contents of an asset “in order to comply to non-disclosure and customer relevance principles”.

We set out above the engineering challenges for the use of product lines. The analysis of the essential problems shows that the intellectual/academic challenge is essentially to support product lines so as to **enable cost-effective certification/approval** (including providing evidence) and to **support multiple stakeholders with different development/instantiation cycles** (challenges 3,5,6,7).

It is further noted that there are constraints of the engineering setting (1,2,4,8) which need to be viewed as constraints in solving the above problems.

5.1.2 Accidental Challenges

In this context, an accidental challenge is one that is a consequence of the particular industrial environment used to deploy and evaluate the approach. Whilst these are not necessarily a direct result of the overall research challenge, they are still important aspects of the work as they are typical examples of the problems faced by academic-to-industrial technology transfer. (It would be interesting to explore the extent to which failures of academic technology transfer are in fact failures to recognise the importance of accidental properties.)

Most organisations and individual engineers are change-averse; an evolutionary approach to process change and improvement is generally preferred to a revolutionary process shift. Being sensitive to an evolutionary change results in a set of challenges and constraints that are enablers to the successful adoption of the approach rather than being required to deploy and evaluate the fundamental research.

We describe below the accidental challenges that arise from the deployment of the approach on the Large Civil FADEC product line:

1. The constraint of the use of UML models for software architecture and design

   The product line processes and design notations needed to be make use of/extend UML to be acceptable to the development organisation. The two most recent projects undertaken by the development organisation used UML class models to develop the software architecture, and automatic generation of the code structure from these models. This means that there was significant level of experience in using class modelling techniques, and a legacy of artefacts that could potentially be “harvested” to ease the transition to a product line approach.

2. The coinstraint of the use of SPARK as the target implementation language

   The product line instantiation processes needed to target SPARK as the implementation language to be acceptable to the development organisation. The development organisation had significant experience in using SPARK as the target implementation language, with more than 5 SPARK applications being successfully developed and approved to DO-178B/ED-12B Level A.

3. Hard Real-Time, Embedded Constraints

   Typically, high-integrity avionics applications are deployed as embedded systems that have the additional challenge of meeting hard real-time constraints. Any design or development approach that adds a temporal (or, to a lesser extent, spatial) overhead to the software system is discouraged. (CPU occupancy is usually at a premium on such systems, and any approach that would increase CPU utilisation is not acceptable.)

4. Restrictions on the available (incumbent) development environments/tools (e.g. ARTiSAN Studio)

   Organisations are naturally averse to any unnecessary expenditure on IT, and wish to maximise their investment in the development environment they already use. As the development organisation had a significant investment in the ARTiSAN Studio tool to support their existing UML modelling and code generation, the product line approach had to work within this constraint.

5. Custom and practice, customer expectations

   Significant (and often unstated/assumed) non-functional requirements can come from both customers and "development stakeholders" (i.e. indirect customers that make use of the system during development, for example engine test and flight test engineers.) Typically, they have requirements for tuning and calibrating the behaviour of the system via separately loadable/settable data values (e.g. Development Variables (DVs), Data "Trims"[4], Data Entry Plug (DEP) etc.), and have

[4] "*A **trimmer** or **preset** is a miniature adjustable electrical component. It is meant to be set correctly when installed in some device, and never seen or adjusted by the device's user*." (From Wikipedia). As control systems transitioned from analogue electronic to discrete software implementation, they retained much of the historic terminology. Therefore, the altering of a nominally constant data value to "tune" the system response is known as "trimming the software" in the same way as adjusting a miniature potentiometer to tune an analogue electronic controller was known as **trimming**.

test and monitoring equipment for such purposes that are used on many projects. Such externally driven compatibility requirements can be easily overlooked, but must be adhered to when instantiating products from the product line.

6. Project management strategies

   Whereas certification evidence is required to support the version of the system that is presented for certification approval, multiple prior development versions of FADEC systems need to be delivered to customers to support the wider engine and aircraft development programmes. This means that the system and software development programme is managed, resourced and scheduled to support incremental development. The chosen product line approach must be able to support the development and deployment of incremental functionality, typically with a sub-set of components in the first instances; otherwise, it will not meet the organisational need to support its customer’s programme.

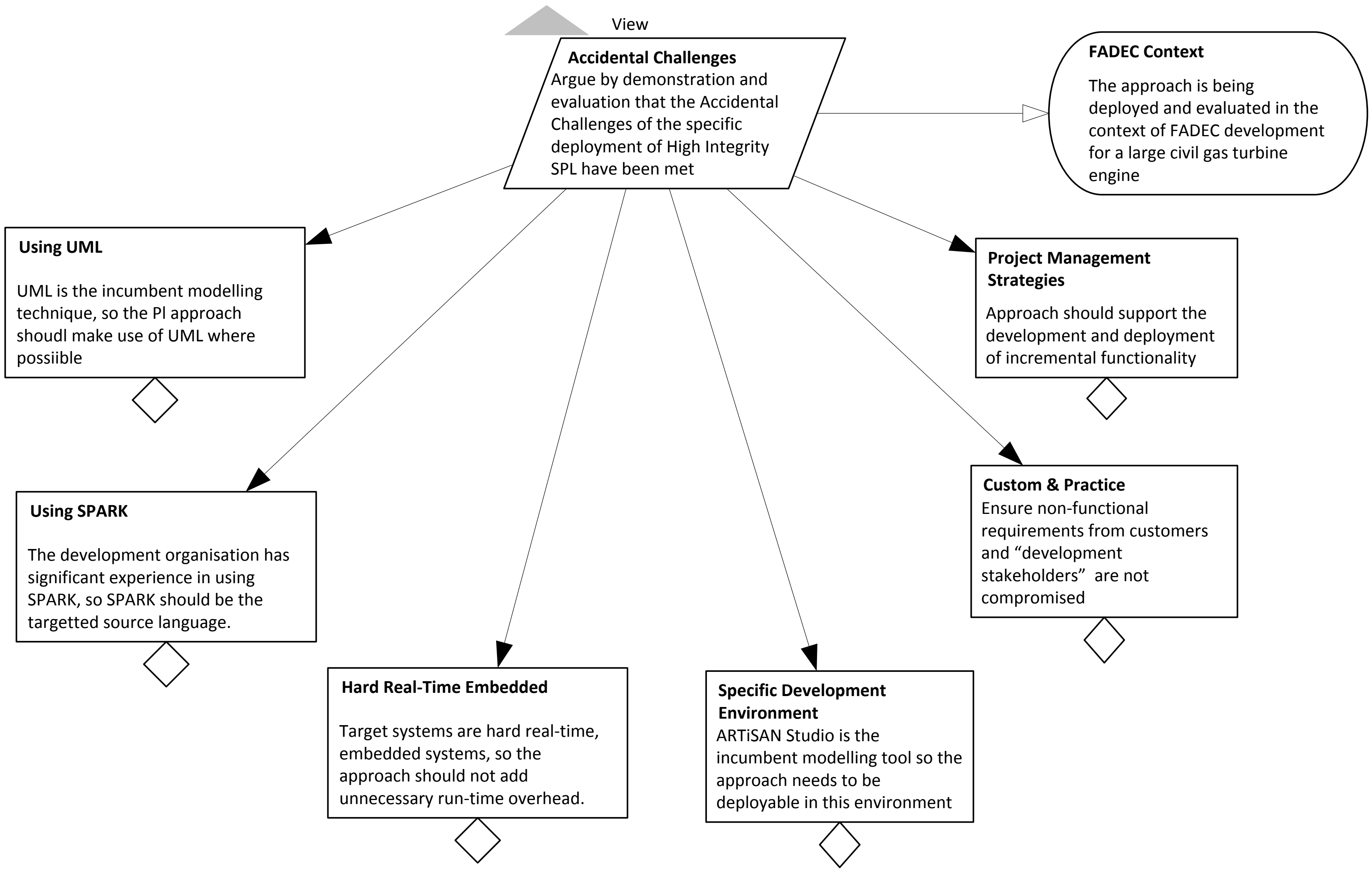

View
Accidental Challenges
Argue by demonstration and evaluation that the Accidental Challenges of the specific deployment of High Integrity SPL have been met
FADEC Context
The approach is being deployed and evaluated in the context of FADEC development for a large civil gas turbine engine
Using UML
UML is the incumbent modelling technique, so the Pl approach shoudl make use of UML where possiible
Project Management Strategies
Approach should support the development and deployment of incremental functionality
Using SPARK
The development organisation has significant experience in using SPARK, so SPARK should be the targetted source language.
Custom & Practice
Ensure non-functional requirements from customers and "development stakeholders" are not compromised
Hard Real-Time Embedded
Target systems are hard real-time, embedded systems, so the approach should not add unnecessary run-time overhead.
Specific Development Environment
ARTiSAN Studio is the incumbent modelling tool so the approach needs to be deployable in this environment

Figure 58 Accidental Challenges of High-Integrity Software product Line Development Practices

5.2 Solution Strategy

We have articulated the essential and accidental challenges posed by the desire to apply a Software Product Line approach to a set of high-integrity software systems. The following section describes the approach selected to develop the product line and create the product instances. We begin by discussing model transformation and how it may be used to instantiate products when using a model-based approach to software development.

We discussed the use of model transformation technology to instantiate product lines in Chapter 2; this included a discussion on the types/taxonomy of model transformations. Here we discuss the design of a set of endogenous, horizontal model-to-model transformations and a final model-to-text transformation that realise the complete product instantiation transformation.

5.2.1 Transformation Technology

In this section, we provide background on the model transformation tools chosen to implement the product line transformations. We briefly introduce and describe the "mechanics" of producing a transformation using the chosen environment; however, the novelty is in the design of the transformations that are encoded using this technique, and this is discussed in section 5.3 onwards.

We required a model-to-model transformation technology that had the following characteristics:

- Deterministic
- Declarative
- Endogenous
- Suitable for repeated application
- Extensible
- Can be augmented with a Model-To-Text transformation

A number of model transformation languages were available, or have been developed over the duration of the research project described in this thesis. Many of these languages, such as ATL (Atlas Transformation Language) , ETL (Epsilon Transformation Language), Operational QVT (Query/View/Transform), have concrete implementations based on the Eclipse Modelling Framework, requiring the underlying models to be MOF/EMF compliant. Whilst it is perfectly possible to implement the transformations described in this thesis using EMF-compliant tools, this was made difficult due to the constraints described earlier (requiring the research to be undertaken using the incumbent modelling tools in use in the sponsoring organisation, which made extensive use of the Artisan Studio UML tool.) Models developed in Artisan Studio are not easily interchangeable with the EMF framework (the provided XMI interchange being both unreliable and lossy with respect to the required model elements.)

Therefore the model transformation technology chosen to realise the product line instantiation was the ACS/TDK (Automatic Code Synchronisation/Template Development Kit) "4G" technology from Atego (formerly ARTiSAN). The ACS/TDK toolset provides the basis for the model-to-text code generation and round-trip model and code development extensions to the ARTiSAN Studio UML environment. The "4G" version of ACS/TDK augmented this with the ability to perform Model-to-Model transformation.

The decision to use ACS/TDK 4G (hereafter known as TDK) was primarily driven by the need to develop an instantiation process that could be used for real on a large, multi-developer avionics project. ARTiSAN Studio was the incumbent modelling tool used on the projects that formed the baseline for the product line development activities and there was a substantial investment in tool licenses, existing product models and user knowledge.

The previous projects used a UML to SPARK code generator that was implemented using OCS (On-Demand Code Synchronisation). OCS is a simple template-based Model-To-Text code generation engine. OCS scripts are developed in a language called SDL and are interpreted by the Studio environment on-demand. The customised OCS SPARK generator makes use of Ada and SPARK profiles which extend the UML class models to capture Ada and SPARK-specific concepts. This approach was used effectively on two large avionics projects (approximately 250K SLOC each). (The SPARK OCS generator was originally produced by ARTiSAN (now Atego), customised by Altran Praxis and subsequently by the author.)

However, OCS was not suitable for development of the product line transformation and code generation for a number of reasons. Firstly, the OCS product had been deprecated by Artisan and replaced by the ACS generator engine. Secondly, OCS had no model-to–model transformation capabilities. However, legacy OCS generators can be ported to/hosted within ACS-based generation schemes. This capability meant that it was easy to create the back end model-to-text transformation shown in Figure 60 from the OCS baseline and this had a degree of provenance from previous project use. The effort could therefore be spent on developing the product line transformation rather than replicating a pre-existing code generator.

In contrast to the interpreted-SDL approach of the OCS generator, ACS generators are compiled to Win32 DLLs and executed either on demand or as part of a continuous generation approach. ACS generators can run in the background during a modelling session and continuously generate code in response to changes in the source model. Round-tripping is also supported where model elements can be created in response to external changes to the source code. However, in the context of high-integrity software development, the generator is used exclusively in forward–engineering mode. (Back-documentation or reverse-engineering of design information from code is not an acceptable high-integrity development practise.)

A specific ACS generator DLL is produced by designing a generator model using the Studio UML tool (augmented with the TDK development kit). A special version of ACS is then used on the generator model to auto-generate the generator code and DLL.

5.3 Implementing SPL Transformations

This section contains an overview of the model transformation designed to instantiate products from the product line. A full description of the transformation design is contained in Appendix B, which also contains an overview of the "TDK 4G" environment used to develop the transformations.

5.3.1 Realising Model Transformation for High-Integrity Product Lines

The overall model transformation process used to instantiate products from the product line is illustrated in Figure 60.

The initial M2M transformation takes the deployment model and produces a product specific model that has all the variation points resolved based upon the selected decision options. This model is then used by downstream transformations to produce the product-specific source code and supporting development artefacts.

Once the reference architecture and product line components have been developed, product instances can be created. Instantiation of products is achieved by the deployment of the appropriate components in a copy of the reference architecture model and the selection of the appropriate decision options for each component (either directly, or as the result of a higher-level feature model selection). Once the components are deployed and the decision options are resolved, then product-specific assets can be generated using model transformation.

5.3.1.1 Model-to-Model Transformation 1 – Reductive Product Line to Product Model Transform

The TDK 4G model transformation environment allows a transform to be described as a declarative class model. Here we describe the form of the class model that describes the product line to product instance reductive transformation

Figure 61 shows the complete transform class model (the detail of which is contained in Appendix B). The instantiation transformation essentially performs the following algorithm:

```
for each component included in the deployment model:
      follow the bind link to the catalogue component;
      for each model element in the catalogue component:
         if it is a variation point then
            if selection expression evaluates True then
              duplicate into deployment model;
            end if;
         else
            duplicate into deployment model;
         end if;
    end for;
end for;
```

The result of this transformation is a complete product specific model under the deployment model "root" which can be passed to the downstream transformations.

The transformation model is built up from a network of associated "Search" classes to isolate the meta-model elements that may exhibit variability. Once these elements are isolated, the selection expressions that guard the inclusion of that element are evaluated for the particular decision options selected for the particular product. Successful evaluation of the expression triggers the duplication of that element into the product line model. Common meta-model elements (i.e. those not stereotyped as variation points) are always duplicated into the target model.

To understand the transformation performed we have to refer back to the decision contract meta- model we introduced in Chapter 4 (shown again here in Figure 59)

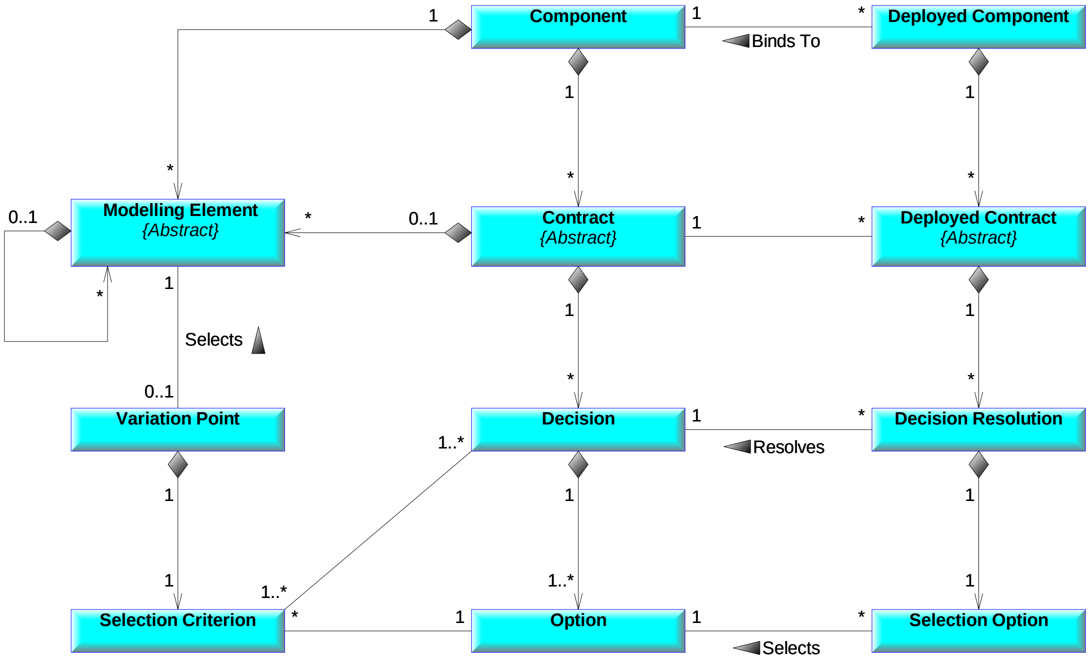
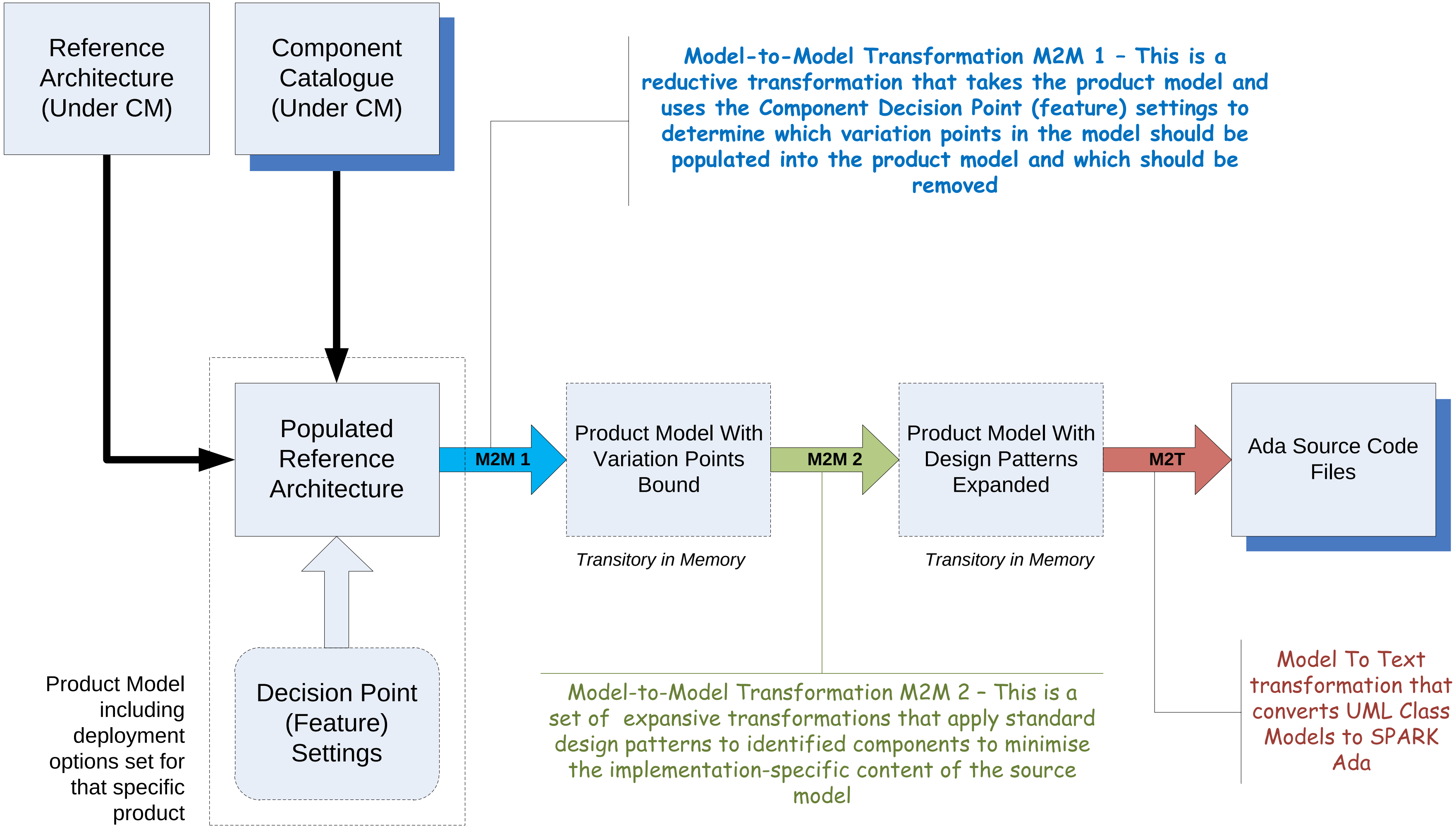


FIGURE 59 PRODUCT LINE META-MODEL USING DECISION CONTRACTS (FROM CHAPTER 4)


Reference Architecture (Under CM)
Component Catalogue (Under CM)
Populated Reference Architecture
Decision Point (Feature) Settings
Product Model including deployment options set for that specific product
M2M 1
Product Model With Variation Points Bound
Transitory in Memory
M2M 2
Product Model With Design Patterns Expanded
Transitory in Memory
M2T
Ada Source Code Files

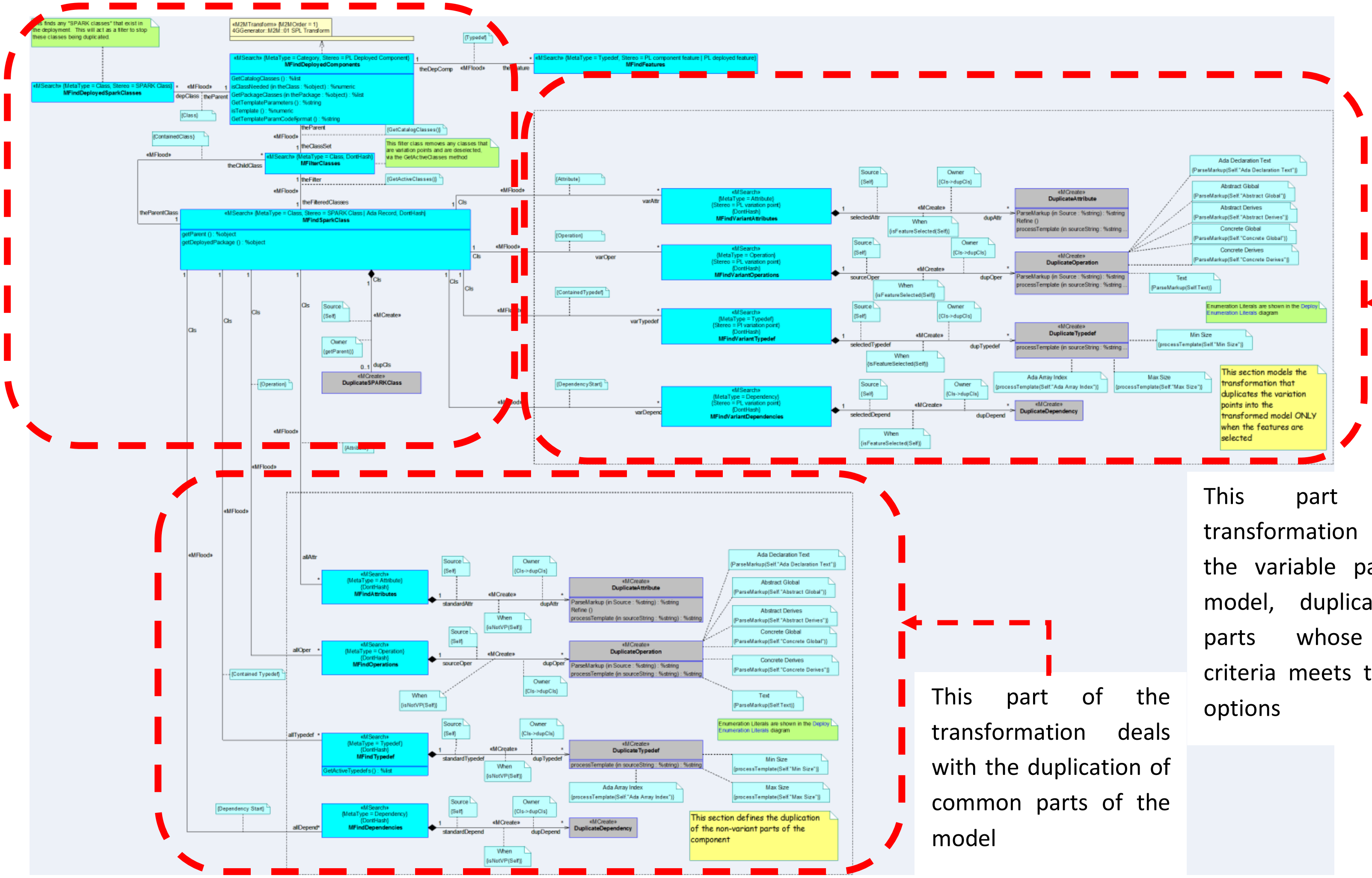
Model-to-Model Transformation M2M 1 – This is a reductive transformation that takes the product model and uses the Component Decision Point (feature) settings to determine which variation points in the model should be populated into the product model and which should be removed

Model-to-Model Transformation M2M 2 – This is a set of expansive transformations that apply standard design patterns to identified components to minimise the implementation-specific content of the source model

Model To Text transformation that converts UML Class Models to SPARK Ada

FIGURE 60 PRODUCT INSTANCE SPARK CODE GENERATION FROM REFERENCE ARCHITECTURE AND PRODUCT LINE COMPONENTS

This part of the transformation selects the classes in the model to which the transformation is applied



This part of the transformation deals with the variable parts of the model, duplicating those parts whose selection criteria meets the selected options

This part of the transformation deals with the duplication of common parts of the model

FIGURE 61 OVERVIEW OF STRUCTURE OF MODEL-TO-MODEL TRANSFORMATION 1 CLASS MODEL

The UML is extended via a special product line UML profile to realise this meta-model. Figure 62 shows a model of this profile.

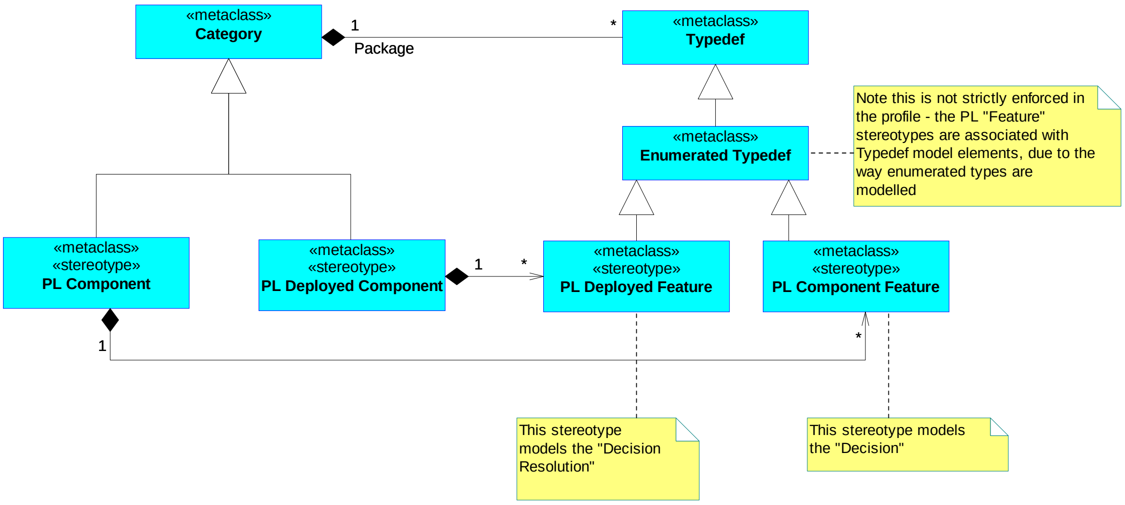

«metaclass»
Category
1
Package
*
«metaclass»
Typedef
«metaclass»
Enumerated Typedef
Note this is not strictly enforced in the profile - the PL "Feature" stereotypes are associated with Typedef model elements, due to the way enumerated types are modelled
«metaclass»
«stereotype»
PL Component
«metaclass»
«stereotype»
PL Deployed Component
1
*
«metaclass»
«stereotype»
PL Deployed Feature
«metaclass»
«stereotype»
PL Component Feature
1
*
This stereotype models the "Decision Resolution"
This stereotype models the "Decision"

FIGURE 62 PL PROFILE MAPPING TO UML META-MODEL ELEMENTS

At this point in the transformation process there now exists a model in memory that represents the deployed component set with all variations points resolved. This now needs to be transformed into a model from which SPARK Ada can be generated. This is achieved by applying a set of design pattern transformations.

5.3.2 Opaque Behaviour and Textual Transformation

As previously identified, a major issue for this approach is that to be complete, component and system models can contain important elements which are not compliant with any defined meta-model. These include many of the operation design elements which have textual or graphical content that is included from other modelling environments; code inserts generated in IDEs or text editors; and informal content from word processing and diagramming tools (in UML terms these are examples of "opaque behaviour" i.e. whose form is not described by the UML meta-model). However, to deliver successfully a complete and correct product-specific component, transformations have to be able to identify and manipulate such content to be able to deliver the required variability. Whenever the transformation engine identifies this type of content, the M2M transformation delegates this to a processor that deals with the opaque behaviour. Figure 63 shows this diagrammatically.

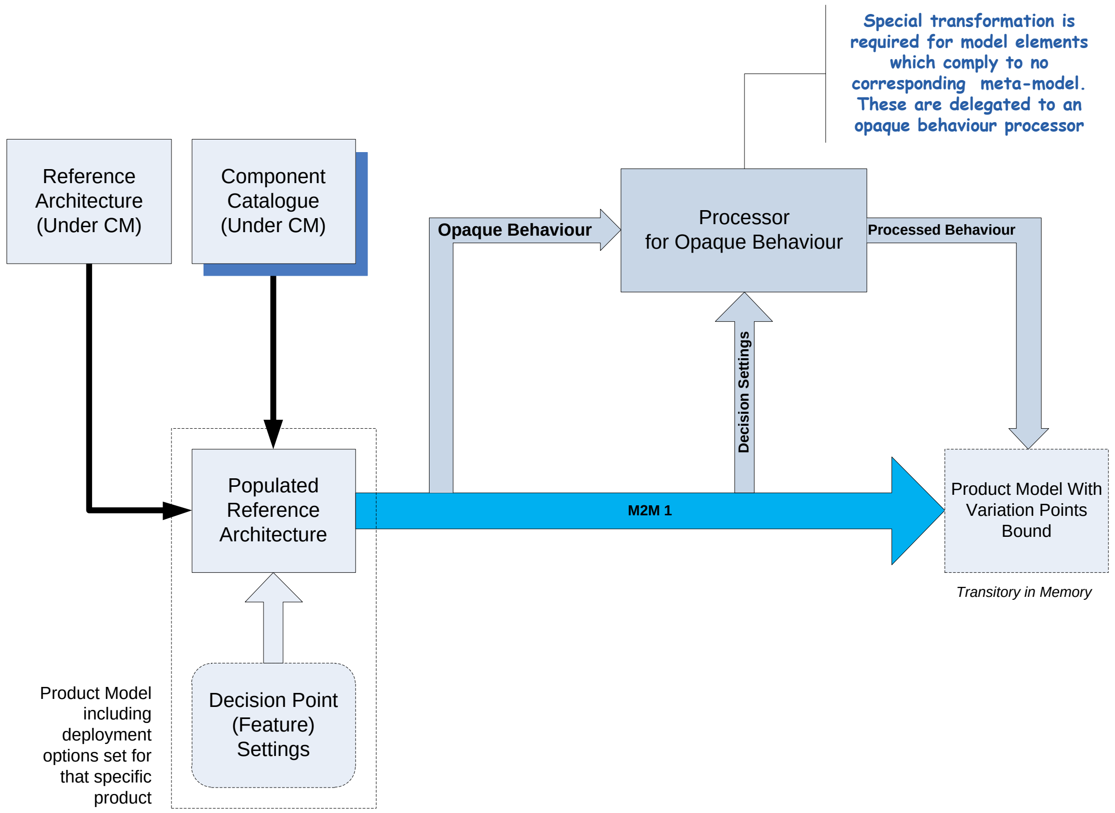

Special transformation is required for model elements which comply to no corresponding meta-model. These are delegated to an opaque behaviour processor
Reference Architecture (Under CM)
Component Catalogue (Under CM)
Opaque Behaviour
Processor for Opaque Behaviour
Processed Behaviour
Decision Settings
Populated Reference Architecture
M2M 1
Product Model With Variation Points Bound
Transitory in Memory
Decision Point (Feature) Settings
Product Model including deployment options set for that specific product

Figure 63 Auxiliary transformations For Model Elements Not Compliant With A Defined Meta-Model (UML Opaque Behaviour)

We can see this in use in the UML operation duplication shown in Figure 64 below. The instantiated operation duplication factory class DuplicateOperation is decorated with constraints to populate elements of the duplicated operation. These elements are modelled as text fields with mark-up to denote the common and variable parts. The internals of the ParseMarkup transformation operation call out to an ANTLR parser [125] that removes the unwanted variation from the text string and returns the product-specific string.

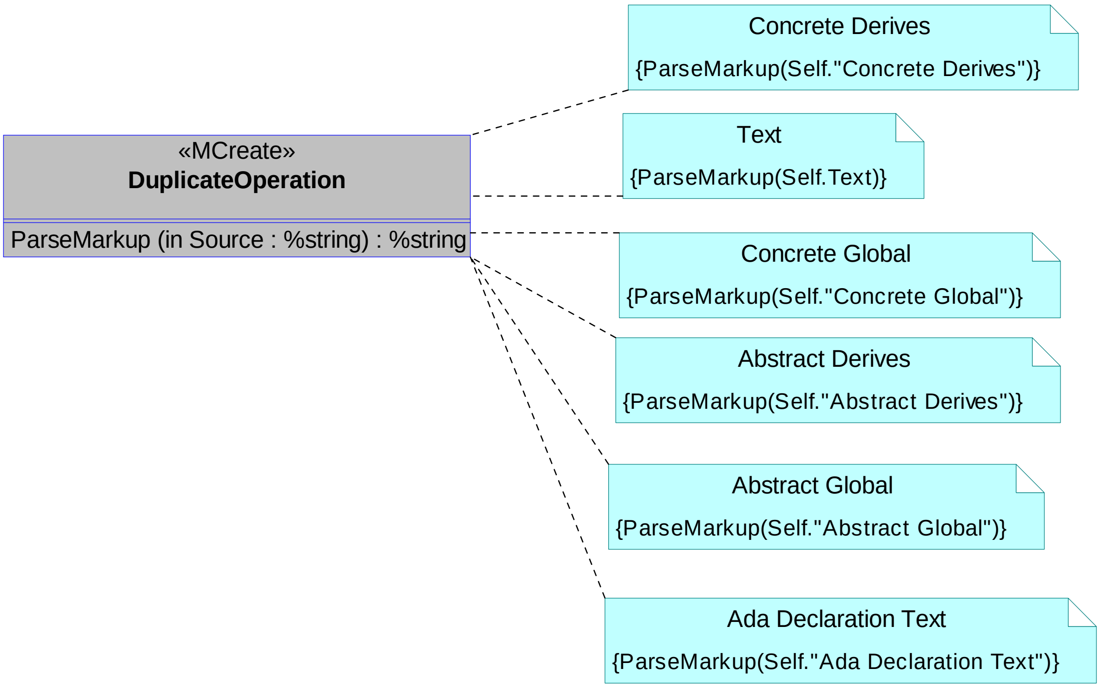

«MCreate»
DuplicateOperation
ParseMarkup (in Source : %string) : %string
Concrete Derives
{ParseMarkup(Self."Concrete Derives")}
Text
{ParseMarkup(Self.Text)}
Concrete Global
{ParseMarkup(Self."Concrete Global")}
Abstract Derives
{ParseMarkup(Self."Abstract Derives")}
Abstract Global
{ParseMarkup(Self."Abstract Global")}
Ada Declaration Text
{ParseMarkup(Self."Ada Declaration Text")}

FIGURE 64 INVOKING THE TEXT TRANSFORMATION FOR NON-META-MODEL COMPLIANT MODEL ELEMENTS

The mark-up language used has a very simple set of keywords, expressions and region markers which allows the identification of regions of text as being common or variable, and provide the expressions which identify whether the region is required in a product variant.

Text strings containing mark-up take the following form:

```
VPBegin
{
       Some common text
}
VPIf DECISION_EXPRESSION
{
       Some variant text
}
VPEndif
{
       Some more common text
}
VPEnd
```

CODE LISTING 1 EXAMPLE VARIATION TEXT 1

The result of parsing this text with DECISION_EXPRESSION evaluating true would yield the following:

```
Some common text
Some variant text
Some more common text
```

The result of parsing this text with DECISION_EXPRESSION evaluating false would yield the following:

```
Some common text
Some more common text
```

The conditional constructs can also include VPElse and VPElsif to provide default and guarded alternatives.

The expression language is a simple set of <decision> = <value> pairs which can be combined with Boolean operators AND and OR to produce more complex selection expressions.

5.3.2.1 Worked Example of Text Transformation

Figure 65 below shows the top-level structure of an IgniterControl component with the main component class expanded to show the run operation, and the set of five decisions that make up the component's variation contract. Each of the five decisions are Boolean selections (true, false).

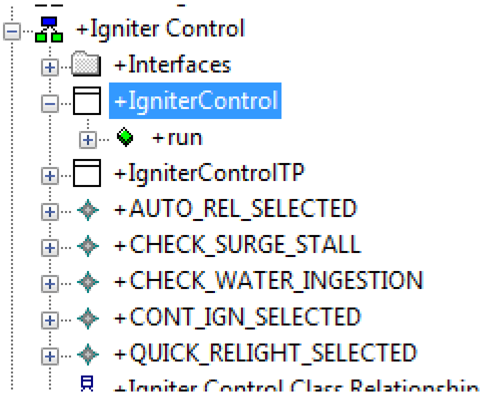


FIGURE 65 IGNITER CONTROL COMPONENT SHOWING DECISIONS

If we now look at the body of the run operation, we can see the use of the mark-up language to include/exclude regions of SPARK code in response to each of the decisions in this component. Code Listing 2 contains the body of the operation and it can be seen that the code is split into three distinct regions. The start and end of the operation contains code common to all variants. For each of the five decisions there are individual regions that can include or exclude the code specific to that decision.

```
VPBegin
{-- Determine whether dual ignition is required

If IThrustControl.Get.fuelDipIgnInhibit.data    or IThrustControl.Get.primingInProgress.data then

   -- Fuel dip or priming requires all ignition to be inhibited
   lclDualIgnCmd   := FALSE;
   lclSingleIgnCmd := FALSE;

else

   lclDualIgnCmd := IStarting.Get.igniterCmdAutostart.data = CommonRecTypes.HIGH_IGN or else
                    IStarting.Get.igniterCmdManualStart.data = CommonRecTypes.HIGH_IGN;
}
VPIf AUTO_REL_SELECTED = TRUE
{
   if IAutoRelight.Get.igniterCmdAutoRel.data = CommonRecTypes.HIGH_IGN then
      lclDualIgnCmd := TRUE;
   end if;
}
VPEndIf

VPIf QUICK_RELIGHT_SELECTED = TRUE
{
   if IStarting.Get.igniterCmdQuickRel.data = CommonRecTypes.HIGH_IGN then
      lclDualIgnCmd := TRUE;
   end if;
}
VPEndIf

VPIf CONT_IGN_SELECTED = TRUE
{
   if IThrustControl.Get.igniterCmdFuelDip.data = CommonRecTypes.HIGH_IGN then
      lclDualIgnCmd := TRUE;
   end if;
}
VPEndIf
```

```
VPIf CHECK_SURGE_STALL = TRUE
{
   if IEngineEvents.Get.igniterCmdSurgeStall.data = CommonRecTypes.HIGH_IGN then
      lclDualIgnCmd := TRUE;
   end if;
}
VPEndIf
VPIf CHECK_WATER_INGESTION = TRUE
{
   if IEngineEvents.Get.igniterCmdWaterIngest.data = CommonRecTypes.HIGH_IGN then
      lclDualIgnCmd := TRUE;
   end if;
}
VPEndIf
{
 -- for a normal start, set single ignition unless dual ignition has already been selected
   lclSingleIgnCmd := not lclDualIgnCmd and
                      (IStarting.Get.igniterCmdAutostart.data = CommonRecTypes.LOW_IGN or
                       IStarting.Get.igniterCmdManualStart.data = CommonRecTypes.LOW_IGN);

end if;
-- Write ignition request for use by the actuation function.
if lclDualIgnCmd then
   IIgniters.Put.igniterCmd (data => CommonRecTypes.IgnitionLevelRecType'(data => CommonRecTypes.HIGH_IGN,
                                                                          flt  => FALSE));
elsif lclSingleIgnCmd then
   IIgniters.Put.igniterCmd (data => CommonRecTypes.IgnitionLevelRecType'(data => CommonRecTypes.LOW_IGN,
                                                                          flt  => FALSE));
else
   IIgniters.Put.igniterCmd (data => CommonRecTypes.IgnitionLevelRecType'(data => CommonRecTypes.NO_IGN,
                                                                          flt  => FALSE));
end if;
}
VPEnd
```

Code Listing 2 Example of use of mark-up to insert variation in code bodies

5.3.3 Template Components & Transformation

During the development of the product-line component designs, it became apparent that the generative programming/M2M transformation approach described earlier in this chapter was not catering for a significant type of variability. This involved components that contained similar or identical functionality but operated on different signals and data sets. Typically, this would be handled in the small by producing parameterised library or utility components. However, it was found that there were significant areas of repeated or "cloned" functionality for which the overhead of parameterising the interfaces would significantly degrade the run-time performance of the component. In addition, the data-types of the parameters may differ between instantiations of the cloned code, making strong-typing of the component contract difficult.

This type of problem is handled in Ada by the use of the Generic mechanism; however, Ada Generics were not supported by SPARK[5]. Therefore, we extended the transformation to provide a mechanism allowing components to be **templated**.

5.3.3.1 Use of Template Components

Template components are deployed in the same manner as "standard" product line components: a "deployed" component is included in the deployment model and is related back to the product line component using a "bind" association. Template components are modelled in terms of their formal template parameters, which are shown in class diagrams as additional decoration on the package icons (see Product Line Component in Figure 66).

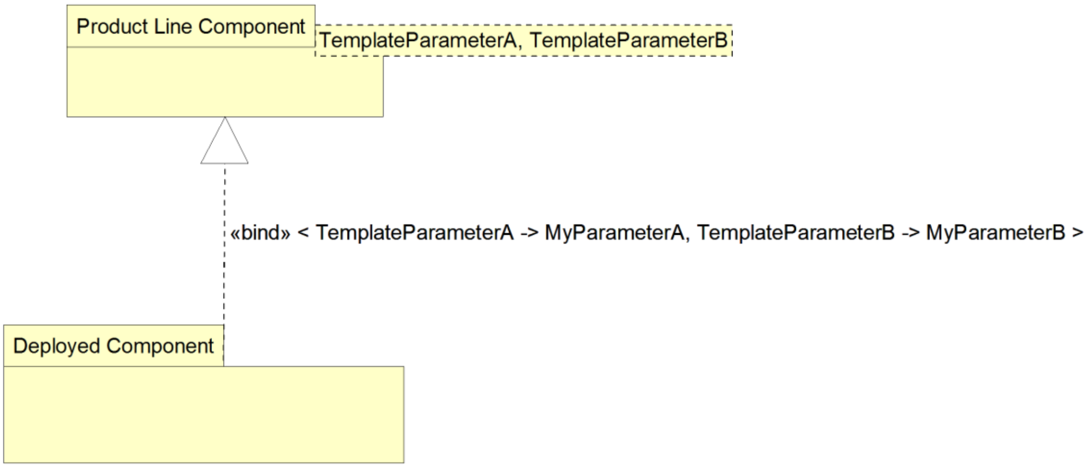

Product Line Component
TemplateParameterA, TemplateParameterB
«bind» < TemplateParameterA -> MyParameterA, TemplateParameterB -> MyParameterB >
Deployed Component

FIGURE 66 SIMPLE EXAMPLE OF TEMPLATE COMPONENT DEPLOYMENT

[5] SPARK did not support Ada generics in 2010 when this generator was implemented. However, as of the 10.1 release of SPARK, limited support for generics has been introduced.

The bind link between the product-line template and deployed components is decorated with the mapping of formal to actual parameters for that instantiation of the template. In this way, the deployed component contains behaviour in terms of the actual parameters in place of the template parameters.

5.3.3.2 Declaration and Transformation of Template Components

The provision of support for the instantiation of templates within the generation transformation used a similar approach to the handling of opaque behaviour. The existence of a decorated bind link from deployed to catalogue component indicates to the transformation tool that template instantiation is taking place. The Formal Parameter -> Actual Parameter mappings are cached from the bind dependency and are used within a simple string replacement function that is applied to specific text fields in the catalogue model as they are duplicated into the transitory deployed model. The model attributes that are allowed to contain formal template parameters are as follows:

- Class Name
- Operation Name
- Attribute Name
- Typedef Name
- Operation Body
- SPARK Annotations

5.3.3.3 Templates and Variability

It is perfectly possible for template components to also contain decisions and variation points. This can be a useful mechanism to cater for small variations within template components. The decision to create two different templates or provide variation in a single template can be difficult to judge, and currently is based on a subjective assessment of the resultant complexity of the variable template component.

5.3.4 Expanding Design Patterns

So far, we have discussed the initial model-to-model transformation (M2M1) shown in Figure 60. Once this transformation has completed, we run a further set of M2M transformations to expand any **design patterns** that have been include in the source components. One of the weaknesses of the architecture-centric approach that was used previously was that the architectural UML class models created to represent the software system were too detailed; in particular, they contained too many target-language specific constructs.

This level of detail was required in the model to enable successful, syntactically complete code generation. However, it was not required to convey any special characteristics or design intent; the model detail was mostly implementation of standard design approaches. With the availability of model-to-model transformation in the code generation process, it was decided to take advantage of this to enable the use of more abstract representations of standard designs. The M2M transformation could then expand these at generation time to produce the syntactically and semantically complete implementations for code generation purposes. This is analogous to a Model Driven Architecture approach that distinguishes between a Platform Independent Model (PIM) and Platform Specific Model (PSM) [124].

The design patterns expanded with this set of M2M transformations are an ordered set; it is entirely possible that one design pattern will contain reference to another pattern that requires subsequent transformation. As we perform the transformations in a single pass, a design pattern may only make use of other patterns processed in a downstream transformation.

We apply the design pattern transformations in the following order:

1. Apply Development Variable (DV) Pattern
2. Apply Testpoint Pattern
3. Apply Interface Pattern
4. Apply OS Interface Pattern
5. Apply Testport Pattern
6. Apply Graphical Data Pattern

Each of these design patterns encodes the details of a particular idiom or code construct that is used within the FADEC software. The details of these used to be completed by hand by the designer, whereas now they can indicate the required style by use of a model stereotype, and the transformation will add the detail to the model.

5.3.5 Source Code Generation (Model-to-Text Transformation)

The final transformation phase shown in Figure 60 is the Model-to-Text transformation that produces the SPARK source code. An important property of this phased transformation approach is that the transitory model presented to the M2T code generator is of the same form as the single-project UML model that was used in previous, single-system projects. Therefore, minimal changes are required to the M2T generator to enable its use on a product line development. As described in section 5.3.3, the original M2T generator was implemented using the OCS tool. It is possible to "host" OCS-based generators within an ACS generator; the OCS templates are imported en-masse as operations in a generator package.

5.3.6 Lifecycle Data Generation ("Model-to-Document" Transformation)

One important aspect in our approach is that the design model and the transformations support the generation of product-specific lifecycle evidence as well as the product source code. We have already discussed the importance of traceability in Chapter 4 Section 4.8, and the process for exporting product-specific traceability was outlined there.

In addition to the code generator and traceability export tools, we have developed document generation transformations that can produce DO-178B/ED-12B "Low Level Requirements" artefacts for each of the components in the instantiated product. The detail of these documentation transformations are not presented in detail in this thesis as they were developed by other members of the FADEC development team; however they were based on the meta-model and transformations developed as part of this research and presented here. The fact that other engineers can understand the underlying meta-model and transformation approach to a level at which they can produce supporting documentation tools is a testament to the usability of the approach we have developed.

5.4 Conclusions and Observations

Model transformation and generative programming approaches are fundamental to software product line developments that exploit commonality and variability to automatically realise product instances. We have demonstrated in this chapter that given a product line model conformant to a reference architecture and core asset components designed that include variation, model transformation can be used to generate product instances based upon the resolution of variability decisions in deployed components.

5.4.1 Addressing the Challenges

We began this chapter by setting out a number of essential and accidental challenges for the deployment of High-Integrity Product Lines. We can now start to review our approach against these challenges to determine qualitatively how well they have been met. We have to assume that that the "single system development" approach on which our product line process is built was fit for purpose; however, this was used successfully on two FADEC development projects prior to development of this approach [82].

5.4.1.1 Accidental Challenges

We start with an assessment of the approach against the accidental challenges described in 5.1.2.

1. Use of UML models for software architecture and design

   Chapter 4 described the architecture and component meta-model that is used to capture our designs within UML. Here we have described how to transform those models to produce product instances. This challenge is addressed successfully, with the caveat regarding the use of test transformations for opaque behaviour as discussed in 5.3.2.

2. Use of SPARK as the target implementation language

   It is clear that our approach is successful in generating SPARK-compliant programs.

3. Hard Real-Time, Embedded Constraints

   There is nothing in the approach that inherently compromises the ability to meet the hard real-time constraints. Indeed the use of reductive transformations at code generation time helps ensure that the source and object code size is minimised, and does not require the run-time (and verification) overhead of deactivation mechanisms.

4. Restrictions on the available (incumbent) development environments/tools (e.g. ARTiSAN Studio)

   We have demonstrated that the ARTiSAN Studio model transformation tools can be used successfully to implement a reductive product line transformation.

5. Custom and practice, customer expectations

   The approach we have taken is to build a product line development environment upon the existing tools and process that are incumbent in the organisation. Much of the business custom and practice has been encoded in the design patterns that are applied during the model transformation process.

6. Project management strategies

   Again, our approach of building a product line development environment upon the existing tools and process should minimise the impact on the project management practices. There is nothing inherent in our approach that requires a change to the strategies used. However, this does not mean that the issues inherent in managing the business aspects of a product line (as articulated in the BAPO model[16]) is alleviated by this approach.

5.4.1.2 Essential Challenges

Assessing our ability to meet the essential challenges from an analysis of the technical approach is more difficult. Whilst we can argue that we have produced an approach that meets the technical constraints of the accidental challenges, for many of the essential challenges we can only argue that we have "done no harm". To truly asses our success against a number of these challenges we must test the approach in the real world. The challenges listed in bold face below require an assessment in use.

1. **Scale & Size of Product**
2. **Deployment into "typical" industrial teams**
3. Enabling the demonstration of requirements satisfaction (validation, traceability, basic integrity)

   Here we have demonstrated an approach that allows the capture of traceability information in a manner that enables product-specific trace data to be produced.

4. Progressive addition of detail

   Modelling the variability as decisions in the components themselves means that they can be developed in isolation, or at least be decoupled from each other. Componentry can be developed through the product and product line lifecycle, and it certainly means that prototype product can be instantiated before a complete product line implementation is defined (also see point 6).

5. Clarity of design

   It can be argued that one of the advantages of the reductive transformation approach is that all the product variability is visible for scrutiny by review, and the result of variability decisions is clear. This is aided by the decision contract approach, where the available decisions and their resultant impact is available for prior verification.

6. Allowing different sources/drivers of variation at different times in the development

   Again, the "component containing a decision contract" approach means that components can be developed at separate times, by separate teams and then integrated into the product line architecture when required. Having an approach that enables a level of flexibility in working arrangement is key to achieving this challenge. **The development process must be capable of supporting the different phases of FADEC, Engine and Aircraft development test as outlined in Chapter 3.**

7. **Credible Certification/Approval**
8. Information Partitioning/"Chinese Walls"

   The ability to produce product-specific documentation as a result of the product line transformation means that any information that is not pertinent to the product itself is removed from the documentation. In the extreme, multiple product line components (or product-specific components) can be produced to separate intellectual property that resides with different parties.

5.4.2 Summary

We have previously shown how it is possible to move from a single-system model-based development to a product line, and, via the appropriate separation of concerns in the model transformation stream (Figure 67), preserve existing code generation strategies where appropriate, thereby reducing the risk of the final product not being fit for purpose.

We have shown in detail the design of a model transformation suite that employs both model-to-model and model-to-text technologies to implement a product line code generator; the resultant generator is not a research prototype, but is actively used to develop avionics control system software products.

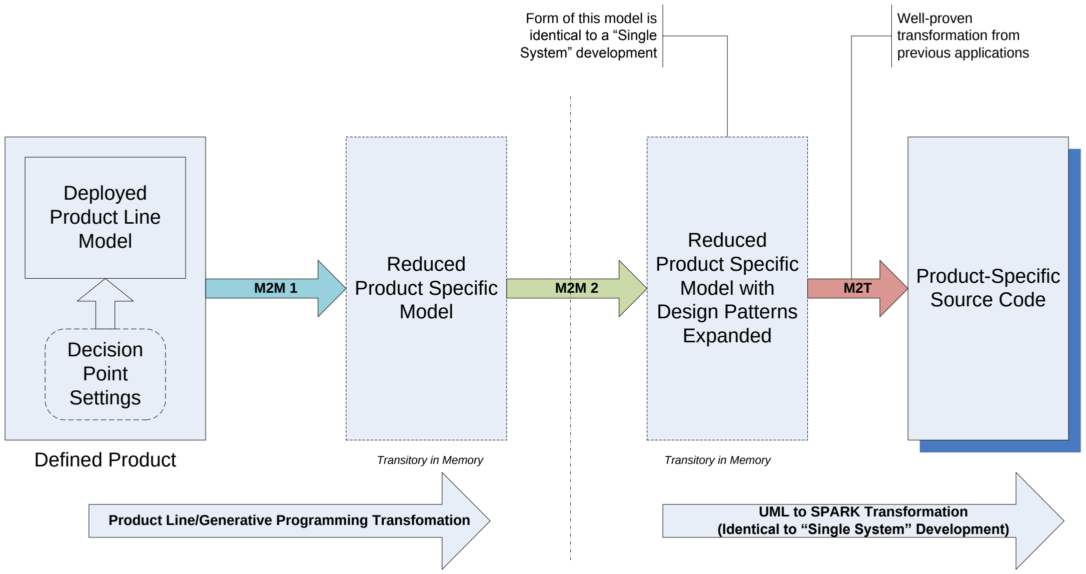

Form of this model is identical to a "Single System" development
Well-proven transformation from previous applications
Deployed Product Line Model
Decision Point Settings
Defined Product
M2M 1
Reduced Product Specific Model
Transitory in Memory
M2M 2
Reduced Product Specific Model with Design Patterns Expanded
Transitory in Memory
M2T
Product-Specific Source Code
Product Line/Generative Programming Transfomation
UML to SPARK Transformation (Identical to "Single System" Development)

FIGURE 67 SEPARATION OF CONCERNS IN THE MODEL TRANSFORMATION STREAM

Any practical approach to product line instantiation using model-transformation must take into account those parts of the product definition that are not meta-model compliant. That has been addressed in the approach described in this chapter. However, a long-term research aim is to develop and/or integrate a set of modelling environments that are rich enough to capture the full range of specification and design descriptions required for current and future systems.

The approach outlined here (and in more detail in Appendix B) was designed to be an industrial strength implementation of a transformational product line approach. This is an approach that utilises many of the concepts that the product line community have been researching and advocating, and is applicable to a high-integrity development environment. To show fully that this approach has met our essential challenges, we now need to review how well this approach works in practice.

6 Evaluation and Validation

This chapter evaluates the trusted product line approach described in the previous chapters using data obtained from industrial use of the technique. The data provides quantitative information on the cost-effectiveness of the approach and qualitative information on the ability of the process to provide product approval evidence. This evaluation will determine if the trusted product line approach is effective in terms of both development cost and product quality.

6.1 Industrial Deployment of Trusted Product Lines

The approach to software product line development described in the previous two chapters has been used to develop an engine control system product line for large civil aerospace gas turbine engine applications. This development began in early 2009, with the first application of the product line commencing flight test on a "flying test bed" aircraft in early 2012.

The development approach used is shown pictorially in Figure 68 below:

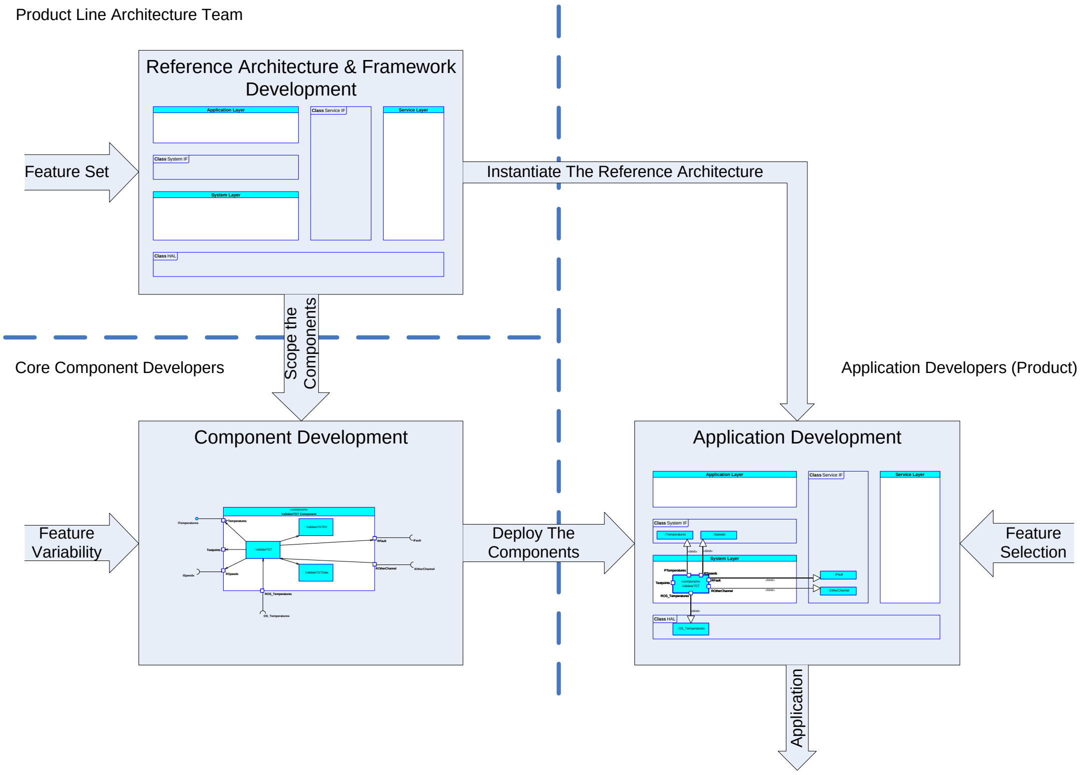

Product Line Architecture Team
Reference Architecture & Framework Development
Feature Set
Instantiate The Reference Architecture
Scope the Components
Core Component Developers
Application Developers (Product)
Component Development
Application Development
Feature Variability
Deploy The Components
Feature Selection
Application

FIGURE 68 SOFTWARE PRODUCT LINE PROCESS FLOW

A specialist team (led by the author) defined the reference architecture; this is as described in Chapter 4 of this thesis. The development of the product line from 2009 onwards focussed on the creation of core software components ("core assets") that could be used to build the product instances using the model transformation approach described in

Chapter 5. The company were hesitant to embrace fully the product lines concept; therefore, the development effort was split into two development teams. A product lines component team was established to develop components compliant with a set of product line software requirements specifications (SRS). These SRSs identified common and variable requirements for the components against the scope of the proposed product line, with variability specified using the PLUSS notation [119]. The components produced contained decision contracts and variation points as described in chapter 4. This team was responsible for approximately 50% of the functionality required for the first application; the set of components developed in this way was defined and agreed with the recipient project in a "scope of supply" (SoS) document. Secondly, a project team was formed to produce the remaining 50% of project-specific components that did not contain variation, and to deploy the full set of components to produce the final application. However, it should be noted that all software components were deployed into the product using the code generation process described in chapter 5. The only difference between "core asset" components and "project specific" components is that the project-specific components typically did not contain variability.

6.2 Evaluation Methods

We employ two complementary evaluation methods to evaluate and assess the effectiveness and success of the deployed approach. Firstly, we evaluate the cost-effectiveness of the approach using a quantitative method based upon time booking (effort) data gathered from the development teams on a weekly basis over a two-year period (Jan 2009 to Feb 2011). We also critically review the findings of independent audit of the project against the objectives of DO-178B/ED-12B to determine the effectiveness of the approach to deliver product assets of the required quality.

6.2.1 Quantitative Evaluation Method

The effort expended by the domain and application teams was collected using a data collection approach known as Process Engineering Language or PEL [126]. PEL requires the definition of a grammar to describe development activities, which allow the data collected to be analysed from a number of different viewpoints and, crucially, those queries do not need to be defined prior to the collection of the data. As described in [126], the PEL *lexicon* provides a constrained vocabulary of terms used to describe a process. This PEL lexicon is divided into four dimensions: Actions, Stages, Products and Representations:

> **Actions** are verbs that describe the task performed, for example *Produce, Review, Maintain.*
>
> **Stages** provide a time or milestone view of the project, and are typically obtained from the project or programme management view of the project, for example *Project A, Delivery D4.1.* This is traditionally the dimension against which effort and cost data is collected for budgetary and billing purposes.
>
> **Products** are the physical or logical components and/or systems that are being produced, for example *Thrust Reverser, Starting System*. It is typically the product breakdown structure as defined in the physical and functional architectures.

Representations are the process outputs and work products, for example *Software Requirements, Code*. This provides the process view on the data and should reflect the process definitions as worked on the project.

The defined PEL lexicon used to collect the effort metrics on the product line development is summarised below:

Action	Stage	Product	Representation
Produce	<List of Project Deliveries>	<Product Component Structure>	Requirements
Review	Asset Development[6]		Architecture
Rework			Design
Re-Review			Code
Support			Builds
Attend			Low Level Test
			Integration Test
			Hardware Software Integration Test
			Documentation
			"Other" (Management)

This allows the development staff to construct cost-booking codes by selecting one item from each of these columns that most accurately reflects the activity they have performed, for example:

PersonA books 10 hours to : "Review" "Delivery 1.1" "Component X" "Design"

[6] Note that in addition to the list of project software deliveries, the Stage dimension has a category of Asset Development that allowed the collection of costs for the production of product line assets.

This method has been used for cost collection on software development within the company since 1996, and was not specially introduced for evaluation of this research.

6.2.2 Qualitative Evaluation Method

During the period of evaluation, the development project was subject to a number of independent audits to determine the compliance to DO-178B/ED-12B and the internal company procedures. These audits provide a view on the development activities that is not prejudiced by the business focus on product lines; they are intended solely to ascertain whether the product as designed and built meets the regulatory objectives and company quality standards. This provides an objective evaluation on the ability of the product line “factory” to deliver assets of the required quality with the necessary supporting evidence.

6.3 Evaluation Results

This section contains the results of the evaluations undertaken as described previously. Firstly, we provide the results of the analysis of the PEL cost-booking data, followed by the qualitative analysis of the audit findings.

6.3.1 Quantitative Evaluation Results - Relative Process Performance

6.3.1.1 Sample Data Set

The effort data analysed comprises the set of time bookings taken between January 2009 and February 2011. This includes both the product lines component development team and the application development team for the first target project. The data set comprises 15,400 individual time booking entries, made by 184 unique individuals. The total effort recorded over this period totals 142,000 hours, which is a significant development activity both in terms of company investment and for arguing the relevance of the data set for analysis purposes.

We can start to use the PEL classifications identified earlier to understand the data content. Figure 69 shows the breakdown of the total hours booked by the process area or “representation”.

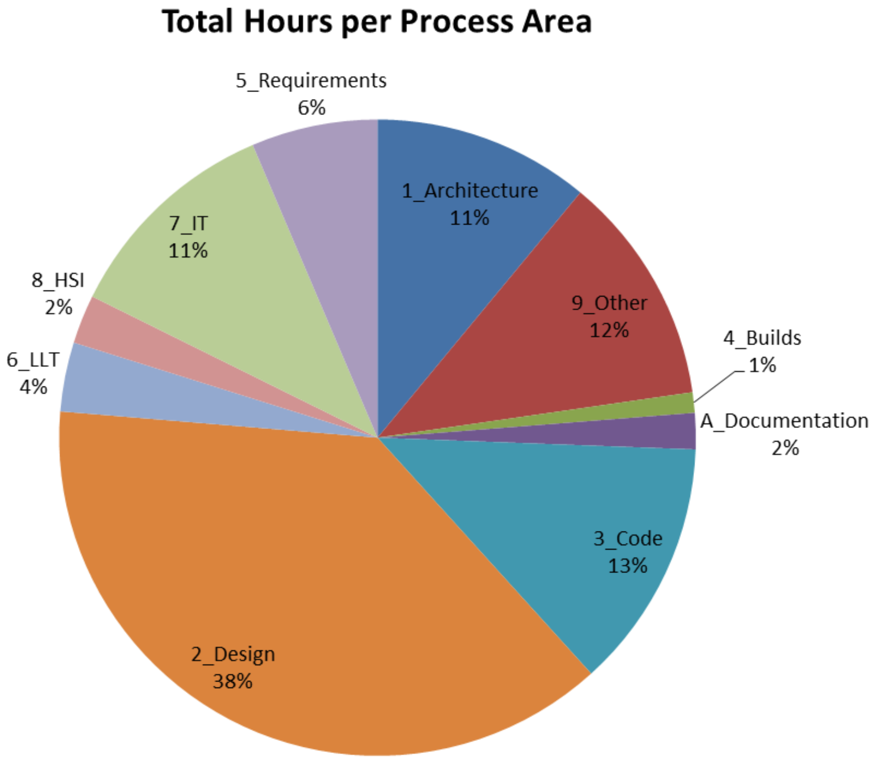


FIGURE 69 BREAKDOWN OF TIME AGAINST "REPRESENTATION"

Here we can see the largest development activity recorded was the design process; in fact, the design/code activities account for 51% of the activities measured. Understanding this breakdown is important in interpreting the data presented:

- The data sample covers the major development activities. The verification by test activities had not started to any great degree over the time period sampled. (Verification by test accounts for 17% of the time booking data analysed. This has typically risen to 50% of the total hours expended by completion of a development project).
- The 6% identified as requirements is primarily requirements review. The SRS set was developed by a "systems engineering" team which, unfortunately for this analysis, does not use the PEL booking system for time recording.

We can now attempt to analyse the data set to determine if we can identify and isolate the effects of development for a product line.

6.3.1.2 Analysis 1 - Total Hours per Process Area by Team

The first comparative analysis provides the breakdown of hours per process representation, shown separately for the project team (Figure 70) and the product lines team (Figure 71). This separation of data was made using the "Asset Development" identifier in the Stage PEL dimension to indicate those hours booked by the product lines development team.

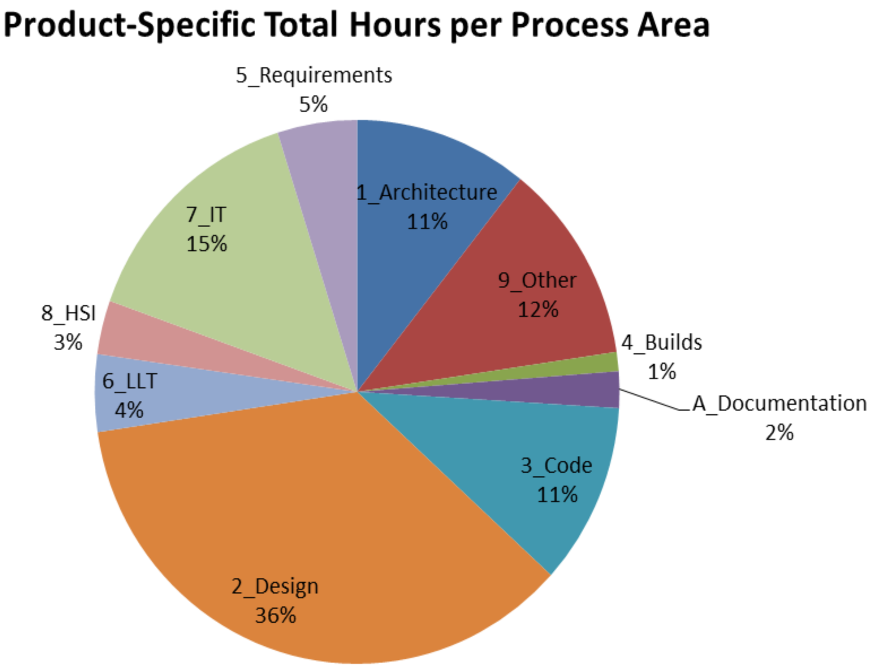


FIGURE 70 BREAKDOWN OF PRODUCT SPECIFIC TIME AGAINST "REPRESENTATION"

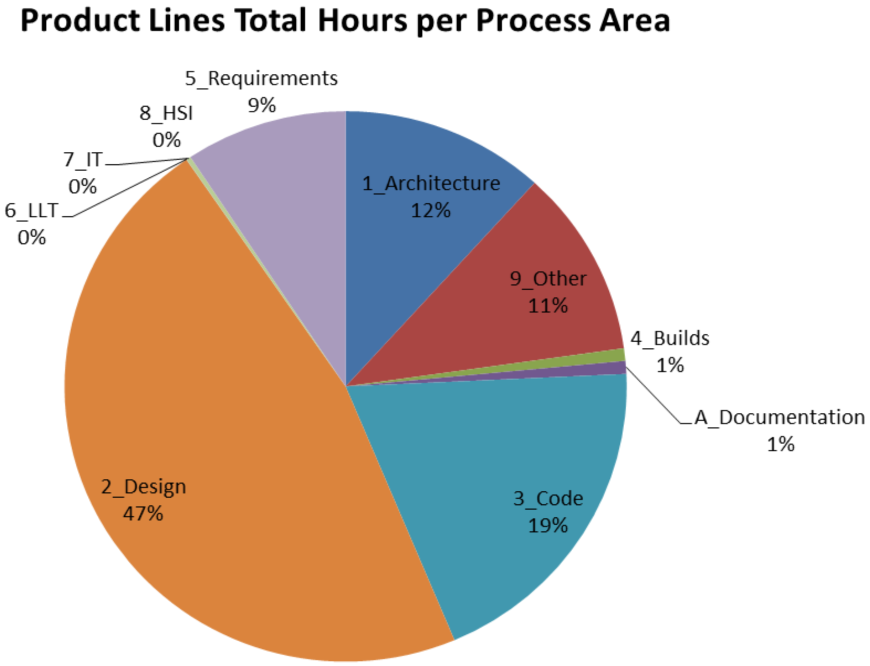


FIGURE 71 BREAKDOWN OF PRODUCT LINES TIME AGAINST "REPRESENTATION"

We can see here that there is no verification by test activity recorded against the product lines development team. This is to be expected, as verification by test requires a "buildable" product against which to run the tests. For components containing variation, these need to be instantiated before a buildable component is available.

A view of this data just containing the development activities (requirements, architecture, design and code) is provided in Figure 72 and Figure 73 below. Using this breakdown, we can compare and contrast the proportion of development effort across the phases for product line and non-product line component development as defined by the cost attribution within the Stage field of the PEL bookings.

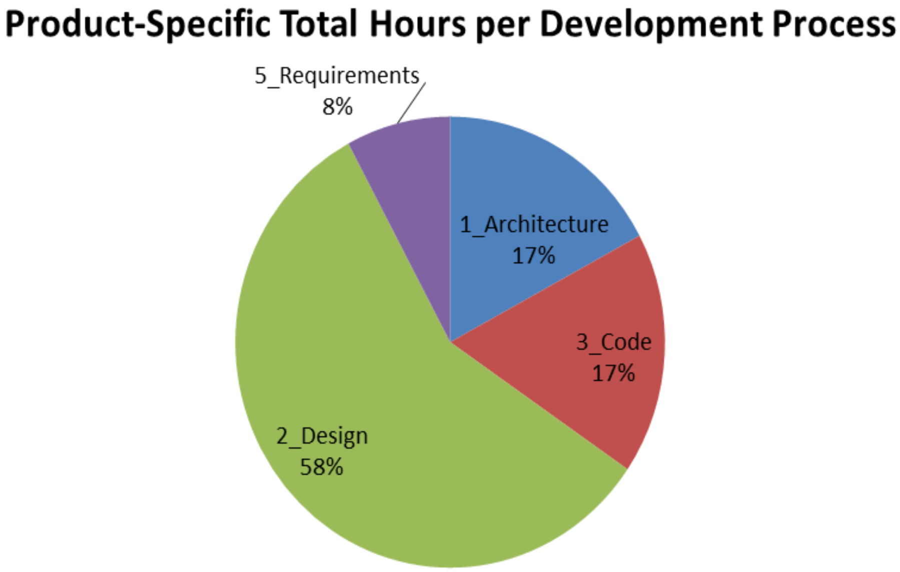


FIGURE 72 BREAKDOWN OF PRODUCT SPECIFIC TIME AGAINST DEVELOPMENT PROCESS TASKS

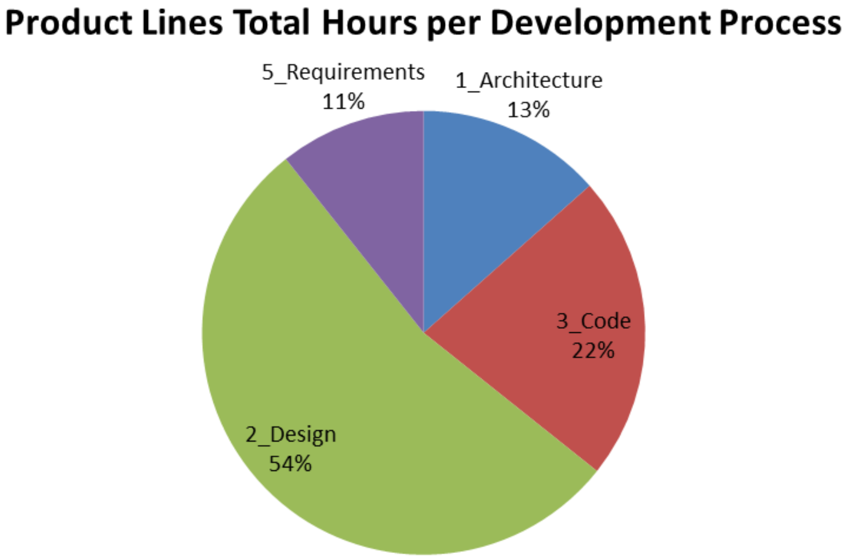


FIGURE 73 BREAKDOWN OF PRODUCT LINES TIME AGAINST DEVELOPMENT PROCESS TASKS

Here we can see that proportionally there is slightly more effort in requirements review and code development in the product lines team compared with the product-specific development team, which we could postulate was due to the added complexity of both reviewing and implementing artefacts containing variability (this is discussed further in section 6.4.1.4.)

This data was sub-divided into Product Lines and Product Specific essentially by identifying the team from which the hours were recorded. This may not necessarily be an absolutely accurate means of distinguishing between product line and product specific developments, although it is a strong indicator. The next set of analyses attempts to provide a greater degree of accuracy in this sub-division.

6.3.1.3 Analysis 2 – Hours by "Scope of Supply"

For this analysis, we use the product breakdown as defined in the project management documentation for the Product Line; namely that defined in the "Scope of Supply" (S0S) document. This was an agreement at the start of the Product Line development that identified those parts of the FADEC software that were to be developed as a product line asset, and those that were to be developed on project. Figure 74 and Figure 75 show the relative levels of process effort between the components identified as Product Line and those developed specifically for the project as defined in the SoS document. Figure 76 and Figure 77 repeat this but purely for the development tasks.

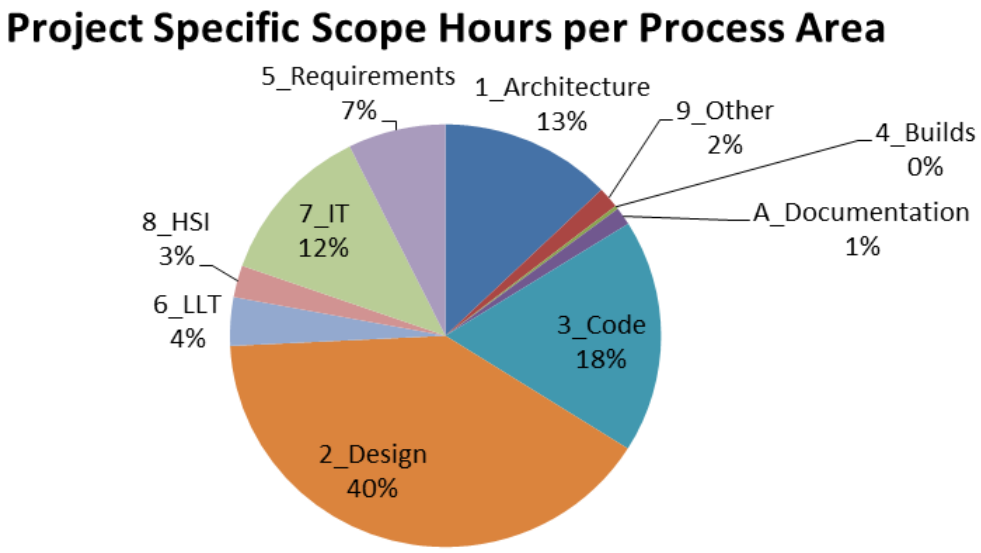


FIGURE 74 BREAKDOWN OF PRODUCT SPECIFIC TIME (AS DEFINED IN SOS) AGAINST "REPRESENTATION"

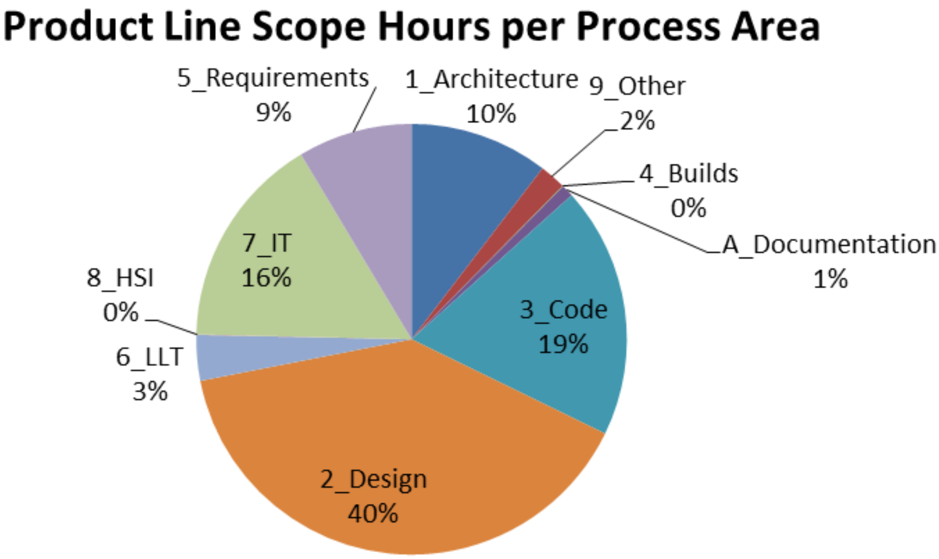


FIGURE 75 BREAKDOWN OF PRODUCT LINE TIME (AS DEFINED IN SOS) AGAINST "REPRESENTATION"

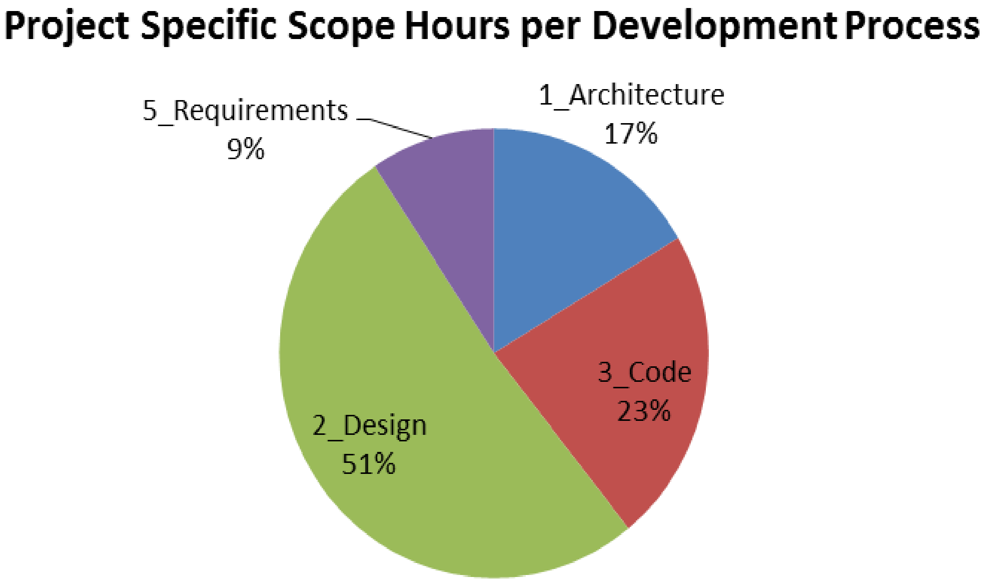


FIGURE 76 BREAKDOWN OF PRODUCT SPECIFIC TIME (AS DEFINED IN SOS) AGAINST DEVELOPMENT PROCESS TASKS

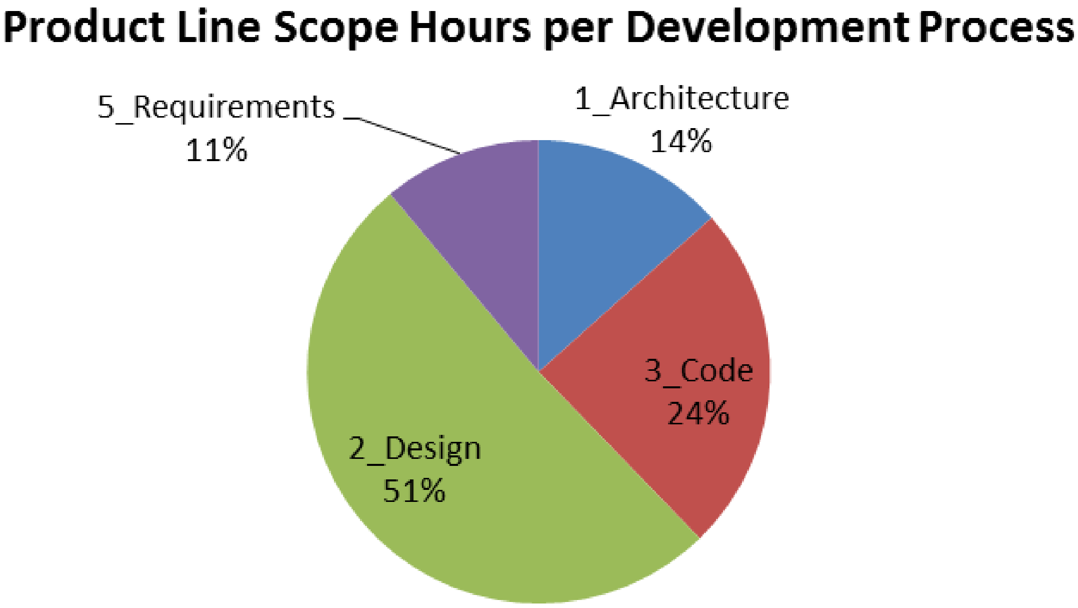


FIGURE 77 BREAKDOWN OF PRODUCT LINE TIME (AS DEFINED IN SOS) AGAINST DEVELOPMENT PROCESS TASKS

Again, this analysis indicates that the proportion of the engineering effort on requirements review and code development is slightly greater for the Product Line components then for the product-specific components; however the differences are quite small.

As with the analysis documented in 6.3.1.2, this breakdown into product line and product specific components is a project management distinction, and does not necessarily distinguish whether technically the components contain variability or not. Therefore, we perform a final comparative analysis, using information extracted from the components themselves.

6.3.1.4 Analysis 3 – Hours By Variability

For this cost analysis, rather than distinguishing between “Product Line” and “Product Specific” components by project management and/or team structure allocation, we actually distinguish between variable and non-variable components. This identification was performed by an automated analysis of the UML model used to develop the set of components. This analysis tool traversed the model in a similar manner to the code generator and identified those components that contained decision contracts. In this way, we can identify definitively the components that have had required extra work to provide variation points.

Figure 78 and Figure 79 show the total process time allocation for non-variable and variable components respectively, and Figure 80 and Figure 81 show the proportions for the development processes only.

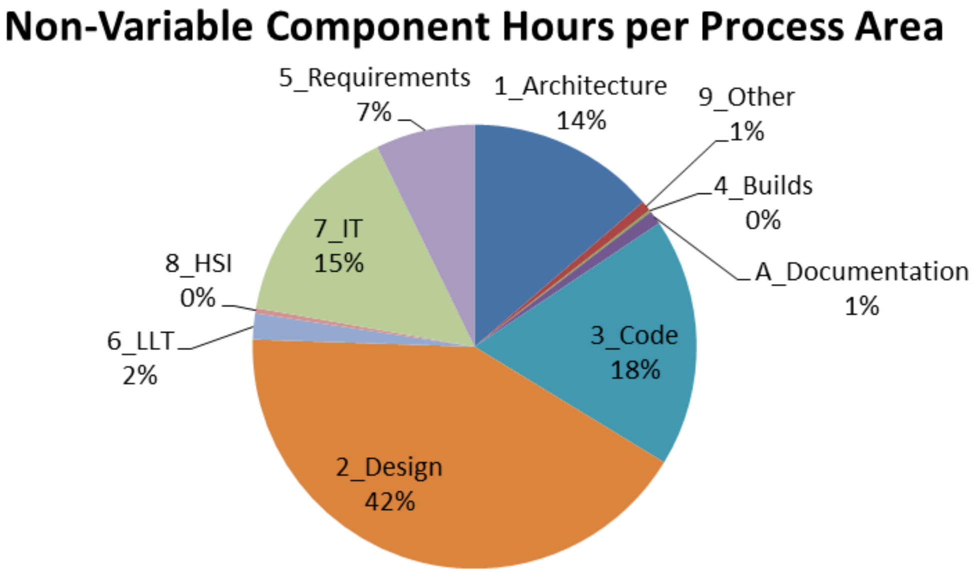


FIGURE 78 BREAKDOWN OF DEVELOPMENT TIME FOR COMPONENTS CONTAINING NO VARIABILITY

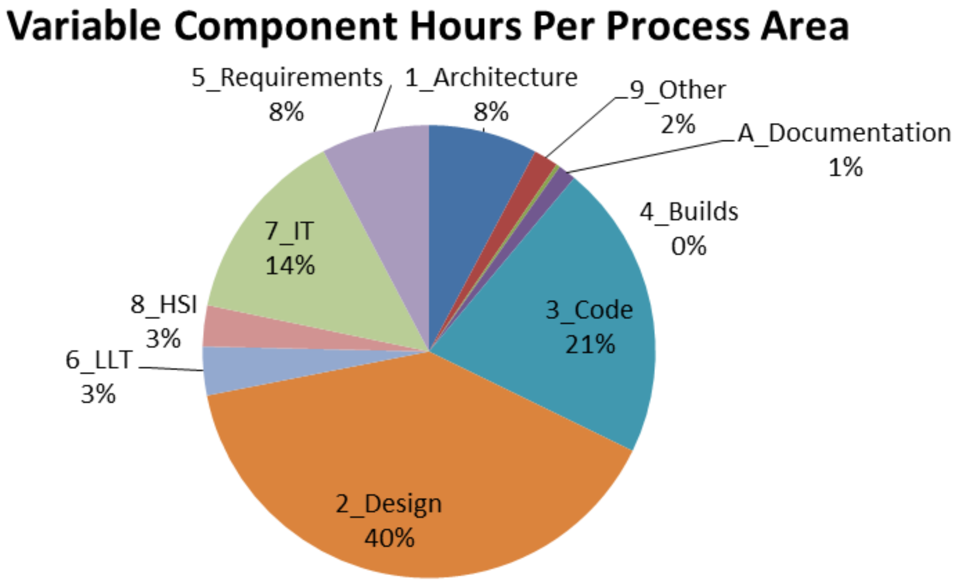


Figure 79 Breakdown Of Development Time for Components Containing Variability

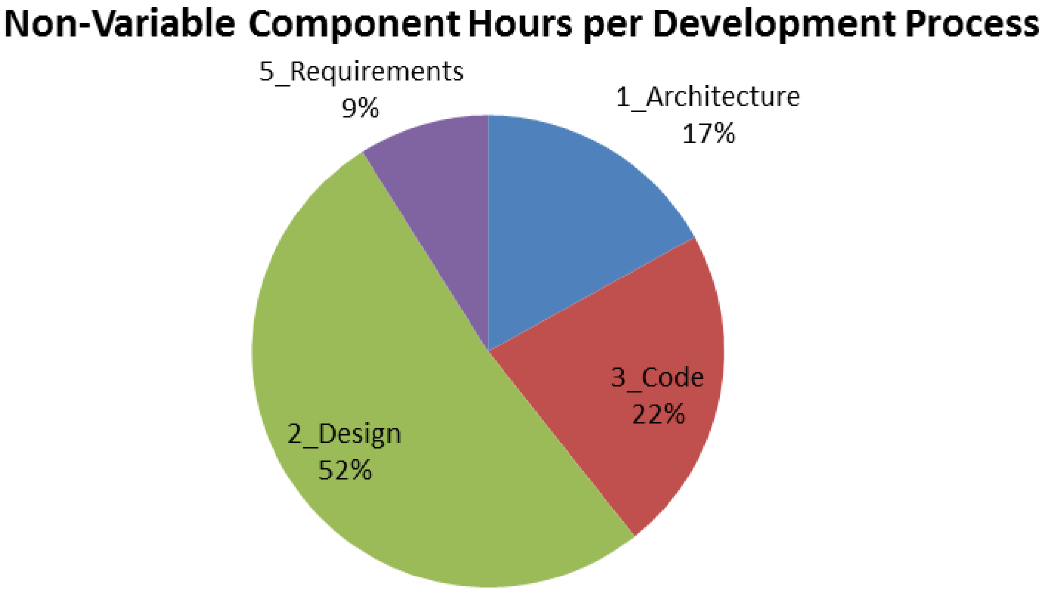


Figure 80 Breakdown of Development Process Time for Components Not Containing Variability

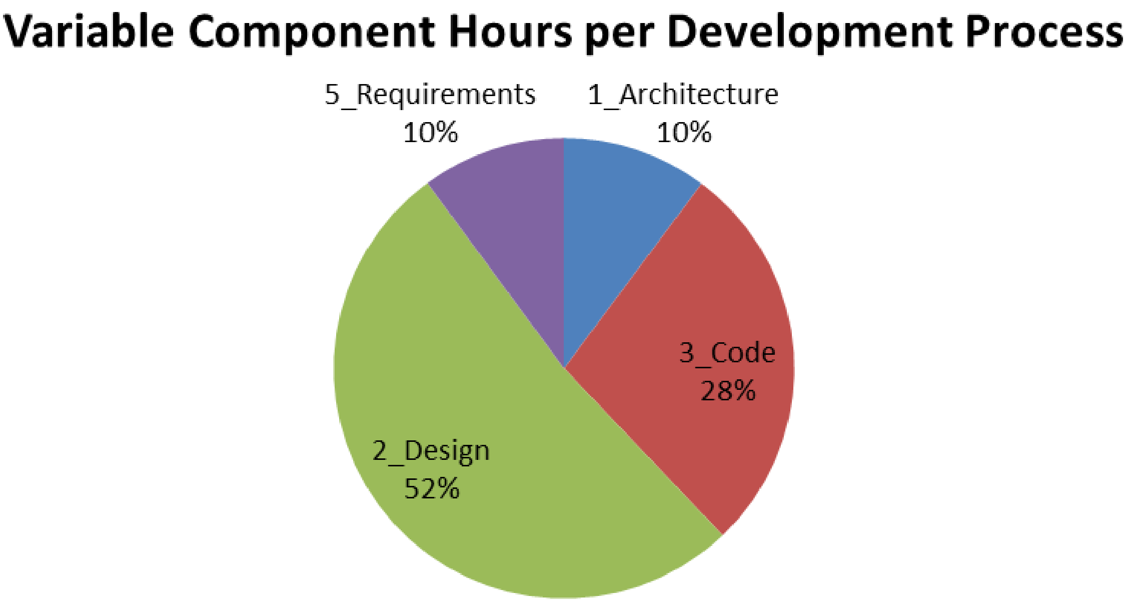


FIGURE 81 BREAKDOWN OF DEVELOPMENT PROCESS TIME FOR COMPONENTS CONTAINING VARIABILITY

Again, we can see a similar pattern to the previous analyses, however the difference in the proportionate effort in code development is even more marked (22% for non-variable components vs 28% for variable). Conversely, there is a marked reduction proportionally for the architectural development effort (17% for non-variable components vs 10% for variable).

Whilst these analyses of relative effort per development phase provide an interesting insight into how the development process may subtly change when developing variable components, they do not provide any indication on the magnitude of the cost difference between the two component types. This is addressed in the next section.

6.3.2 Quantitative Evaluation Results - Absolute Cost Performance

The previous set of analyses concentrated on the relative process cost differences between product line and project-specific component development. Here we look at the absolute difference in the component development costs to ascertain if there is a significant cost differential between the two component types.

Table 4 shows the cumulative development costs for the components on the project, categorised into variable and non-variable components. (Note that in this categorisation, a variable component is one that contains a decision contract, as identified by an analysis of the product UML model.)

TABLE 4 AVERAGE DEVELOPMENT COST PER COMPONENT (VARIABLE AND NON-VARIABLE)

	Variable Components	**Non-Variable Components**
Total Cost (Hours)	37924.05	28883.95
Number of Components Developed	72	140
Average Cost per Component (Hours)	526.7	206.3

Cost Ratio Variable to Non-Variable Component	**2.6**

The "Total Cost" row in Table 4 contains the recorded development effort that can be directly attributable to the components from the PEL cost booking data. Calculating the mean cost per component for each component type, and calculating the ratio between them identifies **a cost ratio of 2.6 between a variable and non-variable component**. Given that the cost data for variable components will include any deployment costs incurred by the project, this is closely in line with the industry-accepted view that a product line approach becomes cost effective at or after 3 deployments.

Whilst this result appears to correlate well with the accepted industry wisdom, we need to determine which factors are influencing the increased cost of the variable components. Is the introduction of variability the dominant factor in the cost of these components, or has the company decided to implement large, problematic or complex components as product line assets?

To help to identify and isolate the root cause of this cost differential, we analyse the relative code size and complexity of the variable and non-variable components. In this analysis, we measure complexity using McCabe's cyclomatic complexity metric. Code size is determined using a number of “source lines of code” (SLOC) counting rules, including “Non-Blank, Non-Comment” (to give a measure of the “value” of the code) and a simple “number of lines in the code file” measurement.

Table 5 shows the relative complexity of the component types for a set of instantiated components (both variable and non-variable). Note that this analysis is post-instantiation; i.e. this is after deployment options have been selected and code generated for that particular option set.

TABLE 5 COMPARATIVE AVERAGE COMPLEXITY (MCCABE) BETWEEN VARIABLE AND NON-VARIABLE COMPONENTS

	Number of Components Analysed	Sum of Mean McCabe	Mean Mean McCabe		Sum of Total McCabe	Average Total McCabe
Variable Components	57	126.48	2.22		1406	24.67
Non-Variable Components	87	164.61	1.89		1422	16.34

Mean McCabe Ratio (Variable/Non-Variable)	**1.17**
Total McCabe Ratio (Variable/Non-Variable)	**1.51**

The difference between total and mean McCabe in Table 5 is explained as follows:

McCabe is measured per sub-program (i.e. SPARK procedure or function). A component may have a number of procedures or functions within its implementation. If the number of subprograms in a component is denoted as n then:

Total McCabe per Component = $\sum_{i=1}^{n} McCabe(i)$

Mean McCabe per Component = $(\sum_{i=1}^{n} McCabe(i))/n$

We can see from the Mean McCabe ratio in Table 5 that the variable components are not, on average, significantly more complex than the non-variable components (1.17 times more complex on average). Certainly, this difference is not enough to account for the difference in development cost.

The Total McCabe ratio shows a more pronounced difference (1.51 times) – this would indicate that the variable components are larger, or at least contain more individual operations than the non-variable components. This may be confirmed by looking at the relative component sizes as indicated by their Source Line of Code (SLOC) counts, shown in Table 6.

Table 6 Comparative Average Code Size (SLOC) Between Variable and Non-Variable Components

	Number of Comps Analysed	Total Code Lines	Total Blank Lines	Total Cmt Lines	Total Lines		Mean Code Lines per Comp	Mean Blank Lines per Comp	Mean Cmt Lines per Comp	Mean Lines per Comp
Variable Components	57	59160	35095	147068	241323		1037.89	615.70	2580.14	4233.74
Non-Variable Components	87	44487	24889	122363	191739		511.34	286.08	1406.47	2203.90
Ratio (Variable/Non-Variable)							**2.03**	**2.15**	**1.83**	**1.92**

Table 6 records the results of analysing the source code of a set of deployed components; a total SLOC count is produced for each component, and this is further sub-divided into the following categories:

- Code Lines – 'Useful' lines of program source (sometimes defined as Non-Blank, Non-Comment)
- Blank Lines
- Comment Lines

Table 6 shows the ratio of average useful Code Lines between Variable and Non-Variable components to be 2.03, i.e. Instantiated variable components are on average twice the size of non-variable components (using this SLOC measure).

This size differential could be accounted for in a number of ways:

1. The components that the company has decided to implement as part of a product line are inherently larger.

2. The variation mechanism used in the product line components results in larger code files post-instantiation (using this SLOC count) then if non-variant code was used.

We postulate that the size increase identified is probably a combination of these two explanations. Because of the SLOC counting convention used in this analysis, the size data is subject to inflation due to changes in the code layout; for example:

```
if A and B then
```

would count as a single line using this convention, whereas the semantically equivalent:

```
if A

and B

then
```

would count as 3 lines. If the "and B" part of that expression was optional as part of a variability point, then the second form of the code would be used to allow the insertion of the variability mark-up code, as shown stylistically below :

```
if A

VpIF option {

and B

}

then
```

This type of construct naturally leads to inflated code sizes when measuring non-blank non-comment lines of code. To counteract these effects a semi-colon-based code count convention (i.e. a count of statement termination) may show that the relative code sizes are not as different as it appears.

6.3.3 Cost Correlation

Now we have identified the relative cost differential between variable and non-variable components, it would be useful to determine if there was any clearly identifiable aspect of a variable component that contributed to the increased cost. To try and identity this we plotted a number of potential component cost-drivers against component cost to see if any were closely correlated.

Figure 82 to Figure 85 show various component variability complexity measures (based upon the number of decisions/options provided to the component user (Figure 82, Figure 83), and how much of an impact those decisions have on the actual variability in the code (Figure 84, Figure 85)).

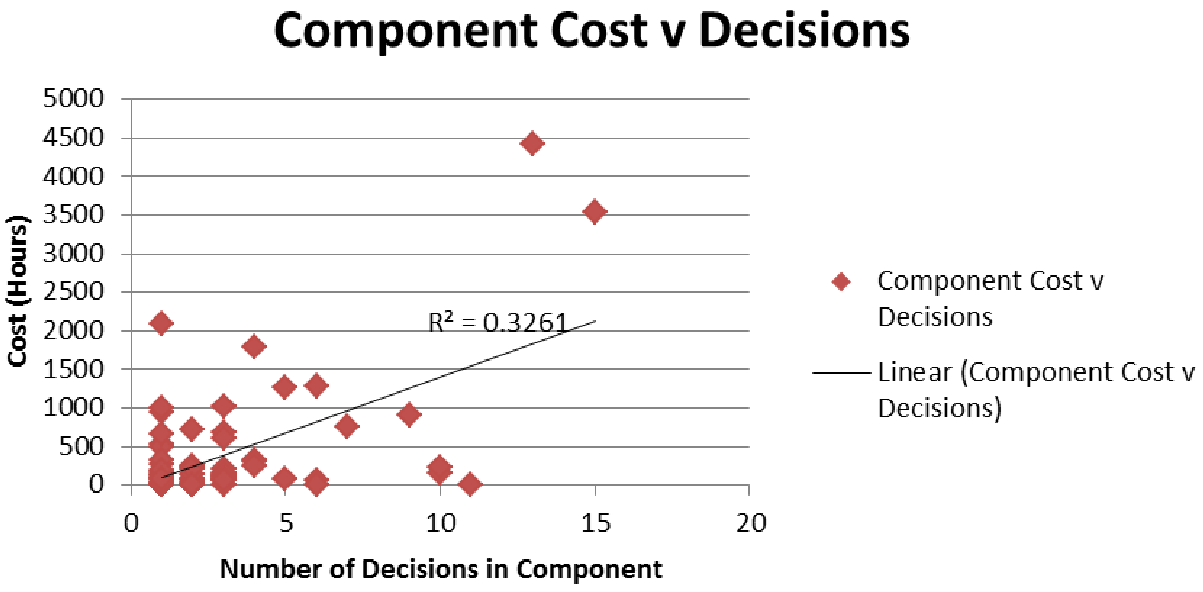


FIGURE 82 NUMBER OF DECISIONS IN A COMPONENT VS COMPONENT COST

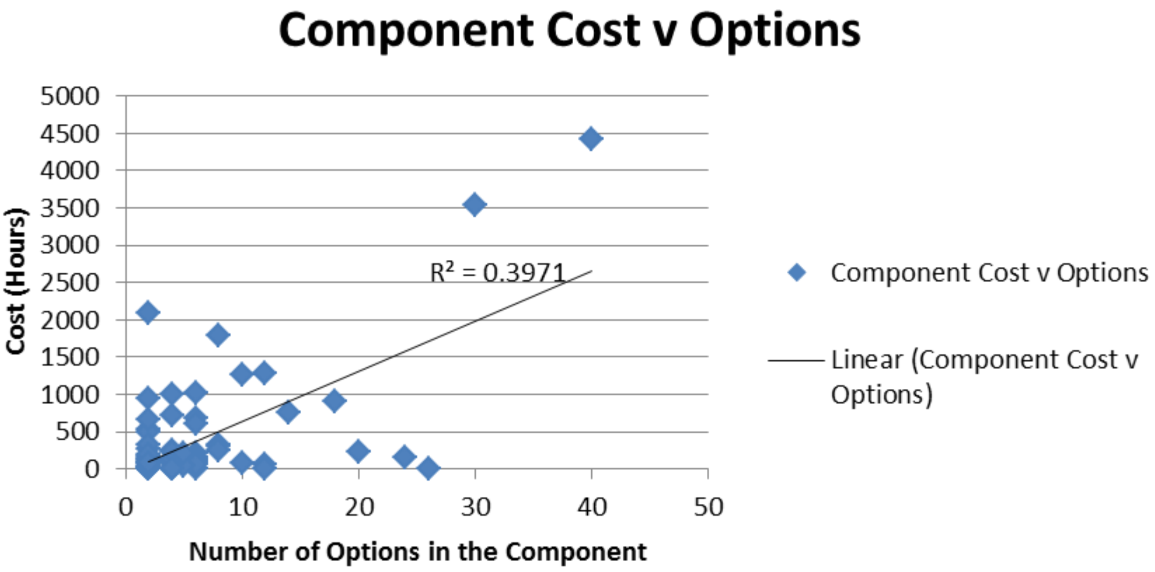


FIGURE 83 NUMBER OF OPTIONS IN A COMPONENT VS COMPONENT COST

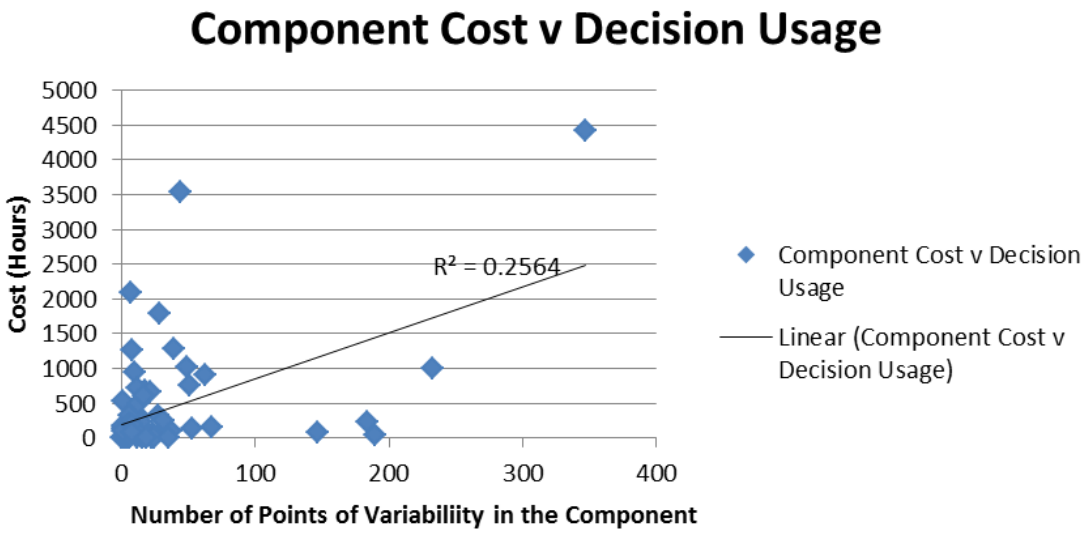


FIGURE 84 NUMBER OF USES OF A DECISION IN A COMPONENT VS COMPONENT COST

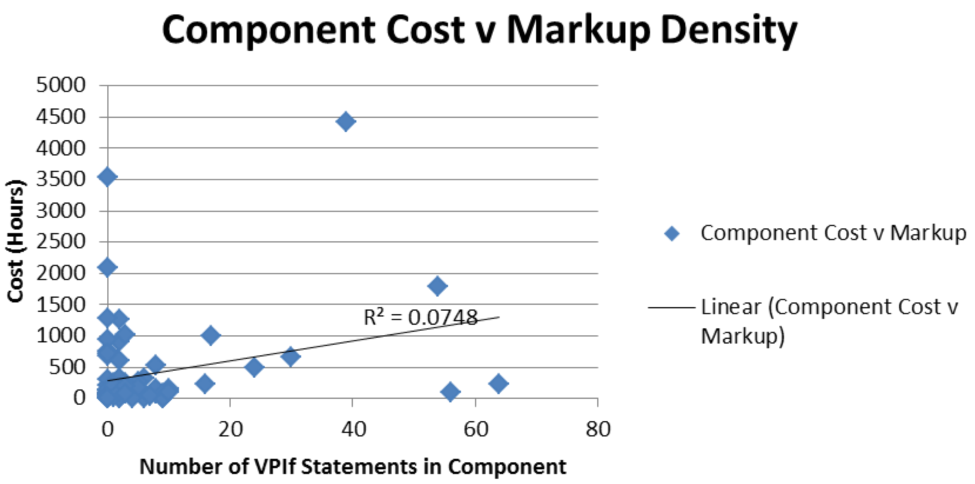


FIGURE 85 AMOUNT OF CODE MARKUP IN A COMPONENT VS COMPONENT COST

We can see from this analysis that none of these factors could be said to closely correlate with the cost of the components. The factor with the closest correlation is the number of options in a component (with an R^2 correlation of approximately 0.4) but this is still very weak.

We also compared the component development costs with both the code size in SLOC and the code complexity (McCabe). This comparison was performed for both variable and non-variable components. These comparisons can be seen in Figure 86 to Figure 89.

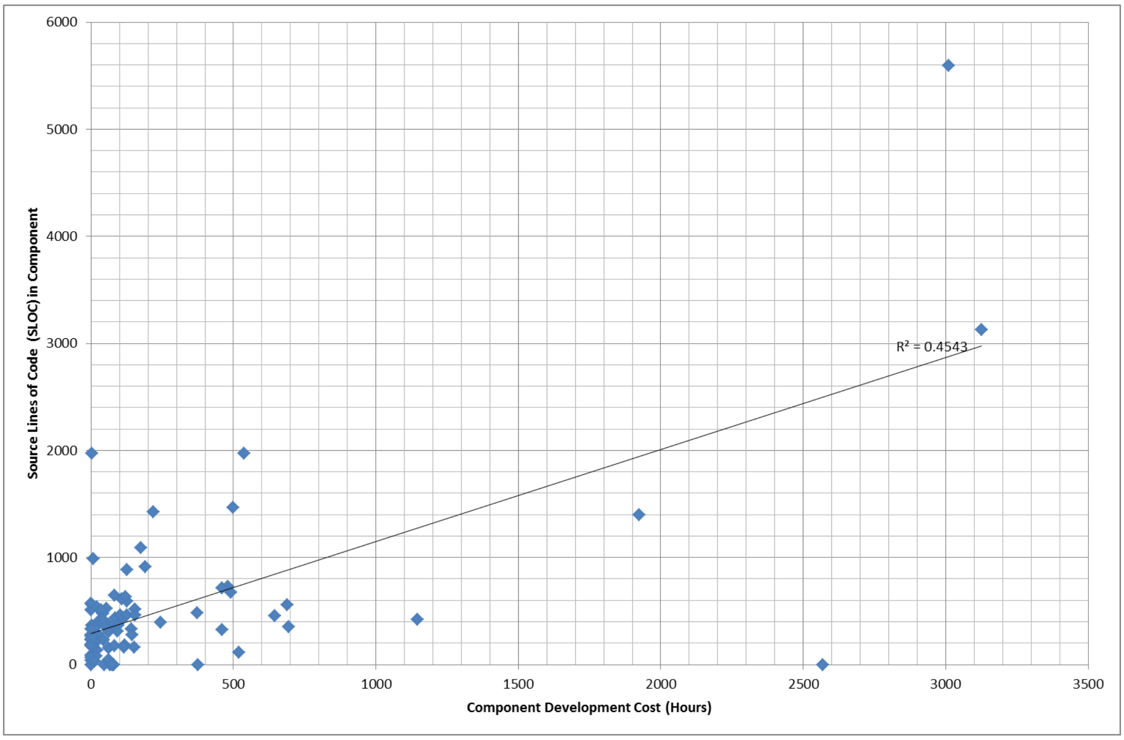


FIGURE 86 SLOC V COMPONENT COST FOR NON-VARIABLE COMPONENTS

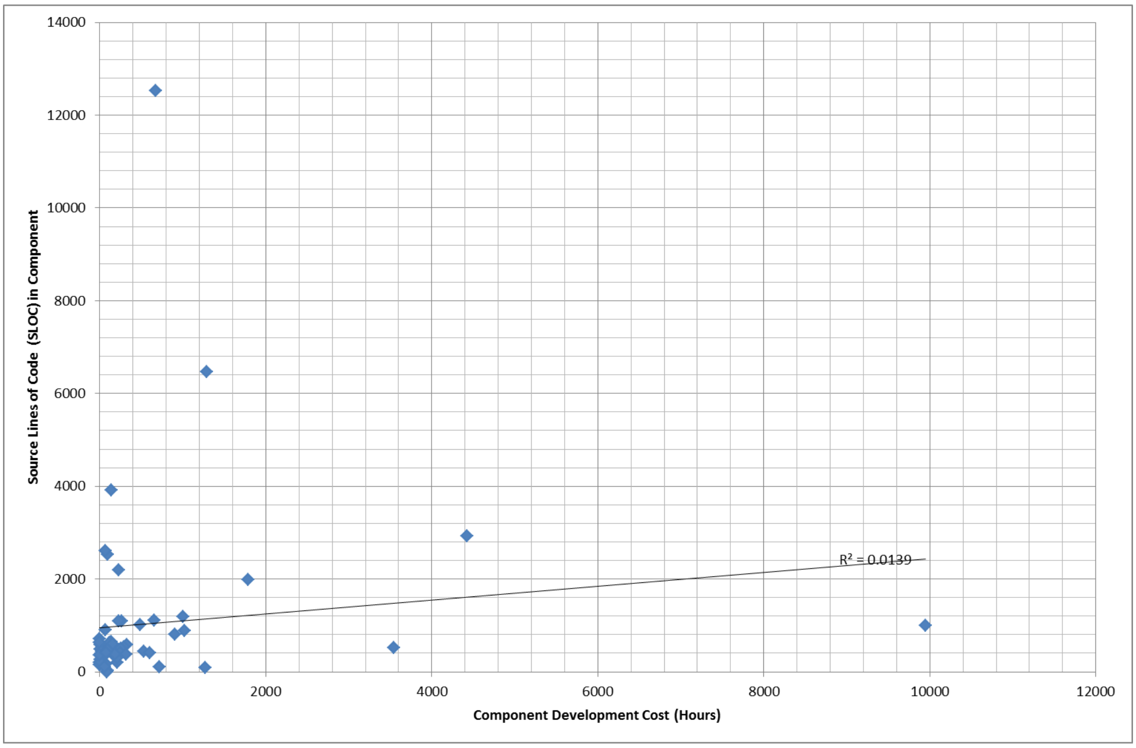


FIGURE 87 SLOC VS COMPONENT DEVELOPMENT COST FOR VARIABLE COMPONENTS

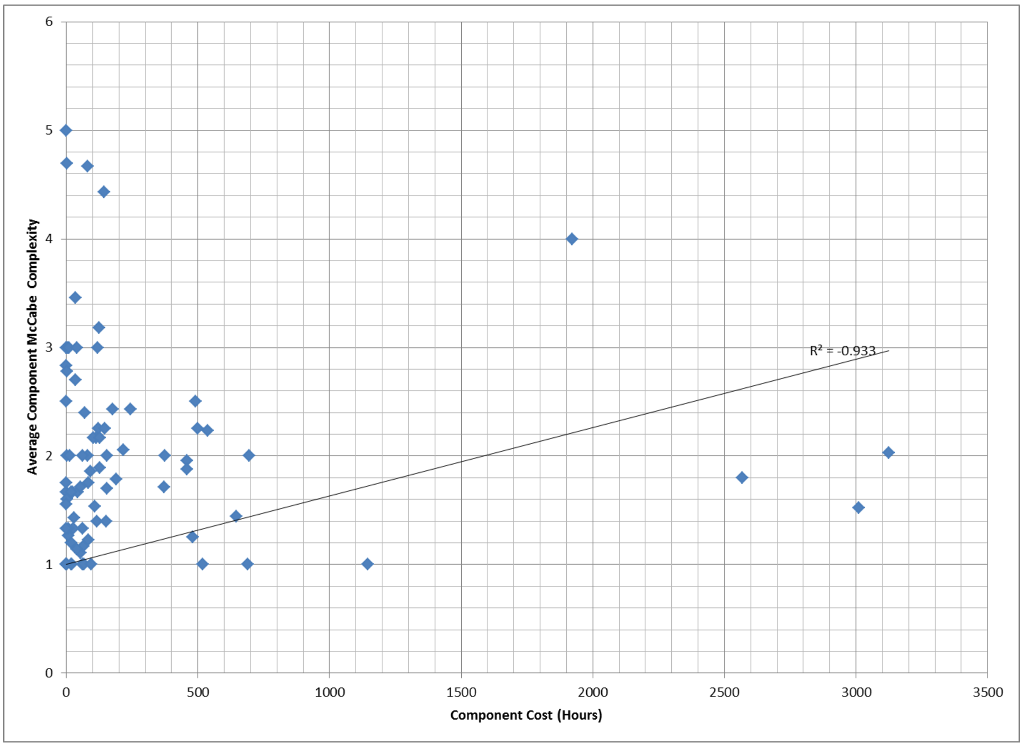


Figure 88 Average McCabe Complexity vs Component Cost for Non-Variable Components

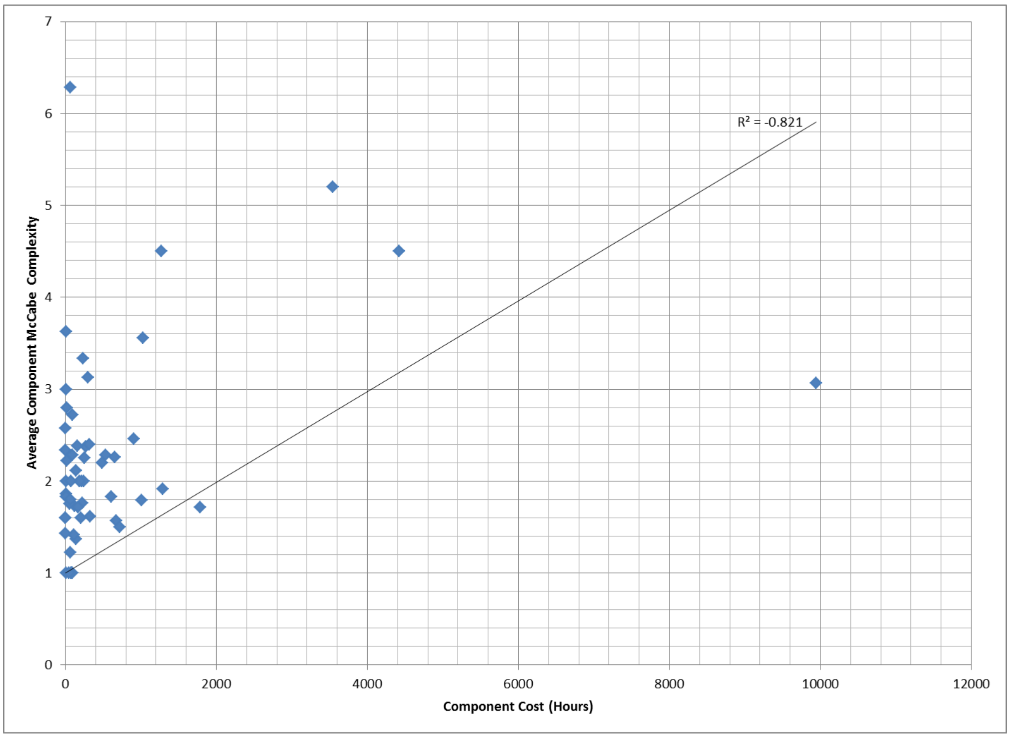


Figure 89 Average McCabe Complexity vs Component Cost for Variable Components

Again, there is no close correlation between size or complexity and component development cost to be seen from this analysis. It is interesting that size and cost for non-variable components are significantly more closely correlated (R^2 = 0.45) than for variable components (R^2 = 0.01).

6.4 Qualitative Evaluation Results

We have seen in the previous section how the Trusted Product Lines approach has affected the costs of the software development process. Now we try to assess the effect of this approach on the quality of the software product produced. This is a qualitative assessment based on the results of a number of internal audits undertaken on the project deploying the product line assets.

6.4.1 Pre-SOI2 Audit Findings

The company has a policy of performing "pre-audits" prior to any regulatory audit. These are held to ensure that the project is ready for that level of scrutiny and are performed by senior Software Quality Assurance (SQA) staff and the company DO-178B experts. Typically, the findings of these audits are more extensive than the regulatory audits as the auditors have more in-depth understanding of the processes and the product.

The criteria used to determine if a project is ready for a pre-SOI2 audit is as follows [127] :

- 50% of Requirements written and reviewed
- 50% of Design written and reviewed
- 50% of Code written and reviewed
- All necessary requirements and design procedures/standards written and reviewed
- All outstanding SQA actions against design processes closed

The auditors expect to be able to follow a number of traceable "threads" through the development artifacts from systems requirement to the implementing code.

Two pre-SOI2 audits were held on the project in question [127, 128], and observations were made regarding the project compliance to the objectives of DO-178B. The relevant audit findings are discussed below.

6.4.1.1 Initial Audit

The initial pre-SOI2 audit was held at the request of the project, although the auditors noted that the criteria for SOI2 had not been fully met and, as a result, they recommended a follow-up audit should be held prior to agreeing the project was fit for scrutiny by the regulator. Amongst a number of issues found during the audit, the following observations were raised that are pertinent to the product line approach [127]:

1) *Ensure that reviews are managed such that evidence exists of who answered each review question.*
2) *Ensure that the evidence of review for the High Level and Low Level software requirements is complete and explicitly addresses the DO178B requirements*

The evidence of review under question was gathered both on the project and during product-line asset generation.

6.4.1.2 Follow-up Audit

The follow-up pre-SOI2 audit was held three months after the original, with the project artefacts in a state compliant with the audit entry criteria. The main product-line related finding of this audit is as follows [128]:

1) *Provide evidence that shows how the review evidence of a generic artefact deployed on a project meets the DO178B requirements.*

This is a more explicit action than raised at the original audit, and it summarises and encapsulates the problems the auditors had with the use of generic and reusable assets in general (especially where transformation was involved). **It was difficult for the project to demonstrate the applicability of the verification evidence for the product-line asset on the deployed project** (to the satisfaction of the auditors).

The response to this was for the deployed product line assets to be re-reviewed by the project to ensure applicable verification evidence was available to support the regulatory approval. This naturally reduces the value of the product line assets to the deploying projects.

6.5 Trusted Product Lines Argument Framework

Chapter 1 summarised the five main challenges/strategies that we identified as being fundamental to the successful application of Trusted Product Lines. As part of the research evaluation, we now describe how these challenges have been addressed by our approach. Figure 90 maps the high-integrity re-use issues raised by the FAA and Leveson and Weiss (as identified and enumerated in Chapter 2) onto the Trusted Product Lines framework to indicate where these need to be been addressed and/or discharged. It should be noted that we have shown via the data analysis in this chapter that the assumption "Assume Minimum of 3 Products" stated in Figure 90 is valid in the context of our Trusted Product Lines approach.

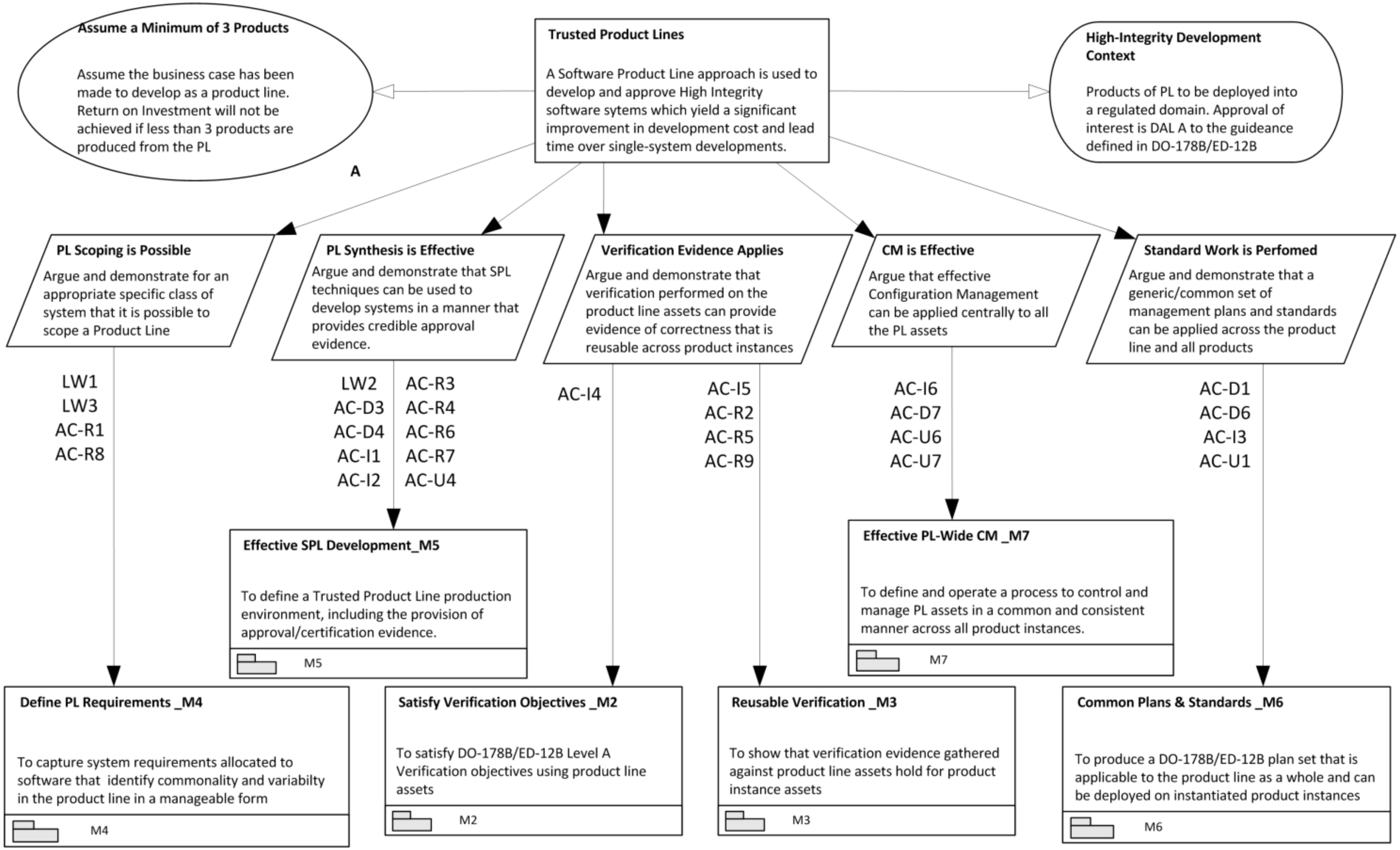

Assume a Minimum of 3 Products
Assume the business case has been made to develop as a product line. Return on Investment will not be achieved if less than 3 products are produced from the PL
A
Trusted Product Lines
A Software Product Line approach is used to develop and approve High Integrity software sytems which yield a significant improvement in development cost and lead time over single-system developments.
High-Integrity Development Context
Products of PL to be deployed into a regulated domain. Approval of interest is DAL A to the guideance defined in DO-178B/ED-12B
PL Scoping is Possible
Argue and demonstrate for an appropriate specific class of system that it is possible to scope a Product Line
PL Synthesis is Effective
Argue and demonstrate that SPL techniques can be used to develop systems in a manner that provides credible approval evidence.
Verification Evidence Applies
Argue and demonstrate that verification performed on the product line assets can provide evidence of correctness that is reusable across product instances
CM is Effective
Argue that effective Configuration Management can be applied centrally to all the PL assets
Standard Work is Perfomed
Argue and demonstrate that a generic/common set of management plans and standards can be applied across the product line and all products
LW1
LW3
AC-R1
AC-R8
LW2
AC-D3
AC-D4
AC-I1
AC-I2
AC-R3
AC-R4
AC-R6
AC-R7
AC-U4
AC-I4
AC-I5
AC-R2
AC-R5
AC-R9
AC-I6
AC-D7
AC-U6
AC-U7
AC-D1
AC-D6
AC-I3
AC-U1
Effective SPL Development_M5
To define a Trusted Product Line production environment, including the provision of approval/certification evidence.
M5
Effective PL-Wide CM _M7
To define and operate a process to control and manage PL assets in a common and consistent manner across all product instances.
M7
Define PL Requirements _M4
To capture system requirements allocated to software that identify commonality and variabilty in the product line in a manageable form
M4
Satisfy Verification Objectives _M2
To satisfy DO-178B/ED-12B Level A Verification objectives using product line assets
M2
Reusable Verification _M3
To show that verification evidence gathered against product line assets hold for product instance assets
M3
Common Plans & Standards _M6
To produce a DO-178B/ED-12B plan set that is applicable to the product line as a whole and can be deployed on instantiated product instances
M6

Figure 90 Annotated Trusted Product Lines GSN Argument

6.5.1.1 PL Scoping is Possible

We described this challenge as being able to clearly identify a product line scope; i.e. be able to define robustly when a specific product is a member of the product line (and equally when a product is **not** a member). In addition, it must be possible to identify common parts of the product line (those aspects that are present in all members) and the variable parts (aspects included/excluded by selection).

We noted in Chapter 1 (1.3.1) that this was primarily an engineering challenge, not requiring significant research investment. Much of the scoping activity is problem-domain related, and thus not addressed explicitly by the work described in this thesis. However, we have provided explicit support to address the solution-domain issues, via our decision contract approach:

- We clearly identify those components that contain variability, via the existence of a decision contract.
- We clearly map those decisions onto the variable parts of the component with navigable associations in the UML model.
- We provide clear traceability back to the High Level Requirements for both common and variable parts of the components.
- We provide automatically, via transformation, the applicable subset of the traceability for the instantiated component.

In this way, the solution-domain issues raised by Leveson and Weiss (LW3) and the FAA in AC-148 (AC-R8) are addressed by our approach.

6.5.1.2 PL Synthesis is Effective

We described this as a demonstration of our ability to apply product line synthesis techniques to the creation of product instances from artefacts developed for the product line. The Trusted Product Line development and synthesis approach has to take into account the characteristics of typical high-integrity development projects. The product line development approach chosen must be capable of providing credible approval evidence for the instantiated product. This is a key message of Trusted Product Lines, and we have successfully demonstrated this in our approach – specifically by the following:

- We base our approach on a proven, certifiable software engineering process based on UML, SPARK and model-to-text code generators that have been demonstrated in use on current systems.
- We have defined a reference architecture that can host components containing variability but also instantiate products whose architecture as deployed reflects current, certified products.
- We include the SPARK information flow annotations in the component design, including explicit support for their variability and instantiation into the product source code.

- We have defined a decision contract approach that makes explicit the allowable variability in the component, with navigable links to all the points of variation in the component to aid review and analysis.
- Traceability links for both common and variable parts of the component are supported, allowing the variation to be justified and reviewed.
- Documentation can be produced for the product line component, allowing review and analysis pre-instantiation, and for the product-instance component, providing the DO-178B/ED-12B Low Level Requirements artefacts to support certification.
- Negative variability ensures that the review and analysis of the product line artefacts sees the full scope of the variation, and understands how particular decision options (and combinations) would affect the instantiated product.
- We have defined a deployment process that captures how products are built – i.e. from product line components allocated to processors, and with specific options selected to resolve the component variability.
- We have produced transformations that create the Low Level Requirements and Source Code artefacts for specific products from an architectural model that contains deployed components and resolved decisions.
- We have demonstrated in practice that this process is deployable on industrial-sized projects with large development teams who, in many cases, were unfamiliar with product line techniques prior to the deployment of this process.

Our approach has specifically addressed the associated PL synthesis issues raised by Leveson and Weiss (LW2 – via the reference architecture and decision contract) and the FAA in AC-148 (AC-D4, AC- I1, AC-I2, AC-R3, AC-R7).

We remove the need to consider the following, as we are developing as a product line and not as a reusable software component: AC-D3, AC-U4, AC-R4 and AC-R6.

6.5.1.3 Verification Evidence Applies

This challenge relates primarily to reducing product verification costs via the product line approach, whilst retaining the ability to demonstrate the applicability of the verification evidence to the product instance being certified. We have not addressed verification by test explicitly in this research, however our approach "does no harm", in that the process produces artefacts that would enable a traditional verification approach to be followed on the instantiated product. Theoretically, we should be increasing the maturity of products instantiated using our approach (reducing the cost of what the company terms "scrap and rework") as we enable review of product line assets prior to instantiation. However, as we have seen in section 6.5, it may be difficult to demonstrate the applicability of this verification to instantiated product. We address this in Chapter 7.

6.5.1.4 CM is Effective

The effectiveness of the Configuration Management and Change Control processes to manage the product assets and associated process evidence is key to the successful application of a Trusted Product Line Approach. As we noted in Chapter 1, this is primarily an engineering challenge rather than one that required novel research, and therefore we have not addressed this further. Furthermore, there is nothing introduced in our approach that should provide additional CM challenges over and above those existent already with the configuration of complex models.

6.5.1.5 Plans, Processes and Procedures are Standardised

The consistent management of the engineering process across a product line development is a significant challenge, as discussed in Chapter 1 (Section 1.3.5). However, it was not specifically the subject of the research described in this thesis and therefore will not be considered further.

6.5.1.6 Remaining High-Integrity Reuse Issues

The following issues raised by the FAA in AC20-148 and Leveson and Weiss are not addressed by the strategies and goals in the trusted product lines approach: LW4, AC-D5, AC-U2, AC-U3, AC-U4. This is because in general these issues need to be addressed at the **system** level rather than the software level.

6.6 Conclusions

In this chapter, we have provided both qualitative and quantitative analyses of the effectiveness of the product line approach as defined previously in this thesis. We have analysed the development effort expended on the various project phases and compared and contrasted this between project-specific and product-line assets. We have analysed the development cost differential between components containing variation and non-variant components. Finally, we have started to assess the quality level and regulatory compliance evidence of the product line assets.

We can draw the following conclusions from this analysis:

1. The relative size of the development process phases does not change to any significant degree when designing product line assets/variable components.
2. The absolute cost of developing a variant component is greater than developing a specific component. The data indicates that the average cost differential is 2.6 times. This would appear to be in line with the product line industry heuristic of a payback on investment after 3 products have been deployed from the product line. It is interesting to question whether the 2.6 times cost differential is peculiar to or dependent on the high-integrity nature of the domain. Given that there has been no distortion or re-profiling of the development process, then it is reasonable to say that all phases of the development process increase by the same relative amount. Therefore, it can be assumed there is nothing inherent in the high-integrity process that contributes to this cost increase. It can be postulated, therefore, that a similar move to product line development within a different

domain would result in a similar relative development cost increase; there is little inherently "high-integrity specific" in this relative cost observation.

3. It has been difficult for projects to satisfy independent auditors that the verification evidence for product-line assets is applicable when deployed.

A casual observer may argue that due to the experimental data presented in this chapter being based on a single product instantiation, this weakens the conclusions that can be drawn. As we discussed in chapter 1, this type of applied industrial research often consists of "N-of-1" studies [13] and, indeed, we have seen that comparative evidence is difficult to gather. However, this weakness is outweighed by the size, complexity and diversity of the product being developed, and the nature of the development process followed. The development was performed using essentially two parallel teams; one undertaking a product-line approach containing variability and the other a single-product based development. This allowed other environmental factors to be discounted when analysing the comparative data. As this development was undertaken by a typical engineering organisation for a sizable, important product further provides credibility to the experimental data.

In summary, the analysis to date has shown that the Trusted Product Lines approach does not significantly weaken the economic case for a product line approach as compared to other industries. However, there are opportunities to ensure the quality level necessary for regulatory compliance that may significantly improve the economic argument. In the rest of this thesis, we discuss approaches that may address the issues related to the applicability of evidence at deployment time.

7 Property-Preserving Transformations

The use of transformations to enable a product to be instantiated from assets designed as part of a product-line is fundamental to the approaches we have discussed so far in this thesis. Evaluation has shown that, whilst this may be a valid approach in general, the ability to claim prior verification evidence against a transformed asset is problematic. This is due to a lack of assurance that the transformation has not introduced an error into the product. A viable Trusted Product Lines approach requires the ability to guarantee the correctness of a transformation with respect to a defined set of properties of the input model. This guarantee must be independent of the particular input model used. This leads us to investigate the implementation of Property-Preserving Transformations.

7.1 The Challenge of Property-Preserving Transformation

Ideally, the instantiation of software products from a product line would be performed by trusted transformation techniques. This would mean that, for a given set of verified input product-line models and a set of product selection criteria (e.g. feature selections), the transformation would be guaranteed to correctly instantiate the product instance, and that instance would be self-consistent, correct, and valid. Current transformation tools do not provide this level of assurance.

Within an aerospace context, such an automated transformation would be regarded as a development tool, and be subject to DO-178B/ED-12B[4] tool qualification objectives if its output was not separately verified (tool qualification is discussed at length in the next section.) However, requiring extensive separate re-verification of the instantiated product would begin to undermine the business case for the product-line approach. The challenge is to make verification evidence gathered for the product-line clearly applicable to a product instance whilst using a cost effective, affordable transformation to perform the product instantiation. As discussed in Chapter 1, we must to construct arguments that a product instantiated from the product line is fit for purpose whilst minimising the economic cost of producing that product. Those arguments have to convince developers, regulators and users that the following hold :

- Applicability – the requested product has been instantiated.
- Conformance – all artefacts conform to the required and declared standards.
- Compliance – all artefacts demonstrably comply to their requirements, specifications and architectual constraints.

In the literature, Jackson et al. [129] discuss an approach to ensuring a reused transformation still preserves the properties of the original transformation, but do not address how to show that the transformation itself retains properties from source to transformed artefact. In fact, their approach is predicated on having performed verification on the original transformation to "ensure that they behave as desired".

7.1.1 Trusting Tools –The Role of Tool Qualification

Civil aerospace is a typical example of a regulated domain, in which software is developed to a set of industry guidelines and is subject to audit and approval by regulatory authority or body. Prior to entry into service, civil avionics software is required to be approved by an airworthiness authority, a process more commonly known as "certification". This approval process typically takes the form of a set of audits designed to demonstrate that the software has been developed in accordance with the guidance of DO-178B/ED-12B "Software Considerations in Airborne Systems and Equipment Certification" [4].

The annex A tables in DO-178B/ED-12B provide guidance on objectives for each of the software development processes and how they vary with assurance level. For example, DO-178B/ED-12B table A-5 lists the objectives associated with "Verification of the output of the software coding and integration processes". Let us assume that most product line developments will include transformations that produce product source code. We need to ensure that the transformation does not destroy or compromise any verification evidence that has been gathered for the product line against these objectives.

The table A-5 objectives for source code are as follows:

- Source Code complies with low-level requirements.
- Source Code complies with software architecture.
- Source Code is verifiable.
- Source Code conforms to standards.
- Source Code is traceable to low-level requirements.
- Source Code is accurate and consistent.

(Note that "Low-level requirements" is DO-178B/ED-12B terminology for the software design data.)

Typically, compliance to these objectives is demonstrated by "review and analysis", where review is usually a checklist-driven peer review of the artefact, and analysis is an automation verification that a given property holds (or otherwise) for an artefact.

Whilst it is perfectly possible to perform this review and analysis process on a product-line asset, if that asset undergoes transformation when instantiated it becomes difficult to argue that the evidence still applies to the resultant asset. This is only possible if the transformation is "trusted" and typically, the basis of that trust would take the form of tool qualification evidence – otherwise additional verification of the transformed asset is required.

DO-178B/ED-12B includes guidance on the use of tools within the software development process. Wherever a tool is used to automate part of the software development activity, and its output is not separately verified, then that tool requires **qualification**. The objectives for tool qualification vary dependent upon whether the tool is a verification or development tool. Verification tools cannot introduce an error into the software product; they can only fail to detect an error. Therefore, the qualification requirements for verification tools are relatively straightforward, and take the form of a simple acceptance test of the tool against a set of operational requirements plus strict revision control. Development tools, however, produce output that forms part of the software product and therefore are capable of introducing an error into the product (for example automatic code generators producing source code). Development tools whose output is not separately verified are required to be developed to the same assurance level as the software product they are used to develop. Development tool qualification is consequently very costly and is sometimes impossible to perform. Any use of pre-developed libraries and operating system components within a tool makes the availability of qualification evidence highly unlikely.

This causes significant problems for organisations wishing to develop and/or use qualified development tools, particularly for level A projects. As DO-178B/ED-12B provides objectives for the software development **process** to follow, it is almost impossible to retrospectively provide qualification evidence for an existing tool. In addition, the high-integrity software development tools market is so small that it is hardly ever commercially viable to develop a tool compliant with DO-178B/ED-12B Level A objectives. Currently the only commercially available development tool that can be qualified to DO-178B Level A is the SCADE “pictures-to-code” environment produced by Esterel [59].

7.1.2 Static Verification of Transformation

If tool qualification is prohibitively expensive (or not even possible) for the transformation environments used to implement product lines, would it be possible to implement separate, automatic verification paths to validate the result of the transformation? This may be possible via performing a type of “regression analysis & review” (analogous to regression testing) on the transformed artefact. This may allow an argument to be made that prior verification evidence still holds, and could be more cost effective than complete re-verification of the artefact. Alternatively, multiple, diverse transformations could be performed and their results compared (analogous to n-version programming). This would provide confidence that a single transformation approach had not introduced error into the product.

A combination of these two approaches, implementing diverse transformation techniques and automated post-transformation analysis, may be a viable and credible approach to producing a sufficient level of evidence without the expense of full tool qualification for the transformation. We illustrate the potential of such an approach in the following example.

Firstly, we look at static verification of transformed assets to determine if this can increase confidence in the transformation. We have already seen that our model transformation process has been used successfully on a large avionics product line to develop components including variability. However, it has been difficult to reduce the project-specific verification effort using this approach. This is due to the problems described earlier; the transformations cannot be trusted to preserve properties from product-line assets to instantiated components. However, we can start to illustrate how static verification may be used to demonstrate the correct composition of components with a simple example.

7.1.2.1 IgniterControl Component Example

Consider the simple software component that controls the ignition demand for a gas turbine engine that we introduced in Chapter 5 (Section 5.3.2.1). We elaborate on the functionality of this component here.

The ignition demand can be in one of three states: no ignition, low ignition or high ignition. Igniters are fitted in pairs, although typically only one is used at any one time to ignite the engine (this is “low ignition”); however both igniters can be commanded on in certain circumstances (this is “high ignition”).

This is a software product line component, and there are some common and variable aspects of the ignition control scheme. The common aspects are the determination of the ignition level from the demands from the automatic start and manual start systems. There are also variable features: high ignition may be demanded for the following optional scenarios:

- Automatic Relight in the event of engine flameout
- Quick Relight in the event of the pilot inadvertently shutting the engine down in flight
- Water Ingestion Protection in the event of significant levels of water passing through the engine core

These optional features can be included in the component in any combination, dependent upon the type of engine and the type of airframe into which the engine is installed.

This component is implemented in SPARK. If we look at the SPARK annotations for the common parts of the component, the global derives annotations look as follows:

```
--# derives IIgniters.igniterCmd from
--# IStarting.igniterCmdAutostart,
--# IStarting.igniterCmdManualStart,
--# IThrustControl.fuelDipIgnInhibit,
--# IThrustControl.primingInProgress;
```

The final value command to the ignition system is derived from the starting system (manual and automatic) and the state of the thrust control system (whether the fuel is being primed or we are specially inhibiting the ignition during a fuel dip manoeuvre). Validation of the requirements and variation points requires engine and control system knowledge; discussion of such issues is outside the scope of this paper.

When we select an optional feature, for example the water ingestion protection, this annotation changes as follows:

```
--# derives IIgniters.igniterCmd from
--# IStarting.igniterCmdAutostart,
--# IStarting.igniterCmdManualStart,
--# IEngineEvents.igniterCmdWaterIngest,
--# IThrustControl.fuelDipIgnInhibit,
--# IThrustControl.primingInProgress;
```

The line in **bold** declares that the water ingestion state of the EngineEvents sub-system now contributes to the derivation of the igniter command. Similar changes to the “derives” annotations occur when other features are included in the component.

To implement this component as part of a product line, we first create a UML component including the optional features as a decision contract (see Figure 91). Within the decision contract, the component publishes the available variability decisions as first-class model elements (the nodes denoted with the grey star icon in Figure 91). We then indicate which parts of the model are variable by use of the «PL variation point» stereotype which can be attached to any model element of relevance to the code generation process. These indicate the parts of the model that are included in a product instantiation if the associated selection expression (an expression in terms of the decisions in the decision contract) evaluates true.

Using model-to-model and model-to-text transformations, we can generate instantiations of these product line components that take into account the selected decisions on a particular project. This works well for first-class UML model elements, however the implementation requires the capture of information that cannot be modelled within the standard UML meta-model – for example the code bodies and SPARK annotations. These are typically captured via text fields in the model, and inserted into the generated code via the model to text generator. Variability in these text fragments is denoted using a simple mark-up language and the code generator delegates the processing of this type of field to a text pre-processor (this process has been described in detail in Chapter 5).

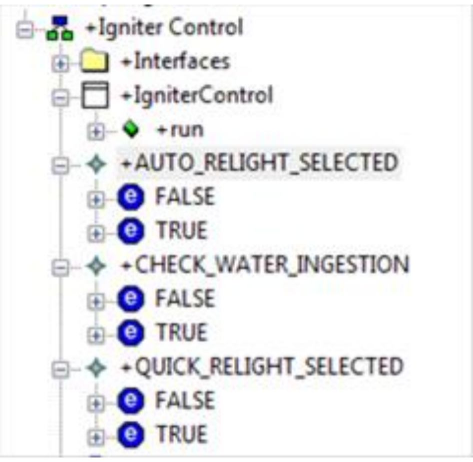


FIGURE 91 COMPONENT STRUCTURE SHOWING DECISION CONTRACT FOR IGNITERCONTROL COMPONENT

This approach has worked well in practice, and can successfully generate product-specific components with matching SPARK annotations. The marked-up SPARK “derives” annotation prior to instantiation is shown below.

```
VPBegin
{# IIgniters.igniterCmd from
}
VPIf AUTO_RELIGHT_SELECTED = TRUE
{# IAutoRelight.igniterCmdAutoRel,
}
VPEndIf
VPIf CHECK_WATER_INGESTION = TRUE
{# IEngineEvents.igniterCmdWaterIngest,
}
VPEndIf
VPIf QUICK_RELIGHT_SELECTED = TRUE
{# IStarting.igniterCmdQuickRel,
}
VPEndIf
{# IStarting.igniterCmdAutostart, IStarting.igniterCmdManualStart,
# IThrustControl.fuelDipIgnInhibit,
IThrustControl.primingInProgress}
VPEnd
```

(Note that the code generator automatically adds the comment marks and “derives” keyword.)

The simple mark-up language used to denote regions of optional text can be clearly seen here. The keywords of the mark-up language start with VP, and the regions of text to be passed through the transform are contained within braces. The mark-up allows expressions to be associated with conditional (VPIf) statements. If the associated expression evaluates true with respect to the product options, then the associated text region is passed through to the product component.

A fragment of the code body that implements the product line component is shown below. This demonstrates the use of the mark-up language to control the inclusion or otherwise of Ada source statements in the final product:

```

VPIf AUTO_RELIGHT_SELECTED = TRUE
{
if IAutoRelight.Get.igniterCmdAutoRel = HIGHIGN then
      lclDualIgnCmd := TRUE;
   end if;
}
VPEndIf

VPIf QUICK_RELIGHT_SELECTED = TRUE
{
   if IStarting.Get.igniterCmdQuickRel = HIGHIGN then
      lclDualIgnCmd := TRUE;
   end if;
}
VPEndIf
```

As can be seen in the above examples, the mechanism for optionally including SPARK annotations and Ada source code is using the same mark-up language and is processed by the same text transformation tools. Therefore, due to the potential for common-mode error in the transformation of both code and SPARK annotation, using a successful information flow analysis on a product component is weak evidence of a correct (property-preserving) transformation. However, there are still advantages to this type of analysis as is discussed in the following section.

7.1.2.2 Detection of Ineffective Product Variants

An ineffective product variant can be defined as one whose particular set of product-specific features (decisions) results in a product instance where one feature implementation is rendered ineffective by another. Deploying each feature in isolation would result in a functional product; it is the particular combination of features that is ineffective. The interaction of features/variation points can be very subtle and difficult to identify using peer review techniques in isolation, but are ideal error categories for identification by a static analysis approach.

In the following simple example, the software designer decides to “optimise” the implementation of the igniter control component to assign directly the lclDualIgnCmd variable to the result of the enumeration checks as follows:

```
VPIf AUTO_RELIGHT_SELECTED = TRUE
{
  lclDualIgnCmd:= (IAutoRelight.Get.igniterCmdAutoRel = HIGHIGN);
}
VPEndIf

VPIf QUICK_RELIGHT_SELECTED = TRUE
{
   lclDualIgnCmd:=(IStarting.Get.igniterCmdQuickRel = HIGHIGN);
}
VPEndIf
```

Instantiating each of those decisions in isolation would result in a functional product implementation, however instantiating a product with both features/decisions selected would result in the following code:

```
lclDualIgnCmd:= (IAutoRelight.Get.igniterCmdAutoRel = HIGHIGN);
lclDualIgnCmd:= (IStarting.Get.igniterCmdQuickRel = HIGHIGN);
```

The first assignment to lclDualIgnCmd is now completely ineffective; however, this would be caught by a data-flow analysis of the instantiated product such as that performed by the SPARK Examiner.

A snippet of the SPARK Examiner report on this code is shown below:

```
78  lclDualIgnCmd := (IAutoRelight.Get.igniterCmdAutoRel = HIGHIGN);
    ^4
!!! (  4)  Flow Error         : 10: Ineffective statement.
```

This is obviously a trivial example that **should** be caught by the peer review of the product-line asset, however one could conceive of much more subtle interactions between features/decisions that would be very hard to detect by code inspection of the pre-transformed asset alone.

The above example could be found by a data-flow analysis of the source code in isolation; however, an information flow analysis against a SPARK contract will catch more instances of this type of error. (This is because the information flow contract provides a more precise definition of the required relationship between input and output, and the subsequent analysis would be more sensitive to ineffective variants.) This analysis is significant, as the "optimisation" set out above is, in many ways, the natural and elegant way to produce the code.

7.1.2.3 Detection of Mal-Transformed Product Variants

We have seen how a data-flow analysis can detect the instantiation of ineffective product variants. A different form of erroneous transformation would be the instantiation of product variants with missing functionality; i.e. an omission error. The automatic detection of omission errors requires a means of identifying the required or expected behaviour to compare against the implemented behaviour. Information flow analysis (as implemented by the SPARK Examiner) may help in this regard; this form of analysis compares the actual information flow (simplistically, the relationship between inputs and outputs as implemented in the source code) with a definition of the expected information flow provided in the form of source code annotations ("derives" annotations in SPARK).

This is a simple example of a "design-by-contract" approach where an abstract definition of requirements ("contract") is shown by analysis to hold in the implementation. Given our simple IgniterControl example, the derives annotations require that the Water Ingestion command `IEngineEvents.igniterCmdWaterIngest` is used in the derivation of the final Igniter Command value `IIgniters.igniterCmd`.

```
--# derives IIgniters.igniterCmd from
--# IStarting.igniterCmdAutostart,
--# IStarting.igniterCmdManualStart,
--# IEngineEvents.igniterCmdWaterIngest,
--# IThrustControl.fuelDipIgnInhibit,
--# IThrustControl.primingInProgress;
```

If the code that implements this is omitted from the instantiated source, it will still compile and show no dataflow errors. However, an information flow analysis against the derives annotation will show that there is missing information. The SPARK Examiner produces the following errors in this scenario:

```
!!! (  1)  Flow Error: 30: The variable IEngineEvents.igniterCmdWaterIngest
is imported but neither referenced nor exported.
!!! (  2)  Flow Error: 50: IIgniters.igniterCmd is not derived from the
imported value(s) of IEngineEvents.igniterCmdWaterIngest.
```

Similarly, the erroneous **inclusion** of functionality (i.e. a commission error) would also be captured by this technique (if the effect of the additional code was not reflected in the contract derives annotations).

It is important to remember that in our current approach, both the derives annotations and the code implementation are transformed using the same transformation engine, which may lead to a common mode failure masking this type of error.

7.2 Diverse Transformation, Contracts and Implementation

We reviewed the literature with regard to different types of model transformation approaches in Chapter 2. In particular, we recognised the two approaches for realizing variability in product lines via model transformation: reductive and additive transformations (shown again in Figure 92).

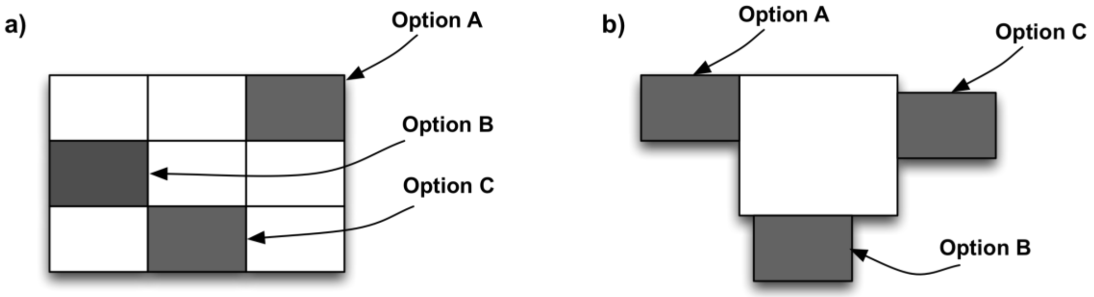


FIGURE 92 REDUCTIVE/NEGATIVE (A) AND ADDITIVE/POSITIVE (B) VARIABILITY EXTRACTED FROM [57]

The transformations currently implemented in our approach (as described in Chapter 5) are reductive, and make use of stereotyped model elements to identify variation points in a UML design model, and the use of text pre-processing to remove unwanted variation in text regions (for example code bodies). However, there remains a potential for common-mode transformation error leading to false-positive static analysis results. This prompted an investigation into whether multiple diverse transformations would lead to a more credible analysis.

This revised approach takes advantage of the properties of both reductive and additive transformations and utilises them with programming languages whose syntax and semantics allow for the separation of contract (interface) and implementation, for example SPARK. This approach is not limited to SPARK however; it is applicable to any language that allows separate contract and implementation.

In general, the component contract specification is declarative; it defines properties that are expected to hold in the implementation of the component. For product line components these contractual properties can have common and variable parts, reflecting the intended variability across the product line. The declarative nature of these contracts makes them ideal for using a positive variability approach. This would involve the declaration of the common part of the contract, then providing additional contract "advice" which is associated with each of the decision options in the decision contract. Given a product configuration (a specific set of decision outcomes) an additive transformation can then construct the product specific component contract.

The detail of the implementation can be created separately to meet the product line contract, but for an imperative language such as Ada (which underpins the SPARK language) this is most easily achieved using negative variability – identifying text regions within the source code that are included or removed when the associated decision options are selected or deselected. A reductive transformation can then be used to generate the product-specific implementation.

Finally a static analysis can be performed to demonstrate conformance of the implementation to the contract. This analysis can be as rigorous as the form of the contract and power of the analysis tools allows. This approach is illustrated in Figure 93.

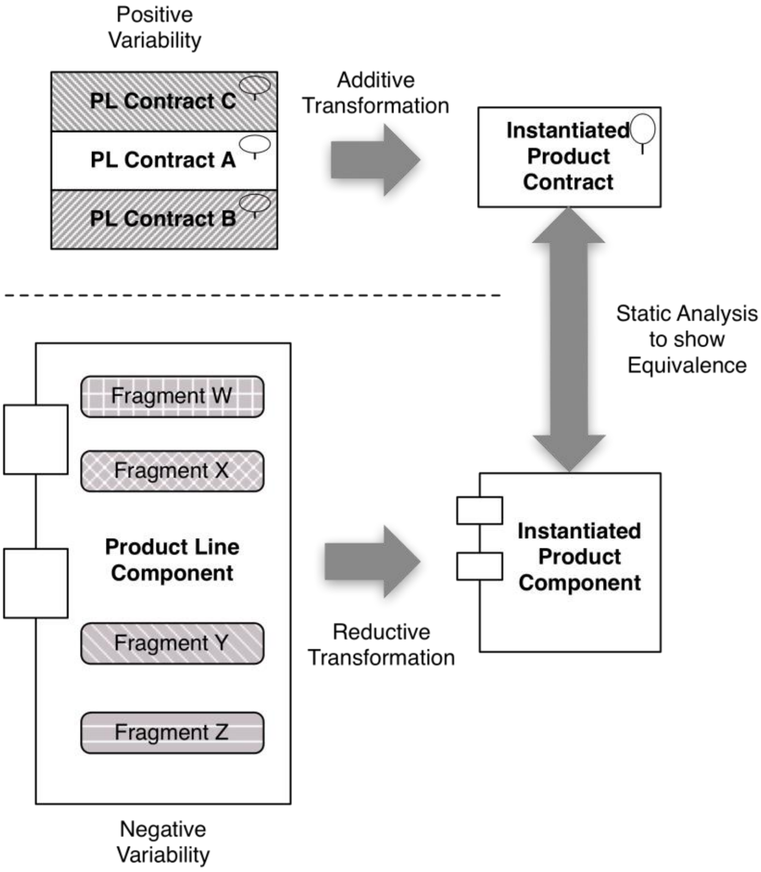


FIGURE 93 VERIFYING EQUIVALENCE VIA STATIC ANALYSIS FOLLOWING THE DIVERSE TRANSFORMATION OF CONTRACT AND IMPLEMENTATION

The potential for using SPARK analysis as a means of demonstrating that properties have been preserved following an instantiation transformation is very attractive, however we have seen in the previous section that due to the common transformation used for both code and annotation, this evidence is relatively weak. It was decided to investigate whether the annotations could be modelled or encoded in a different form, and whether an alternative transformation approach could be used.

It is clear that SPARK information flow annotations are declarative – ordering is not important – and therefore should not be subject to the “point of injection” problems of positive variability. It was also apparent that the information flow effect of decisions in a component could be isolated very easily for the examples we studied. Therefore, it was concluded that rather than modelling annotation variation as negative variability associated with a code sub-procedure, it should be modelled as positive variability associated with the decisions in the components decision contract. Each outcome in each decision would have its associated effect on the information flow within the component explicitly stated. The information flow for a given instantiated sub-procedure would be the combination of the common information flow for that sub-procedure and the set of flows for the decision options selected.

To test this we created a Domain Specific Language (DSL) to capture the positive variability of annotations, and ease the implementation of the compositional transformation. The DSL and associated tools were created using the Eclipse Modeling Project tool XText [130]. The optional annotations for the IgniterControl example discussed earlier can be stated in the new positive variability language are as follows:

```
selecting AUTO_RELIGHT_SELECTED as TRUE {
  operation IgniterControl.run {
    abstract{
        derives IIgniters.ignitionCmd from
         IAutoRelight.igniterCmdAutoRel;
      }
  }
}

selecting QUICK_RELIGHT_SELECTED as TRUE {
  operation IgniterControl.run {
    abstract{
      derives IIgniters.ignitionCmd from
        IStarting.igniterCmdQuickRel;
      }
  }
}

selecting CHECK_WATER_INGESTION as TRUE {
  operation IgniterControl.run {
    abstract{
      derives IIgniters.ignitionCmd from
      IEngineEvents.igniterCmdWaterIngest;
      }
  }
}
```

The language allows for multiple operations to be annotated per decision option, and for abstract and concrete annotations to be provided if required.

This approach has the advantage that it collects together all the annotations that are associated with the decision option into one place. This makes it much easier to review the effect that the selection of a decision option is intended to have on a component, and makes it much easier to spot mistakes in that information flow contract. It brings the declaration of the information flow to where conceptually it should be in the component – the contract.

Once a component has been deployed and the decision options selected, the instantiation annotation is an additive composition of the common information flow per operation, and the annotations for the selected options. This approach has two clear benefits over the text transformation described earlier:

- The total effect of a decision option on the information flow of a component is clear, held in one place and can be peer reviewed and verified in its entirety.

- The information flow for an instantiated component is generated by a different transformation to the component itself. Diversity of transformation means that a successful information flow analysis of the instantiated component provides stronger evidence that the component has been transformed correctly.

This, then, offers the possibility of avoiding common mode failures in tools, and obtaining greater benefit from the product line approach.

7.2.1 Transformation of Behavioural Contracts

So far, we have only considered the use of contracts that describe the intended data and information flow within a component. Languages such as SPARK allow the intended behaviour of components to be modelled as part of the contract. Tool support is provided to enable a formal proof to be performed to show that the component implementation matches its contract specification.

The strongest form of evidence that a component has been transformed correctly would be to use the diverse transformation approach described in the previous section, coupled with a contract that includes a behavioural specification. A formal proof that the transformed component matches its (diversely) transformed specification would provide evidence at least as strong as testing of the component that the transformation had not introduced error into the product.

A fully-worked example of the transformation and proof of behavioural specifications is beyond the scope of this thesis, however the concepts are similar to those we illustrate above.

7.3 Conclusions

We began this chapter with the understanding that a transformation approach to the deployment of product line assets was viable for high-integrity systems, but may not have the level of assurance needed to deliver the full cost-benefits available to commercial product lines. This was due to the need to re-verify the transformed asset to show that errors had not been introduced during the transformation. The level of investment needed to develop a “trusted” transformation has been shown to be prohibitive in most circumstances, and potentially impossible without developing the technology “from scratch”.

We have investigated the possibility of using static verification of the transformed asset to assure its correct transformation, and have shown that this may provide credible evidence that error has not been introduced. In addition the credibility of this evidence may be enhanced by the use of diverse transformations to transform contract and implementation, with their conformance being demonstrated automatically. Finally we have postulated that a contract containing formal behavioural specifications may provide evidence of correctness as strong as function test of the transformed component.

The ability to trust that a transformation has preserved properties of the source model and has not introduced errors into the transformed asset is key to ensuring a Trusted Product Lines approach is cost effective. We have outlined how static verification may be used to provide assurance in the correctness of transformation. Whilst a complete process has not been demonstrated, there is enough experience of the successful uses of SPARK [131] to have confidence in the viability of the approach.

We can contrast the approach outlined here with other approaches to verifying the correctness of model transformation, particularly

- Correctness of transformation by construction
- Testing of transformations

Whilst in general these approaches will increase the quality and reliability of transformations, they would not (in isolation) be acceptable means of compliance with the objectives of DO-178B/ED-12C for qualified development tools. A qualified development tool needs to be developed to the same level as the software it is used to develop. We would need to define “Level A” development and verification processes for the model transformation environments, including, for example, test coverage metrics for the transformation definition languages to ensure there is no “dead transformation code”.

As most transformation environments are open-source, it is very difficult to obtain evidence from their development process to substantiate a tool qualification argument. Therefore, we will continue to rely on the verification of the output of the transformation rather than trust the transformation itself for some time yet.

8 Summary & Conclusions

This final chapter provides a summary of the research described in this thesis and reviews the results obtained against the original hypothesis. The conclusions to be drawn from the research are stated and critically reviewed, and we identify possible further work to expand on the research undertaken.

8.1 Trusted Product Lines Revisited

This thesis has introduced the concept of the Trusted Product Line, where the practices developed and matured in the software product lines community over the past decade can be utilised in a principled manner to develop high-integrity systems. We have highlighted the legitimate concerns that the high-integrity community has over inappropriate software reuse, and have noted that a product lines approach must show how these concerns are addressed or negated in the development processes utilised. The development of a Trusted Product Lines argument has been outlined. This forms a framework for demonstrating that an instantiated product is fit for purpose within a civil aerospace context (it meets all applicable development, verification and management objectives, and product reuse concerns.)

We have shown how a civil avionics product line reference architecture can be defined (as an extension of single-product architectures), and how a component-based approach can be used to populate the architecture with product line assets. In particular we have introduced the novel concept of “decision contracts” which allow a clear definition of the available variability to the user of the asset.

We have developed a model transformation approach that can operate on instances of the reference architecture and components to automatically instantiate a product instance, including instantiating the design documentation to accompany the product. This transformation is not an academic prototype, but has been used to develop a commercial FADEC system of substantial size and complexity over the duration of the research period (and continues to be used at the time of writing). The cost-effectiveness of this approach was assessed via an analysis of the effort data captured during the FADEC system development. The analysis of these metrics indicates that the relative effort expended across the software development process remains similar to the profile for a single-system development, however the cumulative effort for a product-line asset is significantly greater than a product-specific asset. The data analysed appears to be consistent with the industry heuristic of product-line payback over 3 product deployments.

We have discussed the need for property-preserving transformations as a means to ensure that product-line evidence is applicable to product-specific components. We have begun to demonstrate that static analysis may be a cost effective way of demonstrating that a transformed component has retained certain specified properties of the originating product-line component, certainly for languages with a well-defined syntax and semantics (such as SPARK), and especially for languages with mechanisms to separate contact and implementation.

The ability to transform a sub-program's contract and implementation via diverse means and then show equivalence via static analysis would provide a greater degree of assurance in the transformation than a homogeneous transformation of the source code, which is the current state of the art in most software product line support environments today.

There remains work to be done to validate this approach fully, but the initial results appear promising. Firstly, we need to argue precisely which properties of the product line component have been preserved if the static analysis is successful. We can be certain that an instantiated component is valid SPARK and is compliant with its information flow contract. Is this sufficient evidence to be able to claim that, say, a code peer review performed on the product line component is valid against the transformed version and does not need repeating? The precise nature of each verification claim needed to support regulatory guidance (DO-178B/ED-12B [4] for example) needs to be examined in the context of transformed components and the evidence obtained by successful static analysis.

In addition, the applicability of this technique to stronger forms of static analysis needs to be investigated. We briefly discussed the ability of SPARK to perform partial proofs of correctness of the component source code against pre and post-conditions stated in the component contract. Do these pre- and post-conditions lend themselves to compositional transformation in a similar manner to the information flow annotations? If so, this could lead to very strong arguments that the composed component meets its specification via automated proof checking. Whilst there is more to be done, we have outlined an approach to tool qualification that has the potential to improve the payback on use of product lines – perhaps bettering the current industry average of requiring 3 systems to break even on the cost of development.

8.2 Research Hypothesis & Conclusions

To draw suitable conclusions from the research described in this thesis, we need to revisit the research hypothesis described in Chapter 1 (Section 1.4). The hypothesis was stated as follows:

*It is **feasible** to construct product line models which*

> *a) allow the specification of required behaviour (including the identification of common and variable aspects in a product-line)*
>
> *b) allow the definition of a reference implementation architecture which can be transformed into an effective, efficient and analysable product implementation*
>
> *and **enable** suitable supporting **evidence** for certification to be produced, including **effective** verification.*

Through the research described in this thesis, we have demonstrated **feasibility** via the following:

- The definition of a reference architecture for a particular class of high-integrity system (civil avionics FADEC) including a component model that allows the explicit capture and modelling of variability ("decision contracts").
- The design and implementation of a model transformation toolset to support product instantiation.
- The use of the reference architecture, component model and transformation in the development of a commercial FADEC product line.

We have provided an argument framework to **enable** certification claims to be credibly made from product line **evidence**.

We have demonstrated how static analysis techniques may be used to provide **effective** verification of the correctness of transformation when instantiating products.

However, we have also recognised the difficulty of claiming the applicability of verification evidence obtained on a product line asset when used on a product. This was shown in the qualitative review of the product line described in Chapter 6. We have suggested ways of addressing this by adopting transformation processes that can be shown to be property-preserving. Providing cost-effective verification approaches and arguing their applicability remains the biggest impediment to the more widespread use of product line development for high-integrity applications.

8.3 Further Work

A number of necessary or potentially fruitful further areas of research were identified whilst conducting the work described in this thesis. Progress in these areas would enhance the quality and/or productivity of a Trusted Product Line approach.

8.3.1 Heterogeneous Modelling Approaches for Product Lines

The work described in this thesis uses extensions to the UML notation to model the product line reference architecture and components. One of the weaknesses noted in the thesis is the lack of a usable behavioural notation that can be integrated into the model infrastructure and transformed in tandem with the UML structures. This has lead to the component behaviours being modelled as text objects, and transformed using rudimentary pre-processor techniques.

As was discussed in Chapter 2, the embedded, real-time industry makes significant use of tools and notations such as Matlab/Simulink (and, for high-integrity aerospace applications in particular, the Esterel SCADE suite) to model and specify product behaviours. The ability to produce coherent architectural (e.g. UML) and behavioural (e.g. Simulink) models, including a coherent identification of variability, and use these as the basis of product realisation would be of significant benefit to the industry.

8.3.2 Formal Approaches to Product Line Development

We identified the potential advantages of using formal approaches to the construction and verification of components in Chapter 7. This work mainly focussed on the demonstration of correct component construction following transformation. The idea of using formal specifications and performing proof of behavioural correctness of transformed components was postulated, but a fully worked out demonstration of this was beyond the scope of this thesis.

In addition, it may be possible to conceive of formal analysis of product-line components prior to transformation to identify conflicting behaviours and therefore detect invalid product variants.

8.3.3 Compositional Verification for Product Lines

The dynamic testing of a product is one of the primary verification activities required to show the behavioural correctness of the product. It is difficult to perform verification by test on product line assets, especially those containing variation. It is especially difficult to perform integration tests (where components are built into sub-systems and systems) on the product line, when the combinatorial effects of the selected variation compound the problem.

If the tests themselves have to be performed on the instantiated products, is there the potential to provide a set of predefined test cases as part of the product line? In particular, can test cases be compositional; that is, can individual tests be defined for common and variable parts of the components and products, and be composed in a similar manner to component instantiation?

8.3.4 Legacy Support & Obsolescence Management

We have seen, both in this research and elsewhere, that the economic argument for developing a product line is predicated on the development of at least three product instances. In the aerospace industry, the overall airframe development lead-time means that there are relatively few new products developed when compared with commercial industries such as automotive and telecommunications. FADEC manufacturers, for example, may only start a new product development every 2 or 3 years, meaning that could be up to 9 years before 3 instances of a product line enter into service. This is a significant length of time when compared to the rate of change of electronics and software technologies; the product line could be ready for refresh long before the payback period for the initial investment. Indeed it would be difficult to make the initial business case for product lines if the investment did not pay back for 9 years.

However, the rate of change of electronics technology results in another problem for the industry that may be positively affected by a product line approach. Electronics component obsolescence is a major problem when providing systems for aircraft and engines that may have a service life of 30+ years. The continuity of manufacture of electronics systems for these products relies upon strategies such as so-called “lifetime buys” of electronic components. Eventually these systems have to be replaced with more modern versions that can be manufactured and supported. However it is very difficult to make the decision of precisely when to upgrade due to the non-recurring cost of redeveloping the system software. Product line approaches may make the economic case for redevelopment of potentially obsolete systems stronger; if the currently in-service systems are taken as in-scope when developing a product line, then the development cost of a replacement system will be borne primarily by the product line.

Research needs to be undertaken into both the technical and economic arguments for using product lines as a means to cost-effective development of “refresh” and “retrofit” systems. This is clearly an area where product line approaches would make business sense within traditionally long lead-time industries.

8.4 Reflections and Coda

There have been some notable successes with software product lines, particularly in taking cost out of the sustainment of families of complex products. Some of these successes have been achieved in safety-related domains but, to our knowledge, the work described here is the first (published) application of product lines in a domain where formal, independent certification has been carried out, exploiting the product line properties.

The gas turbine control software product line enables the construction of products of the order of 200kLoC of executable code (and nearer 300kLoC, including the SPARK annotations.) At present it has not been possible to demonstrate a true return on investment as insufficient product instances have been produced (recall that new products are only developed every 3-5 years). However, the metrics collected show that the development of the reusable assets costs 2.6 times that of a normal “single product” asset, which suggests that a positive return will be achieved on around three developments, which is the industry norm, despite the extra constraints of certification.

There seem to be several keys to this success, which we believe could transfer to other similar domains, specifically:

- The layering of the architecture, driven by sources of variability and constraints of the physical environment provides a general architectural pattern which could be adopted for other embedded systems;
- The enrichment of the product line concepts with the idea of a decision contract aids in controlling developments and in making the process robust to changes in requirements; this is particularly useful for any development which uses fine-grained components;
- The use of transformational tools helps automate the construction of the product instances, removing some opportunities for human error;
- Where there are certification requirements it is useful to design a verification strategy, which balances the verification activities between generic components and instances (indeed, this may be where the process design should start);
- Attention needs to be given to toolset design to avoid the possibility of single points/common mode failures in the toolset, especially where safety is a concern.

With regard to the latter point, we believe that diversity in transformation helps, but we have yet to demonstrate this fully.

One of the surprising aspects of the introduction of a PL approach has been the attitudes and expectations of the asset designers. The development of the approach concentrated on the technical infrastructure and tools to deliver variation into product designs as explained above. Less effort was dedicated to the training and education of the design staff in the "art" of variability, which has resulted in a number of common issues and misconceptions that have had to be addressed, including:

- "Single products can vary at runtime" – One of the most common misconceptions was that in-built modal or state behaviour was actually variability. If the product had different behaviour, say, on the ground and in flight some designers initially regarded this as variability. It was surprisingly difficult to ensure that they all understood that variability *distinguished between different products* and did not represent different states of the same product.
- "The ability to tune the product performance is variability" – The use of development variables (DVs) enables product instances to be tuned. It was very difficult to get the requirements engineers in particular to understand that the ability to tune and optimise a single product instance was not a variation point (i.e. it does not *distinguish between different products*).
- Inclusion of needless variability – It became clear early in the programme that many of the component designers were including variability that was beyond the scope of the product line. Their rationale was that they been asked to produce a

"reusable" component so they were catering for all (foreseen and unforeseen) eventualities. However, this added cost and complexity to the product assets with unknown (arguably zero) benefit. The scope of the product line needs to be clear to the software development teams, and the component development needs to be closely managed to ensure the variability included is that required to realise the set of products identified.

Whilst these are particular concerns seen in the context of the gas turbine control software product line, these issues may be general enough to serve as guidance (warnings) for those introducing product lines in similar domains.

Future evolution of the product line would need to be carefully planned and executed, particularly with regards to the impact on the re-verification of the instantiated products. As product features evolve, the impact on the solution space components (particularly their coupling and interaction) needs to be understood and managed. However, the decision contract approach should help minimise or decouple the effect of altered, augmented and/or replaced components.

Work is currently under way to apply the process to the reworking of a legacy product, to overcome hardware obsolescence problems. This raises some technical challenges as some of the "accidental constraints", e.g. the use of programming languages, are different. One of the tests of the approach, and perhaps a driver of return on investment, is the ability to deal with such legacy applications.

One key area of future work is verification, in particular to make more use of static analysis (and perhaps formal proof), and to use diverse transformational tools to reduce the need for verification of the delivered product. Another issue is the need to better integrate the different modelling notations, e.g. Matlab/Simulink and UML, to provide a more cohesive functional model of the software. This will avoid the use of model annotations to "supplement" the behavioural description, and enable removal of the model-to-text transformation tools which currently bypass some of the model-to-model transformations. These are both important developments that have the potential to improve the return on investment from the product line approach.

This study of product line approaches for high-integrity software systems was instigated as part of a wider business strategy towards a "family" development approach for gas turbine control systems. It became clear that the application of product line theory in this industry would involve more than a straightforward adoption of understood techniques in a new domain. Although the mode of research may not have been typical for doctoral study, the challenges and approaches described here go beyond "good software engineering" into the advancement of the state-of-the-art and have provided novel and innovative techniques for both the business customers and the wider discipline.

We also wish to continue to enhance and progress Trusted Product Lines; innovative approaches, such as that described for diverse transformation, need to be further researched and demonstrated to fully realise the benefits of product line practices for high-integrity systems.

Appendix A – Development and Modelling of SPARK Programs

SPARK 95 (The SPADE Ada 95 Kernel) [14, 75] (hereafter known as SPARK) is a programming language which aims, by design, to provide a sound basis for the development of high-integrity software systems. SPARK programs, by construction and by analysis, can be shown to be free from certain classes of error, and it is possible to partially prove the correctness of a SPARK program against a formal specification of behaviour. The SPARK language is designed to be compiled using a standard Ada 95 compiler, the compliable parts of the language being a carefully selected subset of the Ada 95 language. SPARK is not just an Ada 95 subset, however; equally important to the language are the **annotations** that are held as stylised comments in the source program. These annotations provide information regarding the intended behaviour of the program, in terms of dataflow, information flow and (optionally) sub-program pre-conditions (predicates expressing constraints on the imported variables) and post-conditions (predicates expressing the relationship between the imported variables and exported variables).

Praxis HIS, the definers of the SPARK language, provides a toolset to support the development and verification of SPARK programs. The SPARK Examiner tool [75] performs various levels of analysis of a SPARK program from simple syntactic checks of SPARK compliance of the source code, through checking conformance of the body of the code to the dataflow and information flow annotations, to producing and partially discharging verification conditions (VCs) to prove compliance of the sub-program to any stated pre and post conditions.

Static Analysis, SPARK and Correctness By Construction

Analysis is the determination that a given system property holds via an inspection (typically automated) of the system development assets. For software assets, this is sometimes termed “Static Analysis” as it does not involve the dynamic execution of the software. Various levels of software static analysis can be undertaken, from simple style checkers through to proof of program correctness against a formal (mathematical) specification of required behaviour. For high-assurance systems, an argument that the software is fit for deployment is aided by the use of programming languages that allow the determination that significant properties hold via automated static analysis.

SPARK is designed to facilitate a “correctness by construction” [132, 133] approach to software development, in which each component in a product is shown to be ”well-formed”. The definition of the well-formedness rules can vary, but they “guarantee a certain consistency between the input and output of each step” within the software development process [133]. SPARK is an annotated subset of Ada – all valid SPARK programs are also valid Ada programs – and are compiled using standard Ada compilers. However, SPARK differs from “full” Ada in two major ways: firstly, the parts of Ada that are “problematic” or difficult to formalise (for example unrestricted tasking) are removed from the language. Secondly, the language supports additional information in the form of annotations, provided as stylised comments. (The Ada compiler ignores the annotations in the compilation process as they form part of the code commentary.) The annotations can be regarded as providing a more complete definition of the software component contract than can be provided in the native Ada programming language. Annotations declare the program intentions with increasing levels of rigour; from declaring the intended data-flow, through information flow to providing a program specification in the form of pre and post-conditions

Barnes [75] contains the following simple example of how SPARK provides additional useful information regarding the intended behaviour of a program. Consider a simple Ada sub-program specification:

```
Procedure Add (X: in Integer);
```

Whilst this is perfectly valid Ada, it provides little in the way of information regarding the programmer's intent for the procedure. SPARK allows the contract for this sub-program to be strengthened with additional information, for example:

```
Procedure Add (X: in Integer);
--# global in out Total;
```

This simple addition of an annotation (as a stylised comment) provides significantly more information than the original prototype. It states that the only global variable the procedure is allowed to access is Total, plus the initial value of Total must be used (in) and a new value of Total must be produced (out). We could be more explicit and provide "derives" annotations that definitively state in the contract how the variables are used in combination, for example:

```
Procedure Add (X: in Integer);
--# global in out Total;
--# derives Total from Total, X;
```

This type of annotation is of more value where the sub-program produces multiple out variables. In addition, we can specify more formal, behavioural contracts that start to specify the required functionality of the sub-program, such as:

```
Procedure Add (X: in Integer);
--# global in out Total;
--# post Total = Total~ + X;
```

Here the post condition states the expected value of Total following execution of the sub-program (the out value) should be the in value of Total (denoted by the trailing ~) plus the value of X

SPARK Examiner tool performs static analysis of SPARK programs to determine whether certain properties hold. The properties analysed are dependent upon the depth of analysis required and the extent of the annotations provided. Firstly, it determines the conformance of the code to the SPARK Ada kernel, i.e. the Ada language subset. It then checks consistency of the code to the provided annotations. This takes the form of an analysis of control, data and information flow. The SPARK toolset can be used to perform partial proof of correctness of a SPARK program against the pre- and post-condition annotations. These typically would be created from a formal specification of the program in a software development process following the full “correctness by construction” method.

This type of analysis demonstrates conformance of a program implementation against a more abstract contract that declares the intended properties of the program. This approach to high-assurance software development may prove useful within a product-line development process context. Of particular interest is the possibility of using static analysis on instantiated product assets to determine that product is complete and correct with respect to defined properties. For high-assurance software, it would be advantageous to use SPARK and the SPARK Examiner to determine that, for example, information flow contracts are met in the instantiated product software. This should provide a high degree of confidence that the product asset has been composed correctly. However if the same transformation is used to instantiate both the program contract AND the implementation, then there is the possibility of common-mode error that may not be detected by the analysis.

The majority of the work described within this thesis uses SPARK as the language of choice for the implementation of the software components. This is for a number of reasons: firstly, the research described here was motivated by the need to develop a product line approach to the development of FADEC systems that currently used SPARK as the implementation language of choice. Secondly, the ability to analyse a SPARK program for conformance with predefined definition of behaviour (in the form of program annotations) is a very useful property when combined with generative programming approaches, as we describe later in the thesis. Finally, for the development of high-integrity software systems, SPARK is natural choice for the principled programmer.

Modelling SPARK Programs with UML

Amey and White [77] describe an approach to augmenting the UML with a profile that allows SPARK language concepts to be represented within class diagrams. This, combined with a standard Ada 95 UML profile, also provides sufficient information in the model to allow the template-driven code generation of SPARK programs from the model representation. Their work is modelled and extended in this chapter to show how product line architectures and components can be represented. Appendix B describes in detail how product instances may be automatically generated from these models.

The UML meta-model can be extended to model task or domain specific concepts using **profiles** [12]**.** UML profiles collect together sets of modelling extensions in the form of **stereotypes** and associated attributes known as **tag definitions**.

In Figure 94 we can see how the UML concept of a **Class** containing 0 or more **Operations** can be extended to represent a **SPARK Class** containing 0 or more **SPARK Operations**; the «SPARK Operation» stereotype providing a definition of the operation’s contract as a set of tag values. Figure 95 shows how the SPARK Contract tag is constructed from a set of UML tag definitions that hold the abstract and concrete “global and derives annotations” for the operation. (Note that we introduce a modelling convention «tag definition» that allows their description as classes rather than attributes of the «stereotype» class. This allows a richer description of their relationships, but has the disadvantage of not showing their fundamental types.)

Abstract annotations are declared in the SPARK Specification (the public part of a SPARK Package) and are therefore visible to users of those operations. Concrete annotations are declared in the SPARK Body (the private part of a SPARK Package) and are hidden from external users of the package. This ability to annotate separately the public and private representations of a public operation allows the designer to hide the internals of the design decomposition from the outside world. (Badly designed SPARK programs can break all notions of information hiding by announcing the hidden parts of an object-based decomposition to the outside world via information flow annotation.)

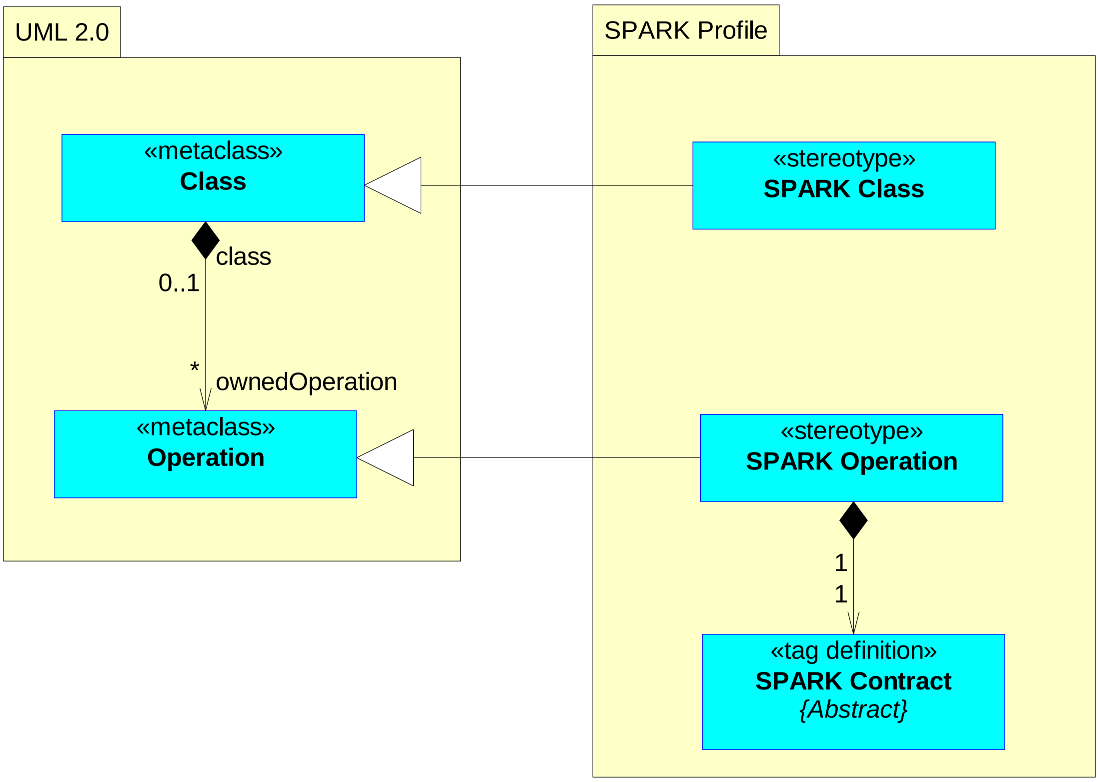


FIGURE 94 SPARK CLASS AND OPERATION

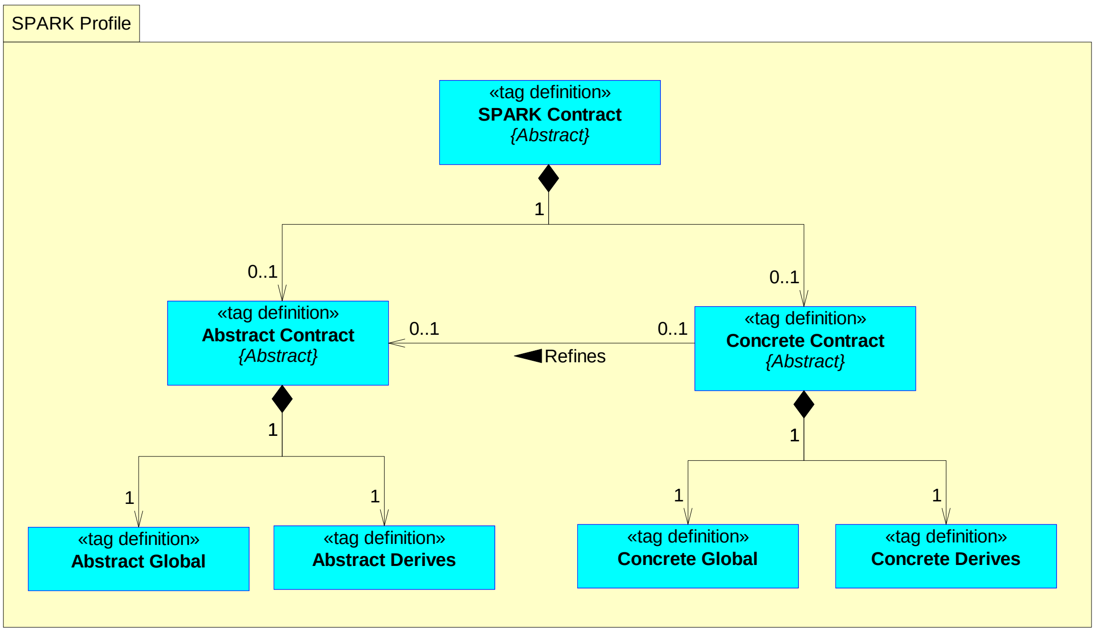


FIGURE 95 INFORMATION FLOW CONTRACT META-MODEL

In addition to «SPARK Class» and «SPARK Operation», the SPARK profile also contains stereotypes to:

- include proof obligations to SPARK Operations («SPARK Proof»).
- construct "Abstract State" («SPARK Abstract State») which hides details of internal state data in the SPARK class.
- identify "Refinement State" («SPARK Refinement State») which shows how abstract state is expanded within a SPARK class.

Figure 96 illustrates the relationship between the SPARK Abstract and Refinement state.

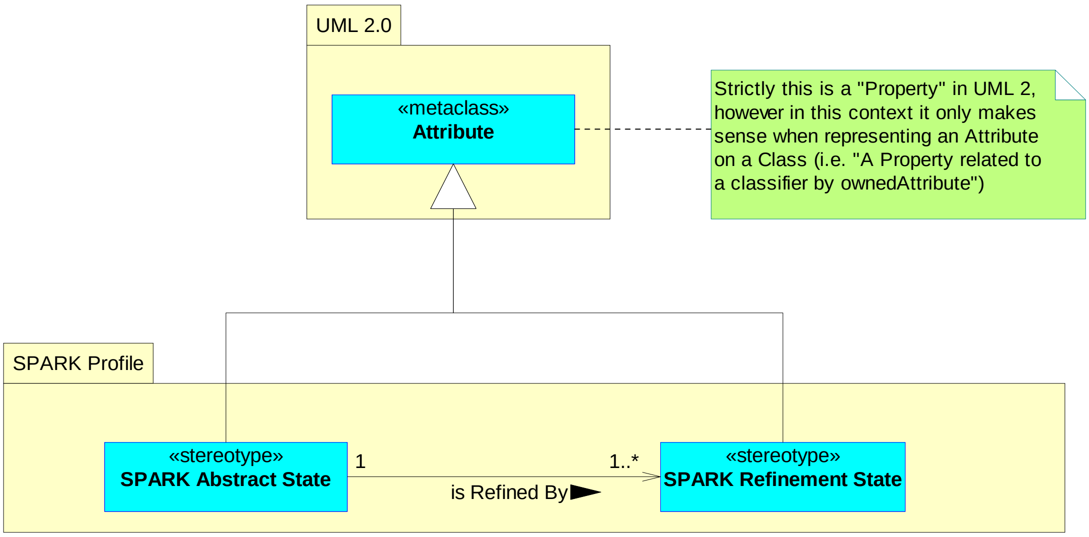

UML 2.0
«metaclass»
Attribute
Strictly this is a "Property" in UML 2, however in this context it only makes sense when representing an Attribute on a Class (i.e. "A Property related to a classifier by ownedAttribute")
SPARK Profile
«stereotype»
SPARK Abstract State
1
1..*
is Refined By
«stereotype»
SPARK Refinement State

FIGURE 96 SPARK ABSTRACT AND REFINEMENT STATE

Other stereotypes are used to control the form of the SPARK code generated at a detailed level.

Using the SPARK UML profile (in conjunction with a more general Ada 95 profile to model the SPARK Ada Kernel) , a UML class model can be used to define the structure of a SPARK program to a level of abstraction that allows the package structure of a SPARK program to be generated automatically.

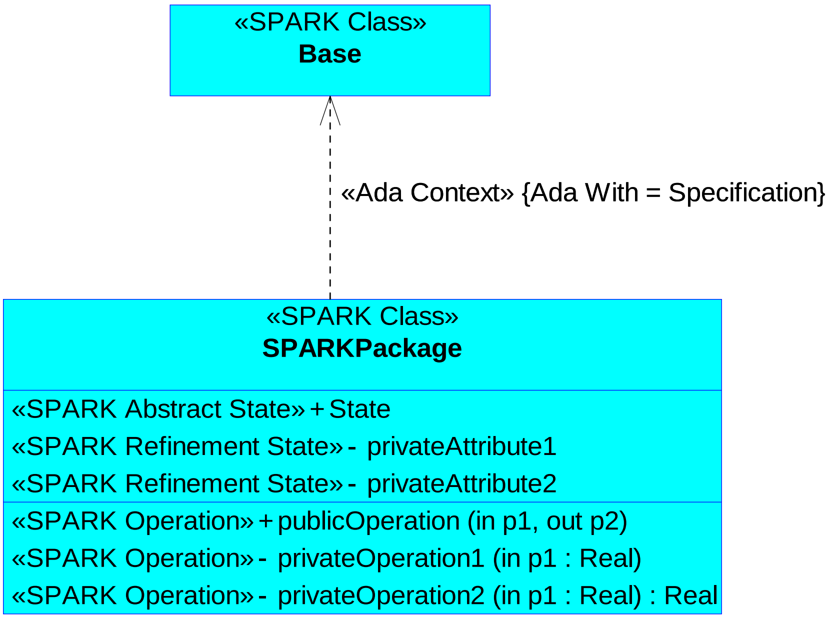
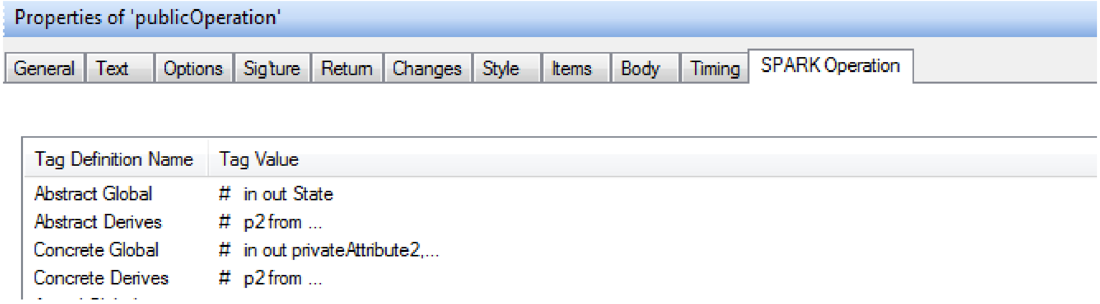

«SPARK Class»
Base
«Ada Context» {Ada With = Specification}
«SPARK Class»
SPARKPackage
«SPARK Abstract State» + State
«SPARK Refinement State» - privateAttribute1
«SPARK Refinement State» - privateAttribute2
«SPARK Operation» + publicOperation (in p1, out p2)
«SPARK Operation» - privateOperation1 (in p1 : Real)
«SPARK Operation» - privateOperation2 (in p1 : Real) : Real

FIGURE 97 SIMPLE EXAMPLE OF A SPARK PACKAGE MODELLED AS A UML CLASS

Figure 97 shows how the SPARK UML profile can be used to model a simple SPARK Class, consisting of a single abstract State definition, two private attributes which refine that state, and three operations, two private and one public. The class itself has a dependency on another SPARK class called Base. This dependency is to be realised in the implementation via an Ada “with” relationship. This is shown as a stereotype on the dependency, with the tag value defining whether the relationship is from the specification or body of the resulting Ada package.

This provides the basic structural information for the SPARK class. However, more information is required to be able to model and generate valid SPARK source code from this. This additional information is contained within the UML tags associated with the stereotypes.

Figure 98 shows a screenshot of the “properties” pane in ARTiSAN Studio for the operation **publicOperation** contained with **SPARKPackage** as shown in Figure 97. Here, it can be seen that the application of the «SPARK Operation» stereotype to the UML operation has resulted in an additional properties tab with the name of the stereotype. The values associated with the stereotype tag definitions are shown in the property pane.

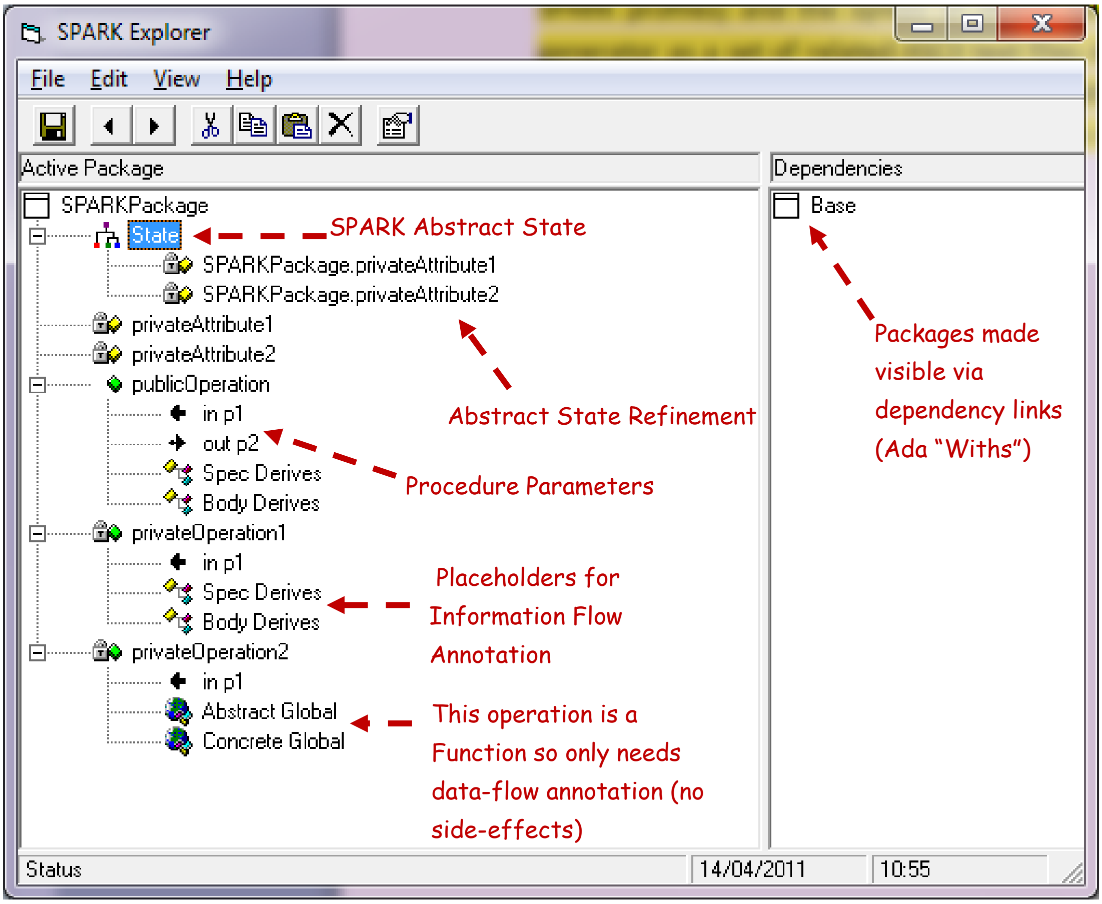


Properties of 'publicOperation'

Tag Definition Name	Tag Value
Abstract Global	# in out State
Abstract Derives	# p2 from ...
Concrete Global	# in out privateAttribute2,...
Concrete Derives	# p2 from ...

FIGURE 98 ARTiSAN STUDIO VIEW OF THE SPARK OPERATION STEREOTYPE TAGS FOR PUBLICOPERATION

To simplify the development and annotation of the modelled SPARK program, the author developed a SPARK editing tool called SPARK Explorer, which visualises the visible program structure and annotation information from ARTiSAN Studio, and allows the user to construct SPARK annotations by drag and drop of state data. The SPARK Explorer view of the **SPARKPackage** example can be seen in Figure 99.



SPARK Abstract State

Abstract State Refinement

Procedure Parameters

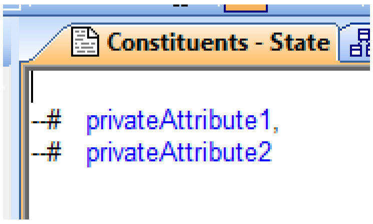
Placeholders for Information Flow Annotation

This operation is a Function so only needs data-flow annotation (no side-effects)

Packages made visible via dependency links (Ada "Withs")

FIGURE 99 SPARKEXPLORER VIEW OF THE INITIAL CONTRACT OF SPARK PACKAGE PLUS STATE REFINEMENT

Figure 99 has a snapshot of the state of the SPARK annotations at a point where only the abstract state refinement has been undertaken. It can be seen that the abstract variable State is refined (realised) by two pieces of concrete state, the private attributes **privateAttribute1** and **privateAttribute2**. This relationship is stored within ARTiSAN Studio as "hyperlinks" within the "Constituents" tag in the «SPARK Abstract State» stereotype applied to the attribute **State**. This can be seen in the screen snapshot shown in Figure 100.

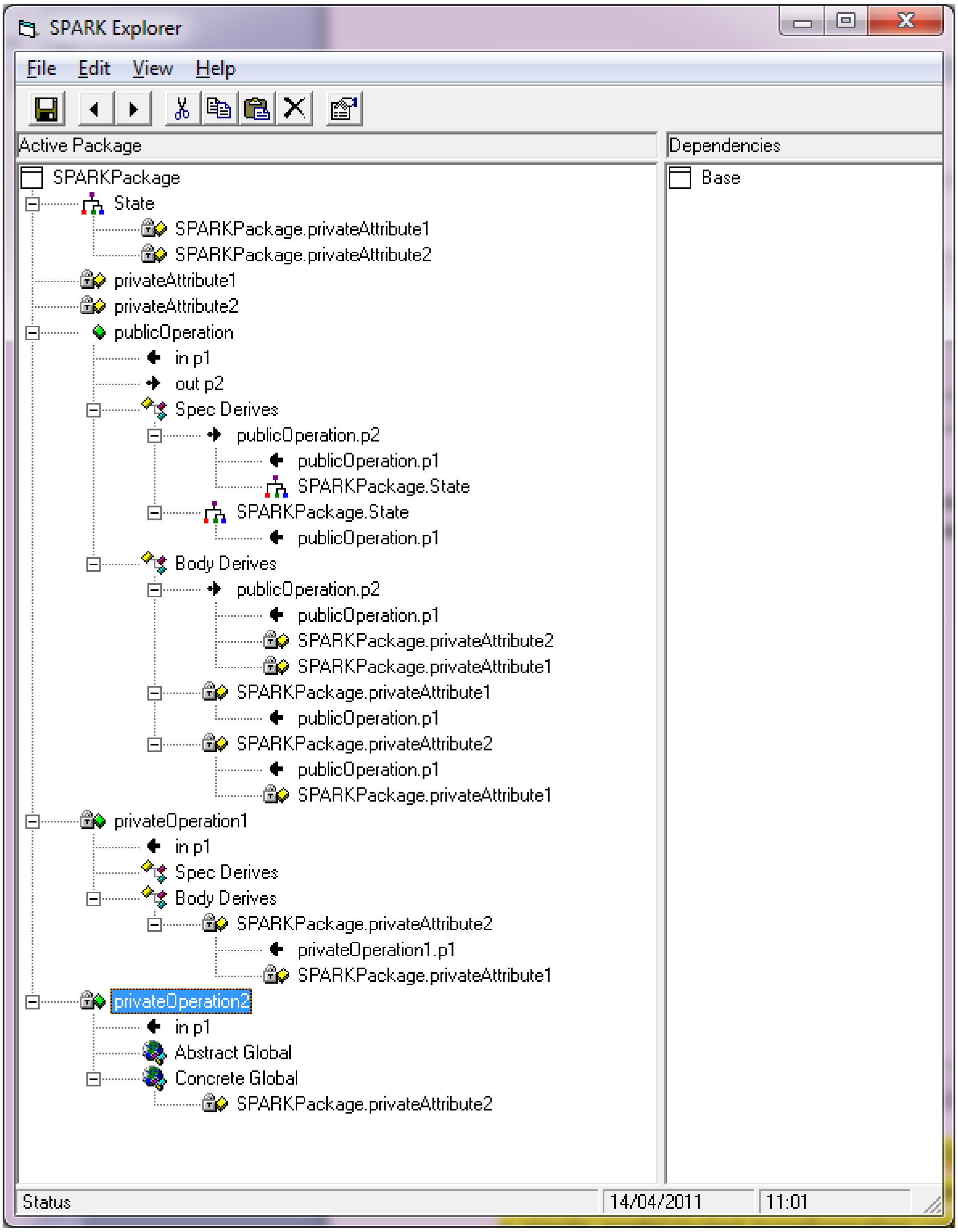


```
--# privateAttribute1,
--# privateAttribute2
```

FIGURE 100 REFINEMENT STATE SHOWN AS HYPERLINKS IN ARTISAN STUDIO

A completed set of SPARK Explorer annotations of **SPARKPackage** can be seen in Figure 101. Here, we can see how the public operation **publicOperation1** declares its visible information flow contract ("Spec Derives") in terms of its parameters and the abstract state, and refines this in a more detailed private contract ("Body Derives") in terms of the parameters and the refinement state variables. In this way, the details of the package internals need not be propagated to the users of the public operations.

Note that private procedures only need to have the private contract as these have no public declarations in the package specification. Also, note that functions only need dataflow annotations to identify the state data the function uses. SPARK does not allow functions to have side effects, and therefore the form of the information flow for a function is always that the return value is based upon the function parameters and any declared state data.



```
SPARKPackage
  State
    SPARKPackage.privateAttribute1
    SPARKPackage.privateAttribute2
  privateAttribute1
  privateAttribute2
  publicOperation
    in p1
    out p2
    Spec Derives
      publicOperation.p2
        publicOperation.p1
        SPARKPackage.State
      SPARKPackage.State
        publicOperation.p1
    Body Derives
      publicOperation.p2
        publicOperation.p1
        SPARKPackage.privateAttribute2
        SPARKPackage.privateAttribute1
      SPARKPackage.privateAttribute1
        publicOperation.p1
      SPARKPackage.privateAttribute2
        publicOperation.p1
        SPARKPackage.privateAttribute1
  privateOperation1
    in p1
    Spec Derives
    Body Derives
      SPARKPackage.privateAttribute2
        privateOperation1.p1
        SPARKPackage.privateAttribute1
  privateOperation2
    in p1
    Abstract Global
    Concrete Global
      SPARKPackage.privateAttribute2
```

```
Base
```

```
14/04/2011    11:01
```

FIGURE 101 FULLY ANNOTATED SPARKPACKAGE AS VIEWED IN SPARKEXPLORER

Now that we have defined the structure of the SPARK packages in a class model (shown in Figure 97) and provided the information flow contracts for the identified operations, we are in a position to generate SPARK-compliant code from the model. The code shown below is the output from applying the model-to-text SPARK transformation on the model described above.

```
with Base;

--# Inherit
--#   Base
--# ;

package SPARKPackage
--# own State;
 is


   --------------------------------------------------------------------
   -- public operation declarations
   --------------------------------------------------------------------

   --------------------------------------------------------------------
   -- publicOperation
   --
   -- Description:
   --------------------------------------------------------------------
   procedure publicOperation(
         p1 : in Base.Real;
         p2 : out Base.Real);
   --# Global
   --#   in out State
   --# ;
   --# Derives
   --#   p2 from
   --#          p1,
   --#          State &
   --#   State from
   --#          p1
   --# ;



private

end SPARKPackage;
```

```
package body SPARKPackage
--# own State is
--#   privateAttribute1,
--#   privateAttribute2
--#  ;
is

   --------------------------------------------------------------------
   -- Private attributes
   --------------------------------------------------------------------

   privateAttribute1 : Base.Real;

   privateAttribute2 : Base.Real;

   --------------------------------------------------------------------
   -- Private operations
   --------------------------------------------------------------------

   --------------------------------------------------------------------
   -- privateOperation1
   --
   -- Description:
   --
   -- Implementation Notes:
```

```
--------------------------------------------------------------------
procedure privateOperation1(
      p1 : in Base.Real)
--# Global
--#   in     privateAttribute1;
--#   out    privateAttribute2
--# ;
--# Derives
--#   privateAttribute2 from
--#          p1,
--#          privateAttribute1
--# ;
is
begin
   null;
end privateOperation1;

--------------------------------------------------------------------
-- privateOperation2
--
-- Description:
--
-- Implementation Notes:

--------------------------------------------------------------------
function privateOperation2(
      p1 : in Base.Real) return Base.Real
--# Global
--#   in     privateAttribute2
--# ;
is
begin
   null;
end privateOperation2;


--------------------------------------------------------------------
-- Public operations
--------------------------------------------------------------------

--------------------------------------------------------------------
-- publicOperation
--
-- Implementation Notes:

--------------------------------------------------------------------
procedure publicOperation(
      p1 : in Base.Real;
      p2 : out Base.Real)
--# Global
--#   in out privateAttribute2,
--#          privateAttribute1
--# ;
--# Derives
--#   p2 from
--#          p1,
--#          privateAttribute2,
--#          privateAttribute1 &
--#   privateAttribute1 from
--#          p1  &
--#   privateAttribute2 from
--#          p1,
--#          privateAttribute1
--# ;
is
begin
   null;
   end publicOperation;

end SPARKPackage;
```

The code generator used to produce this code was developed using the ARTiSAN Studio OCS (On-demand Code Synchronisation) template-driven code generation technology. OCS provides a simple model-to-text transformation using an interpreted template language called SDL. A set of SDL templates define the transformation rules between UML Class models (extended via Ada 95 and SPARK profiles) and the syntax of SPARK source code. The templates are presented to the code generator as a set of related ASCII text files defining the required transformation functions. Code is generated from a particular node in the UML class model tree, the code generator using the OCS templates to guide the transformation of the model fragment to SPARK source. OCS is a simple, interpreted model-to-text transformation engine that suited the development of code from product models, but it lacks the sophistication required for product-line developments.

Appendix B - Instantiating Products using Model Transformation

The technology chosen to develop the transformation for product line instantiation was the ACS/TDK (Automatic Code Synchronisation/Template Development Kit) "4G" technology from Atego (formerly ARTiSAN). The ACS/TDK toolset provides the model-to-text code generation and round-trip model and code development extensions to the ARTiSAN Studio UML environment. The "4G" version of ACS/TDK augmented this with the ability to perform Model-to-Model transformation.

The decision to use ACS/TDK 4G (hereafter known as TDK) was primarily driven by the need to develop an instantiation process that could be used on a large, multi-developer avionics project. ARTiSAN Studio was the incumbent modelling tool; there was a substantial investment in tool licenses, existing product models and user knowledge.

Previous projects used a UML to SPARK code generator that was implemented using OCS (On-Demand Code Synchronisation). OCS is a simple template-based Model-To-Text code generation engine. OCS scripts are developed in a language called SDL and are interpreted by the Studio environment on-demand. As described in Appendix A, the customised OCS SPARK generator makes use of Ada and SPARK profiles that extend the UML class models to capture Ada and SPARK-specific concepts. This approach was used effectively on two large avionics projects (approximately 250K SLOC each).

However, OCS was not suitable for development of the product line transformation and code generation for a number of reasons. Firstly, the OCS product had been deprecated by Atego and replaced by the ACS generator engine. Secondly, OCS had no model-to–model transformation capabilities. However, legacy OCS generators can be ported to/hosted within ACS-based generation schemes. This capability meant that it was easy to create the back end model-to-text transformation from the OCS baseline and this had a degree of provenance from previous project use. The effort could therefore be spent on developing the product line transformation rather than replicating a pre-existing code generator.

In contrast to the interpreted-SDL approach of the OCS generator, ACS generators are compiled to Win32 DLLs and executed either on demand or as part of a continuous generation approach. ACS generators can run in the background during a modelling session and continuously generate code in response to changes in the source model. Round-tripping is also supported where model elements can be created in repose to external changes to the source code. However, in the context of high-integrity software development the generator is used exclusively in forward–engineering mode.

A specific ACS generator DLL is produced by designing a generator model using the Studio UML tool (augmented with the TDK development kit). A special version of ACS is then used on the generator model to auto-generate the specific generator code and DLL.

Describing and Developing Model-To-Model Transformations in TDK

M2M transformations in TDK are described using a decorated form of class model. This model is a declarative statement of the rules used to transform from the source to target meta-models. As TDK is designed primarily to produce code generators, the transformed model is typically transitory in memory; there is the facility to write the transformed model elements back to the source model repository - however, this would be destructive of the source model data. If the transformed model needed to be stored, it would be relatively straightforward to provide a M2T back-end that serialised the model from memory to XMI form, for example. For the purposes of the transformations described in this thesis, the transitory model is perfectly acceptable as it is used purely to facilitate the instantiation and generation of product-specific assets from a product lines model – the product-specific model is never accessed interactively by a user.

To create a transformation and generator model, a special "TDK profile" is included with the generator UML model. The TDK profile augments the UML meta-model as shown partially in Figure 102 and Figure 103.

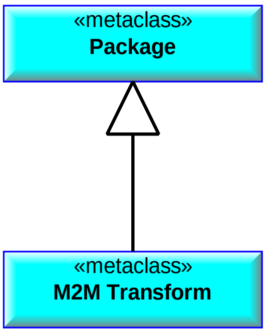


FIGURE 102 TDK M2M TRANSFORM EXTENSION

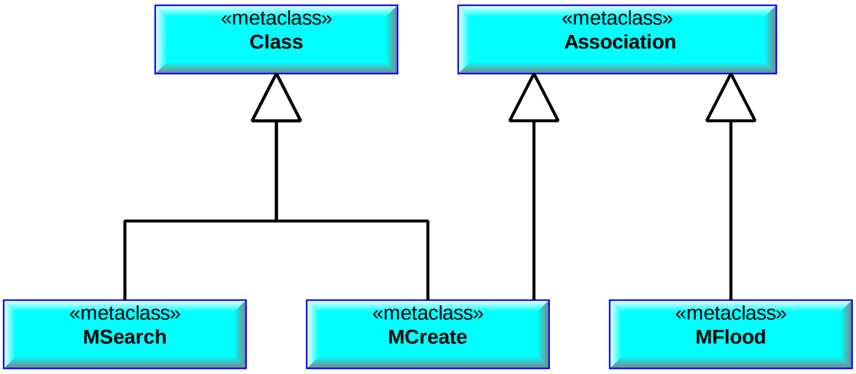


FIGURE 103 TDK M2M CLASS AND ASSOCIATION EXTENSIONS

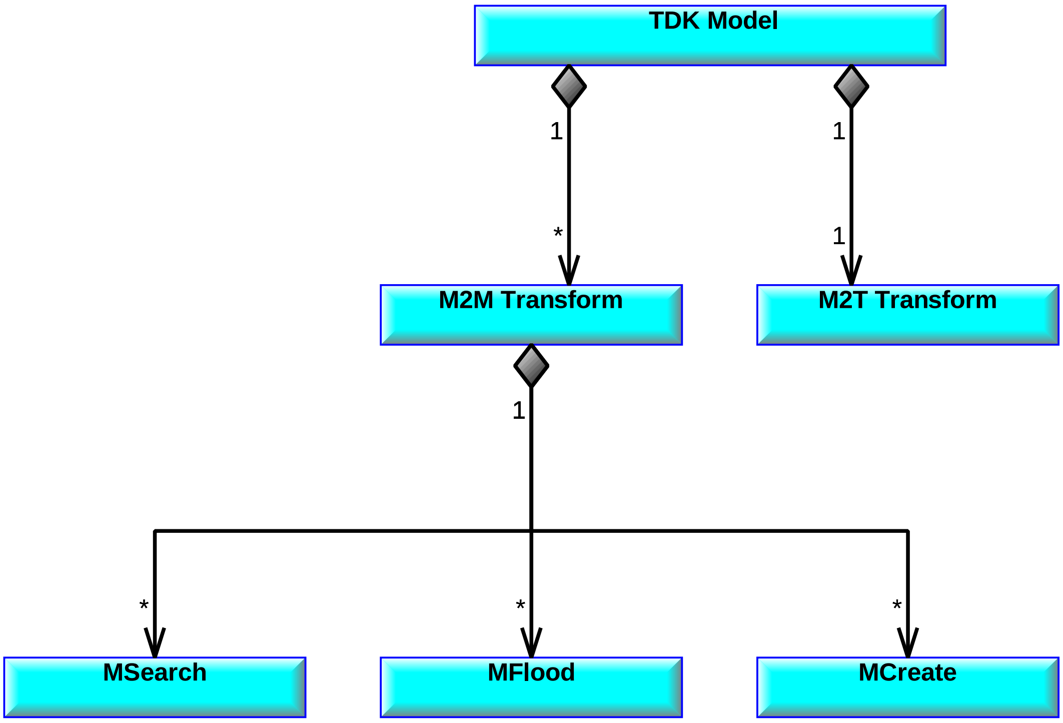


Figure 104 TDK Model Structure

As shown in Figure 104, a TDK generator model typically consists of a single Model-To-Text (M2T) transformation, and optionally a number of Model-To-Model (M2M) transformations. An M2M transform is a stereotyped UML package (Figure 102), which contains a class model representing the transformation rules. If multiple M2M transforms are specified within a TDK model, an order of application can be defined to ensure the cumulative effects of the transformations is predictable.

The rules within a single M2M Transformation package are described in class model form. This model describes a set of search-and-create operations that identify source meta-model elements (via MSearch classes) and, in response, produce target meta-model elements (via MCreate classes). In its simplest form, this could simply find meta-model elements in the source model and duplicate them into the target model. However, much more useful and sophisticated M2M transformations can be realised using this approach. Consider a requirement to add a public accessor (read) operation for each private attribute owned by a class representing an SPARK package (e.g. stereotyped by «SPARK Class»). Figure 105 shows a TDK model describing the transform that attempts to realise this requirement.

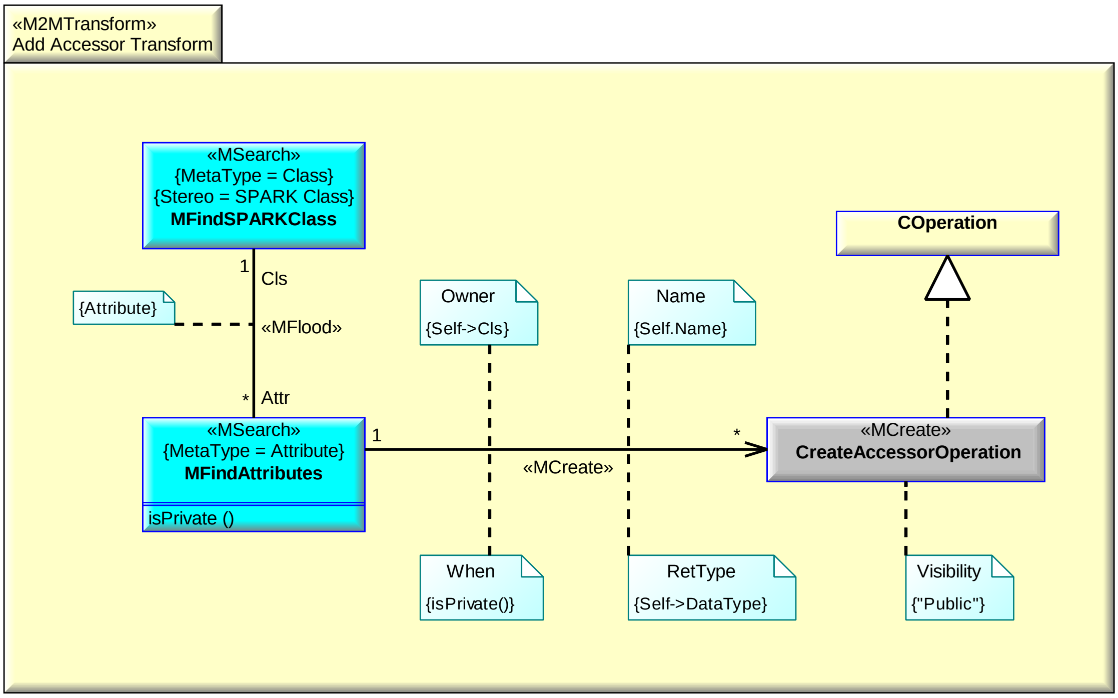


FIGURE 105 TDK M2M TRANSFORMATION TO ADD ACCESSOR OPERATIONS

This transformation structure is typical of TDK M2M models. An initial «MSearch» class **MFindSPARKClasss** collects model elements of a specific meta-type (identified by the "MetaType=" tag), which, in this case, is the set of classes in the model. The set is further reduced by a stereotype filter ("Stereo=" tag) which reduces the search set to those model classes that are identified as "SPARK Class". The **MFindSPARKClass** class is associated with a second «MSearch» class **MFindAttributes** via an «MFlood» association. The «MFlood» propagates a filtered search set from one search class to another and, importantly, maintains a named model association between the elements of the search. In this example, the constraint on the «MFlood» association of "Attribute" defines that all attributes contained within the classes of the **MFindSPARKClass** search set propagate to the **MFindAttributes** set, and the framework maintains a navigable **Cls<->Attr** relationship between them.

We now have the set of attributes owned by SPARK classes within the **MFindAttributes** search class. We can now use the «MCreate» TDK elements to create accessor operations for them within the transformed model. «MCreate» associations link search classes to «MCreate» classes and, like flood associations, they propagate search sets from one class to another. However, «MCreate» elements, as the name suggests, create new model elements in the target model in response to each element in the search set. The model element created does not need to be of the same type as the element in the search set; the element type created is dependent upon the specialisation of the «MCreate» class. As can be seen in Figure 105, the «MCreate» class **CreateAccessorOperation** is a specialised **COperation** class. The TDK framework provide a set of "factory" classes for each creatable UML meta-model element type, from which «MCreate» classes can be derived. The relationship of the newly created model element with the rest of the model is defined by the constraints on the «MCreate» association. The Owner constraint defines which model element should own the newly created operation; here it is defined as **Self->Cls**.

Self is TDK keyword referring to the current object in the related search class, -> traverses an association, and **Cls** is the class object at the end of the «MFlood» association. In simple terms, the created operation is to be owned by the class that owns the associated attribute. The Name and RetType constraints on the «MCreate» association define the name of the created operation and return type respectively.

The requirements for this transformation asked for an accessor to be created for the **private** operations only. There are a number of ways in which this down-selection could be achieved. Operations can be added to search classes that allow procedural SDL code to perform further processing of the search set. An operation could be added to **MFindSPARKClass** that returns all the private attributes of an object, then this operation used as the constraint on the «MFlood» association. Alternatively, an operation could be added to the **MFindAttributes** class that returns true if the current attribute object is private. This is the approach we take in this particular transformation; the isPrivate() operation is used within a **when** constraint on the «MCreate» association. **When** constraints provide a guard on element creation.

Another constraint on the «MCreate» association (RetType) ensures that the return type of the created operation is set to the type of the attribute being accessed. Constraints on the **CreateAccessorOperation** class can be used to set properties of the created operation, as can be seen by the "Visibility" constraint.

Finally, whilst this transformation as designed ensures the correct number and type of accessor operations will be created with Public visibility, they will not be functional, as no implementation body has been provided. This again is a property of the created operation, and can be set by adding a "Body" constraint on the **CreateAccessorOperation** class.

This section introduced the development of model-to-model transformations using TDK. We use these techniques extensively to realise the product-line instantiation transformation, described in detail in the following section.

Realising Model Transformation for High-Integrity Product Lines

The overall model transformation process used to instantiate products from the product line is illustrated in Figure 106. This process was summarised in Chapter 5 of this thesis; we provide more detail on the design of the transformation here. Some of the text and diagrams from Chapter 5 are repeated here to make this appendix understandable stand-alone and avoid the need for the reader to continually cross-reference the information.

Once the reference architecture and product line components have been developed, product instances can be created. Instantiation of products is achieved by the deployment of the appropriate components in a copy of the reference architecture model and the selection of the appropriate decision options for each component (either directly, or as the result of a higher-level feature model selection). Once the components are deployed and the decision options are resolved, then product-specific assets can be generated using model transformation.

Model-to-Model Transformation 1 – Reductive Product Line to Product Model Transform

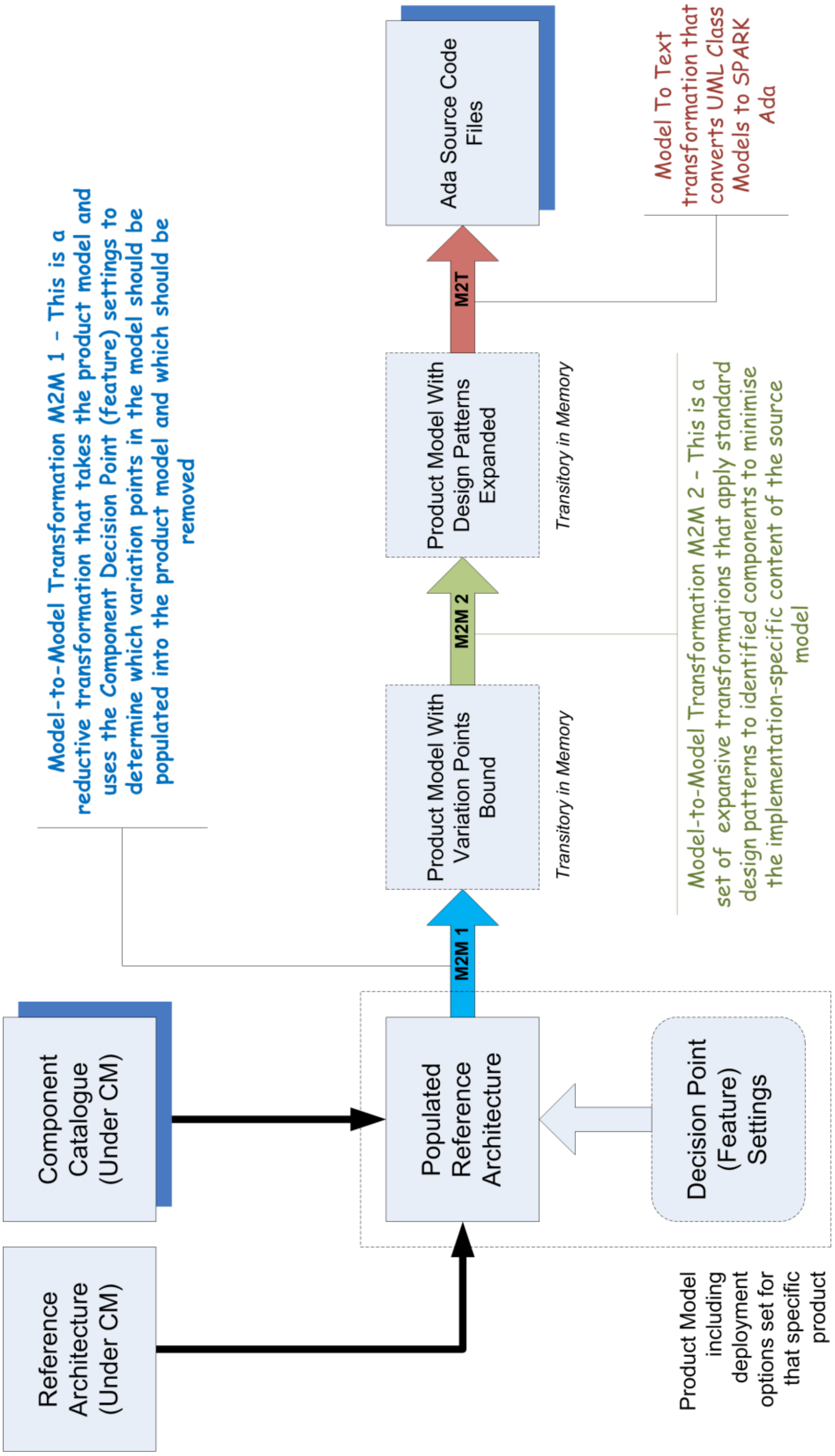
We described earlier how the TDK 4G model transformation describes a transform as a declarative class model. Here we describe the form of the class model that describes the product line to product instance reductive transformation

Figure 107 shows the complete transform class model, however we will be describing fragments of this model in a more readable form throughout this section. The instantiation transformation essentially performs the following algorithm:

```
For each component included in the deployment model:
      Follow the bind link to the catalogue component;
      For each model element in the catalogue component:
         If it is a variation point then
            If selection expression evaluates True then
              duplicate into deployment model;
              end if;
        Else
           duplicate into deployment model;
        end if;
    end for;
end for;
```

The result of this transformation is a complete product specific model under the deployment model “root” which can be passed to the downstream transformations.

The transformation model is built up from a network of associated «MSearch» classes to isolate the meta-model elements that may exhibit variability. Once these elements are isolated, the selection expressions that guard the inclusion of that element are evaluated for the particular decision options selected for the particular product. Successful evaluation of the expression triggers the duplication of that element into the product line model. Common meta-model elements (i.e. those not stereotyped as variation points) are always duplicated into the target model.


Reference Architecture (Under CM)
Component Catalogue (Under CM)
Populated Reference Architecture
Decision Point (Feature) Settings
M2M 1
Product Model With Variation Points Bound
Transitory in Memory
M2M 2
Product Model With Design Patterns Expanded
Transitory in Memory
M2T
Ada Source Code Files

Product Model including deployment options set for that specific product

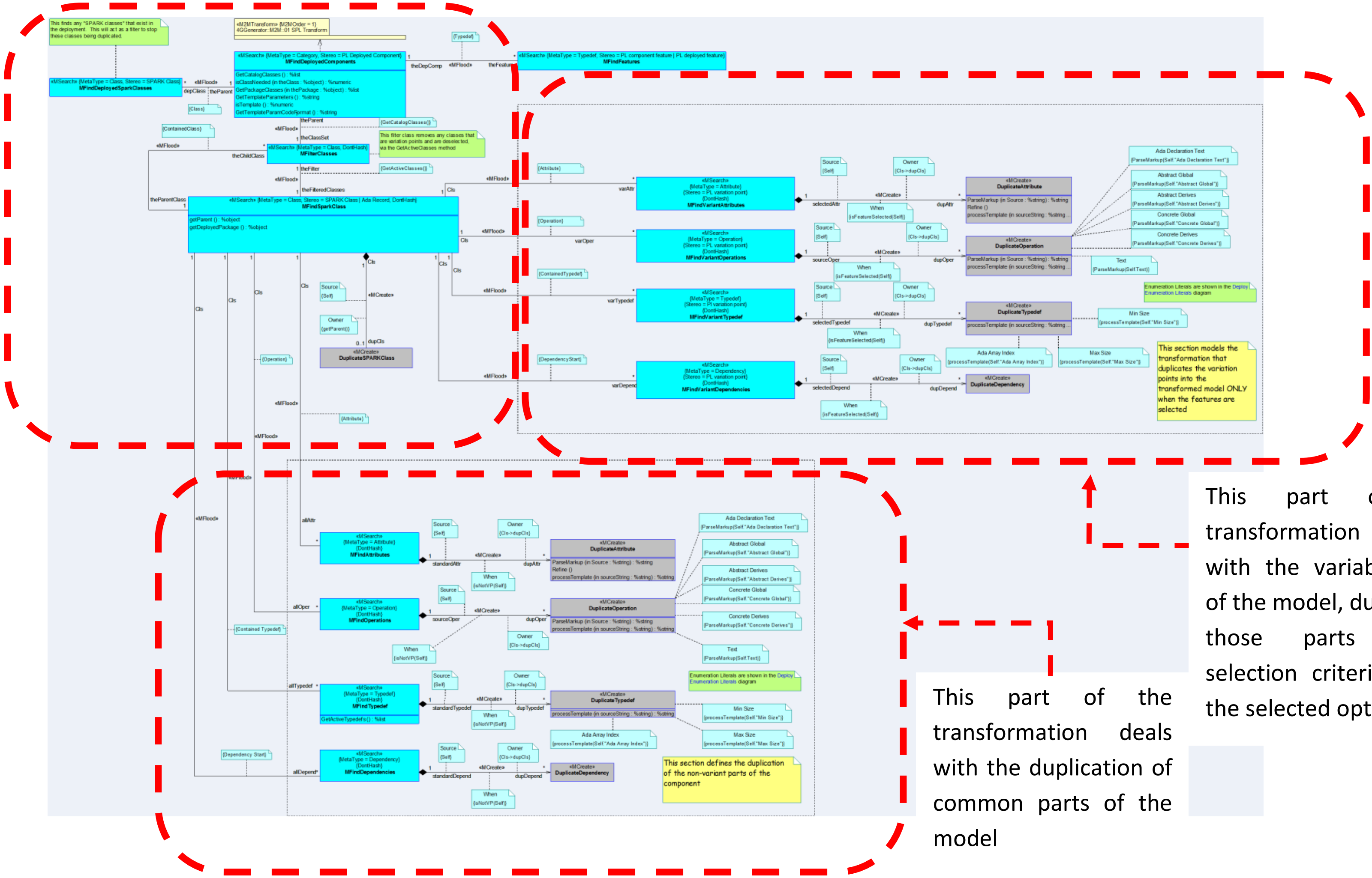
Model-to-Model Transformation M2M 1 – This is a reductive transformation that takes the product model and uses the Component Decision Point (feature) settings to determine which variation points in the model should be populated into the product model and which should be removed

Model-to-Model Transformation M2M 2 – This is a set of expansive transformations that apply standard design patterns to identified components to minimise the implementation-specific content of the source model

Model To Text transformation that converts UML Class Models to SPARK Ada

Figure 106 Product Instance SPARK Code Generation from Reference Architecture and Product Line Components

This part of the transformation selects the classes in the model to which the transformation is applied



This part of the transformation deals with the variable parts of the model, duplicating those parts whose selection criteria meets the selected options

This part of the transformation deals with the duplication of common parts of the model

FIGURE 107 STRUCTURE OF MODEL-TO-MODEL TRANSFORMATION 1 CLASS MODEL

To understand the transformation performed we have to refer back to the decision contract meta- model we introduced in chapter 5 (shown again here in Figure 108).

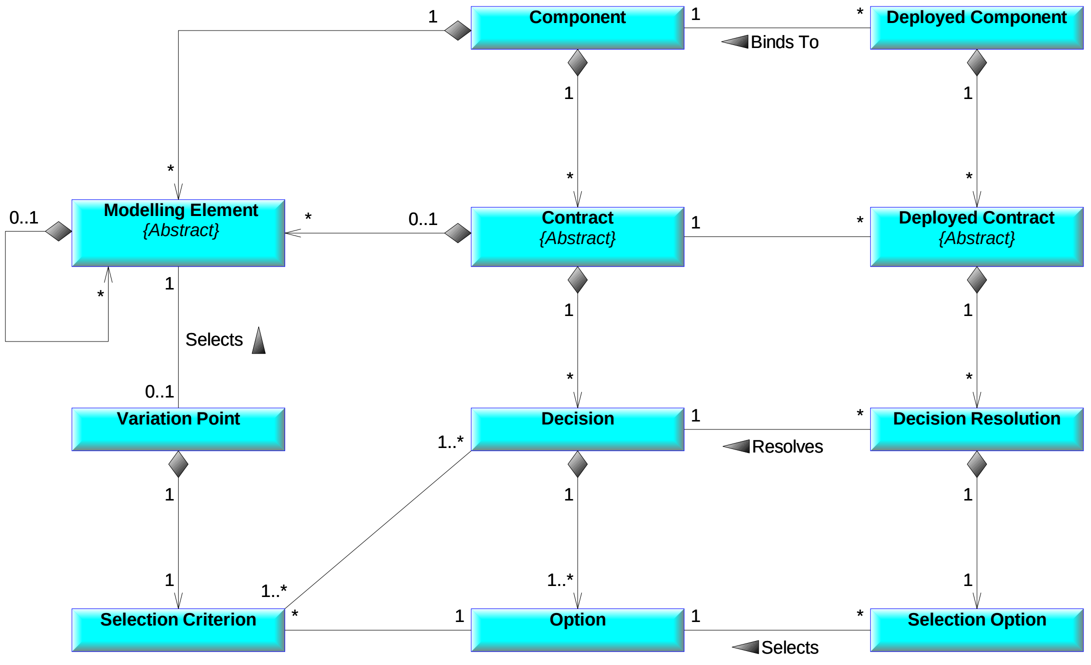
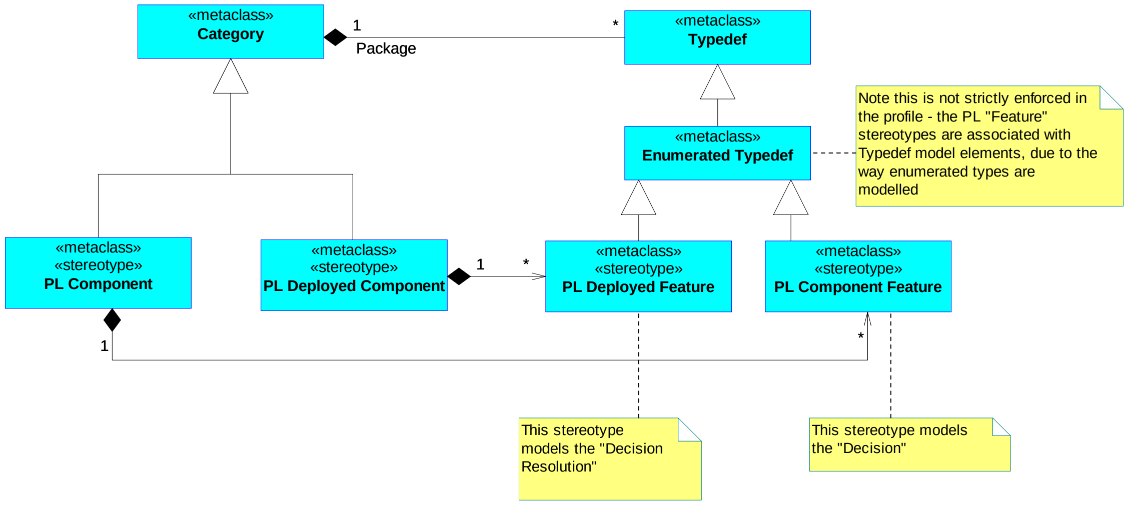


FIGURE 108 PRODUCT LINE META-MODEL USING DECISION CONTRACTS (FROM CHAPTER 5)

The UML is extended via a special product line UML profile to realise this meta-model. Figure 109 shows a model of this profile.


«metaclass» Category
Package
«metaclass» Typedef
«metaclass» Enumerated Typedef
«metaclass» «stereotype» PL Component
«metaclass» «stereotype» PL Deployed Component
«metaclass» «stereotype» PL Deployed Feature
«metaclass» «stereotype» PL Component Feature
1
*

Note this is not strictly enforced in the profile - the PL "Feature" stereotypes are associated with Typedef model elements, due to the way enumerated types are modelled

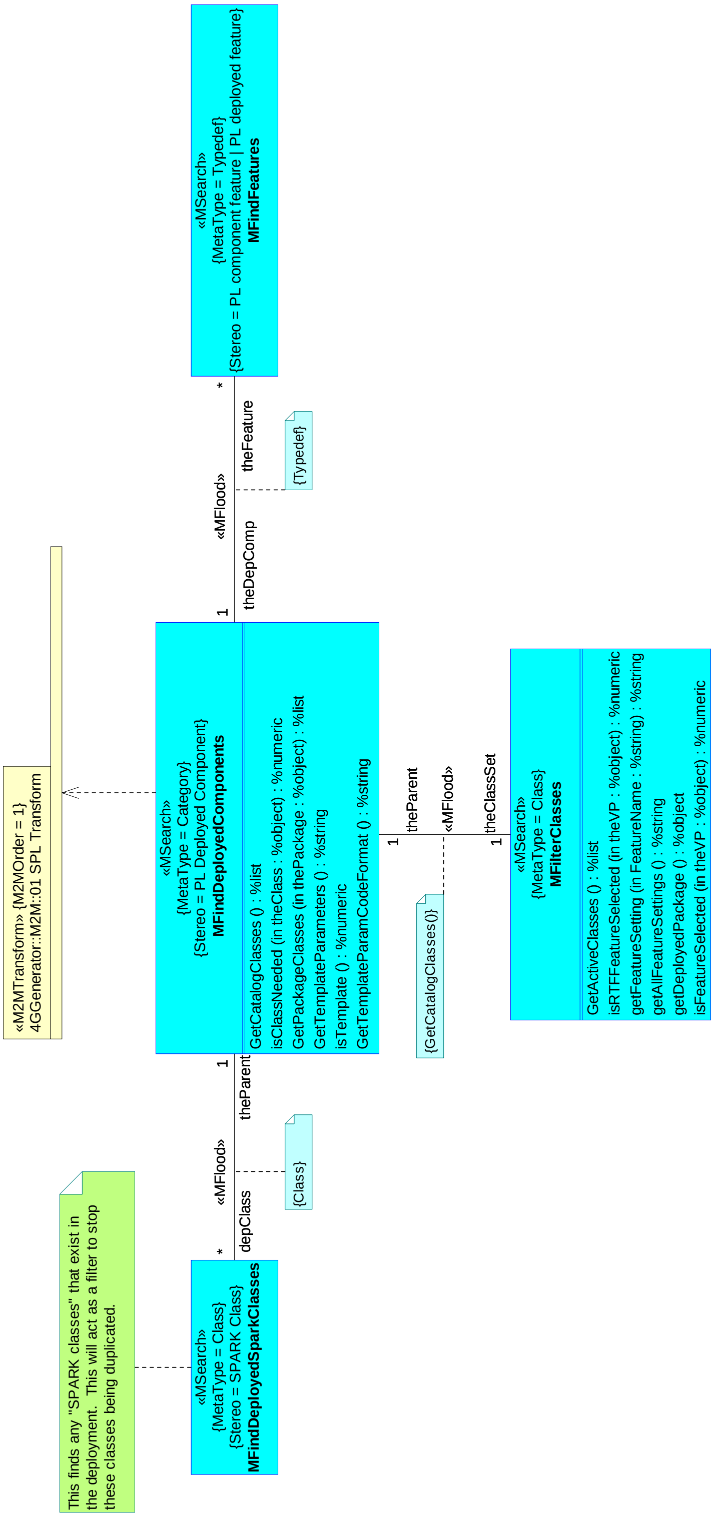
This stereotype models the "Decision Resolution"

This stereotype models the "Decision"

FIGURE 109 PL PROFILE MAPPING TO UML META-MODEL ELEMENTS

We now describe each of the significant classes in the transformation model in detail including their associations

Collecting Deployed Components


«MSearch»
{MetaType = Typedef}
{Stereo = PL component feature | PL deployed feature}
MFindFeatures
*
theFeature
«MFlood»
{Typedef}
theDepComp
1
«M2MTransform» {M2MOrder = 1}
4GGenerator::M2M::01 SPL Transform
«MSearch»
{MetaType = Category}
{Stereo = PL Deployed Component}
MFindDeployedComponents
GetCatalogClasses () : %list
isClassNeeded (in theClass : %object) : %numeric
GetPackageClasses (in thePackage : %object) : %list
GetTemplateParameters () : %string
isTemplate () : %numeric
GetTemplateParamCodeFormat () : %string
1
theParent
«MFlood»
theClassSet
1
{GetCatalogClasses()}
«MSearch»
{MetaType = Class}
MFilterClasses
GetActiveClasses () : %list
isRTFFeatureSelected (in theVP : %object) : %numeric
getFeatureSetting (in FeatureName : %string) : %string
getAllFeatureSettings () : %string
getDeployedPackage () : %object
isFeatureSelected (in theVP : %object) : %numeric
1
theParent
«MFlood»
{Class}
depClass
*
«MSearch»
{MetaType = Class}
{Stereo = SPARK Class}
MFindDeployedSparkClasses

This finds any "SPARK classes" that exist in the deployment. This will act as a filter to stop these classes being duplicated.

FIGURE 110 MFINDDEPLOYEDCOMPONENTS CLASS DIAGRAM

MFindDeployedComponents is the initial class in the transformation network. It is associated with the root of the transformation, and its purpose is to connect all instances of deployed components in the model to be transformed. Deployed components are modelled as UML Packages (Categories) stereotyped as «PL Deployed Component».

A number of the elements owned by the deployed components found by **MFindDeployedComponents** are routed (“flooded”) to supporting «MSearch» classes:

- Typedefs are flooded to **MFindFeatures** to collect the decision resolution for the deployed component
- Classes are flooded to **MFindDeployedSparkClasses** to identify any pre-deployed classes that override the catalogue component.

Note on the class diagram (Figure 110) the GetCatalogClasses() operation is used as the flood constraint into **MFilterClasses** - operations that return object lists can be used in this manner. The significant operations of the **MFindDeployedComponents** class are:

- **GetCatalogClasses() : %list**

GetCatalogClasses() traverses the «bind» link between the deployed and catalogue (Product Line) components (as shown in the meta-model in Figure 59) and returns a list of the classes contained by the catalogue component that require processing. It makes use of the isClassNeeded() operation to determine if the class has already been deployed and removes these from the returned list.

- **isClassNeeded(in theClass : %object) : %numeric**

isClassNeeded() attempts to locate the Class parameter within the set of classes in the named association depClass (see class diagram Figure 110). If not found then theClass is needed and the function returns 1. If found in the set the function returns 0.

MFindDeployedSparkClasses collects any flooded class that is stereotyped as «SPARK Class». This is used to collect classes that already exist in the deployed model and therefore do not need duplication as part of the transformation.

MFindFeatures collects any flooded typedef that is stereotyped as «PL Component Feature» or «PL Deployed Feature». This is used to collect the decision settings in the deployed component that used to guide the downstream reductive transformation.

Identification and Duplication of Classes

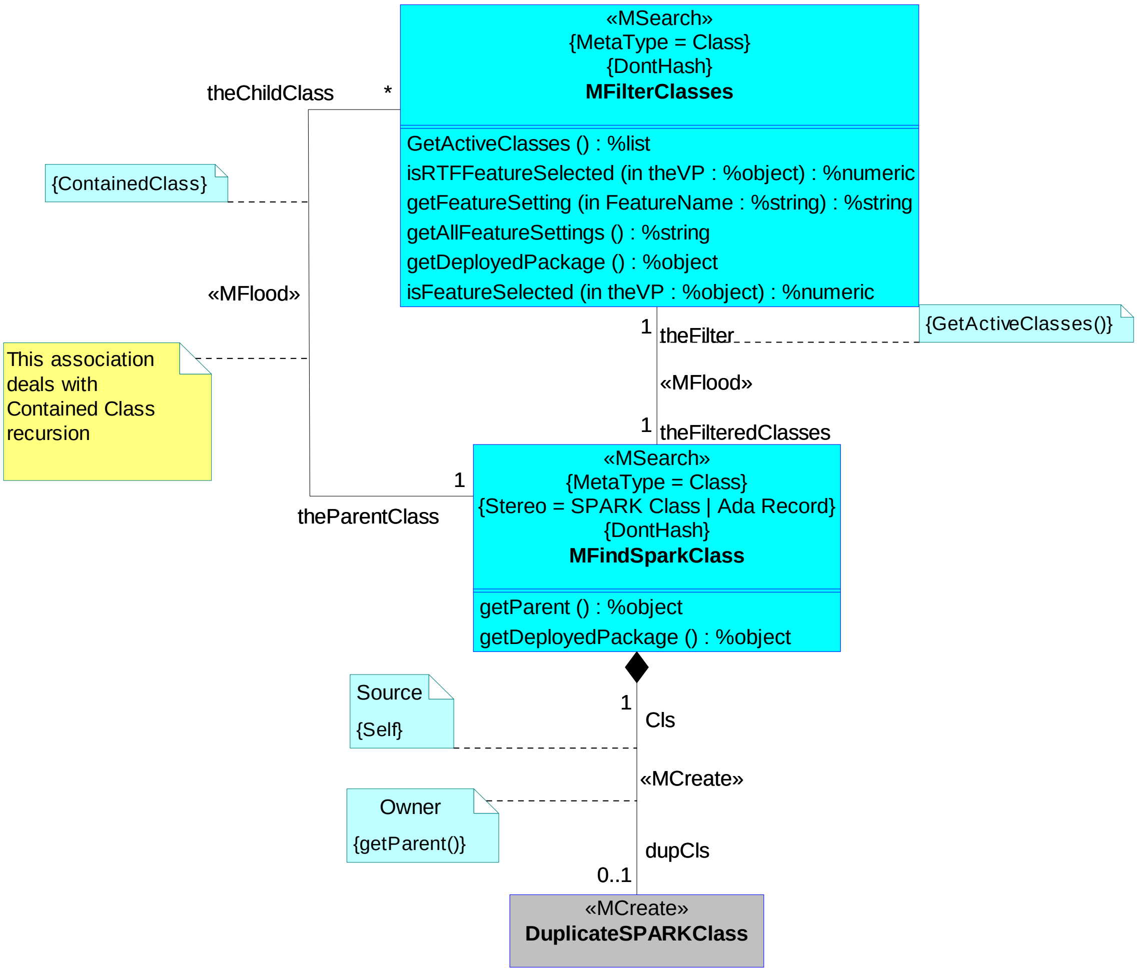

«MSearch»
{MetaType = Class}
{DontHash}
MFilterClasses
GetActiveClasses () : %list
isRTFFeatureSelected (in theVP : %object) : %numeric
getFeatureSetting (in FeatureName : %string) : %string
getAllFeatureSettings () : %string
getDeployedPackage () : %object
isFeatureSelected (in theVP : %object) : %numeric
theChildClass
*
{ContainedClass}
«MFlood»
This association deals with Contained Class recursion
1
theFilter
{GetActiveClasses()}
«MFlood»
1
theFilteredClasses
1
theParentClass
«MSearch»
{MetaType = Class}
{Stereo = SPARK Class | Ada Record}
{DontHash}
MFindSparkClass
getParent () : %object
getDeployedPackage () : %object
Source
{Self}
1
Cls
«MCreate»
Owner
{getParent()}
dupCls
0..1
«MCreate»
DuplicateSPARKClass

FIGURE 111 MFILTERCLASSES CLASS DIAGRAM

MFilterClasses forms the point in the transformation network where the product instantiation decisions start to be made. It forms a gateway that allows through classes that are part of the common product model, or are variation points that have been selected in this particular product instance.

The GetActiveClasses() operation is used to construct the list of common or selected classes and this floods through to **MFindSparkClass.**

The significant operations are described below:

- **GetActiveClasses() : %list**

GetActiveClasses() filters the incoming class list, and passes on to the return list any class that is a common element (i.e. is not stereotyped «PL Variation Point»)

For any class that IS stereotyped «PL Variation Point», the operation isFeatureSelected() is called. If this returns True, then the class is also inserted into the return list. Classes that fail the isFeatureSelected() test are discarded.

- **getFeatureSetting (in FeatureName : %string) : %string**

This function returns the selected option for a given decision in the deployed component. It calls getDeployedPackaget() to return the component being processed and then finds the given decision (FeatureName) in the deployed component contract and returns the selected value.

- **getAllFeatureSettings () : %string**

This operation returns a composite string containing the set of feature settings for the deployed component. The returned string is of the form:

```
FeatureName1:FeatureValue1;FeatureName2:FeatureValue2; ...
```

- **getDeployedPackage () : %object**

This function traverses the model and returns the parent package (category) of the topmost class in the class hierarchy. This represents the deployed component. (Note function takes into account contained class hierarchies)

- **isFeatureSelected(in theVP : %object) : %numeric**

This function determines if a variation point has been selected in the current deployment. Specifically it returns 1 (true) if the object passed has a "PL Select When" expression which returns True when evaluated with the current Deployed component settings.

MFindSPARKClass collects the set of modelled classes that are to be replicated into the deployed model. The flooding operation has performed the down-selection based upon the product decision settings; the set of classes collected in **MFindSPARKClass** are the result of that selection. Note the *theParentClass-theChildClass* association between **MFindSPARKClass** and **MFilterClasses**. This ensure any child classes contained by a class (representing Ada public or private children, or Ada records) are processed by the class decision filter and are replicated as required.

The significant operations are described below:

- **getParent() : %object**

getParent() returns the parent object in the **transformed** model which will own the replicated class.

- **getDeployedPackage() : %object**

getDeployedPackage() returns the UML Package (category) which represents the deployed component. This is the parent in the transformed model for the replicated component classes. It also contains the decision settings for the product instantiation of this component.

DuplicateSPARKClass is a «MCreate» class which instantiates the DClass factory class. The DClass factory class duplicates UML classes into the target model. DClass requires a Source class (to be duplicated) and an Owner object (to own the duplicated class). The **DuplicateSPARKClass** instance is fed by the set of classes collected by **MFindSPARKClass** (as Source) and the Owner object is the result of the getParent() operation on each of the collected **MFindSPARKClass** classes.

We now have defined a transformation that will reduce a product-line class model and replicate the selected classes into a target deployment model based upon the selected decision options. The rest of the transformation concerns the duplication of the other relevant elements of the UML meta-model used to model the SPARK Ada program (i.e. Operations, Attributes, Typedefs and Dependencies.) For brevity's sake, we only describe the operation transformation in detail in this thesis.

Identification and Duplication of Class Contents

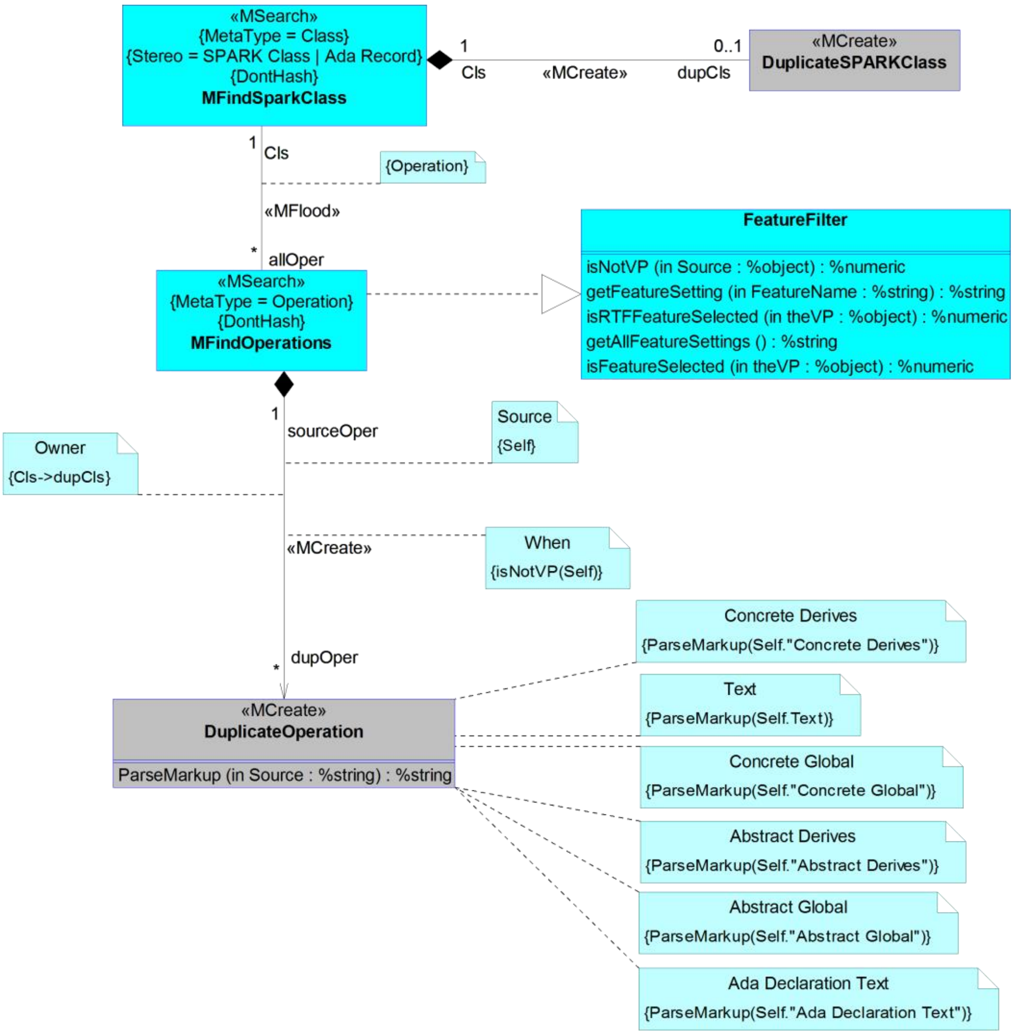

«MSearch»
{MetaType = Class}
{Stereo = SPARK Class | Ada Record}
{DontHash}
MFindSparkClass
1
Cls
«MCreate»
0..1
dupCls
«MCreate»
DuplicateSPARKClass
1
Cls
{Operation}
«MFlood»
*
allOper
«MSearch»
{MetaType = Operation}
{DontHash}
MFindOperations
FeatureFilter
isNotVP (in Source : %object) : %numeric
getFeatureSetting (in FeatureName : %string) : %string
isRTFFeatureSelected (in theVP : %object) : %numeric
getAllFeatureSettings () : %string
isFeatureSelected (in theVP : %object) : %numeric
1
sourceOper
Source
{Self}
Owner
{Cls->dupCls}
«MCreate»
When
{isNotVP(Self)}
*
dupOper
«MCreate»
DuplicateOperation
ParseMarkup (in Source : %string) : %string
Concrete Derives
{ParseMarkup(Self."Concrete Derives")}
Text
{ParseMarkup(Self.Text)}
Concrete Global
{ParseMarkup(Self."Concrete Global")}
Abstract Derives
{ParseMarkup(Self."Abstract Derives")}
Abstract Global
{ParseMarkup(Self."Abstract Global")}
Ada Declaration Text
{ParseMarkup(Self."Ada Declaration Text")}

FIGURE 112 TRANSFORMATION RULE DUPLICATING NON-VARIANT OPERATIONS

Figure 112 defines the transformation fragment concerned with the duplication of non-variant (common) operations. This pattern repeats throughout the transformation for each of the UML meta-model elements that are relevant in the product line models (i.e. attributes, typedefs and dependencies).

The **MFindOperations** class receives the set of operations defined in the selected SPARK classes (from **MFindSPARKClasses**). **MFindOperations** inherits a set of "helper" operations from the utility class **FeatureFilter**, which are primarily used to determine when variation has been selected; in this pattern only the "isNotVP()" operation is actually used. Any operation that is NOT marked as a variation point is duplicated into the target deployment model. The «MCreate» class **DuplicateOperation** inherits the **DOperation** factory class and performs the duplication; this is guarded by a **when** constraint: isNotVP("Self").

The constraints on the **DuplicateOperation** class are interesting. Although the operation itself is common (i.e. has no PL Variation Point stereotype), this does not necessarily imply that the operation itself may not contain variability; it may be a common operation with a variable implementation (i.e. variation with the body of the operation). The constraints on the **DuplicateOperation** class perform the variation processing of the operation contents.

DuplicateOperation declares an operation "ParseMarkup()". This is significant in the overall transformation design. "ParseMarkup()" provides an interface to an ANTLR text processor that removes variation from text fields within the model. A discussion of the design of this processor and its implications is contained later in this chapter. The extent of its use should be noted; the fields that are used within SPARK operations, and which may contain mark-up are:

- Text – containing the Ada source code of the operation body
- Ada Declaration Text – any declarations local to the sub-program (local variables or local sub-programs)
- Abstract Globals – SPARK Abstract Global annotations may contain variation
- Abstract Derives – SPARK Abstract Derives annotations may contain variation
- Concrete Globals – SPARK Concrete Global annotations may contain variation
- Concrete Derives – SPARK Concrete Derives annotations may contain variation

The transformation cannot process these fields using the provided TDK 4G mechanisms as the text they contain does not correspond to any declared meta-model. The implications of this were discussed in the main body of the thesis (Chapters 5 and 7).

Compare this transformation fragment to that shown in Figure 113. This shows the transformation pattern for **variant** operations; again, this pattern is duplicated for all UML meta-model elements relevant in modelling the SPARK program.

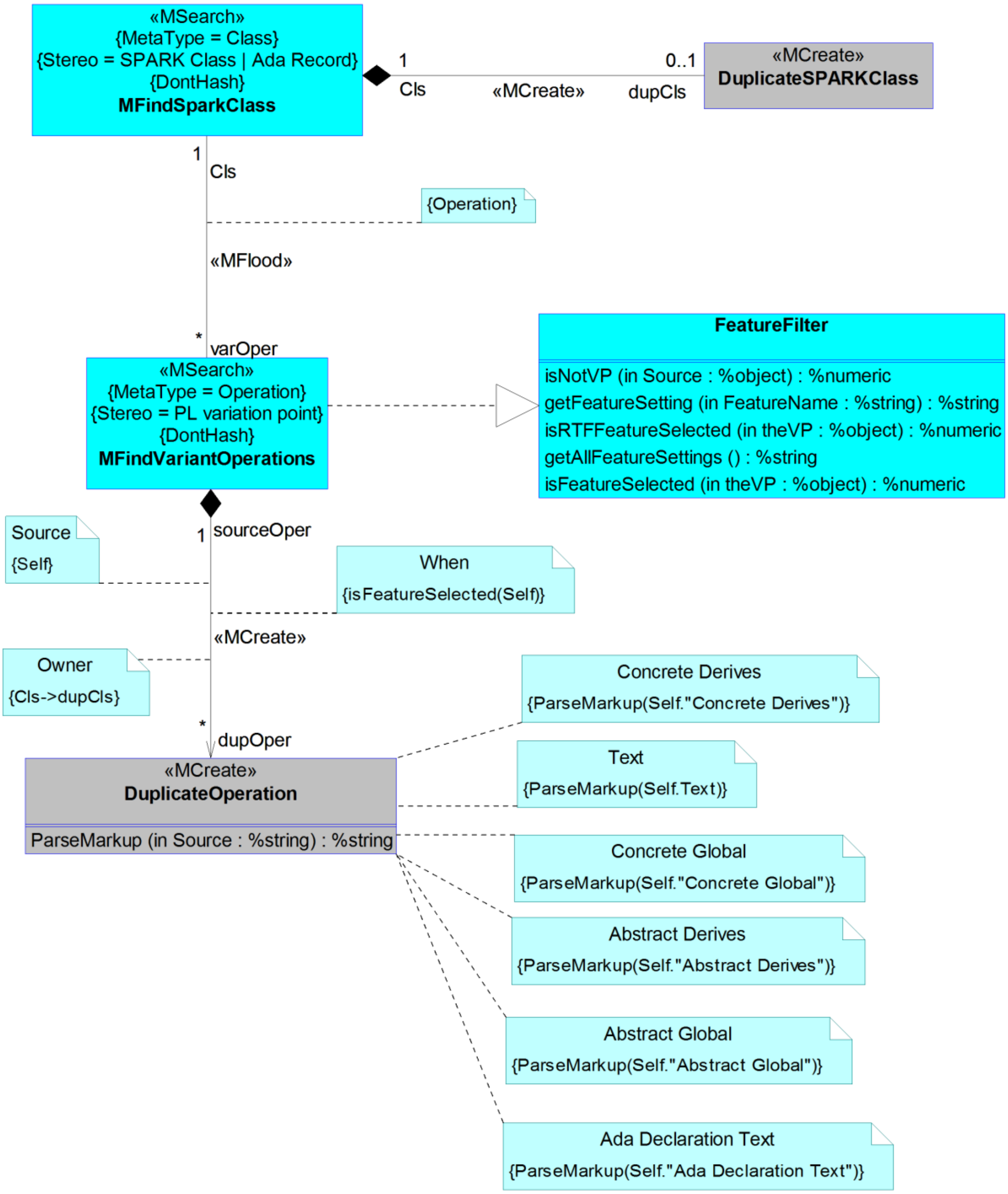

«MSearch»
{MetaType = Class}
{Stereo = SPARK Class | Ada Record}
{DontHash}
MFindSparkClass
1
Cls
«MCreate»
0..1
dupCls
«MCreate»
DuplicateSPARKClass
1
Cls
{Operation}
«MFlood»
*
varOper
«MSearch»
{MetaType = Operation}
{Stereo = PL variation point}
{DontHash}
MFindVariantOperations
FeatureFilter
isNotVP (in Source : %object) : %numeric
getFeatureSetting (in FeatureName : %string) : %string
isRTFFeatureSelected (in theVP : %object) : %numeric
getAllFeatureSettings () : %string
isFeatureSelected (in theVP : %object) : %numeric
Source
{Self}
1
sourceOper
When
{isFeatureSelected(Self)}
«MCreate»
Owner
{Cls->dupCls}
*
dupOper
«MCreate»
DuplicateOperation
ParseMarkup (in Source : %string) : %string
Concrete Derives
{ParseMarkup(Self."Concrete Derives")}
Text
{ParseMarkup(Self.Text)}
Concrete Global
{ParseMarkup(Self."Concrete Global")}
Abstract Derives
{ParseMarkup(Self."Abstract Derives")}
Abstract Global
{ParseMarkup(Self."Abstract Global")}
Ada Declaration Text
{ParseMarkup(Self."Ada Declaration Text")}

FIGURE 113 TRANSFORMATION RULE DUPLICATING VARIANT OPERATIONS

The differences between this and the non-variant pattern are twofold; firstly the central «MSearch» class collecting operations (**MFindVariantOperations**) is filtered on the «PL Variation Point» stereotype – that is, only operations identified as variation points are collected. Secondly, the «MCreate» association with **DuplicateOperation** is guarded by a call to the “isFeatureSelected()” operation. This ensures only operations that valid in the current product are duplicated into the deployment model.

Transformation of Enumeration Literals

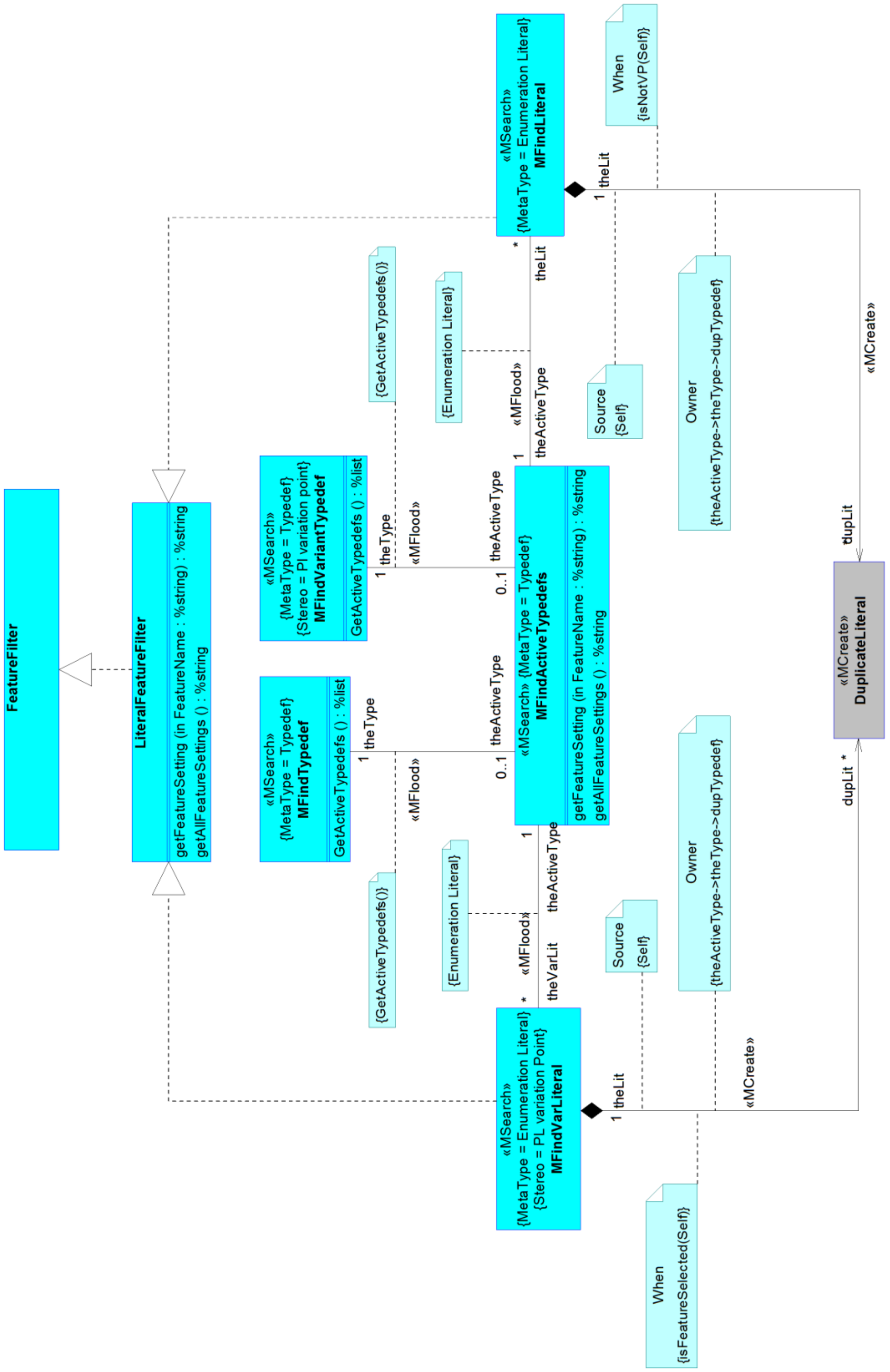


Figure 114 Transformation Rule for Deployment of Enumeration Literals

The final UML meta-model element that requires transformation is the Enumeration Literal. The designed transformation allows literals to be included or removed from type definitions. The type definition itself may be common or a variation point. Figure 114 shows the transformation classes for dealing with enumeration literals.

The **MFindTypedef** and **MFindVariantTypedef** «MSearch» classes contain the set of Typedefs collected during the Typedef transformations. These are collected together into a single set in the **MFindActiveTypedefs** «MSearch» class. Typedef elements may contain the declaration of sets of enumeration literals if the Typedef represents an enumerated type. Two sets of enumeration literals are collected: **MFindLiteral** contains all enumeration literals declared in the active Typedefs. **MFindVarLiterals** collects only those enumeration literals that are stereotyped as PL Variation Point. All common literals are duplicated into the target deployment model via the association between **MFindLiteral** and the **DuplicateLiteral** MCreate class. This is guarded by the” isNotVP()” operation.

(Note that this illustrates a shortcoming in the 4G TDK semantics. The ability to collect a set of elements and then divide this set into two collections, one set including elements with a given stereotype, and a second containing the remaining elements NOT stereotyped is used throughout the product line M2M transformation. However, whilst a collection can be formed declaratively of elements with a given stereotype, the converse is NOT true (e.g. “collect a set of classes NOT stereotyped «SPARK Class»”). This has to be performed by using a procedural operation as a flood parameter, filtering out the stereotyped elements. The ”isNotVP()” operation is an example of this.)

The enumeration literals that are marked as variation points and have been selected for deployment in this product are duplicated into the target deployment model via the association between **MFindVarLiteral** and the **DuplicateLiteral** MCreate class. This is guarded by the ”isFeatureSelected()” operation.

At this point in the transformation process there now exists a model in memory which represents the deployed component set with all variations points resolved. This now needs to be transformed into a model from which SPARK Ada can be generated. This is achieved by applying a set of design pattern transformations.

Opaque Behaviour and Textual Transformation

We discussed the role of text transformations to support variability in “opaque behaviour” regions in section 5.3.2. There we described how the M2M transform delegates these regions of “opaque behaviour” to an ANTLR parser, and we have seen in this appendix how the ParseMarkup operation is used at various points in the transformation. Here we show how the simple mark-up language is defined using an ANTLR grammar that is shown in Code Listing 3 below.

```
grammar VPMarkup;

options {
      language = C;
      output=AST;
      ASTLabelType=pANTLR3_BASE_TREE;
}
tokens {
INSERT;
CONDITION;
FEATURE;
FEATURENOT;
ANDOP;
OROP;
}

// Rules

markupFile  : VP_BEGIN! (ifCommand|contentSkip )* VP_END!;

ifCommand   : IF^ vpSpec theContent (elsifCommand)* (elseCommand)*
ENDIF! ;

elsifCommand : ELSIF^  vpSpec theContent;

elseCommand : ELSE^  theContent;

theContent  : (ifCommand|contentSkip)*;

vpSpec      : variationPoint (boolOp ^ vpSpec)?;

variationPoint    :   featureSetting -> ^(FEATURE featureSetting)  |
               NOT   featureSetting -> ^(FEATURENOT featureSetting);


featureSetting    :  theFeature '=' theValue -> ^(theFeature
theValue) ;

theFeature  : NAME;

theValue    : NAME;

boolOp      : andOp -> ^(ANDOP)
              | orOp-> ^(OROP)  ;

andOp       : 'and' |  'AND' | 'And';

orOp        : 'or' | 'OR' | 'Or';
```

```
contentSkip : CONTENT -> ^(INSERT CONTENT);


// Tokens

VP_BEGIN    : 'VPBegin' | 'VPBEGIN'|'VPbegin'|'vpbegin';
VP_END      : 'VPEnd' | 'VPEND'| 'vpend' | 'VPend';

NOT         : 'not'| 'NOT' | 'Not';

IF          : 'VPIf'| 'VPIF'| 'VPif' | 'vpif';
ELSIF             : 'VPElsIf'|'VPElsif' | 'VPelsif'| 'vpelsif'|
'VPELSIF';
ELSE        : 'VPElse' | 'VPelse' | 'vpelse' | 'VPELSE';
ENDIF             : 'VPEndIf'| 'VPENDIF' | 'vpendif' | 'VPendif' |
'VPEndif';

NAME:    ( 'a' .. 'z' | 'A' .. 'Z' | '_')
        ( 'a' .. 'z' | 'A' .. 'Z' | '_' | '0' .. '9' )*;


OPEN        : '{';
CLOSE       : '}';
CC          : '//';

CONTENT     :   OPEN (options {greedy=false;} : .)* CLOSE;

COMMENT     :   CC (options {greedy=false;} : .)* '\n'
{$channel=HIDDEN;};

WS          : (' '|'\r'|'\t'|'\n')+ {$channel=HIDDEN;};
```

Code Listing 3 Antlr Grammar defining Mark-up Language

This simple grammar allows regions of text to be surrounded by braces ({}). Each text region can be either common (always passed through to the final product-specific variant) or be guarded by an expression (in terms of the component decisions) that identifies whether the region is included. As the main target language for this approach is SPARK (based upon Ada 95), the brace characters were used to identify the text regions as braces are not tokens in the SPARK language. This approach was taken to simplify the parser; instead of having to include the complete grammar of any target language in the mark-up parser, anything within braces is passed through to the output (this is the purpose of the `CONTENT : OPEN (options {greedy=false;} : .)* CLOSE; ) ;` statement).

Template Components & Transformation

As discussed in section 5.3.3, template processing was a late addition to the transformation; this additional functionality was included with the minimal impact on the existing transform via the use of multiple inheritance. The majority of the template handler behaviour was encapsulated in a helper class **TemplateHandle**r, and then inherited by the appropriate «MSearch» classes as shown in Figure 115.

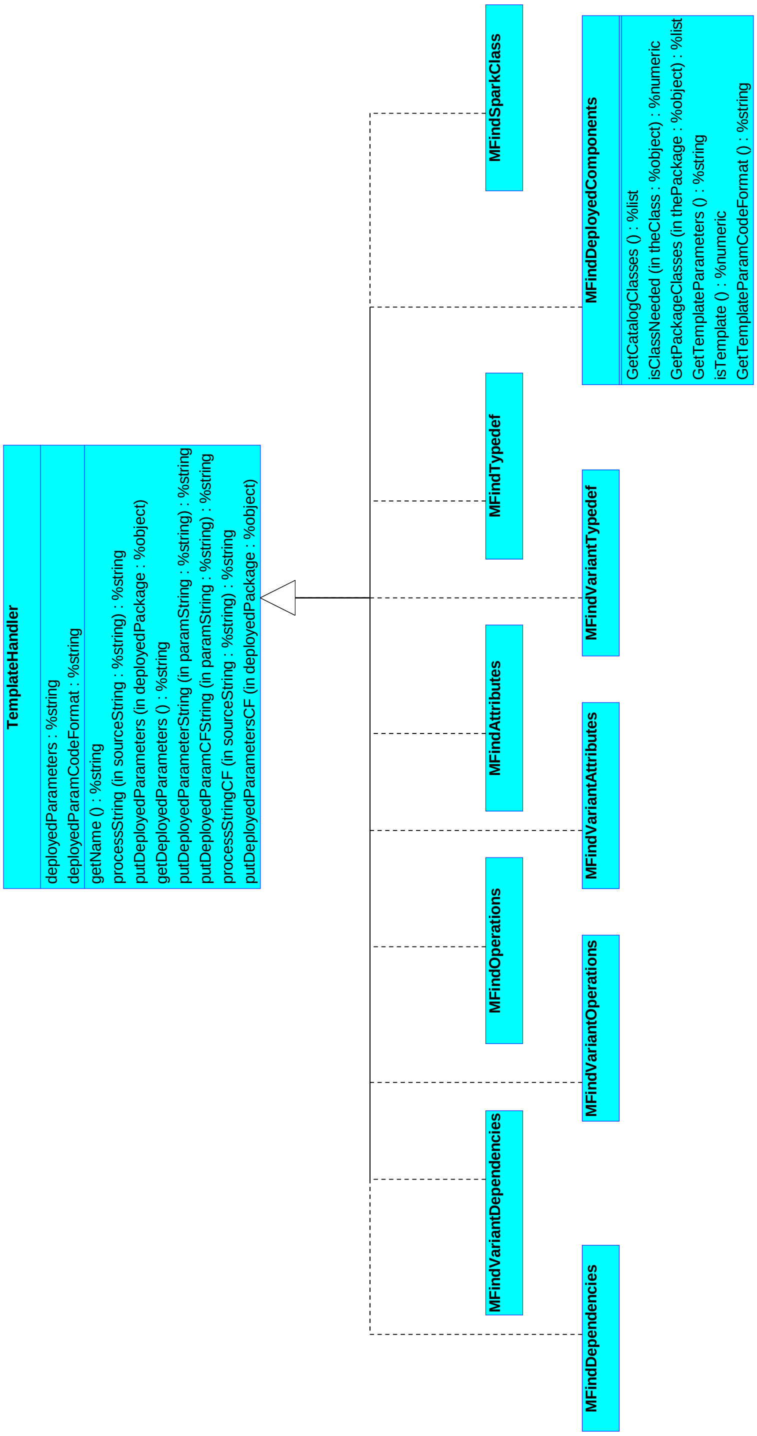

TemplateHandler
deployedParameters : %string
deployedParamCodeFormat : %string
getName () : %string
processString (in sourceString : %string) : %string
putDeployedParameters (in deployedPackage : %object)
getDeployedParameters () : %string
putDeployedParameterString (in paramString : %string) : %string
putDeployedParamCFString (in paramString : %string) : %string
processStringCF (in sourceString : %string) : %string
putDeployedParametersCF (in deployedPackage : %object)
MFindSparkClass
MFindDeployedComponents
GetCatalogClasses () : %list
isClassNeeded (in theClass : %object) : %numeric
GetPackageClasses (in thePackage : %object) : %list
GetTemplateParameters () : %string
isTemplate () : %numeric
GetTemplateParamCodeFormat () : %string
MFindTypedef
MFindVariantTypedef
MFindAttributes
MFindVariantAttributes
MFindOperations
MFindVariantOperations
MFindVariantDependencies
MFindDependencies

Figure 115 Inheriting Template handling Capabilities

Expanding Design Patterns

As discussed in section 5.3.4, the final part of the M2M transformation chain is the expansion of design patterns. We identified that there are six design patterns to be expanded; the transformations that support these are applied in a well-defined order:

1. Apply Development Variable (DV) Pattern
2. Apply Testpoint Pattern
3. Apply Interface Pattern
4. Apply OS Interface Pattern
5. Apply Testport Pattern
6. Apply Graphical Data Pattern

The following section describes the detail of the Development Variable transformation. We omit the details of the other downstream transformations for brevity.

Development Variable Pattern

Development Variables (DVs) provide the means to alter nominally constant data in the program (i.e. they can be regarded as “variable constants”). DVs are set to a default initial value at program reset, but each read of the DV value will be made from RAM, allowing the value to be “soft-trimmed” (i.e. changed at run-time via test equipment). The initial value of the DV can also be altered via a process of “hard trimming” (i.e. downloading a new set of default data values to the controller’s FLASH memory). From a design viewpoint, the only required information for each DV is its name, data type and default value. However, we need to cater for a number of language-level subtleties in the code generation (PSM) model.

Each component that requires a set of Development Variables will contain a “calibration” Ada package named <component>DV. The UML class representation will be stereotyped «DV Class». Typically, the final code form of a DV Ada package containing a single DV will look as follows:

```
with Base;

--# Inherit
--#   Base
--# ;

package ComponentDV
--# own data : DVRecordType;
 is


  ------------------------------------------------------------------
  -- public sequence types
  ------------------------------------------------------------------
  subtype myDVType is Base.Real range 0.0 .. 9999.0;


  ------------------------------------------------------------------
  -- public operation declarations
  ------------------------------------------------------------------

  function myDV return myDVType;

  pragma INLINE(myDV);


  ------------------------------------------------------------------
  -- initialise
  --
  -- Description:
  ------------------------------------------------------------------
  procedure initialise;
  --# Global
  --#   out   data
  --# ;
  --# Derives
  --#   data from
  --# ;


  ------------------------------------------------------------------
  -- testport
  --
  -- Description:
  ------------------------------------------------------------------
  procedure testport;
  --# Derives
  --#
  --# ;


private

end ComponentDV;
```

Code Listing 4 Example Component DV Package Specification

```
package body ComponentDV
is


  ------------------------------------------------------------------
  -- Private record types
  ------------------------------------------------------------------

  type DVRecordType is record

      myDV : myDVType;

  end record;


  ------------------------------------------------------------------
  -- Private typed constants
  ------------------------------------------------------------------

  initial : constant DVRecordType := DVRecordType'(
      myDV  =>    0.5);


  ------------------------------------------------------------------
  -- Private State
  ------------------------------------------------------------------

  data : DVRecordType;

  ------------------------------------------------------------------
  -- Public operations
  ------------------------------------------------------------------

  function myDV return MyDVType
  is --# hide myDV
     --Hidden body accesses state outside SPARK boundary.
  begin
    return data.myDV;
  end myDV;


  ------------------------------------------------------------------
  -- initialise
  --
  -- Implementation Notes:
  --
  ------------------------------------------------------------------
  procedure initialise
  is
  begin
    data := initial;
  end initialise;

  ------------------------------------------------------------------
  -- testport
  --
  -- Implementation Notes:
  --
  ------------------------------------------------------------------
  procedure testport
  is separate;


end ComponentDV;
```

Code Listing 5 Example Component DV Package Body

Prior to model transformation being available, each of the Ada language constructs visible in the source code listing above would need to be explicitly modelled in UML to enable syntactically correct code to be produced. However, the following parts of this pattern are standard and their generation can be automated as part of the design pattern transformation:

- Initialise operation

- Testport operation
- DVRecordType definition
- Initial attribute declarations
- DV function body including SPARK hide
- Inline of DV function prototype

This leaves the following information to be included in the input model:

- DV Classname
- DV Names
- DV Types
- DV Default Values

Figure 116 shows the DV Pattern 4G TDK transformation.

The pattern firstly finds all the «DV Class» classes within the set of deployed components (**MGetDeployedComponentsForDV** and **MFindDVClass**), then finds the set of operations declared in this class. The set of required DVs are modelled as typed operations with the following additional tag:

- InitialValue holds the default initial value for the development variable.

The relatively large number of «MCreate» factory class instantiations in the transform indicates the level of automatically created entities in the final model, and is an indication of the level of abstraction of the input model. All of these entities used to be modelled explicitly by the designer prior to the transformation-based code generator; with the consequential potential for error (e.g. a common mistake was to return the incorrect value in a DV function body as the designer constructed the class contents by copy and paste).

The DV pattern defines a number of standard constructs that must exist in every DV class, there are created for each class found via **MFindDVClass** :

- An Ada record typedef called “DVRecordType” whose element hold the individual DV definitions is created by **theDVRecord** «MCreate» class.
- An “Initial” constant attribute to hold the initial values for the DVs is created by **theInitialAttr**. Note the use of the “After” constraint on the «MCreate» association here. The Initial attribute is of type “DVRecordType” which is also created via transformation. The transformation must create the objects in the correct order to allow them to be subsequently referred to. The “After” constraint ensures that all required objects exist before a creation operation takes place.
- An attribute called “Data” to hold the in-memory values of the DVs is created by **theDataAttr**. This is also of type “DVRecordType” and therefore has the “After” constraint on the creation.
- An operation to initialise the Data attribute is created by **theInitialiseOp**. Note the Text constraint on the factory class which provides the operation body “data := initial;”.

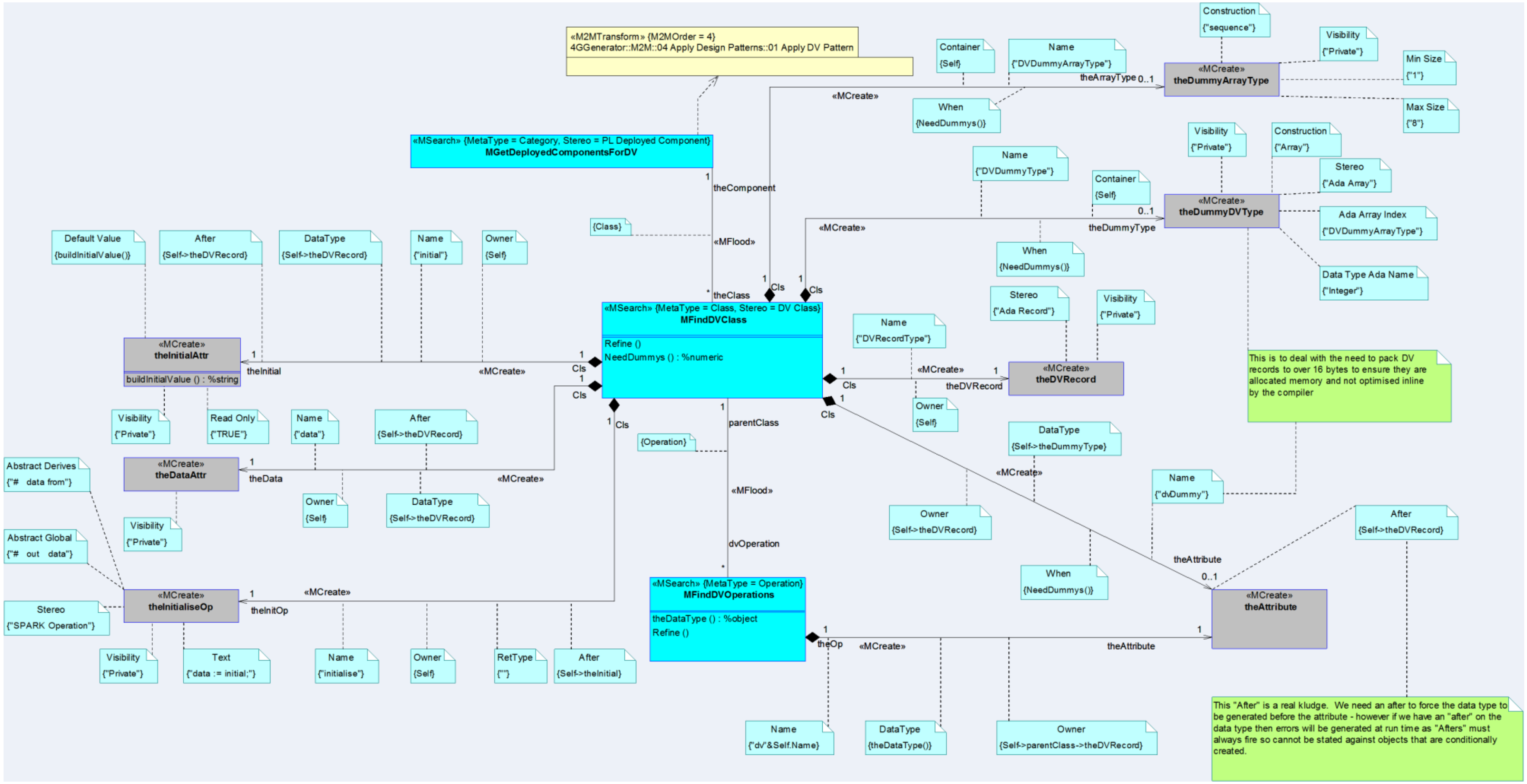
- A data attribute is included in the DVRecordType definition for each identified DV operation in the source model. These are created by **theAtrribute** in response to the operations collected in **MFindDVOperations**.

The power of the use of transformations to hide target-dependent detail from the designer is shown in the elements of the transformation that contain the word “dummy”.

These parts of the transformation exist to provide a work-around due to the behaviour of the particular compiler used on the project. As described earlier, DVs are used to provide “trimmable” constants, whose value may be altered post-compliation. The “hard-trim” process changes the values of these constants in the FLASH memory of the controller. To be able to do this, the DVs need to reside in separate memory regions as defined by the linker process. However, the compiler used will attempt to optimise away any constant value that is less than 16 bytes in size, preferring to locate those values in-line with the program code. To force all DV declarations to be greater than 16 bytes in size, additional “dummy” attributes are inserted into the DV record type declarations for any records that would be otherwise less than 16 bytes. On previous projects, this dummy packing had to be included by the component designers in the source UML model. This was unsatisfactory for a number of reasons, including:

- If this was missed by the designers, the program would still compile, and the problem would only be found when those particular values in the controller were attempted to be trimmed – this could be at customer sites or during costly engine tests.
- This is low-level detail due to specific compiler behaviour – it should not really in the domain of the component designer to address.

The solution to this is to encode the creation of the dummy attributes in the expansion of the design pattern. This makes the process both transparent to the design, and reliable in its implementation.



This is to deal with the need to pack DV records to over 16 bytes to ensure they are allocated memory and not optimised inline by the compiler

This "After" is a real kludge. We need an after to force the data type to be generated before the attribute - however if we have an "after" on the data type then errors will be generated at run time as "Afters" must always fire so cannot be stated against objects that are conditionally created.

FIGURE 116 APPLY DEVELOPMENT VARIABLE PATTERN TRANSFORM

Code Generation (Model-to-Text Transformation)

The final transformation phase shown in Figure 106 is the Model-to-Text transformation that produces the SPARK source code. An important property of this phased transformation approach is that the transitory model presented to the M2T code generator is of the same form as the single-project UML model that was used in previous, single-system projects. Therefore, minimal changes are required to the M2T generator to enable its use on a product line development.

The previous generator used ARTiSAN's OCS (On-Demand Code Synchronisation) technology to transform a decorated UML Class model to SPARK code. OCS uses a set of code templates to transform a UML model to text "on demand". The OCS SPARK generator and associated SPARK UML profile was originally developed by Praxis High Integrity Systems and subsequently modified by Rolls-Royce/AEC. This was used successfully on two FADEC development projects, generating products in excess of 250KSLOC each. A detailed description of the OCS generator is beyond the scope of this thesis (it was not generated as part of this research). Given a UML-to SPARK generator with this level of pedigree, however, it was felt that this should be the basis of the back-end of the product-line code generator. It is possible to "host" OCS-based generators within an ACS generator; the OCS templates are imported en-masse as operations in a generator package. To get this OCS-style generator to transform a transitory in-memory model involved creation of a small "visitor" transformation which traverses the in-memory model, visiting each UML category (package) and class, and invoking the OCS M2T transformation on each class found (via the call to doOCSGenerate() within QCLass). This visitor or mapping model is shown in Figure 117 below.

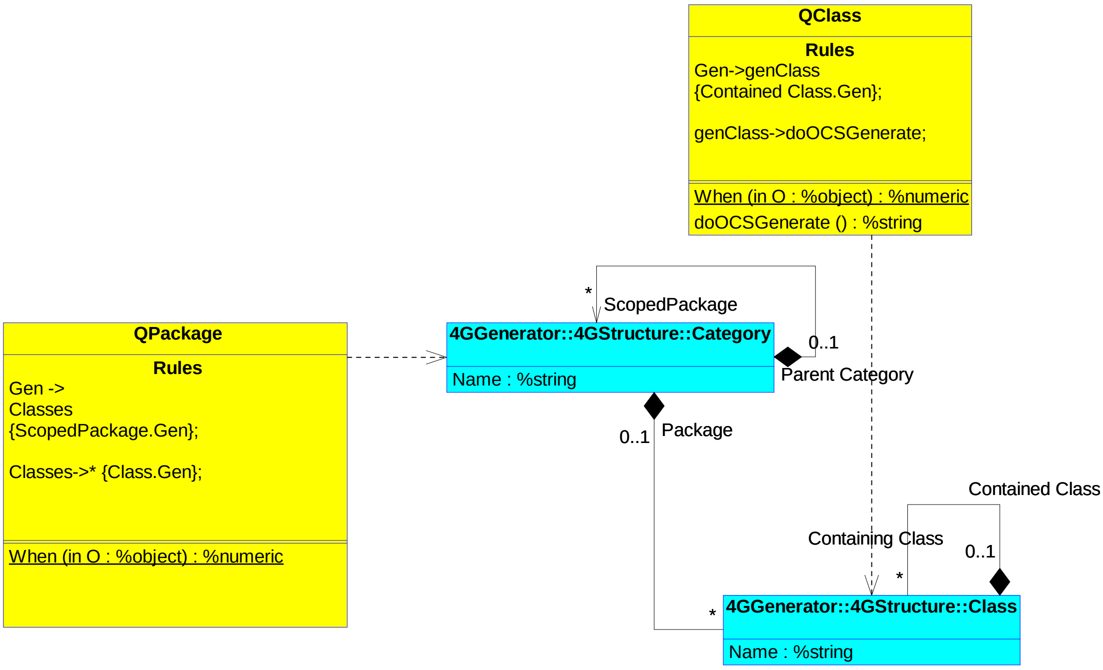
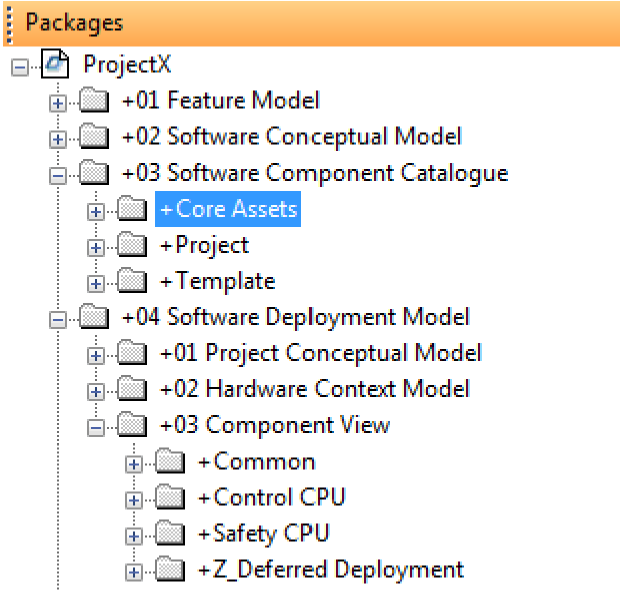


Figure 117 ACS "visitor" transformation which invokes the legacy OCS M2T Code Generator

Appendix C – Case Study

In this appendix we work through an example to illustrate the use of our Trusted Product Lines approach. We show how the reference software architecture and core assets are deployed on a project and how components can be instantiated to meet the requirements of that particular project.

Model Structure



Packages

- ProjectX
  - +01 Feature Model
  - +02 Software Conceptual Model
  - +03 Software Component Catalogue
    - +Core Assets
    - +Project
    - +Template
  - +04 Software Deployment Model
    - +01 Project Conceptual Model
    - +02 Hardware Context Model
    - +03 Component View
      - +Common
      - +Control CPU
      - +Safety CPU
      - +Z_Deferred Deployment

FIGURE 118 MODEL PACKAGE STRUCTURE FOR A PRODUCT LINE PROJECT INSTANCE

Figure 118 shows the top levels of the package structure for a project that uses our product line approach. The very top-level packages represent the following:

- 01 Feature Model – This package provides a link to the high-level software requirements/system requirements implemented by the software. In this case, the contents of this package are simply requirement identification & traceability tags, but this package can contain more complex models of variability, as required by the system-level feature model. Its purpose in our example is to provide a home for requirements identifiers for traceability linkage.

- 02 Software Conceptual Model – This package provides a model of the overall concept of the software, typically used to describe the "high-level" software architecture. It is standardised across all projects that use this reference architecture (it contains the reference architecture definition); specific peculiarities of the project architecture are described in the project conceptual view in the deployment model (see 04 later).

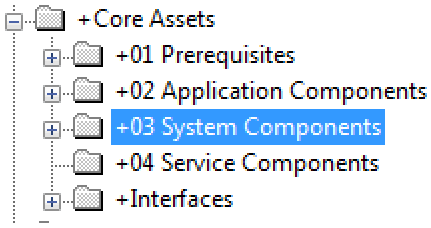
- 03 Software Component Catalogue – This package contains the definitions of all the software components that are used in the project. This is split into two sub-packages: Core Assets and Project. Core Assets contains the set of components that are produced for and managed by the product line initiative. Project contains the set of components produced specifically for the project instance itself. However, both sets of components are produced to the same design standards and are deployed in the same manner. This means that any components that need to migrate (usually from Project into Core Assets when they are deemed applicable to more than this product) can be moved with minimal technical effort.

- 04 Software Deployment Model – This package contains the instantiation rules for this project. It models the specific features of the target hardware for the project, any changes or additions to the software architectural concept and, in the "03 Component View" sub-package, the deployment of the components onto the target CPUs.

This appendix concentrates on the illustrating the relationship between the 03 Component Catalogue and the 04 Deployment Model, and how they work together with the model transformations to instantiate product source code.

Core Asset Components

In chapter 4, we described how the reference architecture for our products identifies a number of layers in the software. These layers are reflected in the package structure holding the component definition.

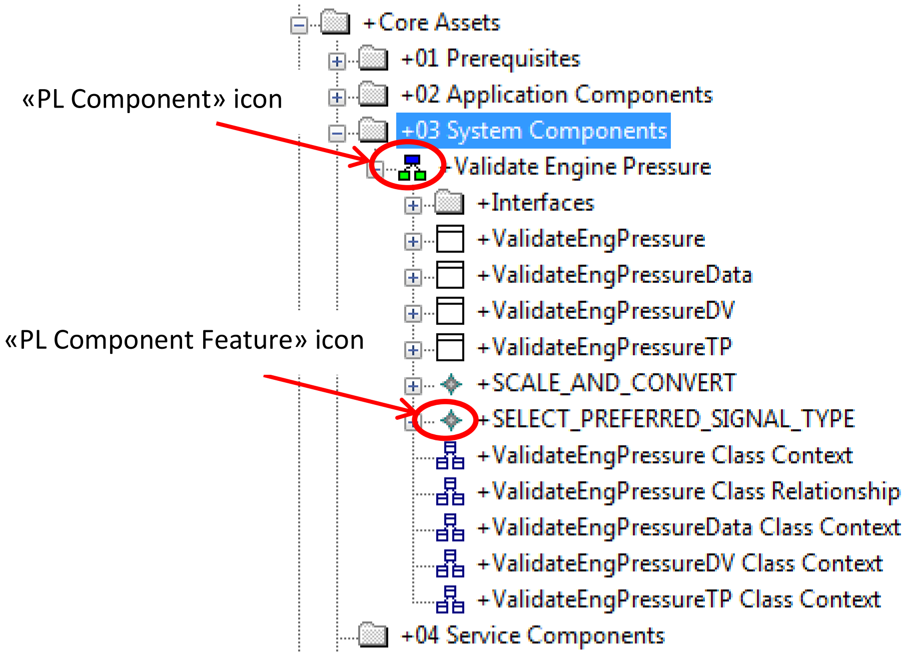


- +Core Assets
  - +01 Prerequisites
  - +02 Application Components
  - +03 System Components
  - +04 Service Components
  - +Interfaces

Figure 119 Layer Structure Reflected in the Core Assets Component Catalogue

We now look at the structure of a component within one these layers. This is a core asset component that lives within the "System" layer, and the purpose of the component is to validate an engine pressure signal. The component is named "Validate Engine Pressure" and its location in the "Core Asset" package structure can be seen in Figure 120.

Here, our use of the ARTiSAN Studio ergonomic profiling can be seen. As described in Chapter 4, special icons are provided to indicate that parts of the model have been specially stereotyped. The Validate Engine Pressure component itself is a UML package stereotyped as a «PL Component», which is associated with the blue/green 3-box icon.



«PL Component» icon

«PL Component Feature» icon

- +Core Assets
  - +01 Prerequisites
  - +02 Application Components
  - +03 System Components
    - +Validate Engine Pressure
      - +Interfaces
      - +ValidateEngPressure
      - +ValidateEngPressureData
      - +ValidateEngPressureDV
      - +ValidateEngPressureTP
      - +SCALE_AND_CONVERT
      - +SELECT_PREFERRED_SIGNAL_TYPE
      - +ValidateEngPressure Class Context
      - +ValidateEngPressure Class Relationship
      - +ValidateEngPressureData Class Context
      - +ValidateEngPressureDV Class Context
      - +ValidateEngPressureTP Class Context
  - +04 Service Components

Figure 120 System Layer Component - Validate Engine Pressure

We can also see in Figure 120 that, in addition to the component's internal classes and diagrams, the component published two features or "decisions", denoted by the grey star icons. These form the "decision contract" for that component, and these must be resolved by any users of the component at deployment time. To illustrate the use of these, we need to understand the requirements the component is designed to fulfil.

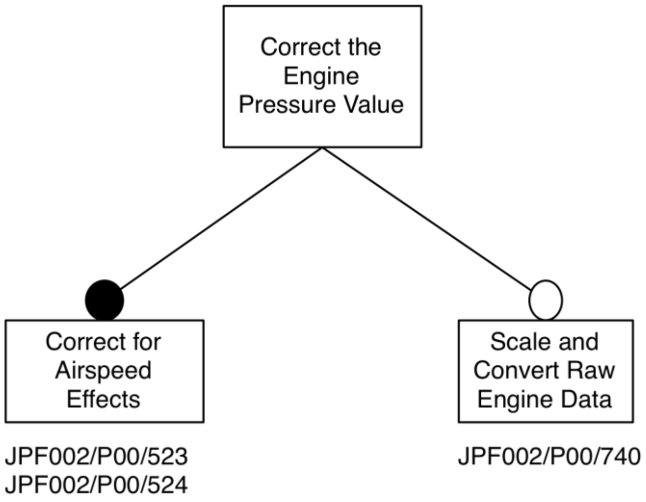

Correct the Engine Pressure Value
Correct for Airspeed Effects
Scale and Convert Raw Engine Data
JPF002/P00/523
JPF002/P00/524
JPF002/P00/740

Figure 121 Feature Model Fragment - Engine Pressure Signal Correction

Figure 121 shows a fragment of the requirements for this component in the form of a feature model; this shows the requirements for the correction of the pressure signal. This has a mandatory feature (the correction of the signal to remove the effect of the airspeed) and an optional feature (to scale and convert the raw engine data). The references provided in Figure 121 under the features are the requirement traceability references ("tag") for the required behaviour (the high-level software requirements). We do not need to understand the technical detail of these requirements for the purposes of this case study.

The optional feature in Figure 121 is represented in the Validate Engine Pressure component by a simple true/false selection in the decision contract, as shown in Figure 122.

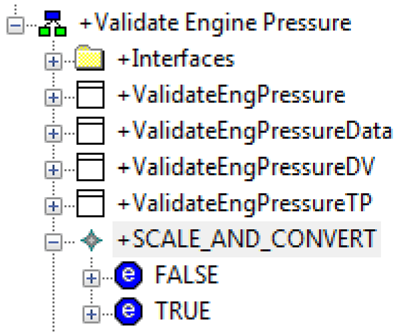
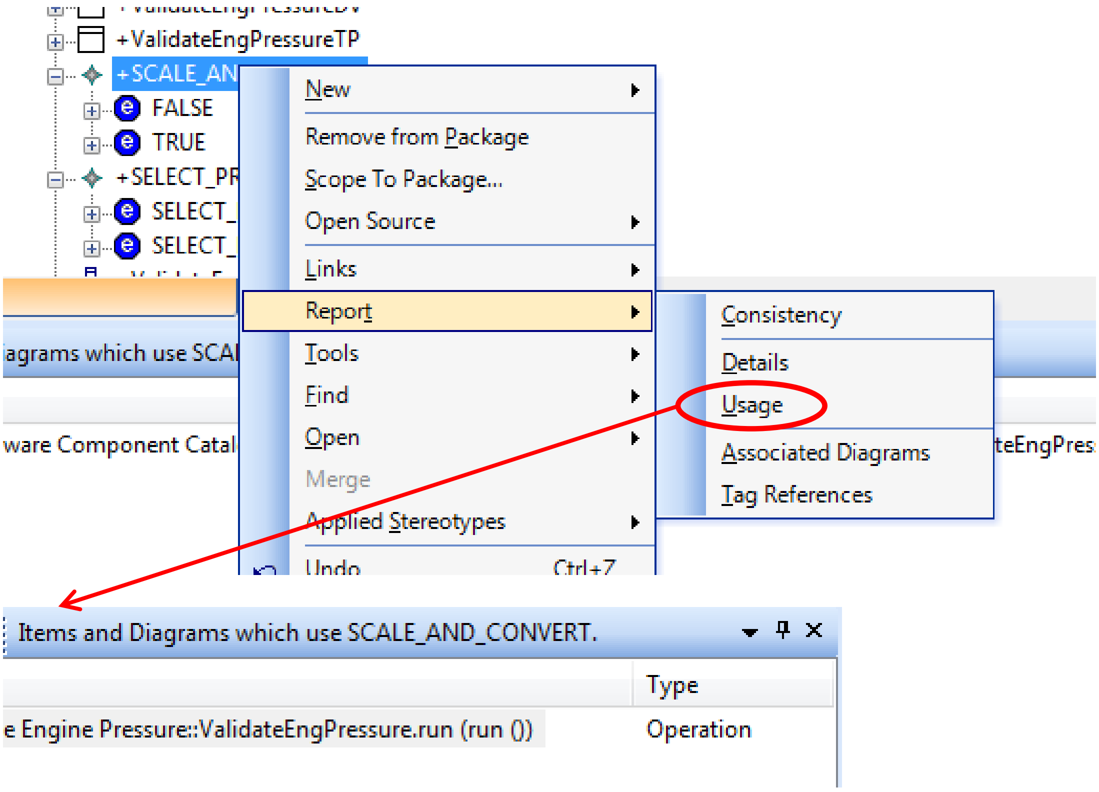


Figure 122 Decision Contract for Scale & Convert Option

One advantage of implementing the decision contracts as stereotyped UML elements is the ability to use the built-in model navigation features of the modelling tools. Within ARTiSAN Studio there is the ability to report the usage of model elements – if we report on the usage of the decision, the tool will provide a list of those parts of the model that are affected by that decision (see Figure 123).

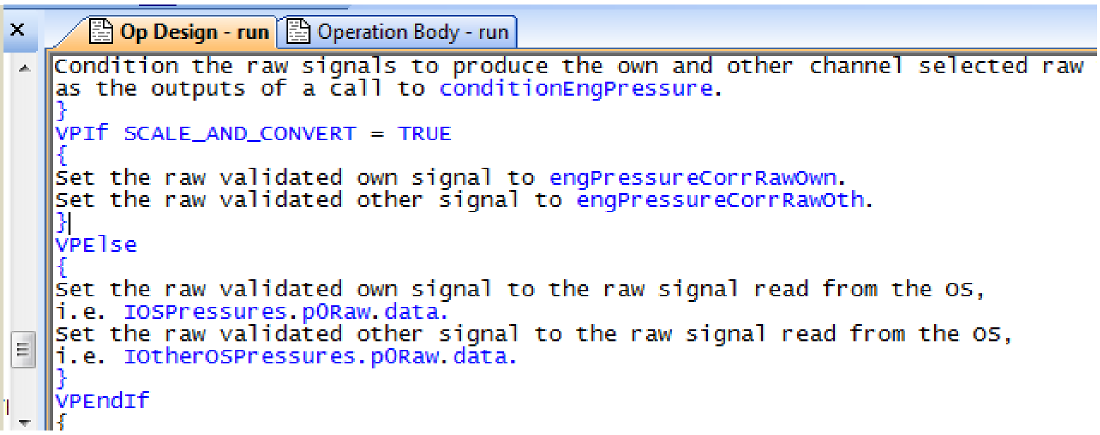


Items and Diagrams which use SCALE_AND_CONVERT.

	Type
e Engine Pressure::ValidateEngPressure.run (run ())	Operation

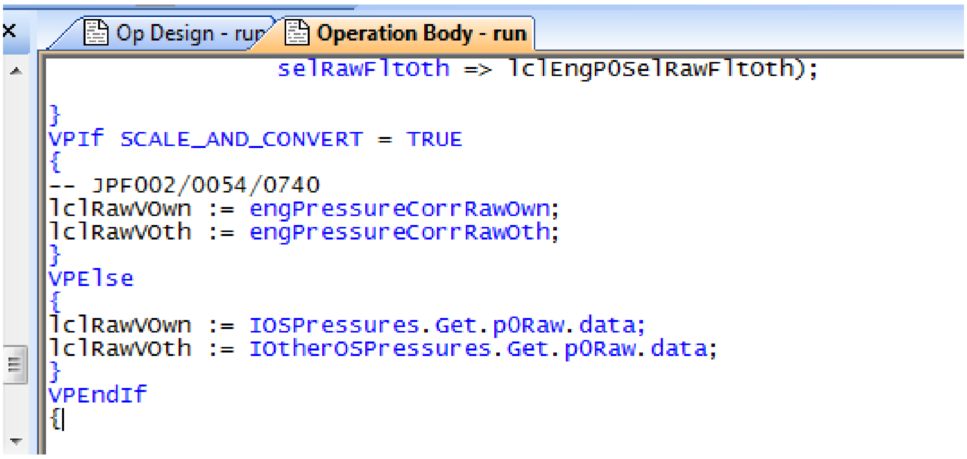
Figure 123 Reporting Usage of Decisions in ARTiSAN Studio

In fact, this decision only affects the body of the “run” operation inside the ValidateEnginePressure class. Figure 124 and Figure 125 illustrate how the design text and SPARK code body for the operation make use of the mark-up text facility to include/exclude the correction code.



```
Condition the raw signals to produce the own and other channel selected raw
as the outputs of a call to conditionEngPressure.
}
VPIf SCALE_AND_CONVERT = TRUE
{
Set the raw validated own signal to engPressureCorrRawOwn.
Set the raw validated other signal to engPressureCorrRawOth.
}
VPElse
{
Set the raw validated own signal to the raw signal read from the OS,
i.e. IOSPressures.p0Raw.data.
Set the raw validated other signal to the raw signal read from the OS,
i.e. IOtherOSPressures.p0Raw.data.
}
VPEndIf
{
```

FIGURE 124 OPTIONAL TEXT IN COMPONENT DESIGN DESCRIPTION



```
                selRawFltOth => lclEngP0SelRawFltOth);

}
VPIf SCALE_AND_CONVERT = TRUE
{
-- JPF002/0054/0740
lclRawVOwn := engPressureCorrRawOwn;
lclRawVOth := engPressureCorrRawOth;
}
VPElse
{
lclRawVOwn := IOSPressures.Get.p0Raw.data;
lclRawVOth := IOtherOSPressures.Get.p0Raw.data;
}
VPEndIf
{
```

FIGURE 125 OPTIONAL CODE IN COMPONENT BODY

We can also use the model reference search facility to follow traceability links, and see where behavioural requirements related to features are implemented. Figure 126 shows the menu selection to query the usage of the optional requirement SRS/P00/740 .

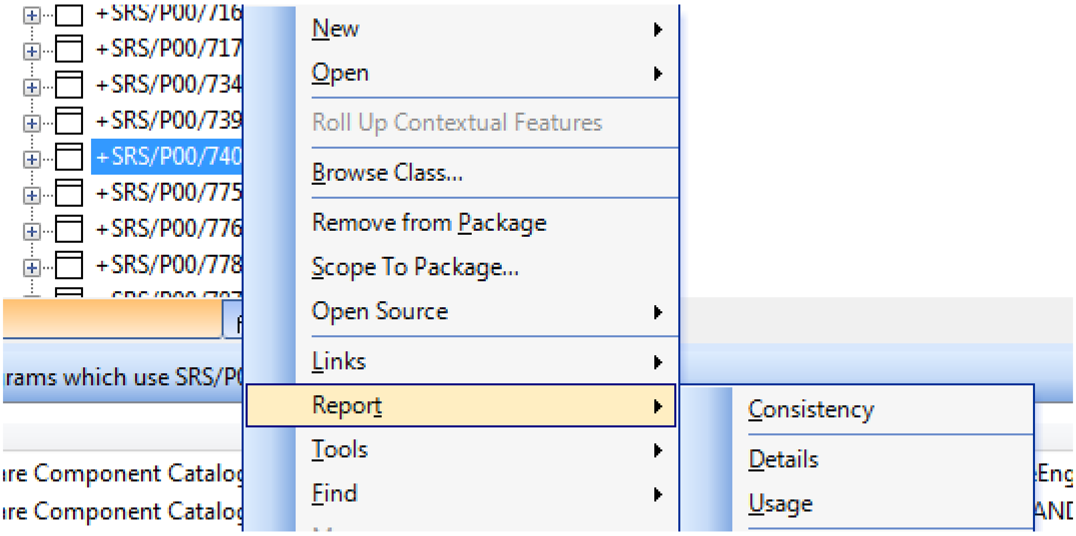
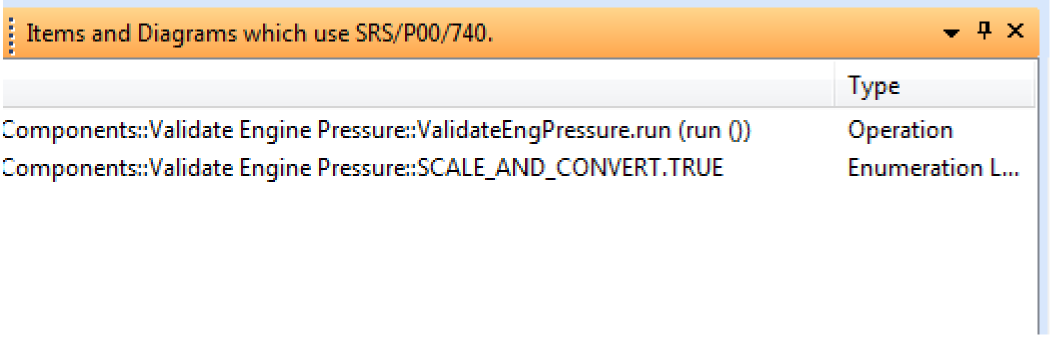


FIGURE 126 REPORT->USAGE ON REQUIREMENT TAGS

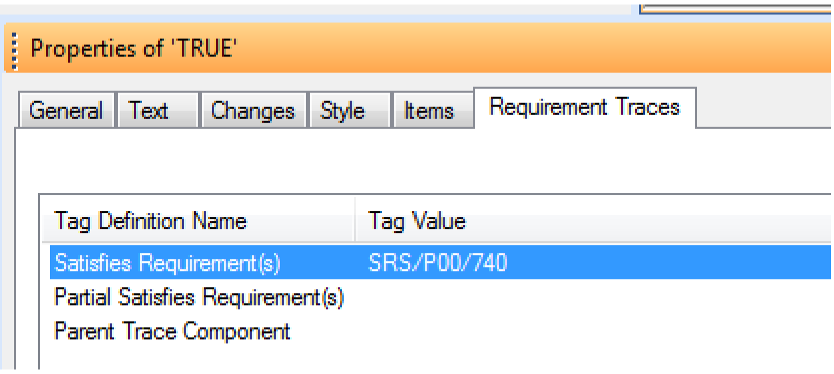
The result of running this query is shown in Figure 127.


Items and Diagrams which use SRS/P00/740.

	Type
Components::Validate Engine Pressure::ValidateEngPressure.run (run ())	Operation
Components::Validate Engine Pressure::SCALE_AND_CONVERT.TRUE	Enumeration L...

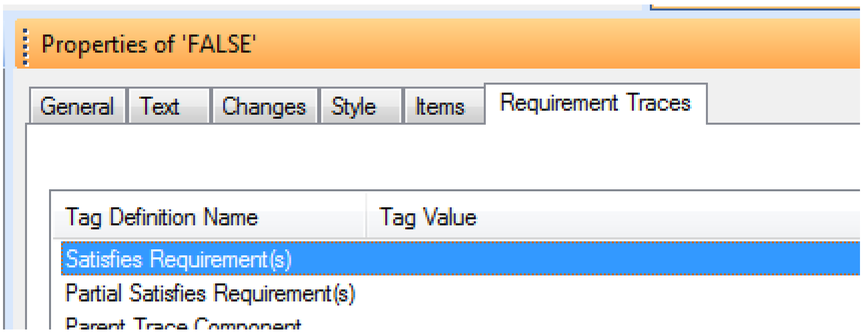
FIGURE 127 RESULT OF REQUIREMENT USAGE QUERY

Here we see that requirement P00/740 is traced in the implementing “run” operation **and** in the decision option SCALE_AND_CONVERT = “True”. This is an example of the mapping of decision options to the requirements they satisfy as described in section 4.8.1 of this thesis. We can see how this is captured in the model in Figure 128, and how the alternative “False” option has no traceability (in Figure 129), as this option results in a simple non-inclusion of the functionality.


Properties of 'TRUE'

Tag Definition Name	Tag Value
Satisfies Requirement(s)	SRS/P00/740
Partial Satisfies Requirement(s)	
Parent Trace Component	

FIGURE 128 REQUIREMENTS TRACEABILITY IN DECISION SCALE_AND_CONVERT OPTION "TRUE"



Properties of 'FALSE'

Tag Definition Name	Tag Value
Satisfies Requirement(s)	
Partial Satisfies Requirement(s)	
Parent Trace Component	

FIGURE 129 REQUIREMENTS TRACEABILITY IN DECISION SCALE_AND_CONVERT OPTION "FALSE"

We will see how this affects the final traceability reporting for the deployed component later in this section. We now look at the other source of variability in this component.

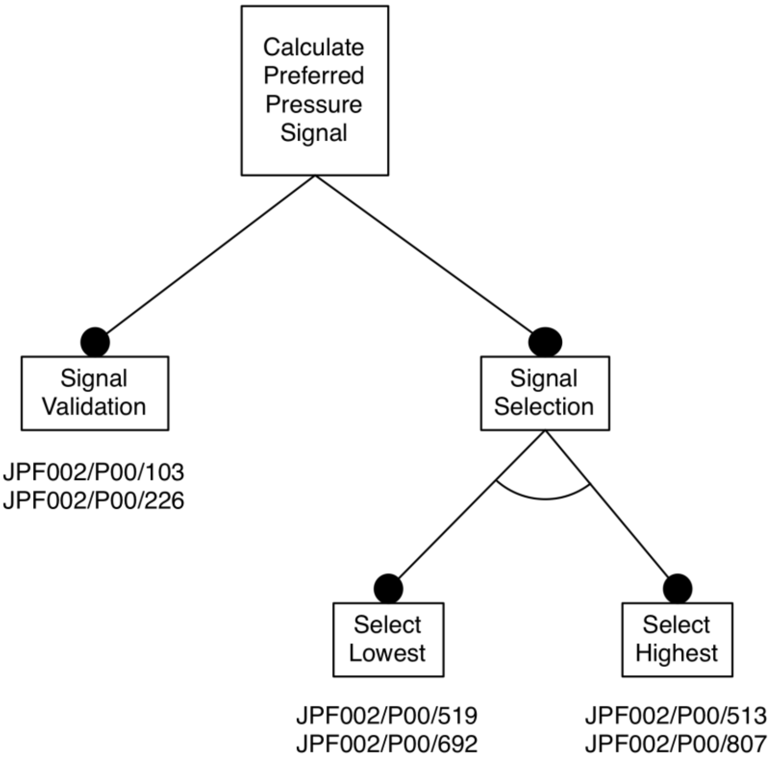
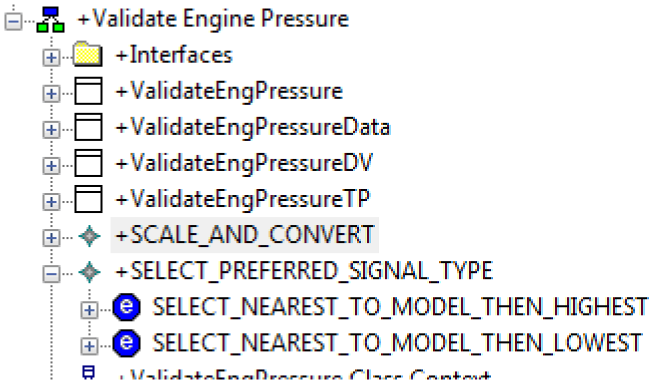


FIGURE 130 FEATURE MODEL FRAGMENT - PRESSURE SIGNAL SELECTION

Figure 130 shows another feature model fragment for the engine pressure validation, which requires variability in the signal selection logic. In essence, there are multiple sources of the engine pressure signal, which need to be validated and then selected between. Where there are multiple valid signals, a means of arbitrating between them is required. One option is to select the lowest of the valid signals, and another is to select the highest. This has been identified as a point of variability in the product line, so the core asset component has to provide these options, and make the selection visible in the component's decision contract. Again, the text under the optional features is the traceability reference for the associated high-level software requirements.

The selection of this variability in the component is via a decision named “SELECT_PREFERRED_SIGNAL_TYPE”, as can be seen in Figure 131.



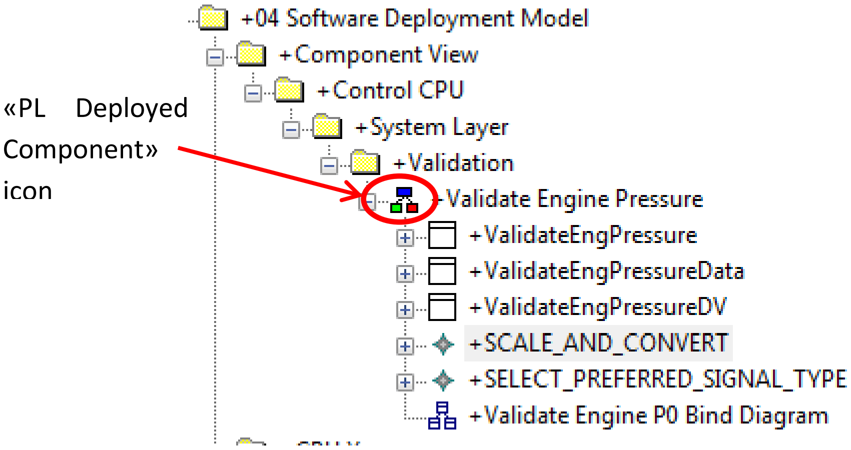
```
+Validate Engine Pressure
    +Interfaces
    +ValidateEngPressure
    +ValidateEngPressureData
    +ValidateEngPressureDV
    +ValidateEngPressureTP
    +SCALE_AND_CONVERT
    +SELECT_PREFERRED_SIGNAL_TYPE
        SELECT_NEAREST_TO_MODEL_THEN_HIGHEST
        SELECT_NEAREST_TO_MODEL_THEN_LOWEST
    +ValidateEngPressure Class Context
```

Figure 131 Signal Selection Options

(Note: the “nearest to model” part of the options shown in Figure 131 is there for consistency with other validation components that have additional options related to modelled (i.e. calculated) values as part of their selection scheme.)

Component Deployment

We have seen how core asset components can be designed and implemented to include variability, and be traced to common and optional software requirements. We now look at how these components can be deployed, instantiated and used on projects.



```
+04 Software Deployment Model
    +Component View
        +Control CPU
            +System Layer
                +Validation
                    +Validate Engine Pressure
                        +ValidateEngPressure
                        +ValidateEngPressureData
                        +ValidateEngPressureDV
                        +SCALE_AND_CONVERT
                        +SELECT_PREFERRED_SIGNAL_TYPE
                        +Validate Engine P0 Bind Diagram
```

«PL Deployed Component» icon

Figure 132 Deployed Component in Package Structure

Figure 132 shows the deployed component in the context of the deployment package structure. The deployed component icon has blue, green and red elements as can be seen, and contains a “bind diagram” which illustrates the model dependency between the deployed component and the product line component, as can be seen in Figure 133.

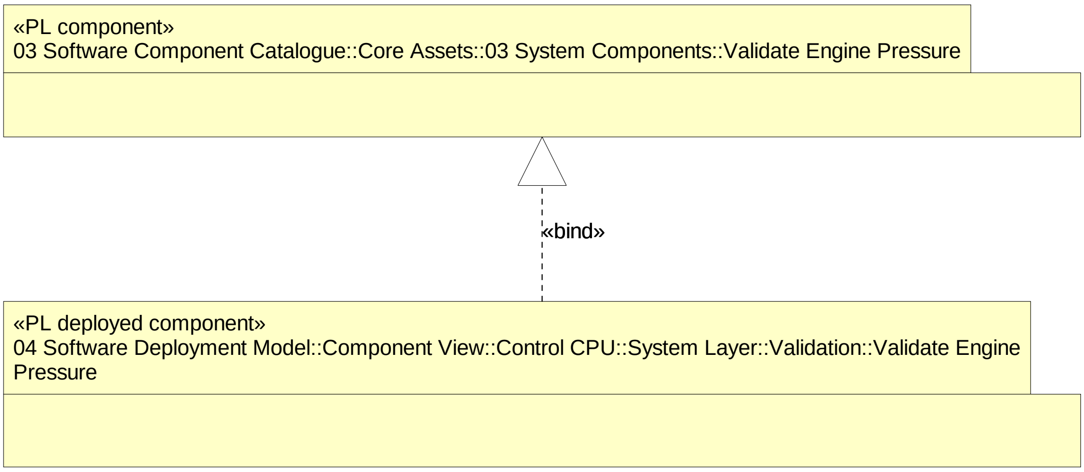
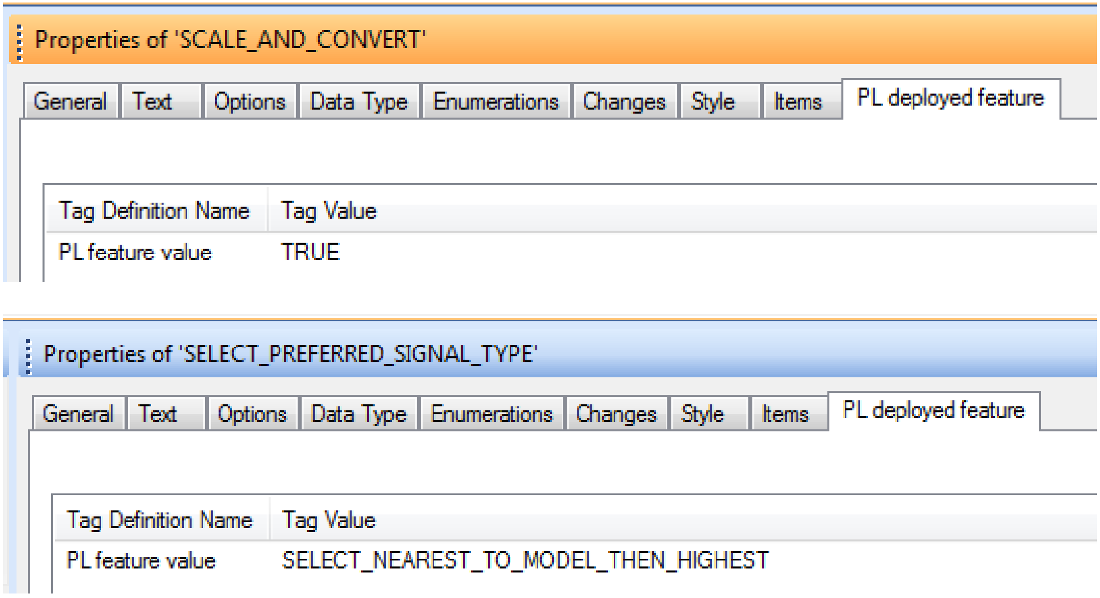

«PL component»
03 Software Component Catalogue::Core Assets::03 System Components::Validate Engine Pressure
«bind»
«PL deployed component»
04 Software Deployment Model::Component View::Control CPU::System Layer::Validation::Validate Engine Pressure

Figure 133 Deployed Component Bind Diagram

This bind dependency provides a means of modelling the relationship between the deployed and the catalogue components, compliant with the meta-model given in section 4.6. It is also a directive to the code generator to traverse the link and generate code from the dependee component.

The deployed component is obliged to resolve all the decisions in the component contract. Each decision in the catalogue component has equivalent in the deployed component, stereotyped as «PL Deployed Feature». An associated tag in this stereotype "PL Feature Value" contains the selected option for that feature, as can be seen in Figure 134.



Properties of 'SCALE_AND_CONVERT'

Tag Definition Name	Tag Value
PL feature value	TRUE

Properties of 'SELECT_PREFERRED_SIGNAL_TYPE'

Tag Definition Name	Tag Value
PL feature value	SELECT_NEAREST_TO_MODEL_THEN_HIGHEST

Figure 134 Resolution of Decision Contracts - Selected Options

Here we see that this project has made the following selections for this deployment:

- SCALE _AND_CONVERT:=TRUE
- SELECT_PREFERRED_SIGNAL_TYPE:=SELECT_NEAREST_TO_MODEL_THEN_HIGHEST

We are now in the position to instantiate the component for the project, by performing the Model-to-Model and Model-to-Text transformations as described in Chapter 5 and Appendix B of this thesis. The following listing shows the log file produced when the Product Line code generator is run on the deployed Validate Engine Pressures component.

```
Starting ACS/TDK...
- Shadow ACS/TDK kit v. 7.0.36
- Loaded [M2M_SPL_SPARK_OCS.dll] code generator
- Root Object(s):
- 04 Software Deployment Model::Component View::Control CPU::System
Layer::Validation [Category]
- Forced Generate. Processing...
- Class Needed : ValidateEngPressure
- Class NOT Needed : ValidateEngPressureData
- Class NOT Needed : ValidateEngPressureDV
- Class Needed : ValidateEngPressureTP
- Class Needed : FaultTypes
- Class Needed : IAcSimulatedPressures
- Class Needed : IAircraftState
- Class Needed : IAirData
- Class Needed : IAirDataEngine
- Class Needed : IChannel
- Class Needed : IEngineState
- Class Needed : IInteractiveMaint
- Class Needed : IOSPressures
- Class Needed : IOtherOSPressures
- Class Needed : ISAV
- Duplicating Class ValidateEngPressure into Validate Engine Pressure
- Duplicating Operation initialise into Class ValidateEngPressure
- Duplicating Operation run into Class ValidateEngPressure
- Duplicating Operation conditionEngPressure into Class
ValidateEngPressure
- Duplicating Class ValidateEngPressureTP into Validate Engine Pressure
- Duplicating Operation engP0AutoFltReset into Class
ValidateEngPressureTP
- Duplicating Operation engP0Pref into Class ValidateEngPressureTP
- Duplicating Operation engP0Corr into Class ValidateEngPressureTP
- Duplicating Operation engP0CorrFactor into Class ValidateEngPressureTP
- Duplicating Operation engP0SelRawOwn into Class ValidateEngPressureTP
- Duplicating Operation engP0SelRawOth into Class ValidateEngPressureTP
- Duplicating Operation engP0SelRawFltOwn into Class
ValidateEngPressureTP
- Duplicating Operation engP0SelRawFltOth into Class
ValidateEngPressureTP
- Parsing attribute ValidateEngPressure.state
- ...parse complete
- Parsing attribute ValidateEngPressure.state
- ...parse complete
- Parsing attribute ValidateEngPressure.xcStatusData
- ...parse complete
- Parsing attribute ValidateEngPressure.autoFltResetTimer
- ...parse complete
- Parsing attribute ValidateEngPressure.selHistFltData
- ...parse complete
- Parsing attribute ValidateEngPressure.firstPass
- ...parse complete
- Parsing attribute ValidateEngPressure.engPressureCorrRawOwn
- ...parse complete
- Parsing attribute ValidateEngPressure.engPressureCorrRawOth
- ...parse complete
- Parsing operation...ValidateEngPressure.initialise
- ...parse complete
- Parsing operation...ValidateEngPressure.initialise
- ...parse complete
- Parsing operation...ValidateEngPressure.initialise
- ...parse complete
- Parsing operation...ValidateEngPressure.initialise
- ...parse complete
- Parsing operation...ValidateEngPressure.initialise
- ...parse complete
- Parsing operation...ValidateEngPressure.initialise
- ...parse complete
- Parsing operation...ValidateEngPressure.run
- ...parse complete
- Parsing operation...ValidateEngPressure.run
- ...parse complete
- Parsing operation...ValidateEngPressure.run
- ...parse complete
- Parsing operation...ValidateEngPressure.run
- ...parse complete
- Parsing operation...ValidateEngPressure.run
- ...parse complete
- Parsing operation...ValidateEngPressure.run
- ...parse complete
- Parsing operation...ValidateEngPressure.conditionEngPressure
- ...parse complete
- Parsing operation...ValidateEngPressure.conditionEngPressure
- ...parse complete
- Parsing operation...ValidateEngPressure.conditionEngPressure
- ...parse complete
- Parsing operation...ValidateEngPressure.conditionEngPressure
- ...parse complete
- Parsing operation...ValidateEngPressure.conditionEngPressure
- ...parse complete
- Parsing operation...ValidateEngPressure.conditionEngPressure

... TRUNCATED LISTING ...

- Parsing operation...ValidateEngPressureTP.engP0SelRawFltOth
- ...parse complete
- Reordering Operations for ValidateEngPressure
- Reordering Operations for ValidateEngPressureTP
- Applying DV Pattern to ValidateEngPressureDV
- Applying Testpoint Pattern to ValidateEngPressureTP
- Applying Testport Pattern to ValidateEngPressure
- Applying Testport Pattern to ValidateEngPressureTP
- Applying Testport Pattern to ValidateEngPressureDV
- Applying Testport Pattern to ValidateEngPressureData
- Applying GData Pattern to ValidateEngPressureData
- Generation Start
- Standard Generation
- Generation End
- Generated.
```

We can see here the various states of the code generation process as described in Chapter 5 and Appendix B. We start with the selection and duplication of the required UML classes, we then deal with the opaque behaviour by parsing the text regions of the operations, and finally (for M2M) we apply the design patterns to those classes that identified as requiring expansion. This transformed model is then passed onto the M2T transformation to produce the matching source code files.

The following fragment of code from the generated version of ValidateEnginePressures.run shows the result of selecting SCALE_AND_CONVERT=TRUE in the deployment model (Compare to the mark-up core asset code in Figure 125).

```
---------- ACQUIRE & CONDITION -----------------------------

-- Condition engine pressure signals
conditionEngPressure (selRawFltOwn => lclEngP0SelRawFltOwn,
                selRawFltOth => lclEngP0SelRawFltOth);


-- JPF002/0054/0740
lclRawVOwn := engPressureCorrRawOwn;
lclRawVOth := engPressureCorrRawOth;


-- JPF002/0054/524
-- Detect if the auto fault reset is confirmed
lclAutoReset := Timers.isConfirmed (data => autoFltResetTimer);

-- Update auto fault reset testpoint
ValidateEngPressureTP.engP0AutoFltReset ( data => lclAutoReset );
```

Changing the SCALE_AND_CONVERT selected option to FALSE and re-running the code generation results in the following code fragment for ValidateEnginePressures.run to be produced:

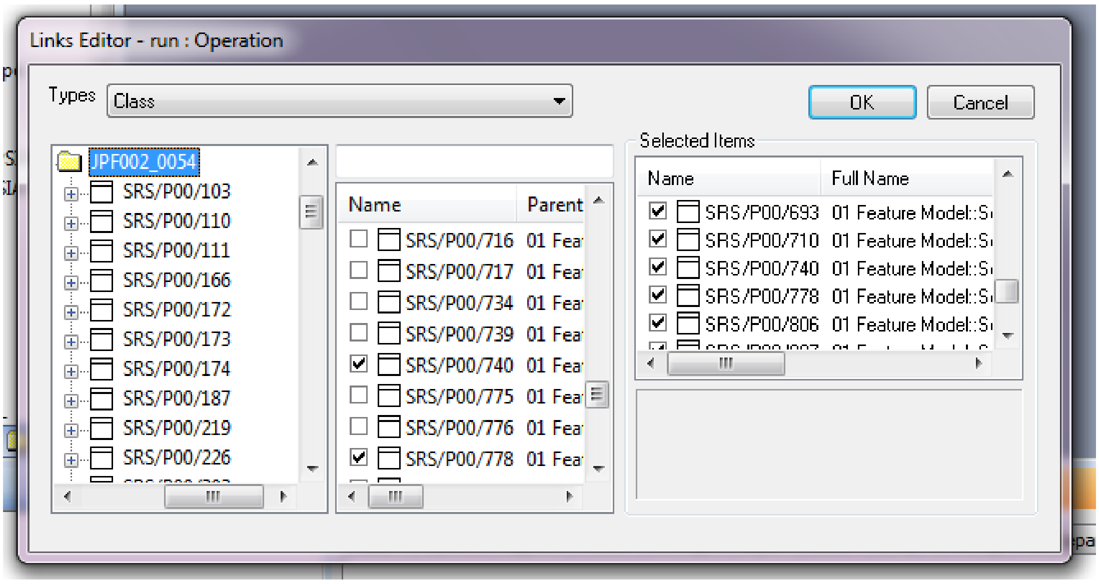
```
---------- ACQUIRE & CONDITION -----------------------------

-- Condition engine pressure signals
conditionEngPressure (selRawFltOwn => lclEngP0SelRawFltOwn,
                selRawFltOth => lclEngP0SelRawFltOth);


lclRawVOwn := IOSPressures.Get.p0Raw.data;
lclRawVOth := IOtherOSPressures.Get.p0Raw.data;


-- JPF002/0054/524
-- Detect if the auto fault reset is confirmed
lclAutoReset := Timers.isConfirmed (data => autoFltResetTimer);

-- Update auto fault reset testpoint
ValidateEngPressureTP.engP0AutoFltReset ( data => lclAutoReset );
```

Comparing the two code fragments shows the effect of selecting/deselecting that option.

Traceability

We discussed the importance of traceability in section 4.8.1 of this thesis. The design and implementation artefacts trace up to their parent requirements to demonstrate that all requirements have been met and there is no unintended functionality.

The traceability provided in the product line components is a superset; each operation traces to its full set of parent requirements. This is irrespective of whether they are common or variable requirements. Figure 135 illustrates how this linkage is performed in the ARTiSAN Studio tool via the “links editor”.



Types: Class

JPF002_0054
SRS/P00/103
SRS/P00/110
SRS/P00/111
SRS/P00/166
SRS/P00/172
SRS/P00/173
SRS/P00/174
SRS/P00/187
SRS/P00/219
SRS/P00/226

	Name	Parent
☐	SRS/P00/716	01 Fea
☐	SRS/P00/717	01 Fea
☐	SRS/P00/734	01 Fea
☐	SRS/P00/739	01 Fea
☑	SRS/P00/740	01 Fea
☐	SRS/P00/775	01 Fea
☐	SRS/P00/776	01 Fea
☑	SRS/P00/778	01 Fea

	Name	Full Name
☑	SRS/P00/693	01 Feature Model::S
☑	SRS/P00/710	01 Feature Model::S
☑	SRS/P00/740	01 Feature Model::S
☑	SRS/P00/778	01 Feature Model::S
☑	SRS/P00/806	01 Feature Model::S

Figure 135 Links Editor Showing Selection of Requirement Traceability for Run Operation

When a component is deployed, however, the traceability needs to reflect only those requirements that are relevant to the particular product options selected. The traceability data for a deployed component is reported by a special traceability extractor tool. This tool implements the algorithm described in section 4.8.2, and constructs the requirement list that is specific to the options selected in the component deployment.

If we look at the set of requirements represented by the feature model fragments in Figure 121 and Figure 130, we can see that the run operation for the product line component must implement the following set of requirements:

- JPF002/P00/103
- JPF002/P00/226
- *JPF002/P00/513*
- *JPF002/P00/519*
- JPF002/P00/523
- JPF002/P00/524
- *JPF002/P00/692*
- JPF002/P00/523
- *JPF002/P00/740*
- *JPF002/P00/807*

The requirement tags in *italics* refer to requirements that are variation points in the high-level requirements suite. However, this is not indicated in the operation traceability links. Instead, as we saw in Figure 128 and Figure 129, the options in the component decision contracts also trace (link) to the requirements satisfied by those options. Using this information, the traceability extraction tools can report the correct set of traceability data for the deployed component.

Table 7 Traceability for Selected Component Options

PL Component “Run” Operation Traces to (Set R)	SCALE_AND_CONVERT Option Traces (Set S)		SELECT_PREFFERED_SIGNAL_TYPE Option Traces (Set P)	
	FALSE (Set SF)	TRUE (Set ST)	SELECT_NEAREST_TO_ MODEL_THEN_HIGHEST (Set PH)	SELECT_NEAREST_TO_ MODEL_THEN_LOWEST (Set PL)
JPF002/P00/103 JPF002/P00/226 JPF002/P00/513 JPF002/P00/519 JPF002/P00/523 JPF002/P00/524 JPF002/P00/692 JPF002/P00/523 JPF002/P00/740 JPF002/P00/807		JPF002/P00/740	JPF002/P00/513 JPF002/P00/807	JPF002/P00/519 JPF002/P00/692

Given the set of traceability data in Table 7, we can see that the deployed traceability for the selected options in the deployed component (SCALE_AND_CONVERT=TRUE, SELECT_PREFERRED_SIGNAL_TYPE= SELECT_NEAREST_TO_MODEL_THEN_HIGHEST) is:

- JPF002/P00/103
- JPF002/P00/226
- *JPF002/P00/513*
- JPF002/P00/523
- JPF002/P00/524
- JPF002/P00/523
- *JPF002/P00/740*
- *JPF002/P00/807*

where the colours relate to the selected options, and the normal typeface set are the common requirements.

As outlined in section 4.8.2, the traceability reporter tool constructs this list by firstly creating the set of common requirements by subtracting the total set of option traces from the set traced by the product line operation (with reference back to Table 7):

$$\text{Common Requirements for "run" \{Set CR\}} = \text{\{Set R\}} - (\text{\{Set S\}} \cup \text{\{Set P\}})$$

This yields a set of common requirements “CR” for the “run” operation of:

- JPF002/P00/103
- JPF002/P00/226
- JPF002/P00/523
- JPF002/P00/524
- JPF002/P00/523

We then add in the selected optional requirements that are **relevant** to the run operation:

$$\text{Optional Requirements OR} = \text{\{Set R\}} \cap (\text{\{Set ST\}} \cup \text{\{Set PH\}})$$

This yields a set of optional requirements “OR” for the “run” operation of

- JPF002/P00/513
- JPF002/P00/740
- JPF002/P00/807

The complete traceability for the deployed run operation is the combination of CR and OR as listed earlier.

Whilst this may seem an overly complex approach for the simple example shown, it is scalable to any level of complexity that is compliant with the decision contract meta-model, in particular:

- Traceability that exists in more than one option, but not all options

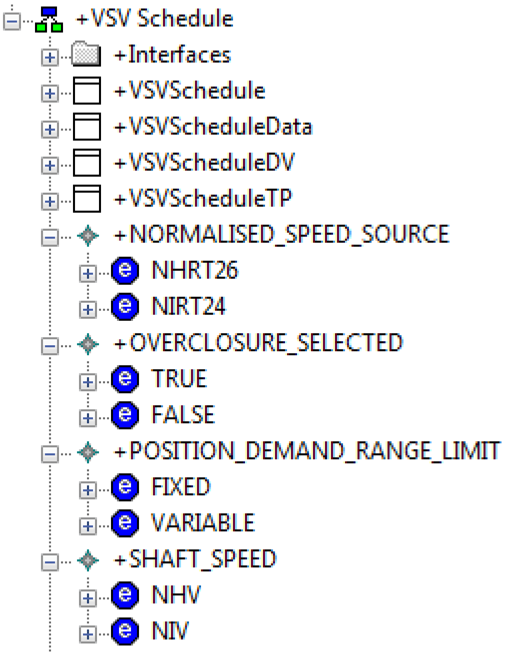
- Traceability that links to part of a component only (i.e. not all operations)

This example has demonstrated the fundamental model structure for core asset components and their deployment, using a very simple example component. We now illustrate the development and deployment of more complex components within this infrastructure.

Core Asset Component with UML Element Variability

The previous example showed a very simple usage of our product lines approach; the only variability in this component was text substitution in the code body. We now consider a more complex component with a greater number of entries in the component decision contract. The number of related variation points is also greater than in the previous example, and the types of element affected cover both text and UML model elements. In the interests of brevity, we will restrict the discussion in this section to the manipulation of UML model elements, as the previous example adequately dealt with text substitution.

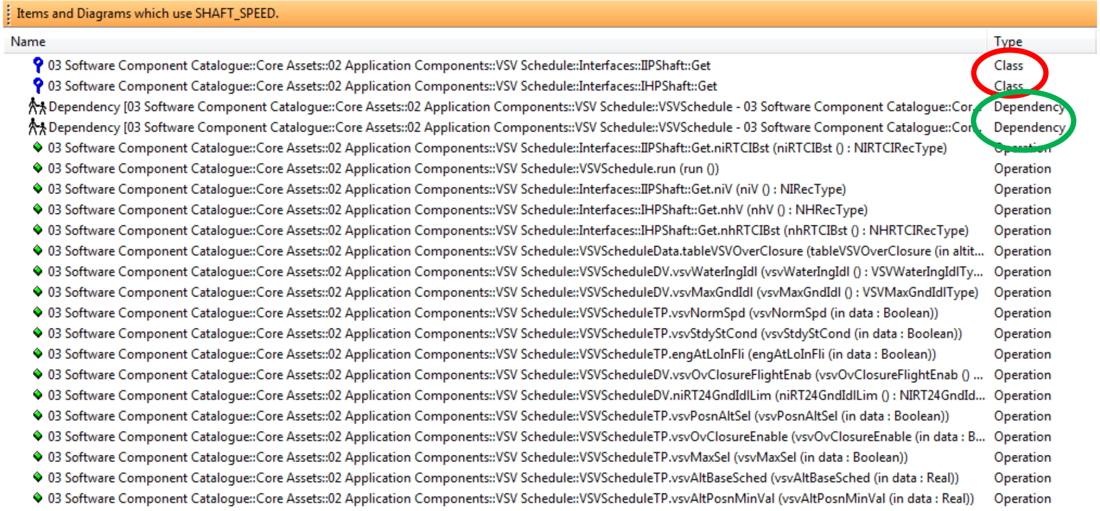
This example concerns the “scheduling” of the engine Variable Stator Vane (VSV) system. Figure 136 shows the component structure for the “VSV Schedule” core asset component, with the set of decisions in the component decision contract expanded to show the available options.



- +VSV Schedule
  - +Interfaces
  - +VSVSchedule
  - +VSVScheduleData
  - +VSVScheduleDV
  - +VSVScheduleTP
  - +NORMALISED_SPEED_SOURCE
    - NHRT26
    - NIRT24
  - +OVERCLOSURE_SELECTED
    - TRUE
    - FALSE
  - +POSITION_DEMAND_RANGE_LIMIT
    - FIXED
    - VARIABLE
  - +SHAFT_SPEED
    - NHV
    - NIV

FIGURE 136 DECISION CONTRACT FOR VSV SCHEDULE COMPONENT

We can illustrate the increased complexity of this component compared to the previous example by reporting the usage of the SHAFT_SPEED decision. Figure 137 shows the set of UML model elements that are affected in some way by the SHAFT_SPEED decision. We can see that, in addition to operations, there is an impact on class and dependency model types.

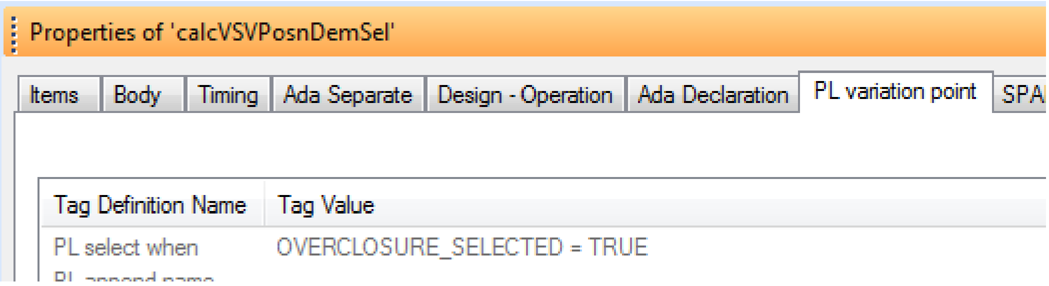


Items and Diagrams which use SHAFT_SPEED.

Name	Type
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::Interfaces::IIPShaft::Get	Class
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::Interfaces::IHPShaft::Get	Class
Dependency [03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVSchedule - 03 Software Component Catalogue::Cor...	Dependency
Dependency [03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVSchedule - 03 Software Component Catalogue::Cor...	Dependency
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::Interfaces::IIPShaft::Get.niRTCIBst (niRTCIBst () : NIRTCIRecType)	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVSchedule.run (run ())	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::Interfaces::IIPShaft::Get.niV (niV () : NIRecType)	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::Interfaces::IHPShaft::Get.nhV (nhV () : NHRecType)	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::Interfaces::IHPShaft::Get.nhRTCIBst (nhRTCIBst () : NHRTCIRecType)	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleData.tableVSVOverClosure (tableVSVOverClosure (in altit...	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleDV.vsvWaterIngIdl (vsvWaterIngIdl () : VSVWaterIngIdlTy...	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleDV.vsvMaxGndIdl (vsvMaxGndIdl () : VSVMaxGndIdlType)	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleTP.vsvNormSpd (vsvNormSpd (in data : Boolean))	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleTP.vsvStdyStCond (vsvStdyStCond (in data : Boolean))	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleTP.engAtLoInFli (engAtLoInFli (in data : Boolean))	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleDV.vsvOvClosureFlightEnab (vsvOvClosureFlightEnab () ...	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleDV.niRT24GndIdlLim (niRT24GndIdlLim () : NIRT24GndId...	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleTP.vsvPosnAltSel (vsvPosnAltSel (in data : Boolean))	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleTP.vsvOvClosureEnable (vsvOvClosureEnable (in data : B...	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleTP.vsvMaxSel (vsvMaxSel (in data : Boolean))	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleTP.vsvAltBaseSched (vsvAltBaseSched (in data : Real))	Operation
03 Software Component Catalogue::Core Assets::02 Application Components::VSV Schedule::VSVScheduleTP.vsvAltPosnMinVal (vsvAltPosnMinVal (in data : Real))	Operation

FIGURE 137 REPORTING USAGE OF SHAFT_SPEED DECISION – CLASSES AND DEPENDENCIES HIGHLIGHTED

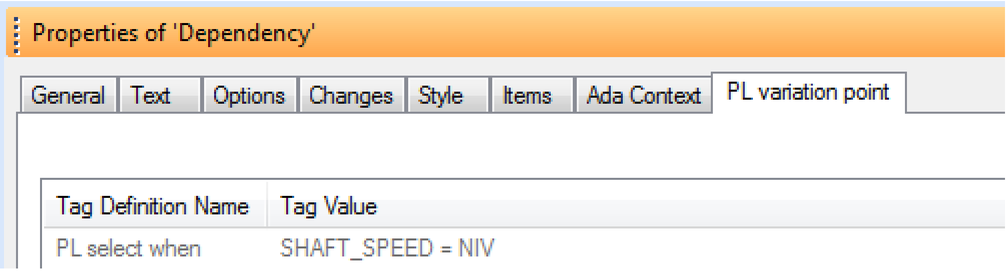
The relationship of the decision to the classes and dependencies are always include/exclude relationships via a “select when” expression. The relationship to operations may be either within mark-up text of operation bodies (as seen previously) or can also be include/exclude via “select when”. Figure 138 illustrates how an entire operation is identified as a point of variation, and the text of the “select when” expression contained within the variation point definition.


Properties of 'calcVSVPosnDemSel'

Tag Definition Name	Tag Value
PL select when	OVERCLOSURE_SELECTED = TRUE
PL append name	

FIGURE 138 UML OPERATION VARIATION

Similarly, Figure 139 shows the definition of a UML dependency as a point of variation and the associated “select when” expression.


Properties of 'Dependency'

Tag Definition Name	Tag Value
PL select when	SHAFT_SPEED = NIV

FIGURE 139 UML DEPENDENCY VARIATION

Component Deployment & Code Generation

The project deployment of this component makes the following selections:

- NORMALISED_SPEED_SOURCE := NIRT26
- OVERCLOSURE_SELECTED := TRUE
- POSITION_DEMAND_RANGE_LIMIT := VARIABLE
- SHAFT_SPEED := NIV

The following log file is generated when running the product line code generation process with the above set of options selected:

```
- Shadow ACS/TDK kit v. 7.0.36
  Saving model ProjectX... Saved.
  Model saved to model cache
- Forced Generate. Processing...
- Class Needed : VSVSchedule
- Class NOT Needed : VSVScheduleData
- Class NOT Needed : VSVScheduleDV
- Class Needed : VSVScheduleTP
- Class Needed : IAcThrustSettings
- Class Needed : IAircraftState
- Class Needed : ICompressorAir
- Class Needed : IEngineEvents
- Class Needed : IEngineState
- Class Needed : IHPShaft
- Class Needed : IInteractiveMaint
- Class Needed : IIPShaft
- Class Needed : ILPShaft
- Class Needed : IThrustSettings
- Class Needed : IVSV
- Duplicating Class VSVSchedule into VSV Schedule
- VP Selected: Expression : OVERCLOSURE_SELECTED = TRUE Evaluated TRUE
: calcVSVPosnAltSel
- Duplicating Operation calcVSVPosnAltSel into Class VSVSchedule
- VP Selected: Expression : POSITION_DEMAND_RANGE_LIMIT = VARIABLE
Evaluated TRUE   : calcVSVMaxSelVal
- Duplicating Operation calcVSVMaxSelVal into Class VSVSchedule
- VP Selected: Expression : OVERCLOSURE_SELECTED = TRUE Evaluated TRUE
: calcVSVPosnDemSel
- Duplicating Operation calcVSVPosnDemSel into Class VSVSchedule
- VP Selected: Expression : SHAFT_SPEED = NIV Evaluated TRUE :
Dependency
- VP NOT Selected: Expression : SHAFT_SPEED = NHV Evaluated FALSE :
Dependency
- Duplicating Operation calcPhaseAdvSpd into Class VSVSchedule
- Duplicating Operation calcShaftSpddotOverP30Filt into Class VSVSchedule
- Duplicating Operation calcVSVShaftSpdReset into Class VSVSchedule
- Duplicating Operation calcVSVSteadyStateDem into Class VSVSchedule
- Duplicating Operation calcVSVPosnResSumNLRT20 into Class VSVSchedule
- Duplicating Operation calcVSVSurgeReset into Class VSVSchedule
- Duplicating Operation calcVSVAccelResetMinVal into Class VSVSchedule
- Duplicating Operation calcVSVReverseReset into Class VSVSchedule
- Duplicating Operation calcVSVDecelReset into Class VSVSchedule
- Duplicating Operation calcVSVPosnDemLim into Class VSVSchedule
- Duplicating Operation calcVSVPosnDemDemDot into Class VSVSchedule
- Duplicating Operation calcVSVResetUnLim into Class VSVSchedule
- Duplicating Operation initialise into Class VSVSchedule
- Duplicating Operation run into Class VSVSchedule
- Duplicating Class VSVScheduleTP into VSV Schedule
- VP Selected: Expression : OVERCLOSURE_SELECTED = TRUE Evaluated TRUE
: engAtLoInFli
- Duplicating Operation engAtLoInFli into Class VSVScheduleTP
- VP Selected: Expression : OVERCLOSURE_SELECTED = TRUE Evaluated TRUE
: vsvAltBaseSched
- Duplicating Operation vsvAltBaseSched into Class VSVScheduleTP
- VP Selected: Expression : OVERCLOSURE_SELECTED = TRUE Evaluated TRUE
: vsvAltPosnMinVal
- Duplicating Operation vsvAltPosnMinVal into Class VSVScheduleTP
- VP Selected: Expression : POSITION_DEMAND_RANGE_LIMIT = VARIABLE
Evaluated TRUE   : vsvMaxSel
- Duplicating Operation vsvMaxSel into Class VSVScheduleTP
- VP Selected: Expression : POSITION_DEMAND_RANGE_LIMIT = VARIABLE
Evaluated TRUE   : vsvNormSpd
- Duplicating Operation vsvNormSpd into Class VSVScheduleTP
- VP Selected: Expression : OVERCLOSURE_SELECTED = TRUE Evaluated TRUE
: vsvOvClosureEnable
- Duplicating Operation vsvOvClosureEnable into Class VSVScheduleTP
- VP Selected: Expression : OVERCLOSURE_SELECTED = TRUE Evaluated TRUE
: vsvPosnAltSel
- Duplicating Operation vsvPosnAltSel into Class VSVScheduleTP
- VP Selected: Expression : OVERCLOSURE_SELECTED = TRUE Evaluated TRUE
: vsvPosnDemOvClosure
- Duplicating Operation vsvPosnDemOvClosure into Class VSVScheduleTP
- VP Selected: Expression : POSITION_DEMAND_RANGE_LIMIT = VARIABLE
Evaluated TRUE   : vsvStdyStCond
- Duplicating Operation vsvStdyStCond into Class VSVScheduleTP
- Duplicating Operation accelTLShaft into Class VSVScheduleTP
- Duplicating Operation phaseAdvSpd into Class VSVScheduleTP

... TRUNCATED LISTING ...
```

```
- Duplicating Operation vsvTransFtr into Class VSVScheduleTP
- Duplicating Operation vsvTransOffsetBasic into Class VSVScheduleTP
- Parsing operation...VSVSchedule.calcVSVPosnAltSel
- ...parse complete

... TRUNCATED LISTING ...

- Parsing operation...VSVScheduleTP.vsvTransOffsetBasic
- ...parse complete
- Reordering Operations for VSVSchedule
- Reordering Operations for VSVScheduleTP
- Applying DV Pattern to VSVScheduleDV
- Applying Testpoint Pattern to VSVScheduleTP
- Applying Testport Pattern to VSVSchedule
- Applying Testport Pattern to VSVScheduleTP
- Applying Testport Pattern to VSVScheduleDV
- Applying Testport Pattern to VSVScheduleData
- Applying GData Pattern to VSVScheduleData
- Generation Start
- Standard Generation
- Generation End
- Generated.
```

As with the previous example, we can see the phases of the model transformation process taking place. This example, however, contains variability that results in the transformation of model elements, not just the processing of marked-up text. This can be seen in the log file above where the transformation process reports “VP Selected” and “VP NOT Selected” and the results of the processing of the “select when” expressions. For example, consider the following log file fragment

```
-  VP  Selected:  Expression  :  SHAFT_SPEED  =  NIV  Evaluated  TRUE     :
Dependency
-  VP  NOT  Selected:  Expression  :  SHAFT_SPEED  =  NHV  Evaluated  FALSE  :
Dependency
```

This illustrates that the core asset contains two dependencies that are mutually exclusive. The selection of the SHAFT_SPEED:=NIV in the deployed component resulted in the inclusion of one dependency and the removal of the other.

Conclusions

In this appendix, we have provided practical examples of how our component construction and deployment approach, and the associated code generation process, can be used to develop real-world software. These examples necessarily provide just a small glimpse of the full system, which contains the hundreds of core asset and project specific components required to implement modern gas turbine engine control system software.

Glossary

ACS	Automatic Code Synchronisation
AOHE	Air/Oil Heat Exchanger
AS	Application Software
ATAM	Architecture Trade-off Analysis Method
ATL	Atlas Transformation Language
BAPO	Business, Architecture, Process, Organisation
CAA	Civil Aviation Authority
CVL	Common Variability Language
DAL	Development Assurance Level
DSL	Domain Specific Language
EASA	European Aviation Safety Agency
EEC	Engine Electronic Controller
EIS	Entry into Service
EUROCAE	European Organisation for Civil Aviation Equipment
FAA	Federal Aviation Administration
FADEC	Full Authority Digital Engine Control
FOHE	Fuel/Oil Heat Exchanger
FPGA	Field Programmable Gate Array
FRAC	Final Review and Comment
HAL	Hardware Abstraction Layer
HP	High Pressure
IDG	Integrated Dedicated Generator
IP	Intermediate Pressure
LP	Low Pressure
MC/DC	Modified Condition/Decision Coverage

MOF	Meta Object Facility
OCS	On-Demand Code Synchronisation
OMG	Object Management Group
OS	Operating Software
PEL	Process Engineering Language
PLUSS	Product Line Use case modelling for Systems and Software engineering
PSAC	Plan for Software Aspects of Certification
LRU	Line-Replaceable Unit
LUCOL	Lucas Control Language
RFP	Request for Proposal
RTCA	Radio Technical Commission for Aeronautics
RTOS	Real-Time Operating System
SAS	Software Accomplishment Summary
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SDL	Scripting language for Artisan Studio Model to Text code generation
SDP	Software Development Plan
SOI	Stages of Involvement
SoS	Scope of Supply
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SRS	Software Requirements Specification
SVP	Software Verification Plan
SEI	Software Engineering Institute
SPL	Software Product Line
SPLC	Software Product Line Conference
TDK	Template Development Kit

TPL	Trusted Product Line
T/R System	Thrust Reverser System
TQP	Tool Qualification Plan
UML	Unified Modelling Language
VSV	Variable Stator Vane

List of References

[1] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*: Addison-Wesley Professional, 2000.

[2] K. Pohl, G. Bockle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*: Springer-Verlag New York Inc, 2005.

[3] P. Clements and L. Northrop, *Software Product Lines*: Addison-Wesley, 2001.

[4] "DO-178B/ED-12B, Software Considerations in Airborne Systems and Equipment Certification," ed: Radio Technical Commission for Aeronautics / EUROCAE, 1992.

[5] F. Van Der Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering*: Springer-Verlag New York Inc, 2007.

[6] P. Clements, "On the Importance of Product Line Scope" in *Software Product-Family Engineering*, F. van der Linden, Ed., LNCS vol. 2290, Springer Berlin / Heidelberg, 2002, pp. 102-113.

[7] J.-M. DeBaud and K. Schmid, "A systematic approach to derive the scope of software product lines," presented at the Proceedings of the 21st international conference on Software engineering, Los Angeles, California, United States, 1999.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute," Carnegie Mellon University 1990.

[9] K. Czarnecki and U. W. Eisenecker, *Generative Programming–Methods, Tools, and Applications*: Addison Wesley, 2000.

[10] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models", *Software Process: Improvement and Practice,* vol. 10, pp. 143-169, 2005.

[11] *Eclipse Modeling Project* Last Accessed: 18th January 2013, Available: http://www.eclipse.org/modeling/

[12] UML 2.0 Superstructure Specification. Available: http://www.omg.org/cgi-bin/doc?formal/05-07-04

[13] A. H. Van de Ven, *Engaged scholarship: A guide for organizational and social research*: Oxford University Press, USA, 2007.

[14] "SPARK 95 - The SPADE Ada 95 Kernel (including RavenSPARK) V4.6," Praxis HIS 2005.

[15] D. L. Parnas, "On the design and development of program families", *IEEE Transactions on Software Engineering,* pp. 1-9, 1976.

[16] F. van der Linden, J. Bosch, E. Kamsties, K. Kansala, and H. Obbink, "Software Product Family Evaluation," in *Third Intenational Conference on Software Product Lines*, Boston MA, 2004, LNCS pp. 107-109.

[17] G. Halmans and K. Pohl, "Considering Product Family Assets when Defining Customer Requirements", *Proceedings of PLEES,* vol. 1.

[18] F. Van der Linden, "Software product families in Europe: the Esaps & Cafe projects", *IEEE Software,* pp. 41-49, 2002.

[19] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and variability in software engineering", *Software, IEEE,* vol. 15, pp. 37-45, 1998.

[20] J. van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, 2001, pp. 45-54.

[21] A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. van der Linden, "Hierarchical Variability Modeling for Software Architectures," in *15th International Software Product Lines Conference (SPLC)*, Munich, 2011.

[22] J. Liebig, S. Apel, C. Lengauer, C. Kastner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, 2010, pp. 105-114.

[23] D. Beuche, "Variants and Variability Management with pure:: variants," in *3rd Software Product Line Conference (SPLC 2004), Workshop on Software Variability Management for Product Derivation, Boston, MA*, 2004.

[24] C. W. Krueger, "BigLever software gears and the 3-tiered SPL methodology," presented at the Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, Montreal, Quebec, Canada, 2007.

[25] C. Kästner, S. Trujillo, and S. Apel, "Visualizing software product line variabilities in source code," in *Proceedings of the SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, 2008, pp. 303-313.

[26] G. Botterweck, L. O'Brien, and S. Thiel, "Model-driven derivation of product architectures," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, Atlanta, Georgia, USA, 2007.

[27] T. Ziadi, L. Hélouët, and J. M. Jézéquel, "Towards a UML profile for software product lines," presented at the PFE 2003 - 5th International Workshop on Software Product-Family Engineering, Siena, Italy, 2003.

[28] H. Gomaa, *Designing software product lines with UML: from use cases to pattern-based software architectures*: Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004.

[29] W. Pree, M. Fontoura, and B. Rumpe, "Product line annotations with UML-F" in *Software Product Lines*, G. Chastek, Ed., LNCS vol. 2379, 2002, pp. 103-148.

[30] "Common Variability Language (CVL) Request For Proposal," Object Management Group (OMG) ad/2009-12-03, 2009.

[31] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger, "XML-Based Feature Modelling" in *Software Reuse: Methods, Techniques, and Tools*, J. Bosch and C. Krueger, Eds., LNCS vol. 3107, Springer Berlin / Heidelberg, 2004, pp. 101-114.

[32] A. Polzer, S. Kowalewski, and G. Botterweck, "Applying software product line techniques in model-based embedded systems engineering," in *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES '09. ICSE Workshop on*, 2009, pp. 2-10.

[33] G. Muller and P. van de Laar, "Researching reference architectures and their relationship with frameworks, methods, techniques, and tools," in *Proceedings of the 7th annual conference on systems engineering research (CSER), Loughborough, UK*, 2009.

[34] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A feature-;oriented reuse method with domain-;specific reference architectures", *Annals of Software Engineering,* vol. 5, pp. 143-168, 1998.

[35] H. Gomaa and M. E. Shin, "Multiple-view meta-modeling of software product lines," in *Eighth IEEE International Conference on Engineering of Complex Computer Systems*, 2002, pp. 238-246.

[36] H. Gomaa and M. Shin, "A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines" in *Software Reuse: Methods, Techniques, and Tools*, J. Bosch and C. Krueger, Eds., LNCS vol. 3107, Springer Berlin / Heidelberg, 2004, pp. 274-285.

[37] O. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding Standardized Variability to Domain Specific Languages," in *Software Product Line Conference, 2008. SPLC '08. 12th International*, 2008, pp. 139-148.

[38] OMG, *Meta Object Facility (MOF) Core Specification 2.0*,Last Accessed: 1 October 2011, Available: http://www.omg.org/spec/MOF/2.0

[39] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*: Addison-Wesley Professional, 2002.

[40] R. van Ommering and J. Bosch, "Widening the Scope of Software Product Lines — From Variation to Composition" in *Software Product Lines*, G. Chastek, Ed., LNCS vol. 2379, Springer Berlin / Heidelberg, 2002, pp. 31-52.

[41] R. van Ommering, "Software reuse in product populations", *Software Engineering, IEEE Transactions on,* vol. 31, pp. 537-550, 2005.

[42] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software", *Computer,* vol. 33, pp. 78-85, 2000.

[43] C. Atkinson, J. Bayer, and D. Muthig, "Component-based product line development: The KobrA approach," in *Software product lines: experience and research directions: proceedings of the First Software Product Lines Conference (SPLC1), August 28-31, 2000, Denver, Colorado*, 2000, p. 289.

[44] H. Obbink, J. Müller, P. America, R. van Ommering, G. Muller, W. van der Sterren, and J. G. Wijnstra, "COPA: a component-oriented platform architecting method for families of software-intensive electronic products," in *Tutorial for the First Software Product Line Conference, Denver, Colorado*, 2000.

[45] J. Wijnstra, "Critical factors for a successful platform-based product family approach" in *Software Product Lines*, G. Chastek, Ed., LNCS vol. 2379, 2002, pp. 15-35.

[46] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants" in *Generative Programming and Component Engineering*, R. Glück and M. Lowry, Eds., LNCS vol. 3676, Springer Berlin / Heidelberg, 2005, pp. 422-437.

[47] F. van der Linden, "Conversation at 15th International Software Product Line Conference," S. Hutchesson, Ed., ed. Munich, 2011.

[48] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer, "Delta Modeling for Software Architectures," in *MBEES*, 2011.

[49] U. Aflmann, *Invasive software composition*: Springer-Verlag New York Inc, 2003.

[50] S. Hutchesson and J. McDermid, "Development of High-Integrity Software Product Lines Using Model Transformation," in *SAFECOMP 2010 - Computer Safety, Reliability, and Security*, Vienna, 2010, pp. 389-401.

[51] J. Perez, J. Diaz, C. Costa-Soria, and J. Garbajosa, "Plastic Partial Components: A solution to support variability in architectural components," in *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, 2009, pp. 221-230.

[52] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming", *ECOOP'97—Object-Oriented Programming,* pp. 220-242, 1997.

[53] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*: Wiley, 2009.

[54] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003, pp. 1-17.

[55] T. Mens and P. Van Gorp, "A taxonomy of model transformation," in *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)* 2006, Electronic Notes in Theoretical Computer Science pp. 125-142.

[56] A. G. Kleppe, J. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[57] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *11th International Software Product Line Conference* 2007, pp. 233-242.

[58] M. Voelter, "Using domain specific languages for product line engineering," presented at the 13th International Software Product Line Conference, San Francisco, California, 2009.

[59] Esterel, *SCADE Suite*,Last Accessed: 27th February 2011, Available: http://www.esterel-technologies.com/products/scade-suite/

[60] "DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification," ed: Radio Technical Commission for Aeronautics / EUROCAE, 2011.

[61] "DO-278A/ED-109A, Guidelines for Communications, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance," ed: Radio Technical Commission for Aeronautics / EUROCAE, 2011.

[62] "DO-248C/ED-94C, Supporting Information for DO-178C and DO-278A," ed: Radio Technical Commission for Aeronautics / EUROCAE, 2011.

[63] "DO-330/ED-215, Tool Qualification Supplement to DO-178C/ED-12C and DO-1278C/ED-109A," ed: Radio Technical Commission for Aeronautics / EUROCAE, 2011.

[64] "DO-331/ED-218,Model-Based Development and Verification Supplement to DO-178C/ED-12C and DO-278A/ED-109A," ed: Radio Technical Commission for Aeronautics / EUROCAE, 2011.

[65] "DO-332/ED-217, Object-Oriented Technologies and Associated Techniques Supplement to DO-178C/ED-12C and DO-1278C/ED-109A," ed: Radio Technical Commission for Aeronautics / EUROCAE, 2011.

[66] "DO-333/ED-216, Formal Methods Supplement to DO-178C/ED-12C and DO-1278C/ED-109A," ed: Radio Technical Commission for Aeronautics / EUROCAE, 2011.

[67] Mathworks, *Matlab Simulink/Stateflow tools*,Last Accessed: 1 October 2012, Available: http://www.mathworks.co.uk/

[68] C. Moler, *The Origins of Matlab*,Last Accessed, Available: http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html

[69] Mathworks, *Simulink - Simulation and Model-Based Design*,Last Accessed: 18 January 2013, Available: http://www.mathworks.com/products/simulink/

[70] Mathworks, *Stateflow*,Last Accessed: 18 January 2013, Available: http://www.mathworks.com/products/stateflow/

[71] B. Dion, "Correct-By-Construction Methods for the Development of Safety-Critical Applications," in *SAE 2004 World Congress & Exhibition, Session: Safety-Critical Systems* Detroit MI, 2004.

[72] Esterel, "Simulink Users Connect with Esterel's SCADE Suite for Safe Embedded Software", *Esterel Technologies White Paper,* 2003.

[73] *ARP4754 Certification considerations for highly-integrated or complex aircraft systems*: Society of Automotive Engineers, 1996.

[74] J. F. Bergeretti and B. A. Carre, "Information-flow and data-flow analysis of while-programs", *ACM Transactions on Programming Languages and Systems,* vol. 7, pp. 37-61, 1985.

[75] J. Barnes, *High Integrity Software, The SPARK Approach to Safety and Security*: Addison-Wesley, London, England, 2003.

[76] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language user guide*: Addison-Wesley Reading Mass, 1999.

[77] P. Amey and N. White, "High-Integrity Ada in a UML and C World," in *Reliable Software Technologies-Ada-Europe 2004*, 2004, pp. 225-236.

[78] P. Amey and D. Allen, "The INFORMED Design Method for SPARK," Praxis Critical Systems 20011, 1999.

[79] X. Sautejeau, "Modeling SPARK systems with UML", *ACM SIGAda Ada Letters,* vol. 25, pp. 11-16, 2005.

[80] IBM, *IBM Rational Rhapsody*,Last Accessed: 29th October 2012, Available: https://www.ibm.com/developerworks/rational/products/rhapsody/

[81] P. Amey and B. Dion, "Combining Model-Driven Design With Diverse Formal Verification," presented at the ERTS 2006, Toulouse.

[82] S. Hutchesson, "An Architecture-Centric Approach To FADEC Software Development," presented at the SPARK User Group 2006, Praxis High-Integrity Systems, Bath, UK, 2006.

[83] www.omg.org. (2003). *Response to the UML 2.0 OCL RfP-Revised Submission OCL RfP (ad/2000-09-03)*. Available: http://www.omg.org/docs/ad/03-01-07.pdf

[84] D. Chiorean, M. Pasca, A. Cârcu, C. Botiza, and S. Moldovan, "Ensuring UML Models Consistency Using the OCL Environment", *Electronic Notes in Theoretical Computer Science,* vol. 102, pp. 99-110, 2004.

[85] M. Richters and M. Gogolla, "Validating UML Models and OCL Constraints", *UML 2000--the Unified Modeling Language: Advancing the Standard: Third International Conference, York, UK, October 2-6, 2000: Proceedings,* 2000.

[86] B. Meyer, "Applying design by contract", *Computer,* vol. 25, pp. 40-51, 1992.

[87] H. E. Eriksson, *UML 2 toolkit*: Wiley, 2004.

[88] S. Flake and W. Mueller, "An OCL Extension for Real-Time Constraints", *Object Modeling with the OCL: The Rationale behind the Object Constraint Language,* vol. 2263, pp. 150–171.

[89] N. G. Leveson and K. A. Weiss, "Making embedded software reuse practical and safe," in *SIGSOFT '04/FSE-12 - 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* 2004, pp. 171-178.

[90] "AC20-148 Resuable Software Components," ed: Federal Aviation Administration, 2004.

[91] I. Habli, T. Kelly, and I. Hopkins, "Challenges of Establishing a Software Product Line for an Aerospace Engine Monitoring System," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*, 2007, pp. 193-202.

[92] F. Dordowsky and W. Hipp, "Adopting software product line principles to manage software variants in a complex avionics system," in *Proceedings of the 13th International Software Product Line Conference*, 2009, pp. 265-274.

[93] F. Dordowsky, R. Bridges, and H. Tschope, "Implementing a software product line for a complex avionics system," in *Software Product Line Conference (SPLC), 2011 15th International*, 2011, pp. 241-250.

[94] D. C. Sharp, "Reducing avionics software cost through component based product line development," in *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, 1998, pp. G32/1-G32/8 vol. 2.

[95] J. D. McGregor, "Testing a Software Product Line," Software Engineering Institute CMU/SEI-2001-TR-022, 2001.

[96] R. Lutz, "Survey of product-line verification and validation techniques," NASA - JPL 2007.

[97] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *Proceedings of the 33rd International Conference on Software Engineering*, Waikiki, Honolulu, HI, USA, 2011, pp. 321-330.

[98] S. Apel, H. Speidel, P. Wendler, A. v. Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 372-375.

[99] S. Trujillo, A. Perez, D. Gonzalez, and B. Hamid, "Towards the integration of advanced engineering paradigms into RCES: raising the issues for the safety-critical model-driven product-line case," in *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*, 2010, pp. 1-4.

[100] I. Habli, T. Kelly, and R. Paige, "Functional Hazard Assessment in Product-Lines–A Model-Based Approach," in *MDPLE'2009 1st International Workshop on Model-Driven Product Line Engineering CTIT PROCEEDINGS*, 2009, p. 26.

[101] I. M. Habli, "Model-based assurance of safety-critical product lines," Doctoral Thesis, Department of Computer Science, University of York, 2009.

[102] B. Hamid, A. Radermacher, A. Lanusse, C. Jouvray, S. GÈrard, and F. Terrier, "Designing fault-tolerant component based applications with a model driven approach", *Software Technologies for Embedded and Ubiquitous Systems,* vol. 5287, pp. 9-20, 2008.

[103] J. Liu, J. Dehlinger, and R. Lutz, "Safety analysis of software product lines using state-based modeling", *Journal of Systems and Software,* vol. 80, pp. 1879-1892, 2007.

[104] R. Braga, O. Trindade Junior, K. Castelo Branco, L. Neris, and J. Lee, "Adapting a Software Product Line Engineering Process for Certifying Safety Critical Embedded Systems" in *Computer Safety, Reliability, and Security*, F. Ortmeier and P. Daniel, Eds., LNCS vol. 7612, Springer Berlin / Heidelberg, 2012, pp. 352-363.

[105] S. Hutchesson and J. McDermid, "Towards Cost-Effective High-Assurance Software Product Lines: The Need for Property-Preserving Transformations," in *Software Product Line Conference (SPLC), 2011 15th International*, 2011, pp. 55-64.

[106] *VARIES Project*,Last Accessed: 31 October 2012, Available: http://www.artemis-ia.eu/project/index/view/?project=42

[107] *The Jet Engine*: Rolls Royce plc, 2005.

[108] FAA, *Conducting Software Reviews Prior to Certification*,Last Accessed: 18 January 2013, Available: http://www.faa.gov/aircraft/air%5Fcert/design%5Fapprovals/air%5Fsoftware/media/jobaid-r1.pdf

[109] J. MCNAMARA, C. LEGGE, and E. ROBERTS, "Experimental full-authority digital engine control on Concorde", *AGARD Advan. Control Systems for Aircraft Powerplants 17 p(SEE N 80-26306 17-07),* 1980.

[110] I. O'Neill, D. Clutterbuck, P. Farrow, P. Summers, and W. Dolman, "The formal verification of safety-critical assembly code," in *IFAC Symposium on Safety of Computer Control Systems 1988*, 1988, pp. 115-120.

[111] C. Fields, "Task force on defense software", *Defense Science Board,* 2000.

[112] W. Lam, J. McDermid, and A. Vickers, "Ten steps towards systematic requirements reuse", *Requirements Engineering,* vol. 2, pp. 102-113, 1997.

[113] W. Lam, "A case-study of requirements reuse through product families", *Annals of Software Engineering,* vol. 5, pp. 253-277, 1998.

[114] S. Hutchesson and N. Hayes, "Technology transfer and certification issues in safety critical real time systems," in *Real-Time Systems (Digest No. 1998/306), IEE Colloquium on*, 1998, pp. 2/1-2/4.

[115] A. R. Behbahani, "Advanced, Adaptive, Modular, Distributed, Generic Universal FADEC Framework for Intelligent Propulsion Control Systems (Preprint)," DTIC Document 2007.

[116] A. R. Behbahani, "Achieving AFRL Universal FADEC Vision With Open Architecture Addressing Capability and Obsolescence for Military and Commercial Applications (Preprint)," DTIC Document 2006.

[117] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.

[118] Foliage, "ATAM Assessment of Trent 1000 Software Architecture," Rolls-Royce plc., Internal Company Report 2009.

[119] M. Eriksson, J. Börstler, and K. Borg, "Managing requirements specifications for product lines-An approach and industry case study", *Journal of systems and Software,* vol. 82, pp. 435-447, 2009.

[120] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture: a system of patterns*: Wiley, 1996.

[121] K. D. Muller-Glaser, G. Frick, E. Sax, and M. Kuhl, "Multiparadigm modeling in embedded systems design", *Control Systems Technology, IEEE Transactions on,* vol. 12, pp. 279-292, 2004.

[122] K. Schmid and I. John, "A customizable approach to full lifecycle variability management", *Science of Computer Programming,* vol. 53, pp. 259-284, 2004.

[123] EASA, "Software Aspects of Certification," EASA CM - SWCEH – 002 ed, 2011.

[124] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, "MDA Distilled: Principles of Model-Driven Architecture," ed: Addison-Wesley, 2004.

[125] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL (k) parser generator", *Software: Practice and Experience,* vol. 25, pp. 789-810, 1995.

[126] A. L. Powell, "Right on time: Measuring, modelling and managing time-constrained software development," PhD Thesis, Department of Computer Science, University of York, 2001.

[127] "Trent XWB EEC Pre-SOI2 Audit Report," Internal Company Report, Aero Engine Controls, 2010.

[128] "Trent XWB EEC Pre-SOI2 Follow-Up Audit Report," Internal Company Report, Aero Engine Controls, 2010.

[129] E. Jackson, W. Schulte, D. Balasubramanian, and G. Karsai, "Reusing Model Transformations While Preserving Properties," in *Fundamental Approaches to Software Engineering*, 2010, pp. 44-58.

[130] T. E. Foundation, *Xtext - Language Development Framework*,Last Accessed: 27 February 2011, Available: http://www.eclipse.org/Xtext/

[131] R. Chapman, "Industrial experience with SPARK", *ACM SigAda Ada Letters,* vol. 20.4, pp. 64-68, 2000.

[132] P. Amey, "Correctness by Construction: Better Can Also Be Cheaper", *CrossTalk: the Journal of Defense Software Engineering,* vol. March 2002, pp. 24–28, 2002.

[133] M. Croxford and J. Sutton, "Breaking through the V and V bottleneck" in *Ada in Europe*, LNCS vol. 1031, Springer, 1996, pp. 344-354.