A COMPONENT-BASED APPROACH TO MODELLING SOFTWARE PRODUCT FAMILIES WITH EXPLICIT VARIATION POINTS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science & Engineering

2017

Simone Di Cola
School of Computer Science

Contents

Word count xxxxx

List of Tables

List of Figures

The University of Manchester

Simone Di Cola
Doctor of Philosophy
A Component-based Approach to Modelling Software Product Families with Explicit Variation Points
January 20, 2017

Software Product Line Engineering (SPLE) concerns the engineering of a family of software products in a given problem domain. Products in a family are variants of one another; they contain different features but also share common ones. SPLE consists of two phases: (a) domain engineering and (b) application engineering. In domain engineering, domain requirements are analysed to create artefacts, or assets, that can be harnessed to construct product variants. The key assets include: (i) a variability model, which specifies commonality and variability within the product family; (ii) a functional model, that specifies the behaviour of all product variants; (iii) a product family architecture (PFA),that defines the architecture of the product family and the architecture of each product therein; (iv) components, that can be assembled into products as specified in the PFA. In application engineering, domain engineering artefacts are harnessed to construct individual product variants.

Clearly, the quality of domain assets determines the effectiveness and efficiency of application engineering. However, creating all the domain engineering assets is a challenging task in the first place. As a result, existing SPLE approaches do not create all domain engineering assets, but use pragmatic substitutes instead. For example, most SPLE approaches do not create a PFA, which is a key domain engineering asset, as it defines an architectural template for all product variants. Instead, they either adopt a programming approach for creating product variants directly or a meta-programming approach for configuring product variants from coding templates. Another commonly missing domain engineering asset is a functional model, which is essential for constructing the PFA, as it specifies the behaviour of all product variants whose architectures should be defined by the PFA. SPLE approaches that do not create a PFA also do not create or use a functional model.

In this thesis, we address the issue of modelling software product families with a view to modelling all domain engineering assets and thereby enabling application engineering from a full set of such assets. Specifically, we present a component model that is designed for modelling and constructing software product families by providing facilities for defining and constructing a functional model and hence a PFA that is based on a feature model, the most widely used kind of variability model, as well as components that result from domain analysis.

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

To my mum

List of Publications

[1] S. Di Cola, K.-K. Lau, C. Tran, C. Qian and M. Schulze. Modelling Software Product Families with Explicit Variation Points. Manuscript submitted to the International Journal on Software and Systems Modeling (SoSym). 2016.

[2] S. Di Cola, K.-K. Lau, C. Tran, C. Qian and M. Schulze. A Component Model for Defining Software Product Families with Explicit Variation Points. In *Proc. of the 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. ACM, 2016.

[3] C. Tran., K.-K Lau and S. Di Cola. SOA Real-time System Development: an Automotive Case Study. In *Proc. of EMC ^{2} Summit at CPS Week 2016*. ERCIM, 2016.

[4] S. Di Cola, K.-K. Lau, C. Tran, C. Qian. Towards Defining Families of Systems in IoT: Logical Architectures with Variation Points. In *Proc. of the 1st EAI International Conference on Cloud, Networking for IoT systems*. Springer-Verlag, 2015.

[5] S. Di Cola, K.-K. Lau, C. Tran and C. Qian. An MDE Tool for Defining Software Product Families with Explicit Variations Points. In *Proc. of the 19th International Software Product Line Conference*. ACM, 2015.

[6] S. Di Cola, K.-K. Lau and C. Tran. A Graphical Tool for Model-Driven Development Using Components and Services. In *Proc. of the 41th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2015.

[7] S. Di Cola, K.-K. Lau, C. Tran, A. Celesti, and M. Fazio. A Heterogeneous Approach for Developing Applications with Fiware GEs. In *Proc. of the 4th European Conference on Service-Oriented and Cloud Computing*. Springer, 2015.

[8] K.-K. Lau and S. Di Cola. (Reference) Architecture = Components + Composition (+ Variation Points). In *Proc. of the 1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures.* ACM, 2015.

Chapter 1

Introduction

"In a tailor-made service, you fit the cloth according to the size and taste of the customer; not just the taste and strength of the designer."
— Dateme Tamuno

We live on a planet built on many products that are constantly changing and evolving. As a result, many companies no longer focus on building single products; they create a portfolio of related products treated as a single entity. In short, they build *product lines*.

Prior the introduction of product lines items were hand-crafted for individual customers. That is, no two goods were identical as each product was built from scratch.

After their adoption, product lines enabled *mass production* based on assembly lines in which standardised components were produced individually and assembled to create a finite product. Mass production significantly improved productivity compared to hand-crafting, but at the expense of customisation possibilities. However, today's customers may not accept whatever is offered to them; they often demand tailored solutions. Sure enough, nowadays the famous Henry Ford's statement "any customer can have a car painted any colour that he wants so long as it is black [1]" would not be accepted.

Thus over the years, due to a rising demand for personalised products, industries shifted from mass-production to *mass-customization* [2, 3]. For instance, PC manufacturers as in Fig. 1.1 offer variations of the same laptop by combining different reusable parts according to the customer's specifications chosen among a predefined set of possible options (*e.g.* memory size, processor and screen size).

A *product line*, therefore, amounts to a portfolio of related products built from a set of reusable components that share important similarities [4]. The systematic engineering[1] of such a portfolio using a common platform is known as *product line engineering* (PLE) [6].

The adoption of a common platform[2] implies that the whole development process must start with flexibility in mind, foreseeing all possible family members, identifying and describing their differentiation points. Considering the example in Fig. 1.1, building those family of laptops requires the implementation of a shared platform, which not only specifies its peculiarities but also must accommodate the customer's wishes for further customisations (e.g. screen coating).

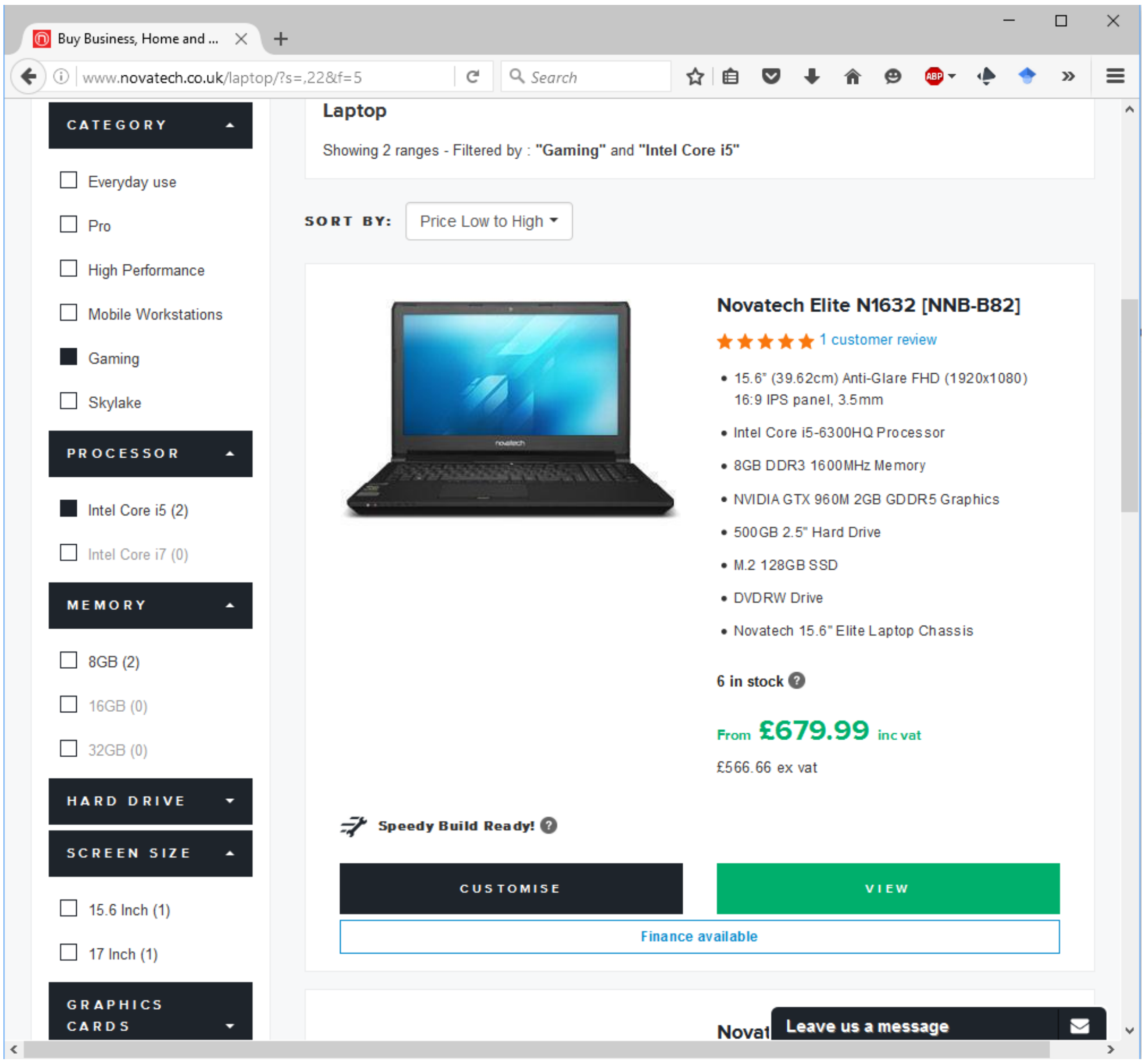
Similarly to the hardware industry, the software one has witnessed a shift from hand-crafted to standardised products. Initially, software was tailored for specific hardware and often sold with it. Nevertheless, during the years as customers' needs increased, so did the software complexity. An increase in software complexity implies an increase in the difficulty of its maintenance and evolution, with a high probability that the same functionality is developed and repeated at different places.

As a consequence, the software industry adopted standard platforms to reuse existing artefacts. In line with the hardware industry, the adoption of standard platforms resulted in mass production of standard *one-size-fits-all* products (*e.g.* Microsoft Office, Adobe PhotoShop) at the expense of customisation possibilities.

Overwhelmed with undesired features and tired of the lack of the desired ones, today's customer require software tailored to their needs [8]. Fitting our analogy with the laptop configurator of Fig. 1.1, suppose one wants to illustrate the software he has written to a potential customer. It is very unlikely that he will start by describing the classes or packages composing his product; the potential customer is not interested in that level of detail. More realistically, he will start off by describing the features the software can offer. The potential client can then understand if and how the software can satisfy her requirements. By the presence or absence of features, software, like any

[1]The term engineering comprises the activities involved in "planning, producing, delivering, deploying, sustaining, and retiring [5]" products.

[2]Open platforms and ecosystems constitute the core strategy of corporations such as Google and Apple, as they allow to expand their platforms outside their organisational boundaries [7].



Laptop

Showing 2 ranges - Filtered by : "Gaming" and "Intel Core i5"

☐ Everyday use
☐ Pro
☐ High Performance
☐ Mobile Workstations
■ Gaming
☐ Skylake

■ Intel Core i5 (2)
☐ Intel Core i7 (0)

☐ 8GB (2)
☐ 16GB (0)
☐ 32GB (0)

☐ 15.6 Inch (1)
☐ 17 Inch (1)

Novatech Elite N1632 [NNB-B82]

★★★★★ 1 customer review

- 15.6" (39.62cm) Anti-Glare FHD (1920x1080) 16:9 IPS panel, 3.5mm
- Intel Core i5-6300HQ Processor
- 8GB DDR3 1600MHz Memory
- NVIDIA GTX 960M 2GB GDDR5 Graphics
- 500GB 2.5" Hard Drive
- M.2 128GB SSD
- DVDRW Drive
- Novatech 15.6" Elite Laptop Chassis

6 in stock

From £679.99 inc vat

£566.66 ex vat

Speedy Build Ready!

Novat

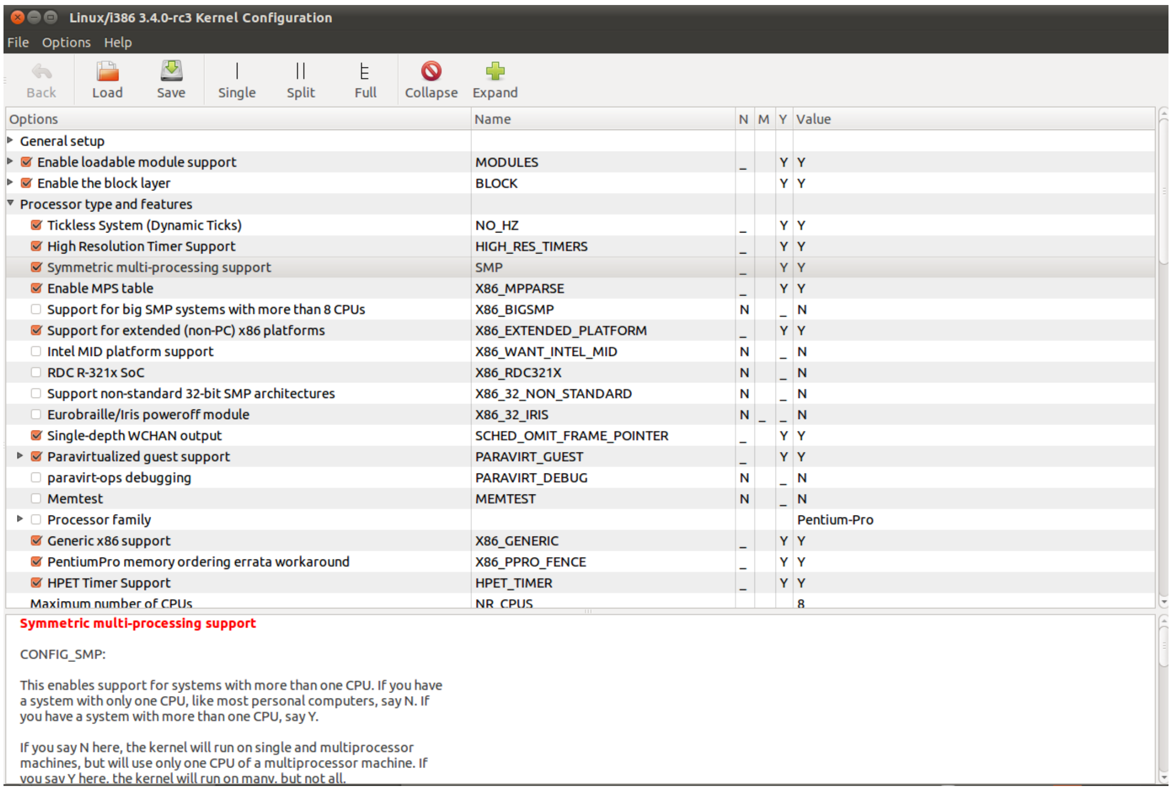
Figure 1.1: Novatech's latptop configurator.

other manufactured product, can come in different flavours. An entry level product, for example, offers a minimal feature set, while the deluxe version offers the most.

1.1 Research Problem

In order to introduce the research problem, let us take as an example the Linux kernel [9]. Able to run on different hardware platforms (from embedded systems to mainframes) and serve the needs of heterogeneous applications, the Linux kernel counts more than 10,000 features (v2.6.35 comprises 11,057 features [10]) configurable by means of a tool called KConfig (Fig. 1.2).

In order to model such a product line one would expect the same paradigm used for



Options	Name	N	M	Y	Value
General setup					
✓ Enable loadable module support	MODULES	_		Y	Y
✓ Enable the block layer	BLOCK			Y	Y
Processor type and features					
✓ Tickless System (Dynamic Ticks)	NO_HZ	_		Y	Y
✓ High Resolution Timer Support	HIGH_RES_TIMERS	_		Y	Y
✓ Symmetric multi-processing support	SMP	_		Y	Y
✓ Enable MPS table	X86_MPPARSE	_		Y	Y
☐ Support for big SMP systems with more than 8 CPUs	X86_BIGSMP	N		_	N
✓ Support for extended (non-PC) x86 platforms	X86_EXTENDED_PLATFORM	_		Y	Y
☐ Intel MID platform support	X86_WANT_INTEL_MID	N		_	N
☐ RDC R-321x SoC	X86_RDC321X	N		_	N
☐ Support non-standard 32-bit SMP architectures	X86_32_NON_STANDARD	N		_	N
☐ Eurobraille/Iris poweroff module	X86_32_IRIS	N	_	_	N
✓ Single-depth WCHAN output	SCHED_OMIT_FRAME_POINTER	_		Y	Y
✓ Paravirtualized guest support	PARAVIRT_GUEST	_		Y	Y
☐ paravirt-ops debugging	PARAVIRT_DEBUG	N		_	N
☐ Memtest	MEMTEST	N		_	N
☐ Processor family					Pentium-Pro
✓ Generic x86 support	X86_GENERIC	_		Y	Y
✓ PentiumPro memory ordering errata workaround	X86_PPRO_FENCE	_		Y	Y
✓ HPET Timer Support	HPET_TIMER	_		Y	Y
Maximum number of CPUs	NR_CPUS				8

Symmetric multi-processing support

CONFIG_SMP:

This enables support for systems with more than one CPU. If you have a system with only one CPU, like most personal computers, say N. If you have a system with more than one CPU, say Y.

If you say N here, the kernel will run on single and multiprocessor machines, but will use only one CPU of a multiprocessor machine. If you say Y here, the kernel will run on many, but not all,

Figure 1.2: Linux kernel configuration tool.

the hardware industry. Specifically, according to domain requirements, components are constructed and assembled conforming to a product family architecture (PFA)[3] in agreement with a customer's feature selection. Returning to our example of Fig. 1.1, once a customer choose her laptop's configuration, memory, CPU and other components are harnessed to a motherboard, which constitutes the product family architecture.

In fact, software product line engineering (SPLE) does distinguish between a phase in which domain assets are constructed and a phase in which those assets are assembled to build the desired product variants. In particular, SPLE distinguishes between a *domain engineering* phase and an *application engineering* one.

In order to prepare a platform that can deal with variability required by the defined product family, four key assets are constructed during the domain engineering phase: (i) a variability model, which specifies commonalities and variabilities within the product family; (ii) a functional model, which specifies the behaviour of all the family members; (iii) a PFA, which defines a family-wide architecture[4] and consequently

[3]Traditionally known as a reference architecture in structured analysis.
[4]While a variety of definitions of architecture have been suggested, this work will use the one of

the architecture of each product variant; (iv) components, which can be harnessed according to the PFA to construct individual family members.

Clearly, the quality of domain assets determines the effectiveness and efficiency of the construction of variants in application engineering. However, creating all the domain engineering assets is a challenging task in the first place. As a consequence, existing SPLE approaches do not model all domain engineering assets, but use pragmatic substitutes instead.

For example, most SPLE approaches do not create a PFA, which is a key domain engineering asset, as it defines an architectural template for all product variants (and indeed in more mature engineering domains such as automotives, the presence of a PFA distinguishes a product line from the development of *ad hoc* systems). Instead, they either adopt a programming approach for creating product variants directly or a meta-programming approach for configuring product variants from coding templates. Another commonly missing domain engineering asset is a functional model, which is essential for constructing the PFA, as it specifies the behaviour of all product variants whose architectures should be defined by the PFA. SPLE approaches that do not create a PFA also do not create or use a functional model.

As discussed by Bosh *et al.* in [12], this results in two key issues: (i) lack of first-class representation of variability; (ii) implicit dependencies between architectural elements and features. Consequently, it is difficult to assess the impact of variability at requirements and realisation level as it is often not clear the artefacts needed for a specific variant.

The ability of adding and removing features implies that features are treated as first-class citizens among a set of reusable parts. The most suitable and widely used methodology to analyse a domain in terms of features is FODA [13–15], acronym of feature oriented domain analysis [16, 17]. FODA defines commonalities and variabilities of a desired product family using several models. Among those the most important ones are: (i) feature model; (ii) functional model; (iii) product family architecture (Fig. 1.3).

A feature model is a hierarchical decomposition of features which establishes their

Bass *et al* [11]: 'the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.'

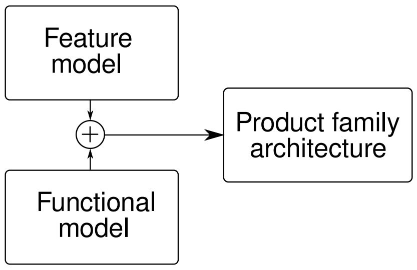


Figure 1.3: FODA models.

relationships and usage constraints, whereas a functional model specifies their behavioural aspects (in the notation of state-chart and activity-chart). Ideally, feature and functional models should lead to the construction of a product family architecture (PFA) and its constituent parts. A PFA determines the concepts, structure and textures necessary to achieve the variability expressed in the feature model, as well as the behaviour specified in the functional model, ensuring that the products share the maximum amount of parts in the implementations.

However, since feature and functional models do not readily translate to elements of a PFA [18, 19], its construction is not straightforward, and existing SPLE modelling approaches do not fully support this crucial phase (see chapter 3). Indeed, the construction of a PFA is still an outstanding engineering challenge [20].

A key challenge is the mapping between the feature model and the architecture (and hence its parts). There are two main issues here: (i) structure (ii) variation points. Firstly, if the structure of the architecture is not a tree, then it is not obvious how to map it to the feature model. Secondly, if the architecture does not contain explicit variation points, or if it does not include the full standard set of variation points in feature models, then mapping to the feature model will also be problematic.

As a result, current SPLE modelling approaches either adopt a programming approach for creating product variants directly or a meta-programming approach for configuring product variants from coding templates. Consequently, SPLE approaches that do not create a PFA also do not create or use a functional model.

For instance, Fig. 1.4 depicts how variability in the Linux kernel is expressed and its relation to (a simplified) feature model. In here, a code-base is annotated according to the features name. In deriving a variant, code fragments relative to unselected features are removed by the compiler.

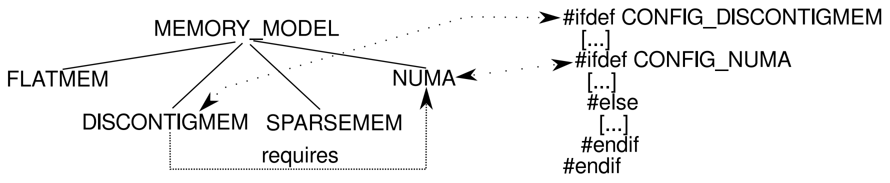


Figure 1.4: Variability in Linux kernel (adapted from [10]).

To summarise, most of the existing SPLE approaches do not adopt the FODA approach to modelling software product families; i.e. they do not model all domain engineering assets, missing most notably a functional model and a PFA. The consequence of missing domain engineering assets is that in application engineering, product variants are not assembled according to an architecture, but instead have to be configured one at a time as an instance of some template which does not capture the architecture of the whole product family. Indeed,the lack of a PFA in domain engineering results in the lack of a product variant architecture, as well as the incapability of analysing a product family at design time.

1.2 Research Aim and Objectives

The primary aim of this research is to investigate the feasibility of establishing a systematic approach for modelling scalable, maintainable and evolvable product family architectures. This can be achieved by combining the complementary strengths of component-based software engineering [21, 22] and software product line engineering in the context of software reuse, as they tackle this problem within a different granularity spectrum. Component-based engineering (reuse in the small) provides the flexibility required for a product line (reuse in the large) for building robust component-based product family architectures. Therefore, significant advantages are expected from their integration.

The choice of the SPLE methodology is somewhat simplified by the fact that FODA is the most widely used approach for SPLE. Despite its success, FODA mainly focuses on providing a detailed analysis of characteristics and behaviour of a product family. However, it is quite vague about the construction of a reference architecture.

As a component model, we have chosen X-MAN [23, 24], a component model enforcing strict hierarchical and compositional construction, to yield a scalable method for software product line engineering. The primary motivations behind this choice are that X-MAN is strictly hierarchical as its composition is algebraical, and unlike other component models [23], control and computation are separated, thus leading to a highly factored system design. Furthermore, and even more interestingly, an X-MAN architecture is executable, resulting in a deployable software product.

The aim is to combine FODA variability model, or in other words the feature model, with a functional model specified in X-MAN in order to define a component-model to systematically model PFAs. In order to achieve this aim, this work presents the following objectives:

- analyse current SPLE modelling approaches and use this investigation as a background for current research;
- define the rules to enable the automatic translation of an X-MAN architecture to its equivalent functional model;
- define a new component-model based on X-MAN to enable the systematic construction of a PFA, which explicitly contains the variability contained in the relative feature model. The new component model is called FX-MAN;
- implement a tool for FX-MAN to demonstrate the validity of the proposed solution;
- evaluate whether FX-MAN enables the systematic modelling of all the domain assets as specified by FODA;
- evaluate with an industrial use case if FX-MAN realises an effective and efficient SPLE modelling approach.

1.3 Research Contributions

This research led to the following contributions:

- the definition of systematic and structured framework for comparing software product line modelling approaches from three perspective: (i) modelled domain assets (table 3.1); (ii) engineering life-cycle (fig. 4.4); (iii) variability management (table 8.1(b)). This framework has worked as a 'red thread' for the research methodology to justify the need for complementing the existing product line engineering methods and underlying models.

- the creation of FX-MAN, an algebraic component model that supports the modelling of product families as hierarchical architectures with explicit variation points like in feature models. As a result, our model construct a product family as a tree which maps naturally to the corresponding feature model (Fig. 1.5).

- the denotation of an (automatic) mapping between FX-MAN architectures to a state chart, alongside the associated activity chart. As a result, to define a functional model, it is sufficient to construct an FX-MAN model of the product family according to the feature model for the given domain.

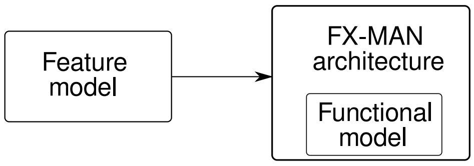


Figure 1.5: Research contribution.

- the development of a tool-set for FX-MAN licensed and commercialised by the University of Manchester Intellectual Property (UMIP) (see appendix I). It is available on the Click2Go platform at the following address `http://www.click2go.umip.com/i/software/x_man.html`.

- a research collaboration with pure::systems' (see appendix H), maker of the most used variability SPLE modelling tool, in order to integrate FX-MAN in a complete SPLE modelling environment.

1.4 Research Methodology

The research methodology adopted for this work is depicted in Fig. 1.6. The approach is a cross-road of research and software engineering as it is not only theoretical, but also empirical. In the following sections we briefly describe the aim of each stage.

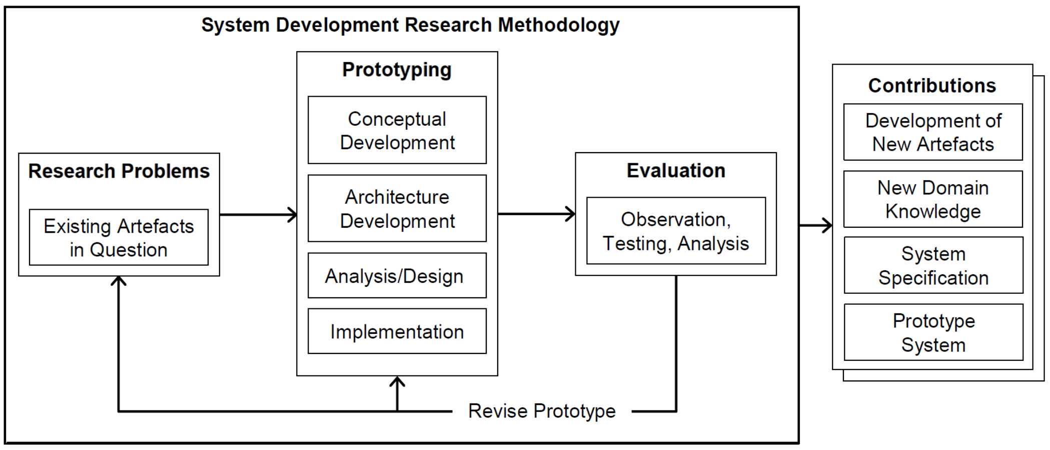


Figure 1.6: Research methodology (adapted from [25]).

Identification the Research Problem

At this stage, the research problem and pertinent research questions are identified and their significance justified. In addition to existing research artefacts, ideas from other disciplines should be explored and eventually incorporated in the proposed solution. Finally, the validity of proposed solution should be evaluated by means of a prototype.

Prototyping

At this stage, a software prototype is developed as a result of the following phases:

- *Conceptual development* - during this phase a conceptual framework is developed to investigate the requirements of the prototype under development.
- *Architecture development* - during this phase both the prototype's components are defined and their relationships established.
- *Analysis/Design* - during this phase the design of data structures and the specification of functions is defined.
- *Implementation* - during this phase observations about the defined framework, architecture and design are evaluated and used in re-designing the prototype.

Evaluation of the Results

At this stage, based on the conceptual framework and design specifications, the prototype is evaluated to verify if it satisfies the research problem. If it does, the research could yield several *contributions*. Otherwise, the cycle is repeated by either further developing the prototype, or by revising the research problem until it is satisfied.

1.5 Thesis Outline

The rest of the thesis is structured as follows.

- Chapter 2 (*Software Product Line Engineering*) introduces the background needed to understand the technical aspects involved in the development of a software product line. It continues by discussing the peculiarities of this approach against other software engineering techniques.

- Chapter 3 (*Modelling Software Product Families*) defines the research problem by presenting a taxonomy of the related SPLE modelling approaches. For each category of the taxonomy it describes its key characteristics, focusing on the domain artefacts modelled and evaluating it against the idealised SPLE modelling approach defined by the feature oriented domain analysis (FODA). This evaluation motivates this research by identifying omissions in the state of the art. The chapter concludes with the introduction of our component-based modelling approach (FX-MAN) and its comparison to FODA.

- Chapter 4 (*Component-based Software Modelling*) introduces the two elements forming the cornerstone of every component-based modelling approach: components and component model, which defines what components are and their composition mechanisms. Section 4.3 presents a taxonomy of current component models and their evaluation against the component-based software engineering (CBSE) desiderata. The evaluation results' motivate the choice of the component model used in this research.

- Chapter 5 (*A Component-based Approach to Modelling Software Product Families*) details the syntax and semantics of FX-MAN' elements, their relations to members of a functional model and how they participate in the modelling of a product family.

- Chapter 6 (*Tool Support*) gives an overview of the tool-set that supports the modelling of product families with FX-MAN. It describes the technology stack at its foundation, motivating its selection, and demonstrate its usage employing a vehicle control system example. The meta-model for each member of the tool-set is detailed in appendices B to D.

- Chapter 7 (*Use Case: External Car Light Family*) illustrates the use of our modelling approach against an industrial use case provided by pure-systems GmbH, maker of the market leader tool for variability management pure::variants (that our tool integrates with). It details all the steps involved in the construction a family of 28688 external car light controllers.

- Chapter 8 (*Evaluation*) assess against the family evaluation framework (FEF), the potentials of our approach in enabling organisations to realise a software product line. Precisely, it uses the architectural dimension of FEF, called FEF-A, to evaluate the maturity level achieved by our approach compared to the current SPLE modelling approaches analysed in Chapter 3.

- Chapter 9 (*Conclusion and Future Work*) ends the thesis by summarising the contributions and challenges of this research. Moreover, it discusses the limitations of the current work as well as directions for the future research.

  Finally, Fig. 1.7 depicts the thesis' outline and the recommended reading path.

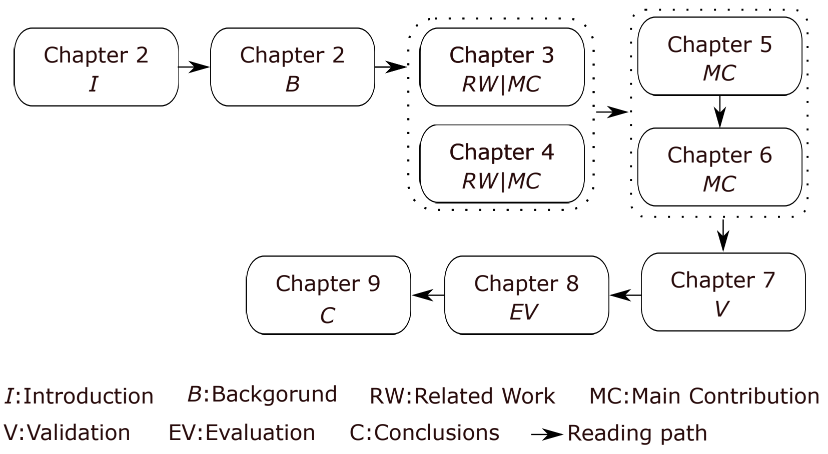


Figure 1.7: Thesis outline - absence of arrows indicates no preferred reading path.

Chapter 2

Software Product Lines Engineering

Begin at the beginning, the King said gravely,
"and go on till you come to the end: then stop."
—Lewis Carroll, *Alice in Wonderland*

2.1 Introduction

Since 1968, when McIlroy envisioned the possibility of establishing a software component market [26] and even more in the 90's mainly due to the feature-oriented domain analysis (FODA) methodology introduced by Kang *et al.* [16], software product lines (SPLs) have gained strength in the software industry. Indeed, over the years, more and more industrial case studies [27–36] have shown that constructing families of related products starting from pre-existing assets results in a considerable gain in quality, scalability, developers productivity, time-to-market and costs compared to a product-centric development.

In this chapter we introduce the background of software product line engineering starting from the definition of a software product line. For Clements *et al.* [27], a software product line (SPL) refers to: "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."

Software product line engineering (SPLE) is then a paradigm to tailor software products using platforms according to the requirements of individual customers [4].

Building products using a common platform means that reuse does not happen accidentally, but it is planned and enforced during the whole development life-cycle. As depicted in Fig. 2.1, a SPL is achieved by the exploitation of two distinct development life-cycles: *domain engineering* (development for reuse) and *application engineering* (development with reuse). In the ideal case, the two life cycles are loosely coupled and synchronised by platform releases.

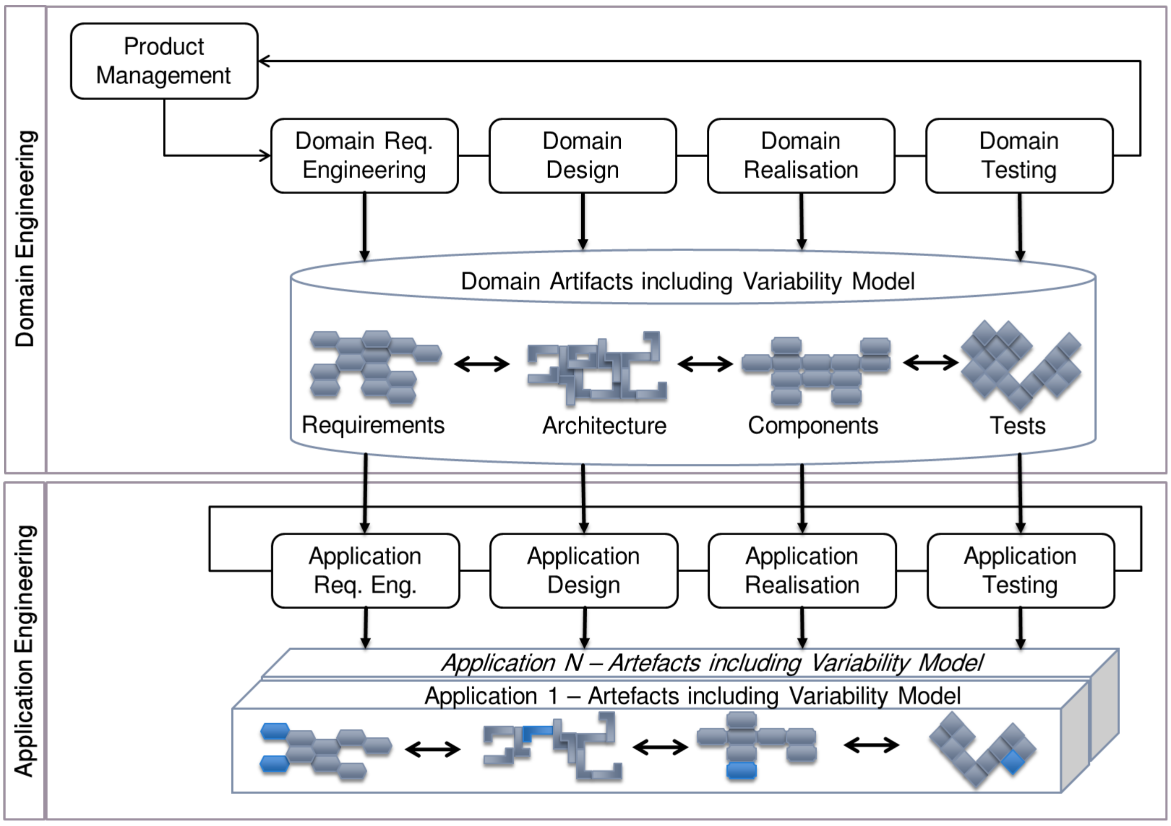


Figure 2.1: SPLE development life-cycles [4].

The activities within each development phase are described in the sections 2.2 and 2.3 using a family of library systems as an example (adapted from [37]). Finally, since at first glance many approaches can be confused with a software product line one, in section 2.4 we discuss what a SPL is not.

2.2 Domain Engineering

Domain engineering is the process where both commonalities and variabilities of a product family are identified and developed. Its aim is to establish the reusable platform in which domain artefacts (*i.e.* requirements, architecture, software components and tests)[1] are developed and stored. The concept of platform results in a standardisation process that affects all levels of an organisation. This explains why SPLE represents the first infra-organisational software reuse approach that has proven successful [29, 30]. The domain engineering activities are described in the following subsections.

2.2.1 Product management

This activity aims at identifying the product family, the variability among its members and their economic aspects. Its input is represented by the business objectives defined by the top management, whereas its output is a product map. A product map sets the products release dates, establishes their commonalities and variabilities and provides a list of possible existing artefacts that can be reused for creating the product line platform.

		City Library	Research Library	University Library
Customer Mngmt	Registration	×	×	×
	Unregistration	×	×	×
	Registration Change	×	×	×
Loan Management	Loan	×	×	×
	Return	×	×	×
	Report Loss	×	×	×
	Item Reservation		×	×
	Item Suggestion			×
Acc.	Billing	×		

Figure 2.2: Library product map.

As an example, Fig. 2.2 depicts the (simplified) product map of a family of library systems. It determines the characteristics of three types of products, namely a *city library*, a *research library* and a *university library*. They all offer the capability for a customer to manage his/her registration and book loans. However, only the research

[1]Also referred as core assets.

and the university library systems allow reserving an item that is not available. Moreover, the university library system supports the process of managing items suggested by other library users. Finally, only the city library system enables the librarian to collect fees from the library customers.

2.2.2 Domain requirement engineering

This activity focuses on requirements elicitation of the scoped product line. As input, it takes the product map. As output, it produces the requirement specifications for the product family, along with an initial variability model used in further development steps.

During the years, several notations have been proposed to represent a variability model (*e.g.* decision model [38], orthogonal variability model [4], common variability language [39] and feature model [17]), each of which with slightly different focus and goals[2]. However, they all share the notion of *variation point*, *variant*, *variability dependencies* and *constraint dependencies.*

A *variation point* outlines a location at which the designed variation will occur, enriched by contextual information [42].

A *variant* represent an instance of a variable item that can satisfy a variation point.

Variability dependencies are used to denote the relationships between different variants that satisfy a variation point.

Finally, *constraint dependencies* denote relationships among variant selections (possibly belonging to different variation points). They can be of two types: *requires* and *excludes.* The former specifies that a variant requires the selection of another variant, whereas the latter indicates the opposite.

Feature diagrams [43, 44] (a graphical actualisation of FODA feature models) are the de-facto standard notation to represented variability models in SPLE [13]. As illustrated in Fig. 2.3, a feature diagram captures common and variable characteristics (as well as their usage constraints) of a product family as nodes in a tree. Variability is expressed by *optional*, *alternative* and *or* variation points.

A *optional* variation point defines a relationship in which a parent feature can be

[2]A comparison between variability models is beyond the scope of this thesis. See [40, 41] for more details.

chosen independently from its child, but the latter can be chosen only if its parent is selected.

An *alternative* variation point denotes a one-out-of-many relationship between children of a common parent: exactly one child must be selected when the parent is chosen.

On the contrary, an *or* defines a some-out-of-many relationship where at least one child feature must be selected when its parent is chosen. Furthermore, cross-tree constraints depict relationships (*e.g.* *requires* and *excludes*) between features belonging to different sub-trees as direct arrows.[3]

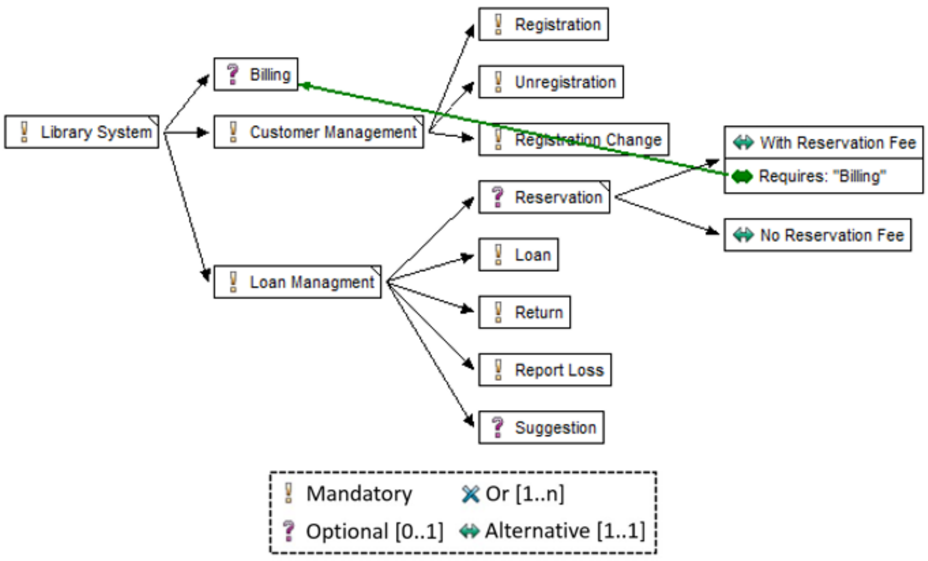


Figure 2.3: Library SPL feature diagram.

In our example, the library feature diagram of Fig. 2.3 establishes the relationships among mandatory and variable characteristics of the product family initially scoped by the product plan of Fig. 2.2 and detailed by the requirements elicited during this activity in Fig. 2.4. For instance, the requirement *OPT-2* states that a library system can optionally allow a customer to reserve a book, whereas *ALT-2-1* and *ALT-2-2* specifies that a customer can either pay a fee for this service or reserve a book for free. This is captured in the library feature diagram by the optional sub-tree *Reservation*, which contains two alternative nodes *With Reservation Fee* (which requires *Billing*) and *No Reservation Fee*.

[3]In theory, any dependency among features can be expressed by cross-tree constraints over a set of optional features. In reality, feature decomposition is preferred and cross-tree constraints are only used to model feature relationships that do not fit into the hierarchy.

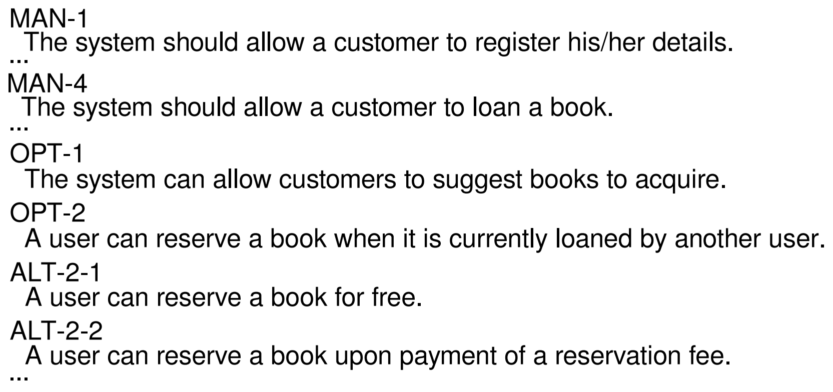

MAN-1
The system should allow a customer to register his/her details.
...
MAN-4
The system should allow a customer to loan a book.
...
OPT-1
The system can allow customers to suggest books to acquire.
OPT-2
A user can reserve a book when it is currently loaned by another user.
ALT-2-1
A user can reserve a book for free.
ALT-2-2
A user can reserve a book upon payment of a reservation fee.
...

Figure 2.4: Library SPL requirements.

2.2.3 Domain design

This activity focuses on constructing the structural and behavioural domain design models needed for the realisation of the product family architecture. Its input is the variability model produced by the domain requirement engineering activity. The output is a product line architecture[4] (PLA) for the identified family.

In literature, the term PLA is often interchangeably used with the reference architecture (RA) one. However, from their definitions, two subtle but critical differences between them exist [45].

Although contrasting opinions still exist [46, 47], there appears to be some agreement that an RA refers to "a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flow between them [11]".

On the other hand, Gomaa [48] defines a PLA as " an architecture for a family of products, which describes the kernel, optional, and variable components in the SPL, and their interconnection." Pohl *et al.* [4] complete this definition by specifying a PLA as a "core architecture that captures the high-level design for the products of the SPL, including the variation points and variants documented in the variability model." This definition entails that a PLA is one of the most important artefacts that form the platform, as it provides a common, high-level structure for all possible products of the product line.

According to those definitions, we can identify two main differences between an RA

[4]While a variety of definitions of architecture have been suggested, this work will use the one of Bass *et al* [11]: 'the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.'

and a PLA. Firstly, while an RA provides standardised solutions for a broader domain, a PLA is focused on a smaller family of related systems. Secondly, a PLA deals with variabilities among products, while an RA does not. In order to avoid confusion, in the rest of the thesis we will use the term product family architecture (PFA) instead of product line architecture.

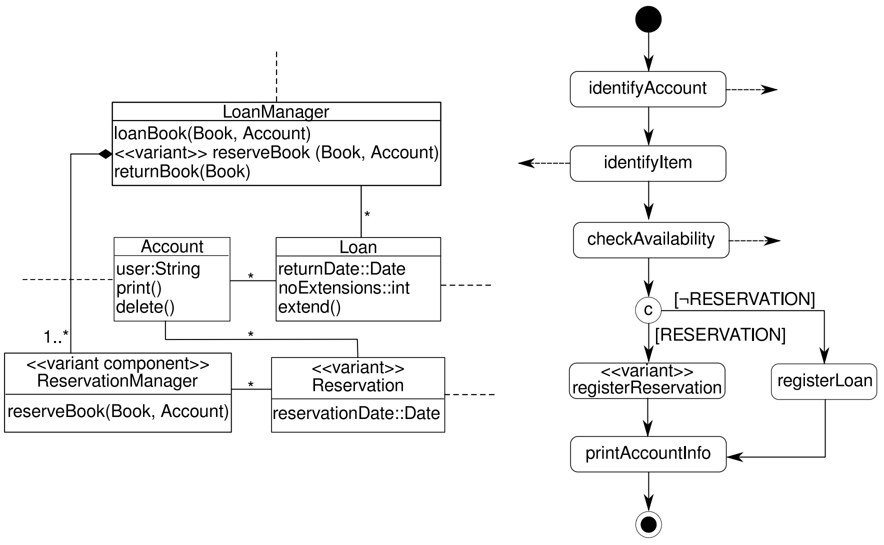


Figure 2.5: Library SPL architecture/design.

Fig. 2.5 depicts a portion of the library family architecture and its behavioural model in the notation of a state-chart. In here, the classes *Reservation* and *ReservationManager* are indicated as optional by means of the stereotypes *«variant»* and *«variant component»* respectively. Similarly, the transition from the state *checkAvailability* and the optional state *registerReservation* is guarded by the condition *[RESERVATION]*, which indicates the presence of the reservation feature.

2.2.4 Domain realisation

This activity involves all the sub-activities required to design and implement the reusable software components. As detailed and discussed in section 4.3, several definitions of software component exist. However, in this work the term component will be used solely when referring to 'a software element [with contractually specified interfaces] that conforms to a component model and can be independently deployed and composed without modification according to a composition standard [21].'

```
class LoanManager{
 #ifdef RESERVATION
   ReservationManager *reservationMgr;
 #endif
 public:
  int loanBook (Book *book, Account *account);
  int returnBook(Book *book);
 #ifdef RESERVATION
   int reserveBook(Book *book, Account *account);
 #endif
 ...
 #ifdef RESERVATION
   int LoanManager::reserveBook(Book *book, Account *account){
     return reservationMgr -> reserveBook(book,account);
   }
 #end
 ...
```

Figure 2.6: Library SPL source code.

The input of this activity consists of the PFA along with a list of software components that need to be developed. The output amounts to a set of loosely coupled, highly configurable software components and their detailed documentations. It is important to notice that the developed components are only stored within the repository and not assembled to create a running application.

Returning to our example of a family of library systems, Fig. 2.6 depicts a detail of the *LoanManager* implementation. According to the PFA, the presence or the absence of the variant *Reservation* is managed using the pre-processor directive *#ifdef RESERVATION* as variability mechanisms. In appendix A a taxonomy of the current variability mechanisms is presented and discussed.

2.2.5 Domain testing

This activity deals with validation and verification of reusable components against their specifications. The input for this activity comprises the PFA, its software components and the domain requirements. The output amounts to a set of reusable test artefacts.

Whereas at this stage there is no a working application, there are several strategies for testing integrated components [49–51]. For instance, it is possible to test a sample product or just a part of it.

2.3 Application Engineering

Application engineering is the process where products[5] are concretely built by reusing the artefacts forming the common product line platform. Its activities (as depicted in Fig. 2.1) are described in the following subsections using as an example a city library system (a family member identified in the product map of Fig. 2.2) as the variant to be built.

2.3.1 Application requirements engineering

This activity aims at determining the requirements for an identified product variant, taking into account the delta between what is required and what is available. Its input consists of the domain requirements, the product map, the variability model and any additional requirements not captured during the domain engineering phase. The output consists of the requirements specification of the desired family member along with a "resolved" variability model, in which a stakeholder has selected the required features.

For instance, Fig. 2.7(a) depicts the list of requirements for the city library variant and its resolved variability model (in the notation of a feature diagram). According to the product map of Fig. 2.2, the city library allows accounting, therefore the optional feature *Billing* is selected. However, it does not allow reservation for an item. Therefore, the optional feature *Reservation* is not selected.

2.3.2 Application design

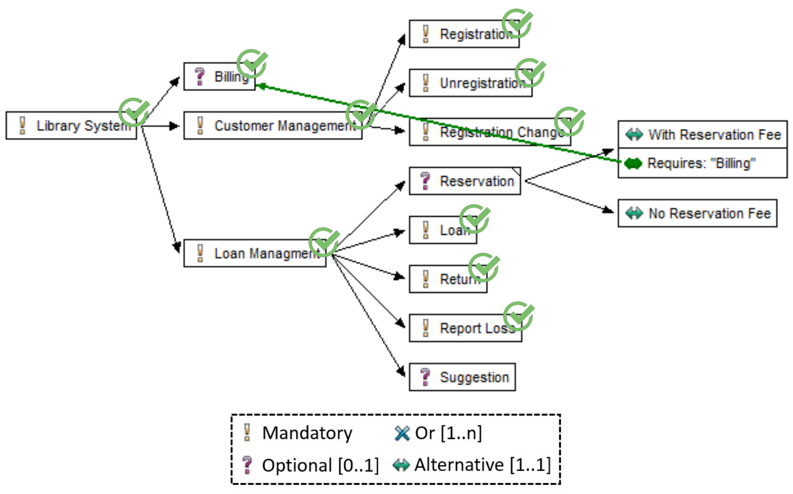
This activity derives an instance of the PFA (and related design models) according to the requirements identified in the previous step. It basically selects and configures the required characteristics from the product family architecture, taking into account the application specific adaptations. Therefore, the input of this activity comprises the PFA as well as the product requirement specifications. The output consists of the application architecture and its related design models specific for the product at hand.

Since new requirements can be added, at this stage each ad-hoc structural change must be evaluated and eventually rejected if too expensive.

[5]Also refereed to as family members, variants or product variants.

The system should allow a user to register his/her details.
...
The system should allow a user to loan a book.
...

(a) City library requirements.



(b) City library feature diagram.

Figure 2.7: City library specifications.

Fig. 2.8 depicts the architecture and design of the city library specified in the previous step. According to its feature diagram, this variant does not offer the capability to reserve a book. Therefore its architecture does not include the classes *Reservation* and *ReservationManager*. Similarly, its state-chart does not include the variant state *registerReservation*.

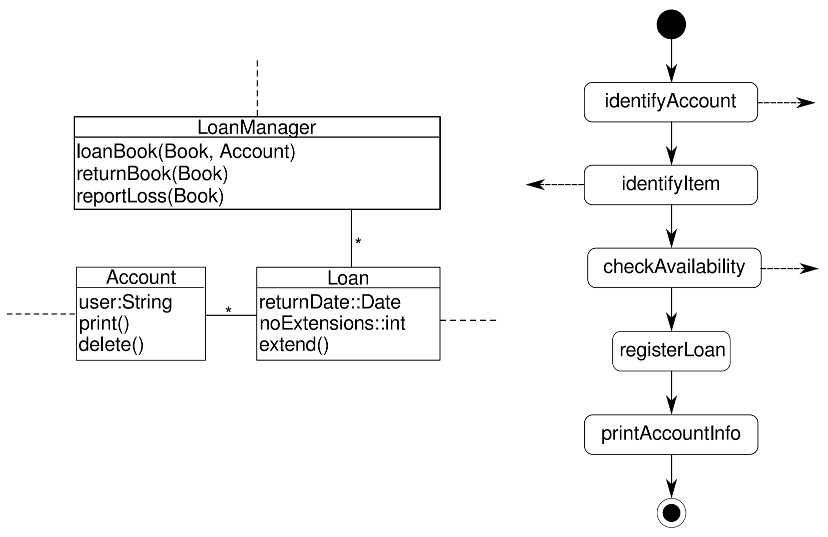


Figure 2.8: City library architecture/design.

2.3.3 Application realisation

This activity subtends the operations needed to select and configure the required software components, along with the implementation of the application-specific ones.

According to the PFA composition mechanism, components are assembled in order to build the desired family member. The input of this activity includes the application architecture and its design models, together with the components retrieved from the domain artefacts. The output is represented by the working product and its documentation.

Fig. 2.9 illustrates a detail of the class *LoanManager* for the city library in which the compiler has removed the code fragments related to the *Reservation* feature.

```
class LoanManager{
 public:
  int loanBook (Book *book, Account *account);
  int returnBook(Book *book);
 ...
```

Figure 2.9: City library source code.

It is important to remark the fact that, even though each system built in a software product line is a system in its own right, it is built by taking applicable components from a common repository. These components are then adapted through planned variability mechanisms and then assembled according to the rules of a product family architecture. Such variability mechanisms are presented and discussed in appendix A.

2.3.4 Application testing

This activity validates against the application requirements if the intended product variant has been created. The input of this activity comprises the product requirements and implementation along with the domain tests artefacts. The output amounts to a report with the results of the performed tests along with a detailed description of any problem discovered.

2.4 What Software Product Lines Are Not

At first glance, many approaches can be confused with a software product line. Let us see which ones are not.

2.4.1 Accidental reuse

For almost four decades, IT industries have tried to accelerate the software development process by using pre-built software units. Even though component-based software engineering is a reality (see section 4.3), true component-based reuse envisaged by McIlroy [26] is still an exception rather than a rule. Historically, we can identify three fundamental causes of this failure: lack of good components that live up to their expectations; lack of easy and precise retrieval technologies; difficulty to verify a component without a context. As a result, time saved through reusing needs to be spent in searching, selecting and verifying the candidate components. Not surprisingly, organisations tend to build their own components from scratch. Unlike accidental reuse, in a software product line reuse is envisaged, enabled and required at all levels of the organisation.

2.4.2 Single-system development with reuse

In developing variations of an existing product, perhaps the simplest solution is the so called clone-and-own [52]. In this approach, new products are built by creating, and then modifying, an exact copy of an existing product. Since a new clone begins a separate maintenance trajectory, this technique presents both advantages and disadvantages. The clear advantage is that once cloned, the new project is developed independently from the original one. Nevertheless, this initial advantage is hindered by the lack of traceability between the shared functionalities. Indeed, as the two projects may diverge further during the time, a bug fixed in the original project may not be replicated to the clone, so the maintenance costs become unsustainable [53]. Compared to clone-and-own, a product line differs mainly in two aspects. Firstly, software product line assets are engineered for being reused. Secondly, a product line is seen as a whole, not as many distinct products.

2.4.3 Configurable architecture

In any non-trivial system, the software architecture represents a pivotal part [11, 54]. In fact, it captures a system high-level design choices, including the composition mechanism, as well as the directives for its maintenance and evolution. Due to its important role, realising a configurable architecture that can be reused in different projects and reconfigured as necessary is a wise choice. However, in a software product line, an architecture is just one artefact in the core assets repository.

2.4.4 Product releases

Any company, at intervals more or less established, releases new versions of its own products. A new release is typically built by evolving previously built core assets. At first glance, this is very similar to a software product line approach. However, in a software product line the evolution of a single product must be contextualised with the evolution of the product line as a whole. Moreover, an early release in a software product line is nothing more than just an instantiation of the core assets. Therefore, old releases are not discarded, but kept as a valuable part of the product family.

2.4.5 Technical standard

A technical standard is an established set of norms designed to promote interoperability and lower the costs related with maintenance and support of a system. Certainly a company that undertakes a product line approach must establish a technical standard, but this is just a part of the product line, no more.

2.4.6 Component-based development

In mature industry domains such as automotive or electronic engineering, the term component refers to a small standard and exchangeable part. In informatics, as introduced in the previous section, a component is defined as “a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard ” [21].

Software product lines do rely on component-based development, but it goes further by introducing the concept of variability. That is, in component-based development, whenever a variation is involved, it is usually maintained separately. Moreover, component-based development fails to provide those technical and organisational management aspects crucial to the success of a software product line.

2.5 Summary

Software Product Line Engineering (SPLE) has proven to be a successful approach for the construction, maintenance and evolution of families of software products that share commonalities and variabilities in terms of features.

Building families of related products using a common platform means that reuse does not happen accidentally, but it is planned and enforced by the exploitation of two distinct development life-cycles: *domain engineering* (development for reuse) and *application engineering* (development with reuse).

This separation of concerns has the advantage of building a solid platform on the one hand, and of building customer-tailored products on the other. Indeed, domain engineering is in charge of assuring that the available variability is adequate for building the identified product family, while a large part of application engineering exploits this variability for binding artefacts according to a predetermined composition model.

Chapter 3

Modelling Software Product Families

"A problem well put is half solved."

—John Dewey

3.1 Introduction

Modelling a product family amounts to the development of its domain assets, or rather (i) a variability model, which specifies commonalities and variabilities within the product family; (ii) a functional model, which specifies the behaviour of all the family members; (iii) a PFA, which defines a family-wide architecture4 and consequently the architecture of each product variant; (iv) components, which can be harnessed according to the PFA to construct individual family members.

In this chapter we define the research topic. We classify existing software product family modelling approaches according to their underlying variability mechanisms into 5 categories. For each category, we analyse the modelled domain assets and their use during product derivation. Finally, we compare this results against FODA,which defines the idealised feature-oriented method for modelling such assets.

The resulting taxonomy, summarised in Table 3.1, shows that whereas the FODA approach to modelling software product families creates all domain engineering assets, most existing SPLE techniques do not.

Section 3.2 describes FODA and defines the comparison criteria. The 5 categories forming the taxonomy are detailed in sections 3.3 to 3.7, whereas section 3.8 introduces our approach, how it differs from existing modelling approaches and compares to FODA.

3.2 FODA

In SPLE the notion of a feature represents a concern of primary interest as it expresses concepts that span both *problem space* and *solution space*. In the *problem space* features define characteristics of a product visible by non technical stakeholders (e.g. end users), while in the *solution space* features guide the structure and implementation of reusable artefacts (e.g. components and design models).

Consequently, in literature a feature has been defined in different ways. Kang *et al.* [16] describe a feature as ‘a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems’. Similarly, [55–59] define a feature as a means of communication used to identify the capabilities of family members in a product line. On the contrary, [8, 60–62] treat a feature as ‘an optional or incremental unit of functionality’, or rather as a concept to be used at design and implementation level. To capture the essence of those definitions, we define a generic feature as *a distinguished capability visible to an end-user, implemented (where possible) as a unity of functionality.*

The most suitable and widely used methodology to analyse a domain in terms of features is FODA [13–15], acronym of feature oriented domain analysis [16, 17]. With respect to the SPLE development life-cycles of Fig. 2.1, Fig. 3.1 illustrates the artefact constructed by FODA in order to model a product family.

In *domain engineering problem space*, domain knowledge is analysed and captured by a set of requirements and a variability model. The latter, in the notation of a feature diagram, identifies a product family by modelling the relationships among its set of features (*e.g.* the feature diagram for a family of library systems in Fig. 2.3). Consequently, a valid sub-set of those features (i.e. valid with respect to the relationships among them) identifies a family member (*e.g.* the city library variant in Fig. 2.7(b)).

Domain knowledge is then used in *domain engineering solution space* to determine

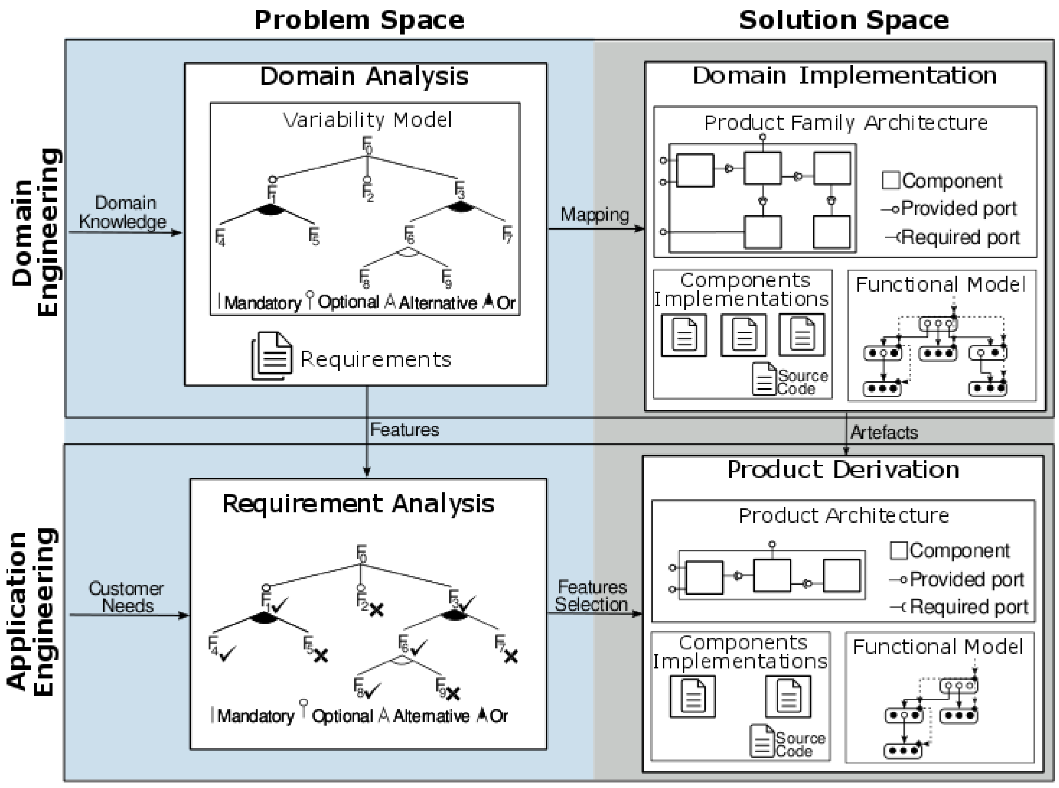


Figure 3.1: SPLE using FODA for modelling product families.

the constituent assets of the product family. From the feature model, FODA defines a functional model parametrised on the features. It describes domain capabilities from a structural and a behavioural perspective. The former details the functional components and how data flows among them, whilst the latter specifies in terms of states and transitions when, and under what circumstances, the functional components are triggered. These can be expressed using different notations.

Structured analysis [63–65] defines a functional model as a hierarchical data flow diagram (DFD) together with a state transition diagram (STD) for each of its levels. A generic DFD is shown in Fig. 3.2. Within a DFD, activities are specified as *data transformations* (DTs) interconnected by data flows. A DT is not fully specified until decomposed into its *primitive* data transformations (PDTs). DTs are activated by *control transformations* (CTs). A CT denotes a finite state machine, modelled by a STD, where control flow input represents events, and control flow output represents prompts (i.e. trigger event).

STATEMATE [66, 67] uses *state-charts*, an extension of STDs, for CTs, and *activity charts* instead of DFDs (activities being DTs). As illustrated in Fig. 3.3, an activity-chart (successively) decomposes activities into sub-activities, and identifies data flows

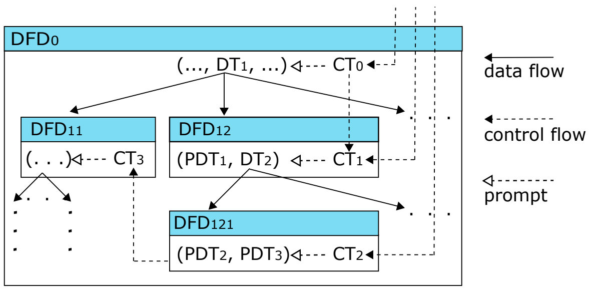


Figure 3.2: A generic DFD.

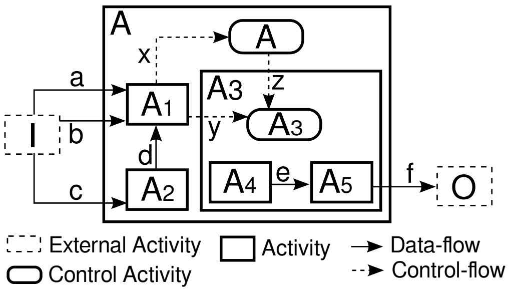


Figure 3.3: A generic activity chart.

between them (drawn as solid arrows). Each activity is controlled by at most one control activity (cf. CT in DFD), specified in the language of state-charts. When an activity requires no further decomposition and its behaviour can be described by a state chart alone, then the control activity is its only sub-activity. Entities external to the top level activity are referred to as external activities and depicted as dashed-line boxes.

UML [68] defines functional models using an object-oriented adaptation of state-charts and DFDs in terms of state machines and activity diagrams respectively.

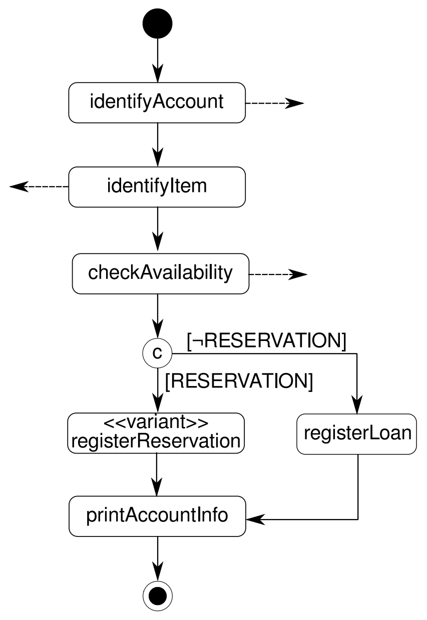
Finally, Simulink [69] provides *Stateflow* [70] as an environment for modelling and simulating functional models based on state machines and flow charts.

Whilst the above approaches are well established for defining functional models of single products, a functional model for a family scoped by a feature diagram is defined as a feature-oriented state-chart together with all the associated activity-charts.[1] This is illustrated in Fig. 3.4(a), where a feature-oriented sate-chart for the family of library systems presented in the previous chapter contains *mandatory* states, corresponding to mandatory features, and a *variant* state *registerReservation*, corresponding to the optional feature *Reservation*.

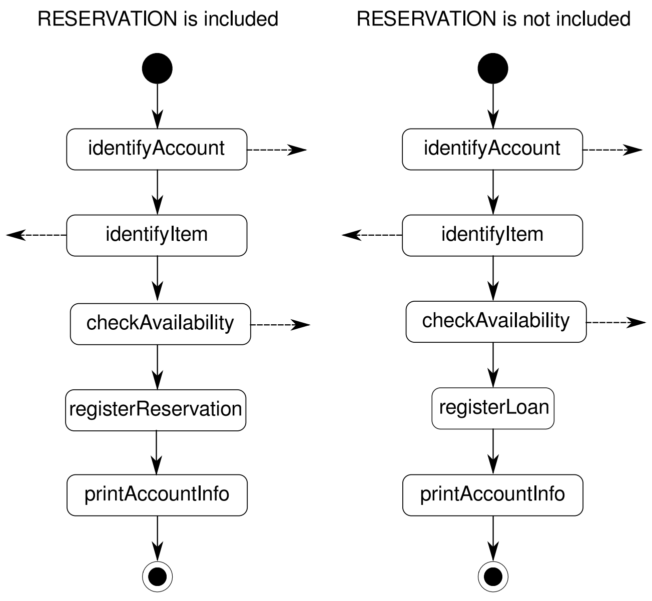
Thus as shown in Fig. 3.4(b), this state-chart can be instantiated into two state-charts according to the inclusion or exclusion of the feature *Reservation*.

From the feature and functional models, FODA suggests ways of constructing a PFA [17, 52] (Fig. 3.5(a)). A PFA determines the concepts, structure and textures necessary to achieve the variability expressed in the feature model, as well as the behaviour specified in the functional model, ensuring that the products share the maximum amount of parts in the implementations.

[1]A comprehensive example can be found in [16].



(a) State chart



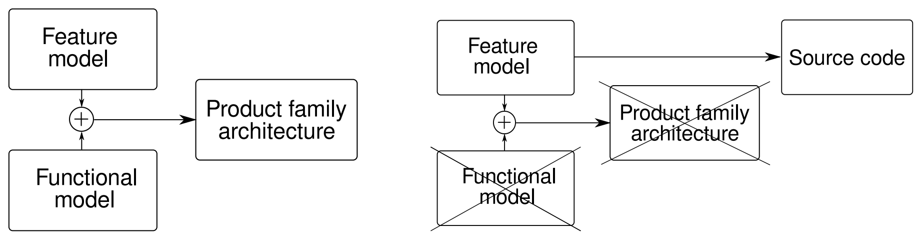
(b) Instances

Figure 3.4: A feature-oriented state-chart and its instances for the library SPL.

Precisely, FODA shows how the functional model can be used to identify application processes and their constituent modules: it uses state charts to pinpoint application processes, and decomposes each process into modules containing functions and data objects specified in activity charts. However, it remains vague on how to map these processes and modules to architectural elements (and hence its parts) [18, 19].

There are two main issues here: (i) structure (ii) variation points. Firstly, if the structure of the architecture is not a tree, then it is not obvious how to map it to the feature model. Secondly, if the architecture does not contain explicit variation points, or if it does not include the full standard set of variation points in feature models, then mapping to the feature model will also be problematic. As a consequence, the construction of a PFA is still an outstanding engineering challenge [20]. Therefore, the picture in Fig. 3.1 remains an idealised one.

Existing SPLE approaches do not model all domain engineering assets, missing most notably a functional model and a PFA (Fig. 3.5(b)). Thus product variants are not assembled according to an architecture, but configured one at time either adopting a programming approach or a meta-programming approach for assembling product variants as instance of coding templates, which do not capture the architecture of the whole product family (Fig. 3.5(b)). Consequently, SPLE approaches that do not create a PFA also do not create or use a functional model.



(a) FODA idealised modelling approach.

(b) Current SPLE modelling approach.

Figure 3.5: Overview of SPLE modelling approaches.

For instance, Fig. 3.6 depicts how variability in the Linux kernel is expressed and its relation to (a simplified) feature model.

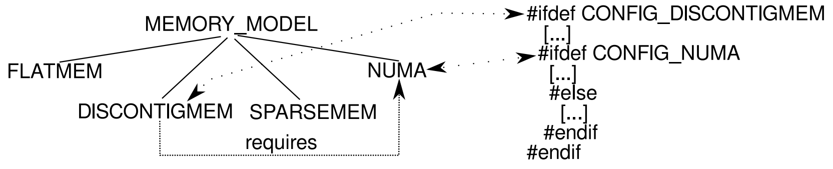


Figure 3.6: Variability in Linux kernel (adapted from [10]).

The lack of a PFA in domain engineering results in the lack of a product variant architecture, as well as the incapacity of analysing a product family at design time (*i.e.* products need to be instantiated in order to be analysed).

This is summarised in Table 3.1, which shows for existing SPLE approaches (categorised according to their underlying variability mechanism) the domain engineering assets they model. In the table, **M** stands for a model that has been constructed

Software Product Family Modelling Approach	**Domain Engineering**				**Application Engineering**	
	Feature Model	Functional Model	Product Family Architecture	Components	Product Architecture	Product Variant
FODA	**M**	**M**	**M**	S	M	S
In Annotation-based SPLE	**M**	**M**	**T**	**S**	—	S
In Weaving-based SPLE	**M**	—	—	**S**	—	S
In Superimposition-based SPLE	**M**	—	—	**S**	—	S
In Δ-based SPLE	**M**	—	—	**S**	—	S
In ADL-based SPLE	**M**	—	**T**	**S**	T	S
In our approach	**M**	M	**M+S**	**M+S**	M+S	M+S

M = constructed model
T = constructed template
S = constructed software

M = model derived from **M**
T = instance of **T**
S = software derived from M or T

Table 3.1: Comparison of approaches to modelling software product families.

during domain engineering (e.g. state-chart and activity-chart); whereas M stands for a model that can be derived from an existing **M**. **T** stands for a coding template; whereas T is an instance of an existing **T**. **S** stands for software constructed from scratch, whereas S stands for software that is derived from an existing M or T.

In SPLE that adopts the FODA approach to modelling software product families, a feature model, a functional model as well a PFA are constructed during domain engineering. These are shown as **M** in Table 3.1, whereas components are shown as S, as they are derived from the specification of functional components in a functional model. In application engineering, the architecture for each product can be derived from the PFA. Therefore, in Table 3.1 the product architecture is denoted by an M. Similarly, software for a product variant (S in Table 3.1) is derived from domain engineering assets .

As can be seen in Table 3.1, whereas the FODA approach to modelling software product families creates all domain engineering assets, most existing SPLE techniques do not. Details of the latter are provided in the following sections.

3.3 Annotation-based SPLE

Widely used in industry [13] as supported by leading tools like pure::variants [71] and Gears [5], annotation-based SPLE approaches [55, 72] consider models, often called 150% models [73, 74], as templates designed with built-in variation points to represent the entire solution space of a product family (Fig. 3.7). As defined by Coplien [75], such a modelling technique implements negative variability. Variant annotations like conditional compilation [76] and UML stereotypes [48] identify parts of the models that need to be removed in order to derive a product during the application engineering phase. Each variation point is defined by naming the features under which each configuration applies. In this way, the mapping between features and assets is consistent and traceable across the full SPLE life-cycle.

Conditional compilation is the prevailing mechanism for modelling variability in a code base [77] since it is natively supported by many programming languages. As an example, the popular C pre-processor *cpp*,[2] provides directives that allow inclusion of

[2] `http://gcc.gnu.org/onlinedocs/cpp/`

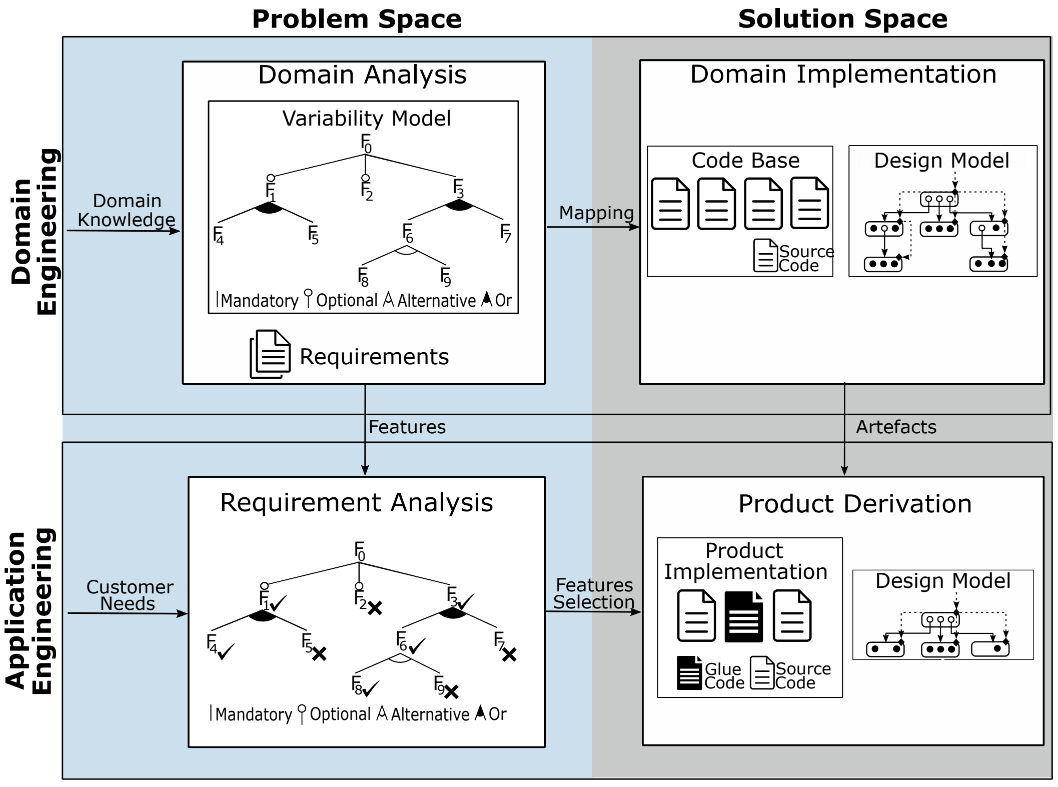


Figure 3.7: Software product family modelling in annotation-based SPLE.

header files (#include), line controls (#line), macros (#define), as well as conditional compilations (#ifdef, #endif).

Pre-processor annotations can be uniformly extended to any kind of textual and non-textual artefacts (*e.g.* design model). Indeed, approaches proposed in [78–80], along with Eclipse projects like fmp2rsm[3] and FeatureMapper[4] allow annotations and pre-processing static and behavioural models using UML stereotypes. Therefore, a functional model can be modelled by a 150% UML model.

Despite their undoubted simplicity, artefacts annotation has been heavily criticised [81–83] as its undisciplined use may compromise artefact quality and maintainability. In fact, pre-processor directives are usually not confined by the mechanism of the host language[5] and they can be applied with any arbitrary level of granularity. This implies that scattered annotations used in undisciplined ways may jeopardise separation of concerns,*i.e.* encapsulation. As a consequence, code implementing a feature could be disseminated across the code base and intermixed with other features' code. Moreover,

[3]`http://gp.uwaterloo.ca/fmp2rsm`

[4]`http://featuremapper.org/`

[5]For instance, *cpp* works on the basis of directives that control syntactic program transformation. Therefore it is not limited to *C* code.

Software Product Family Modelling Approach	Domain Engineering				Application Engineering	
	Feature Model	Functional Model	Product Family Architecture	Components	Product Architecture	Product Variant
FODA	**M**	**M**	**M**	S	M	S
In Annotation-based SPLE	**M**	**M**	**T**	**S**	—	S

Table 3.2: Product family modelling in annotation-based SPLE vs FODA.

as pre-processor annotations can be defined, and re-defined, in different places within the code, even in the case of feature code partitioned into distinct files, it is hard to understand when a determined code fragment will or will not be compiled. Although disciplined annotation [84, 85] and tools support [86, 87] mitigate these weaknesses, they do not solve all of them. Instead of integrating variability into existing artefacts, Pohl *et al.* [4] propose an orthogonal variability model (OVM) in which variability is documented using a dedicated model. Like feature models, OVM consists of variation points, variants, and constraints. However, unlike them, OVM describes variability using directed acyclic graphs, not trees.

Explicit links are drawn between variants and artefacts in domain engineering solution space. According to customer needs, variants and their associated artefacts are removed when not selected. However, the presence of a large number of features and artefacts leads to an explosion on the number of links [88].

As shown in table 3.2, like FODA, annotation-based SPLE does construct a feature model and a functional model (**M**). However, it does not construct a PFA, but just a template (**T**), called family model in pure::variants,[6] to organise the family solution space. A family model describes the solution space in terms of components, parts and source elements. A component is a named entity, which can be further decomposed into sub-components or parts, that in turn are built from source elements (**S**) from the code base. Thus, in application engineering only software for a product variant can be derived (S), but not its architecture.

[6]Without loss of generality, we use pure::variants notation henceforth.

3.4 Weaving-based SPLE

In contrast to annotation-based SPLE approaches, weaving-based ones model product families using positive variability: varying model parts are attached to a base model based on the presence of features (Fig. 3.8). Here the ability to decompose domain artefacts into manageable and comprehensive concerns[7] is crucial for realising positive variability. Aspect-oriented software development (AOSD) [89, 90] has been deployed as a practice to cope with cross-cutting concerns (i.e. aspects of a program that affect other concerns) which are difficult to model with traditional development approaches such as the object-oriented ones. AOSD focuses on the modularisation of cross-cutting concerns into separate functional units (aspects) along with their automated composition (weaving) into a working system. Aspect-oriented modelling (AOM) extends the idea of separation of concerns to the level of software models [91]. It applies AOSD concepts to automatically compose model parts into various base models. Many AOM approaches [92–100] have been proposed for both static and behavioural models (design models). As for AOSD, they all revolve around the concepts of model-based aspects and model weaving techniques.

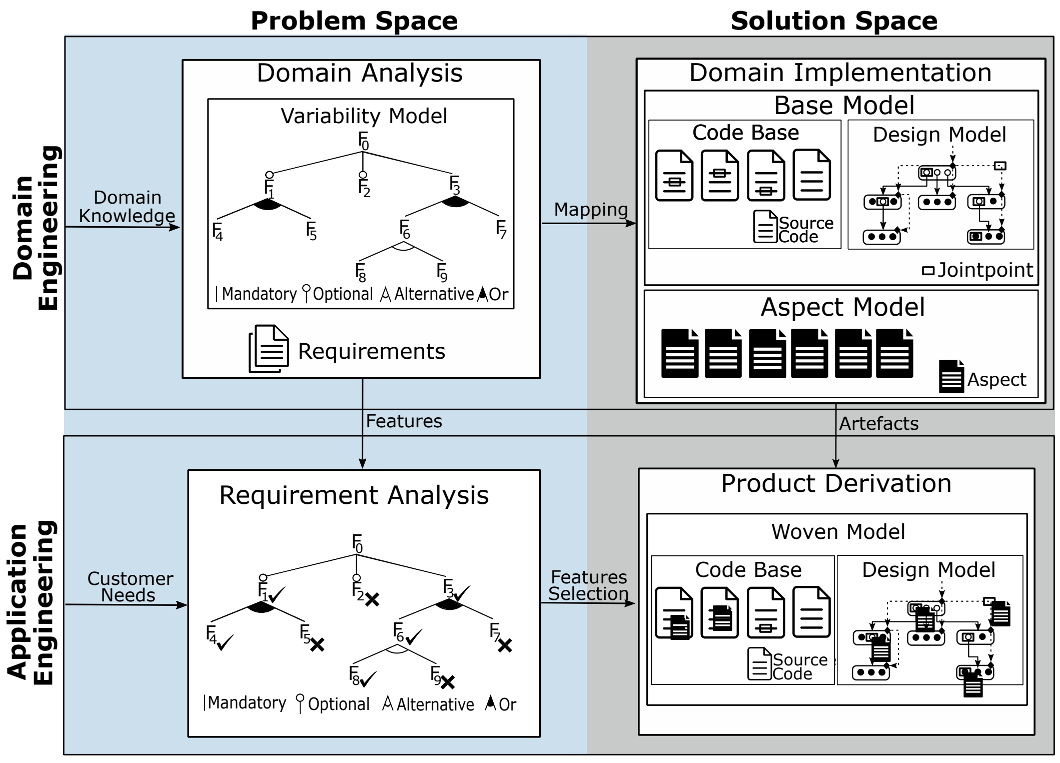


Figure 3.8: Software product family modelling in weaving-based SPLE.

[7]A concern is an area of interest or focus in the system.

A model-based aspect defines pointcuts and advices to specify where and how it affects the base model. A pointcut is an expression that matches several join-points (specific locations in the base model) and associates them with one or more advices. A common practice to specify model-based aspects is to use UML diagrams with wildcards (*e.g.* '*' to match any sequence of characters or '?' to match any sequence of arguments) [100]. During the model-weaving process, a product is derived by weaving advices into a base model each time a pointcut matches.

A model weaving approach allows a clear separation between mandatory and optional domain artefacts.[8] This improves traceability between artefacts in problem and solution space. However, many semantic problems regarding the aspect weaving process have been identified [102–105]. For instance, interference between aspects can invalid both structure and behaviour of a derived product. Interferences between aspects happen when a woven aspect modifies the selection of join-points by previously weaved aspects. This is usually due to aspects obliviousness. Indeed, if aspects are developed acknowledging each other, one can expect alteration of a base model. On the contrary, obliviousness between aspects invalidates their assumptions. As a result, the derived product may not necessarily reflect the corresponding feature configuration [106].

Software Product Family Modelling Approach	Domain Engineering				Application Engineering	
	Feature Model	Functional Model	Product Family Architecture	Components	Product Architecture	Product Variant
FODA	**M**	**M**	**M**	S	M	S
In Weaving-based SPLE	**M**	—	—	**S**	—	S

Table 3.3: Product family modelling in weaving-based SPLE vs FODA.

Compared to FODA, as depicted in table 3.3, weaving-based SPLE modelling approaches do not construct a PFA, nor a functional model. Indeed, base models and aspect models do not describe the behavioural and structural views of a whole product family. Aspect components (**S**) are woven only during product variant derivation (S) in the application engineering phase.

[8]Mandatory features representing a homogeneous concern may be specified as aspects and woven to a base model during the weaving process. A homogeneous cross-cutting concern applies the same advice to several join-points. By contrast, a heterogeneous one adds different pieces of advice to different join-points [101]. Homogeneous and heterogeneous optional features are always defined as aspects in order to localise variability.

3.5 Superimposition-based SPLE

Similar to AOM, superimposition [107–110] is a simple and language independent approach for modelling product families in which features are decomposed into independent artefact fragments and subsequently composed by merging their sub-structures on demand according to the provided feature configuration (Fig. 3.9).

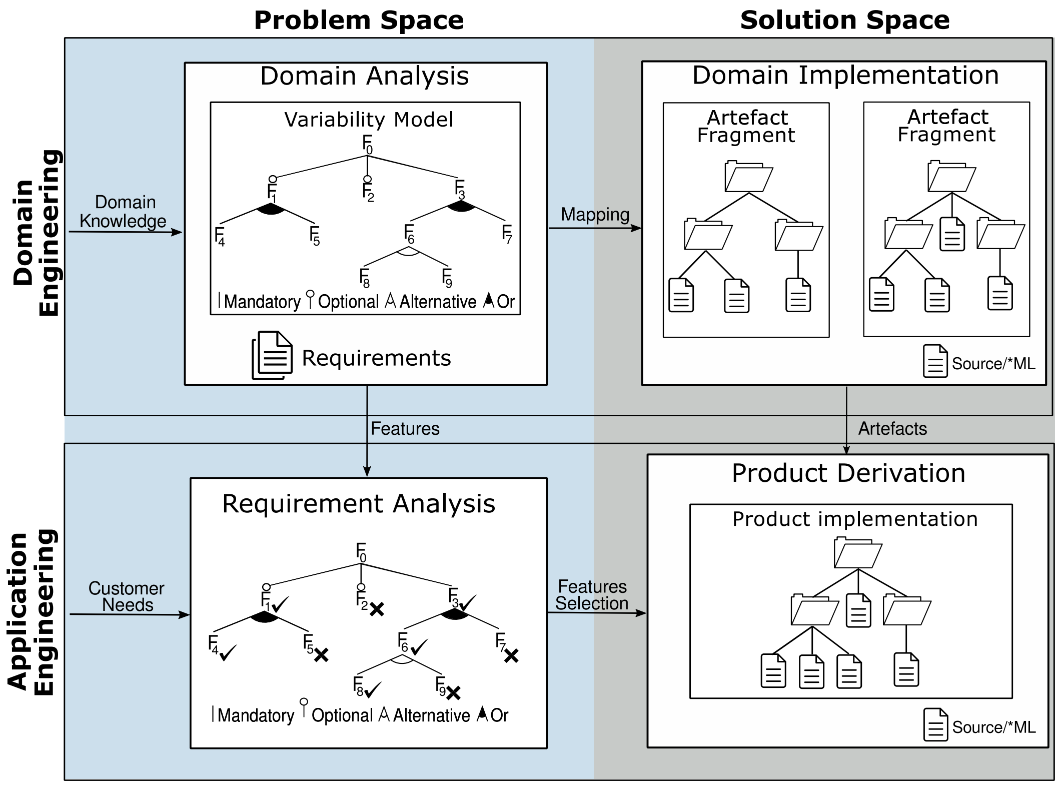


Figure 3.9: Software product family modelling in superimposition-based SPLE.

The internal structure of an artefact fragment is detailed by its relative feature structure tree (FST). Nodes of an FST are detailed by name and type. Given two FSTs, superimposition recursively merges nodes at the same level, having the same name and type. The resulting node has the same name and type as the merged ones. In case a node has no correspondent to be merged with, it is added as a child of the composed parent node. In other words, superimposition composes artefact fragments by merging their elements (via replacement or concatenation) on the basis of their structural and nominal similarities. Many approaches to feature composition rely on this technique [58, 108, 111, 112].

Jak, short for "Jakarta" [58], is a Java extension that enables superimposition by introducing the concepts of layer and class refinement. A layer defines the features a class belongs to, while a class refinement is a form of mixin-based inheritance [113, 114] that can be applied to different concrete super-classes. Each feature has its corresponding implementation directory, called feature module, which contains all classes and refinements. The semantic of feature modules composition is as follows:

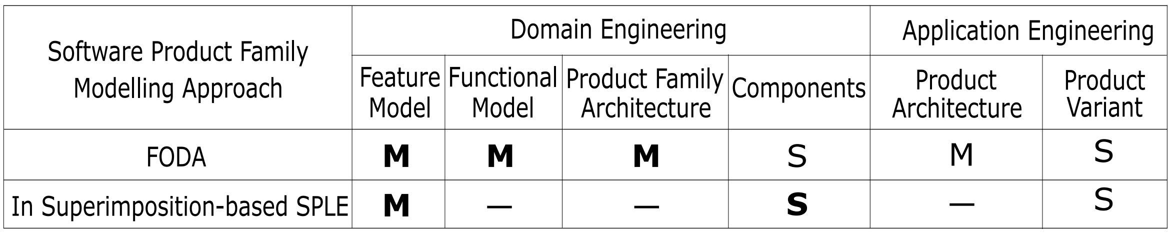
1. classes of any feature modules are put together in a single program;
2. each class is merged with its refinements;
3. in the case a refinement overrides a method, it can be called by the overriding method via the keyword `Super`.

To generalise the idea of superimposition and to abstract implementation details, several models have been developed. GenVoca [115, 116] is an example of such. Based on stepwise refinement, GenVoca is an algebraical model that allows constructing a product family by progressively adding details to base programs. Programs are constants represented as a set of classes. Feature refinements are functions that take in input a program and produce a feature-augmented one as return by adding or overriding attributes and methods of the target class. Names given to programs and feature refinements correspond to the features they implement. As a result, multi-featured application amounts to a named expression.

AHEAD [58], acronyms of Algebraic Hierarchical Equations for Application Design, and FEATUREHOUSE [117] scale up the idea of stepwise refinement to arbitrary domain artefacts. For instance, Apel *et al.* in [118] analyse the use of FEATUREHOUSE regarding superimposition of UML structural and behavioural models.

Although superimposition has been successfully used to model product families [99], its use is limited to particular scenarios. Indeed, artefact fragments are composed according to their structural similarities. Therefore, the substructure of a fragment modelling a feature must be a hierarchy of modules. Moreover, since superimposition is not aware of the underlying artefacts' semantics, in the case of name clashing it cannot perform automatic renaming. It follows that scenarios such as black-box composition, or the integration of structurally incompatible components, are less suited for superimposition and should be handled by alternative composition approaches such as component composition.

Being similar to weaving implies that superimposition-based SPLE approaches model the same domain engineering assets as in weaving-based SPLE. Therefore, its comparison to FODA of fig. 3.10 is the same as that in table 3.3.



Software Product Family Modelling Approach	Domain Engineering				Application Engineering	
	Feature Model	Functional Model	Product Family Architecture	Components	Product Architecture	Product Variant
FODA	**M**	**M**	**M**	S	M	S
In Superimposition-based SPLE	**M**	—	—	**S**	—	S

Figure 3.10: Product family modelling in superimposition-based SPLE vs FODA.

3.6 Δ-based SPLE

Weaving and superimposition, or in general SPLE modelling approaches based on step-wise refinement are referred to as feature-oriented programming (FOP) approaches. Delta-oriented programming (DOP)[9] [120–122] is a relatively new paradigm for developing product families. Similar to FOP, DOP supports a modular implementation of a product family by distinguishing between a base model and a set of delta models (Fig. 3.11). However, a delta encapsulates all the changes a configuration of features applies to a base model, including modification and removal of its parts (thus realising both positive and negative variability). Application conditions (i.e. propositional formulas over feature names) contained in each delta determine under which feature configurations the specified modifications have to be carried out.

Specifically, application conditions are Boolean constraints over features contained in the feature model, hence building the mapping between problem and solution space. This is in contrast to FOP where the mapping between the two spaces is relegated to external tools (e.g. *guidsl* in AHEAD [61]). During application engineering, delta models with valid application conditions are incrementally applied to the target core models. To minimise conflicts between two deltas targeting the same parts, first all additions, then all modifications and finally all removals are performed [119]. As previously stated, delta modelling is a relatively new paradigm for developing SPLs.

[9]Like FOP, concepts of DOP can be applied to other programming or modelling languages. In [119] a seamless delta-oriented model-driven development is proposed.

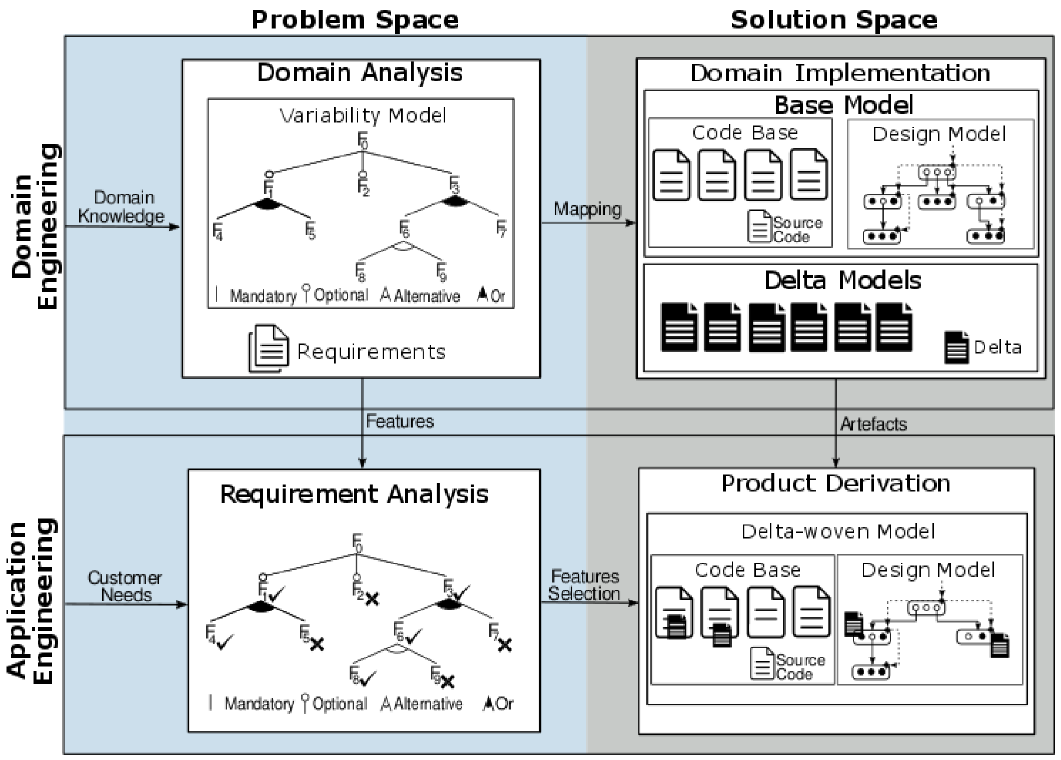


Figure 3.11: Software product family modelling in Δ-based SPLE.

Although this approach can model both positive and negative variability, its lack of tool support and evaluation in industrial scenarios [123] limits its application in modelling a functional model (e.g. 150% UML model) and a template for a product family. Therefore, compared to FODA (Fig.3.12), Δ-based SPLE modelling approaches only construct a feature model (**M**) and components as Δ-models (**S**), from which product variants are derived (S) according to the selected features.

Software Product Family Modelling Approach	Domain Engineering				Application Engineering	
	Feature Model	Functional Model	Product Family Architecture	Components	Product Architecture	Product Variant
FODA	**M**	**M**	**M**	S	M	S
In Δ-based SPLE	**M**	—	—	**S**	—	S

Figure 3.12: Product family modelling in Δ-based SPLE vs FODA.

3.7 ADL-based SPLE

Modelling approaches seen so far lack the construction of the key element of a product family: its architecture. For instance, while annotation-based approaches construct a 150% model, and weaving-based ones build a core model subsequently refined by aspect components, they do not construct a PFA.

Architectural description languages (ADLs) [124, 125] are formal languages that provide elements to represent the architecture of a software-intensive system in terms of components, their external interfaces (ports) and communication connections between them. As described in section 4.3, a software component is an encapsulated software element (which may contain sub-components) with explicit provided and required services (via interfaces) that can be independently deployed and composed without modification (according to a composition standard) by binding provided services with the matching required ones.

Of the many ADLs that have been developed [126–131] (see section 4.3.2 for details) only a few are explicitly designed to model the variability required in a product family (Fig. 3.13).

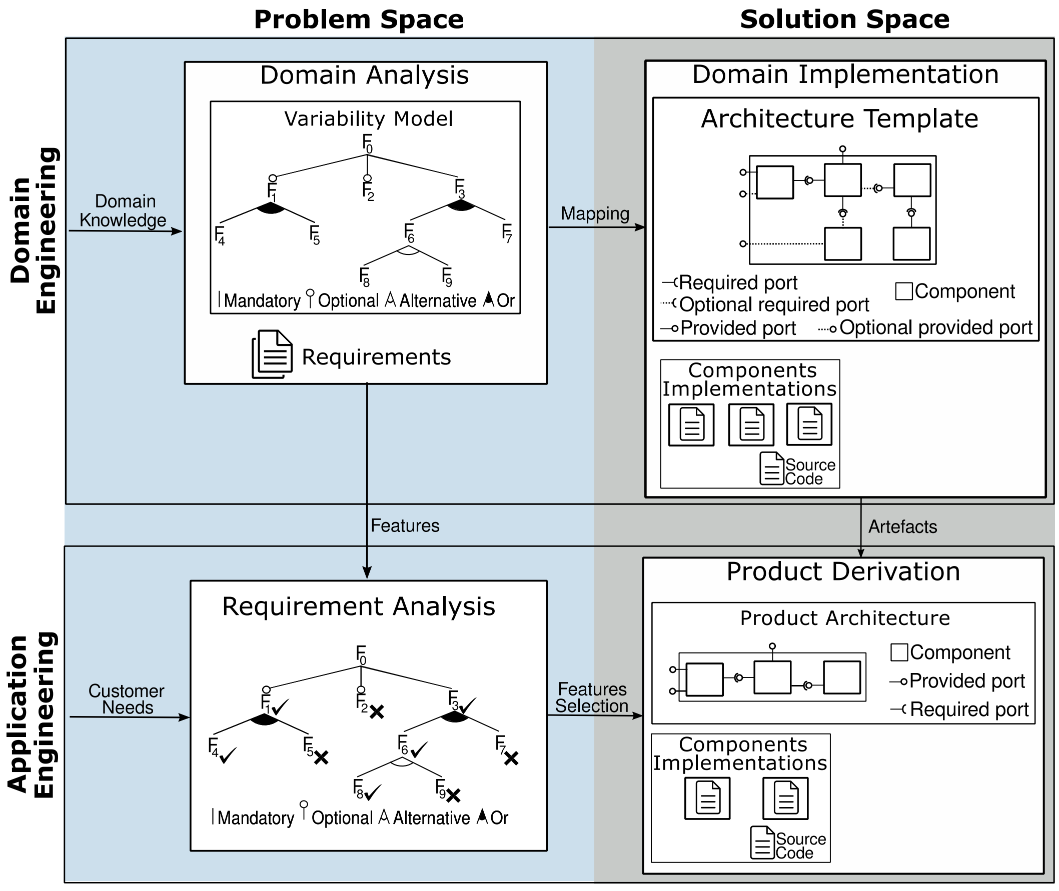


Figure 3.13: Software product family modelling in ADL-based SPLE.

In particular, we can distinguish between two categories of ADLs used in SPLE.

Firstly, there are ADLs that directly construct a PFA in solution space without mapping to elements in the variability model. Product variants are extracted from the PFA by providing configuration parameters, used to solve variability at both architectural and component level. Secondly, there are ADLs that construct a PFA and its components considering the variability model in the solution space, thus providing a mapping between the two spaces.

ADLs belonging to the first category are: Mae [132], xADL2 [133], MontiArc^{HV} [134], Com^2 [135], Koala [136, 137] and Koalish [138].

Mae [132] models variability in an architecture by means of *variant components*. A *variant component* is a component that encapsulates sets of alternative component instances, one of which it is used at a time. An instance is selected according to a property value associated with an instance. The *variant component* exposes the unique interfaces of all its components. Selection of a particular component may result in an illegal architectural configuration if another component uses interfaces that are not exposed by the selected variant component.

xADL2 [133] models architectures as instances of predefined XML schemas. Schemas capture basic architectural elements (i.e. component type, port type, connector type), along with their variability in terms of *optional elements*, *variant types*, and *optional variant elements*. Intuitively, an *optional element* is an element that may or may not be instantiated during product derivation. A *variant type* relates to the possibility of choosing one out of many element types. Optional elements and variant types can be combined to realise *optional variant elements*. Whether or not an element is instantiated depends on the satisfiability of its associated guard condition, which must be mutually exclusive in the case of variant elements.

Based on a generic ADL called MontiArc [139] and its framework MontiCore [140], MontiArc^{HV} [134] models components' variability via *variation points*. A component containing variation points is indicated as *variable component*. Each variation point specifies the set of possible variants (*i.e.* architectural elements) that realise it, along with their cardinality. Variants do not contain variation points but may enclose variable components. This allows encapsulating variability at any level of hierarchy (HV stands for hierarchical variability). A configuration specifies the selection of variants and the desired product's set-up.

Com^2 [135] expresses and handles variability both at architectural and at component level. At architectural level, Com^2 distinguishes between optional and alternative components to be bound to an interface. The inclusion of an optional component or selection of a variant one is evaluated upon a configurable variable. At component level, internal variability resides in source code and/or build scripts and it is described merely as configuration variables. During application engineering, features selection determines the actual value to assign to each configuration variable.

Koala (one of the few ADLs conceived and applied in industry) presents a strict separation between component development and configuration. Indeed, developers make no assumptions about the environment in which components will be used. In the same way, configurators are not allowed to modify the internals of a component to accommodate their configuration. A configuration is an interface-less component characterised by a set of parameters. A component may change its internal structure (via *cpp* directives) according to parameters received from a special, yet standard, kind of required interface called *diversity interface*. At architectural level, *switches* use parameters to re-route connections between interfaces. Koala can automatically remove unreachable components and implementation code therein.

Koalish extends Koala through a number of variability mechanisms stemming out from the product configuration domain [141]. For contained components, a component definition may include a number of parts, each defining a non-empty list of possible component types and their cardinality. To restrict the set of valid individual systems, Koalish allows the definition of constraints to express conditions that must hold for a component to be instantiated.

ADLs belonging to the second category are: Δ-MontiArc [142], Kumbang [143], LightPL-ACME [144], COSMOS*-VP [145], Plastic Partial Components [146] and ADLARS [147].

Based on MontiArc, Δ-MontiArc applies the concepts of DOP to model architectural variability. A core architecture developed using MontiCore is modified by delta models. A delta may add, remove, rename and replace ports, components and parameters. Moreover, it may connect or disconnect interfaces as well as remove unreachable ports and subcomponents. A feature is associated with one or more delta that realises it. For a particular features selection, a product configuration containing the set of associated delta is derived and then applied to the core architecture to derive a concrete product.

From the same authors of Koalish, Kumbang is a domain ontology that unifies feature and the architectural modelling of a product family. Features are modelled in Forfamel [148] (a feature modelling language), while architectures are expressed in Koalish. Kumbang specifies features' realisation by means of implementation constraints. Like any other constraint, an implementation constraint must hold for any family member derived from a Kumbang model.

LightPL-ACME is an Acme[10] [126] extension which introduces elements for SPL architecture description. The *productLine* element represent the SPL. Within it, *Feature* elements determine names and types (mandatory, optional etc.) of features composing the SPL. *Product* element represents a variant by specifying the features it includes. The mapping mechanism between architectural elements (built in ACME) and features is specified in the *System* element using the keyword *mappedTo*. ArchSPL-MDD [149] provides tool support to transform and refine models specified in LightPL-ACME, as well as a source generator to derive the skeleton of specified variant.

COSMOS*-VP uses aspects to extend COSMOS*[150], a generic ADL. Base-level components implemented as COSMOS* components are advised by aspect-level components. However, in order to reduce coupling and maximise reusability, aspect-level components do not specify which base-level component they advise. Binding between the two levels is mediated and encapsulated into aspect-connectors called CONNECTOR-VPS. According to features selection, CONNECTOR-VPS advise the base model with the required aspect.

Plastic Partial Components supports variability within components using the invasive software composition principles [151] as variability mechanism. As in AOP, invasive composition adapts and extends components at hooks (*i.e.* variability points) by transformation. A plastic partial component is a specialisation of component partially defined in the core architecture as a fragment box that hooks a set of code fragments, each implementing specific features. Cross-cutting features are implemented as aspects, while non cross-cutting ones into separate entities called *features*. During

[10]A generic software architecture description language.

application engineering, plastic partial components are completely defined by means of selection of aspects and/or *features* for each variability point.

ADLARS models a PFA in terms of *system*, *task* and *component* types. Systems are defined by a collection of task template instances. A task template is defined in terms of input interfaces, internal component architectures, an enumeration of the associated features along with their relationships to components. Instantiation of a task template requires the provision of the actual feature sub-set that the instance is required to support. The relationships captured within the task template definition enable the internal structure of the instance to be derived. Component instances, like task instances, are created by providing an actual feature sub-set. A component template embodies a collection of possible configurations and directly relates these to features in the feature model. In the application engineering phase, a system is derived by instantiating templates according to the specified feature set for the intended variant.

In ADLs, architectural units can be used to represent features and port connections to represent variations points. However, due to the binary nature of port connection (either connected, or disconnected), such mechanism can only represent *optional* and *alternative* variation points as first-class entities, not *or*.[11] For instance, Koala-based approaches use *switch* to redirect port connection at architectural level, while Com^2 uses *optional components*. The lack of the full set of explicit variation points implies that existing ADLs can only define a PFA in a limited number of cases.

An architecture defined by the aforementioned ADLs is therefore a template that has to be configured according to (i) the values of the provided parameters; and (ii) the selected features. Therefore, as shown in table 3.4, the modelled template (**T**), and its constituent components (**S**) are configured during production derivation to obtain a product variant architecture (T) and its implementation (S).

3.8 FX-MAN-based SPLE

As we have seen, all the existing SPLE approaches do not model the PFA and hence the architecture of an individual product variant (Fig. 3.1). Annotation-based SPLE

[11]Cardinality, aspects and deltas can model inclusive or, but cannot be considered first-class variability mechanisms as they participate at component level.

Software Product Family Modelling Approach	Domain Engineering				Application Engineering	
	Feature Model	Functional Model	Product Family Architecture	Components	Product Architecture	Product Variant
FODA	**M**	**M**	**M**	S	M	S
In ADL-based SPLE	**M**	—	**T**	**S**	T	S

Table 3.4: Product family modelling in ADL-based SPLE vs FODA.

uses a coding template for the PFA from which the code for an individual product can be derived, but the template is not an architecture. ADL-based SPLE defines an architecture template for the PFA which can be instantiated into a template for an individual product, but these architecture templates do not contain the full set of standard variation points. Consequently, all these approaches do not fully link variability to architecture. Apart from ADL-based SPLE, these approaches do not even have a first-class representation of variability.

As discussed in [12], first-class representation of variability and explicit dependencies between architectural elements and features are two key issues in modelling product families. Illustrated in Fig. 3.14, the approach we present in this thesis seeks to address both issues.

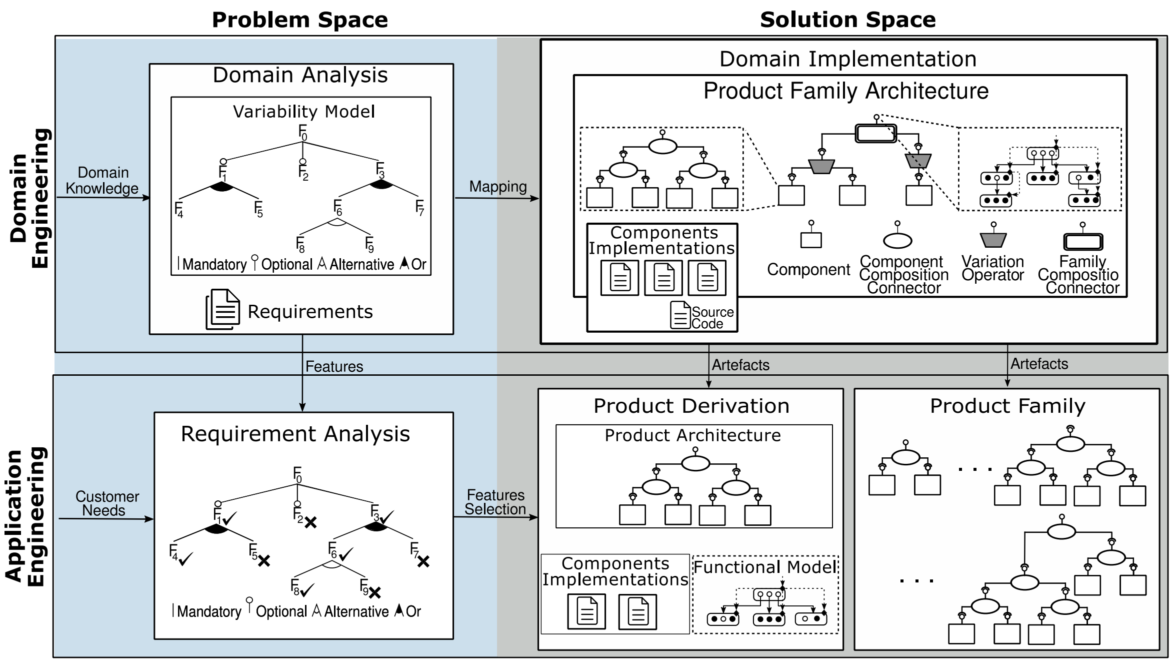
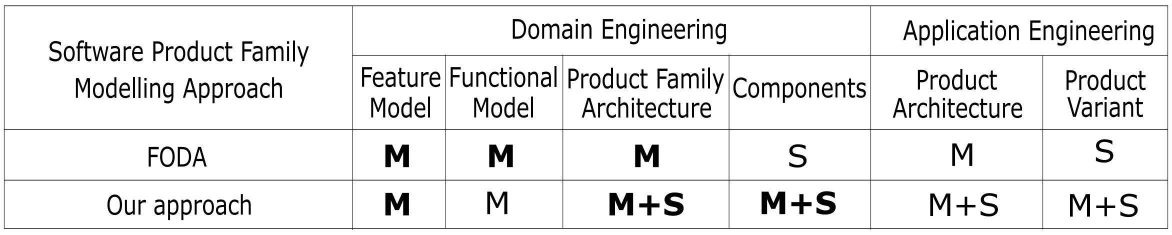


Figure 3.14: SPLE using our approach to modelling software product families.

Our proposal is based on a new component model, called FX-MAN [152], that can be used to construct a product family from components (that represent features and product sub-families), a full set of explicit variation operators (that represent variation points in a product family), and composition connectors that compose and coordinate sub-families. An architecture created in our model contains a family of (sub-families of) fully formed, executable products. A key property of our component model is that any architecture it defines can be mapped (automatically) to a state chart, alongside the associated activity chart. This means that to define a functional model, it is sufficient to construct an FX-MAN model according to the feature model for the given domain.



Software Product Family Modelling Approach	Domain Engineering				Application Engineering	
	Feature Model	Functional Model	Product Family Architecture	Components	Product Architecture	Product Variant
FODA	**M**	**M**	**M**	S	M	S
Our approach	**M**	M	**M+S**	**M+S**	M+S	M+S

Figure 3.15: Our-approach vs FODA.

As shown in Fig. 3.15 where, compared to FODA, our approach constructs a model and an implementation for the PFA (**M+S**) and for components (**M+S**), whilst a functional model is (automatically) derived from the model for the PFA (M). During application engineering, models and software for each product variant are derived from the constructed PFA (M+S).

The significance of having a model, as opposed to a template, for the PFA is that a model contains variability (as defined in the feature model), family and product structure (composition), as well as behaviour (components), whereas a template is just a structure of place-holders for components.

In the rest of the thesis we present the details of our approach.

3.9 Summary

The quality of domain assets determines the effectiveness and efficiency of the construction of variants in application engineering. However, creating all the domain assets is a challenging task in the first place. As a consequence, existing SPLE approaches do not model all domain engineering assets, but use pragmatic substitutes instead.

Annotation-based approaches use 150% models as templates, where products are configured by removing fragments that do not reflect feature selections. Despite its simplicity, artefact annotation has been heavily criticised, as its undisciplined use may compromise artefact quality and maintainability.

Feature-oriented modelling approaches gradually refine a base model with aspects (as for weaving-based approaches) or similar (sub)structures of artefacts (as for superimposition-based approaches) corresponding to a particular features selection. Such modelling approach allows a clear separation between mandatory and optional domain artefacts, so to improve traceability between problem and solution space. However, many semantics problems regarding the step-refinement process have been identified.

Similarly, Δ-based SPLE modelling approaches refine a base model (including modification and removal of its parts) by means of delta models according to the selected features. Although this approach can model both positive and negative variability, its lack of tool support and evaluation in industrial scenarios limits its application in modelling a functional model and a template for a product family.

Finally, ADL-based SPLE modelling approaches use architectural units to represent features and port connections to represent variations points. They enable a systematic way to build PFAs. However, due to the binary nature of port connection (either connected, or disconnected), such mechanism can only represent optional and alternative variation points as first-class entities, not the inclusive or.

In contrast, our proposal is based on a new component model, called FX-MAN, that can be used to construct a product family from components (that represent features and product sub-families), a full set of explicit variation operators (that represent variation points in a product family as in the feature model), and composition connectors that compose and coordinate sub-families. A PFA created in our model contains a family of fully formed, executable products, with the key feature that each architecture can be automatically translated to its equivalent functional model.

Chapter 4

Component-based Software Modelling

"If you wish to make an apple pie from scratch, you must first invent the universe."
— Carl Sagan

4.1 Introduction

Component-based software modelling (CBD)[1] [21, 153] aims to compose systems from pre-built software units, or *components*. A system is developed not as a monolithic entity, but as a composite of sub-parts that have already been built separately. Such an approach reduces production cost by composing a system from pre-existing components, instead of building it from scratch. It also enables software reuse, since components can be reused in many systems. Thus CBD promises the benefits of: (i) reduced production cost; (ii) reduced time-to-market; and (iii) increased software reuse. These benefits have long been sought after by the software industry. This chapter firstly introduces the foundations of CBD: *software component* in section 4.2 and *component model* in section 4.3. Secondly, since a component model defines syntax and semantics of components and their composition, in section 4.4 we provide a taxonomy of current models to identify the most suitable one for this work by evaluating their compliance to the idealised component life cycle's engineering phases.

[1]Various abbreviations have been used, CBD and CBSE being the main ones. We choose CBD.

4.2 Software Component

In mature domains such as automotive or electronic engineering, the term component refers to a small standard and exchangeable part. Transistors illustrate this point clearly: they are designed to be used in numerous applications, and are composed by plugging them into a circuit board. Moreover, a single transistor can be replaced by a new one, as long as the latter satisfies the same requirements.

Informatics engineering tries to imitate this idea by introducing the concept of software component. Although its general accepted definition is still lacking [23], the most widely adopted one is provided by Szyperski *et al.* [153], which define a component as '*a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parts.*' A component is then defined a unit of composition with explicit interfaces. However, nothing is said about their composition mechanism. Meyer *et al.* [154] specify a component as '*a software element satisfying the following conditions: it can be used by other software elements; it possesses an official usage description, which is sufficient for a client author to use it; it is not tied to any fixed set of clients.*' Here, a component is defined as a modular unit with a usage description that serves as implicit interface. As in the previous definition, nothing is said about their composition mechanism. Heineman *et al.* [21] fill this gap by describing a software component as '*a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*' This definition relies on a component model to define both components and composition. However, a component model is only implicitly specified as a composition standard.

From the aforementioned component definitions (and others [155–158]), a generally accepted view has emerged: *a component is a software unit with provided services (lollipops) and required services (sockets), which constitute its interfaces and the only access point* (left Fig. 4.1). Intuitively, provided services are operations performed within a component, while the required ones determine the operations needed from the environment, usually to yield the provided services.

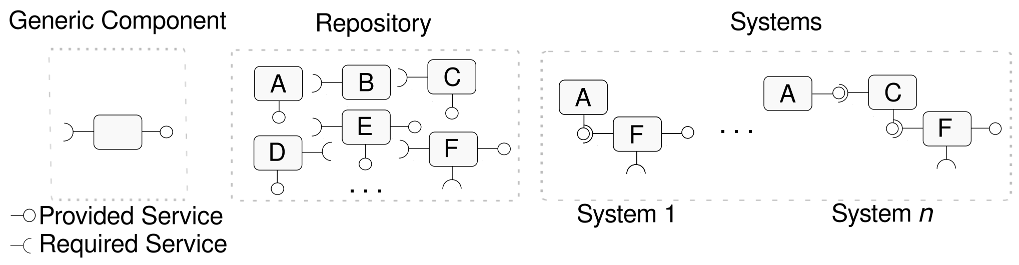


Figure 4.1: Composing generic software components.

Generic components (retrieved from a repository[2] (centre Fig. 4.1)) can be composed into a system by connecting provided services with matching required ones, and vice versa (right Fig. 4.1). The interface of a system can be derived from the ones of its sub-components.

Different types of components are defined as distinct instances of the generic component, together with their corresponding composition mechanisms. As defined by Heineman *et al.* [21], such definitions are provided by the component model, which constitute the cornerstone of every component-based software engineering approach.

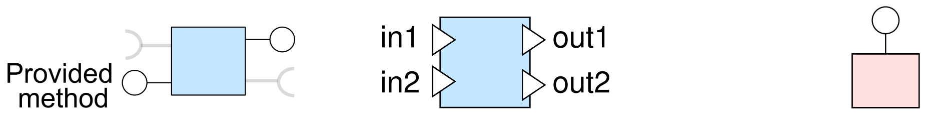
4.3 Component Model

The term 'component model' is used by Luer *et al.* [159] to refer to '*a combination of a component standard that governs how to construct individual components and a composition standard that governs how to organize a set of components into an application and how those components globally communicate and interact with each other.*' Therefore, a component model determines: (i) syntax and semantics of components, or rather their definition, representation and construction; (ii) the composition mechanism for composing the defined components. It follows that being compliant to a component model is one of the properties that distinguish components from other forms of packaged software [160].

In the last twenty years, both academia and industry have developed a large number of component models [22]. Nevertheless, based on component type, we can distinguish between three categories of component models: models based on *objects* (Fig. 4.2(a)), models based on *architectural units* (Fig.4.2(b)) and models based on *encapsulated*

[2]The use of a repository for building systems suggests that their construction is essentially bottom-up (composition) rather than top-down (de-composition).

components (Fig. 4.2(c)).



(a) An object (b) An architectural unit (c) An encapsulated component

Figure 4.2: The three main types of components currently used in practice.

4.3.1 Models based on objects

In a component model based on objects, an object resembles the generic component (left Fig. 4.1) expect that it does not have, or show, any required services (hence the blurred sockets in Fig. 4.2(a)). Objects compose by method delegation, *i.e.* direct method calls, or direct message passing. Current models belonging to this category are: JavaBeans [161–163], Enterprise JavaBeans (EJB) [164–167], COM [168–170], CCM [171, 172], .NET [173, 174], Web-Services [175–177] and OSGi [178–180].

In JavaBeans a component is a bean, which is just any Java class that has *methods*, *properties* and *events* hosted by a container such a BeanBox [181]. A bean 'composes' with another bean by sending a message through delegation of events. The bean container automatically generates, compiles, and loads event adaptor classes for logistics of events.

In EJB a component is a Java object hosted and managed by an EJB container within a J2EE server [182]. For an EJB, its Java class defines the methods of the bean, while its interface exposes bean capabilities for (remote) client applications to access the bean (over a network). There are 3 kinds of EJBs: i) entity beans, which model business data; ii) session beans, which model business processes; iii) message-driven beans, which model message-related business processes. Enterprise beans are composed (in the EJB container) by method and event delegation via interfaces.

In COM (Component Object Model) a component is a unit of compiled code on Windows Registry. Services in a component are invoked via pointers to the functions that implement them. For each service provided there is an interface (a COM component can implement multiple interfaces), specified in Microsoft IDL. Every component must implement an IUnknown interface, which relates to the COM component's life cycle. COM components are composed by method calls via interface pointers.

In CCM (CORBA Component Model) a component is a CORBA meta-type hosted by a CCM container on a CCM platform such as OpenCCM [183]. A CORBA meta-type is an extension and specialisation of a CORBA Object [184]. Component interfaces are made up of ports: *facets* (provided services), *receptacles* (required services), *event sources* and *event sinks.* CCM components are assembled by method and event delegations in such a way that: (i) facets match receptacles; (ii) event sources match event sinks.

In Microsoft .NET a component is a binary unit called *assembly* supported by a Common Language Runtime (CLR). A .NET component is made up of metadata and code specified in Intermediate Language (IL). The metadata contains the description of assembly, types and attributes, while the IL code is compiled and executed by the CLR. .NET components are composed by directly referencing other assemblies via direct method calls within their IL code.

Web services are web application components that can be published, found, and used on the Web. A web service contains: (i) an interface in WSDL (Web Service Description Language), which describes the functionalities the web service provides; (ii) a binary implementation (the service code). Web services composition is realised by orchestration [185], or rather a coordination of their invocations through SOAP (Simple Object Access Protocol) [186] or JSON (JavaScript Object Notation) [187, 188] messages.

In OSGi (Open Services Gateway Initiative) components are Plain Old Java Objects (POJOs) within *bundles.* A bundle is physically distributed as a JAR file, with a manifest file containing its meta-information like imported and exported packages (i.e. services). OSGi bundles do not compose, but POJOs within them do via direct method invocation.

4.3.2 Models based on architectural units

In a component model based on architectural units, an architectural unit is a primary unit of computation and/or data, which interacts with other components via a set of ports. An architectural unit is just the same as a generic component, except that it uses ports instead of services. A port is a feature of a component that specifies a distinct interaction point between itself and its environment. Current models belonging to this category are: Acme-like ADLs [189], UML 2.0 [190–192], SOFA 2 [193, 194], Kobra [195, 196], Palladio [197, 198], PromoCom [199, 200], Fractal [201, 202].

Acme [126] is a prototype architecture description language (ADL). It typifies first-generation ADLs, *e.g.* Darwin [203, 204], UniCon [205], Wright [127], SCA [206] and ArchJava [207]. In Acme-like ADLs, a component is an architectural unit that represents a primary computational element and data-store of a system. Interfaces are defined by a set of ports. Each port identifies a point of interaction between the component and its environment. A component may have multiple interfaces by using different types of ports. In Acme-like ADLs, components are composed (into composite components or systems) by connectors via port connection.

In UML 2.0, a component is an architectural unit with input ports (required services) and output ports (provided services). Composition, via port connection, is achieved within a component by using assembly connectors, whereas between components by delegation connectors (which realise *port forwarding*).

In SOFA 2, a component is an architectural unit determined by its frame and architecture. The frame defines provided and required interfaces, as well as properties of the component. The architecture describes the component's internal structure. SOFA components are composed via connectors by using the following communication styles: (i) remote procedure call; (ii) asynchronous message passing; (iii) streaming; (iv) shared memory.

In KobrA, a component is defined by a set of UML diagrams, which describe its specification and implementation. The former specifies what a component does (its interfaces), while the latter how it does it. KobrA components are composed by direct method calls.

In Palladio, a component can be just an abstract specification and does not necessarily have an implementation. A component consists of an interface (service signatures) and optional behavioural specifications. In ascending order of specifications concreteness, Palladio defines three basic component types: (i) *provided type*; (ii) *complete type*; (iii) *implementation type*. Composition is realised via connectors at ports level.

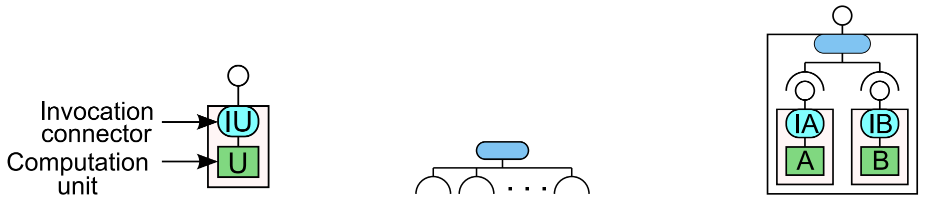
ProCom is a two-layered component model. At the system layer, *ProSys* components are subsystems, or rather active, distributed components composed via explicit (asynchronous) message channels. At the subsystem layer, Prosys components are internally modelled by *ProSave*. A ProSave component is a unit of functionality, designed to encapsulate low-level tasks. It exposes its functionality via services, each consisting of: (i) an input group of ports, which contain the activation trigger and required data; (ii) an output group of ports, which make available the data produced. ProSave distinguishes between data and control flow. Therefore it uses several connectors for more elaborate control: *control fork*, *control join*, *control selection*, *control or*, *data fork*, and *data or*.

In Fractal, a component is a unit of encapsulation and behaviour. It consists of two parts: (i) *content*, which is made up of a finite set of sub-components; (ii) *membrane*, which encapsulates the content and contains both internal and external interfaces of the component. Components are composed via port bindings. The latter can either be primitive (if the bound interfaces are internal) or composite (if the bound interfaces are external). Composite binding is embodied in a binding object which itself takes the form of a Fractal component.

4.3.3 Models based on encapsulated components

In component models based on encapsulated components, components enclose the computation of the services they provide, without external functional dependencies. This is depicted in Fig. 4.2(c), where a component belonging to this category only has a provide port (lollipop), which exposes the services it provides. As a direct consequence, an encapsulated component, like a transistor, can be easily replaced since it is not coupled, leading to a significant gain in reusability. Moreover, it can be designed, implemented and verified independently by different teams. Encapsulated components compose via exogenous connectors, which coordinate the control flow between them.

The only component model belonging to this category is X-MAN [208, 209]. An X-MAN *component* can be either *atomic* or *composite*. An atomic component (Fig. 4.3(a)) is a unit of composition and computation, which exposes a set of *services*. Its *computation unit* (U) fully encapsulates the (functional) implementation of the services it



(a) Atomic component (b) Composition connector (c) Composite component

Figure 4.3: Exogenous composition in X-MAN.

exposes via the *invocation connector* (IC). Atomic components are (recursively) composed into composite ones by means of *composition connectors* (Fig. 4.3(b)). Composition connectors coordinate the execution of the components they compose: a *sequencer* (*SEQ*) provides sequencing, while a *selector* (*SEL*) branching.

Single components in X-MAN can be adapted by *adapters* such as *loop* (*L*) and *guard*.[3] The former provides looping, while the latter gating. Invocation and composition connectors form a hierarchy where each composition preserves encapsulation; the resulting composite component (Fig. 4.3(c)) behaves like an atomic one.

Input and output are both carried through data channels associated with each component's service. Data routeing can be horizontal (within a composite) and vertical (among components). Furthermore, a data channel can be initialized with a default value and has two possible read policies: destructive and non-destructive.

4.4 A taxonomy of current component-models

The cornerstone of any component-based development (CBD) approach is its underlying component model, which defines syntax and semantics of components and their composition mechanism. Precisely, what components are and what desirable properties they should have has been discussed at length [21, 153, 154, 210] and summarised in table 4.1.

Firstly, system development should start from pre-existing components stored (in design phase) and subsequently retrieved (in deployment phase) from a shared *repository*.

Secondly, in order to safeguard reusability, components should be built and used

[3]By providing sequencing, branching and looping, X-MAN is Turing complete.

Desideratum	**Design Phase**	**Deployment Phase**
Components should pre-exist	Deposit components in repository	Retrieve components from repository
Components should be produced independently	Use builder	——
Components should be deployed independently	——	Use assembler
It should be possible to copy and instantiate components	Copies possible	Copies and instances possible
It should be possible to build composites	Composition possible	Composition possible
It should be possible to store composites	Use repository	——

Table 4.1: Desiderata for component-based software development.

by different parties. Therefore, during the design phase, components are built (using a *builder* tool) and deposited in a repository. During the deployment phase, using an *assembler* tool, a different party composes instances of components retrieved from the repository.

Thirdly, it should be possible to copy (in design phase) and instantiate components (in deployment phase), so to maximise their reuse.

Fourthly, it should be possible to build (in a systematic way) composite components both in design and deployment phase. This requires that composite components can also be deposited in and retrieved from a repository.

Lastly, it should be possible to perform composition in both design and deployment phases. This implies the flexibility of realising composition entirely in one phase, or partially in both.

An idealised component life cycle (depicted in Fig. 4.4), derived from the aforementioned desiderata, has been proposed by Lau *et al.* [23].

In the design phase, by means of a *builder* tool, new components are constructed and then deposited in a *repository* (e.g. A in Fig. 4.4). Such components can be retrieved from the repository, and further composed into well-defined *composites* (e.g. BC in Fig. 4.4) using suitable *composition operators*, ideally supported by a composition theory. Composites should be stored in, and retrieved from the repository in order to be further composed like any other component.

In the deployment phase an *assembler* tool can be used to compile and assemble components retrieved from the repository into a deployable system (e.g. A, B, D and BC in Fig. 4.4). As in design-phase, composition should be carried out via

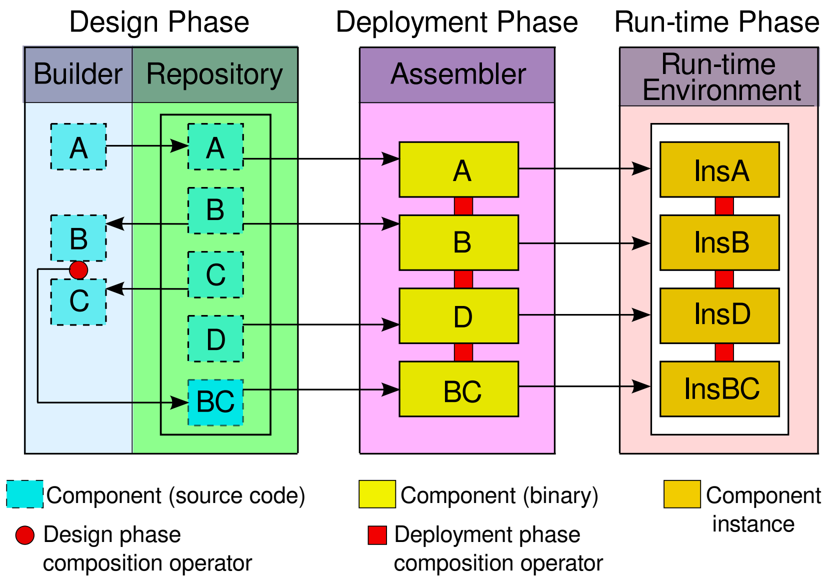


Figure 4.4: An idealised component life cycle.

composition operators. Such operators should allow coordination between components to fulfil application specific requirements. Resulting composite components should have *interfaces* (generated during the composition process) that allow them to be instantiated and executed in the run-time phase.

In the run-time phase an assembled system is deployed and managed in a run-time environment (e.g. instances of A, B, D and BC in Fig. 4.4). Although there is no further composition in this phase, it should be possible to perform some kind of adaptation on component's *instances* at run-time.

The idealised component life cycle provides the basis for comparing and classifying composition support of existing component models. For instance, we can distinguish between component models that do not have composition in the design phase and those that have it in the deployment phase. Fig. 4.10 gives the five categories that cover all the eighteen major existing component models previously described.

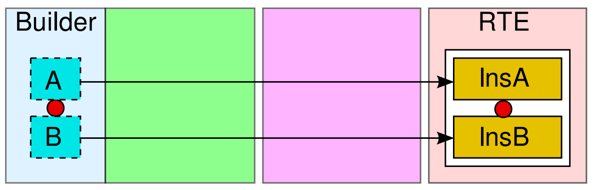


Figure 4.5: Category 1: design without repository.

In category 1 (*design without repository* – Fig. 4.5), the lack of a repository implies lack of reuse. That is, components are constructed from scratch and composed only in the design phase. All simple Acme-like ADLs belong to this category, as do models such as UML 2.0, PECOS, which are based on Acme-like ADLs.

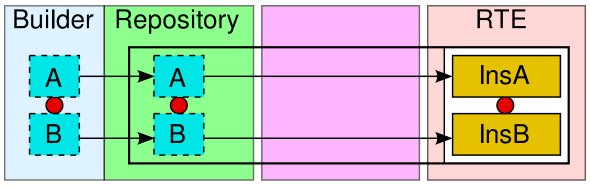


Figure 4.6: Category 2: design with deposit-only repository.

In category 2 (*design with deposit-only repository* – Fig. 4.6), new components constructed in design phase can be deposited in, but not retrieved from a repository. This implies that a deposited component can only be referenced by a newly created one. Therefore, composition is possible (only in the design phase), but composite cannot be retrieved as they do not have their own identity. Component models belonging to this category are EJB, COM, .NET, CCM and Web Services.

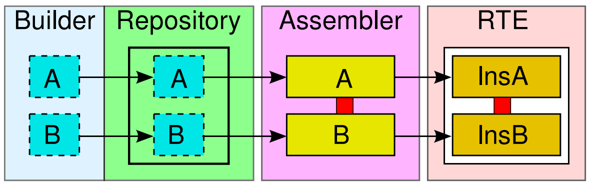


Figure 4.7: Category 3: deployment with repository.

In category 3 (*deployment with repository* – Fig. 4.7), new components can be deposited in a repository, but cannot be retrieved from it. However, composition is not possible in the design phase. Therefore, the repository can only contain atomic components. In deployment phase, components are retrieved from the repository, compiled and composed. The only model belonging to this category is JavaBeans.

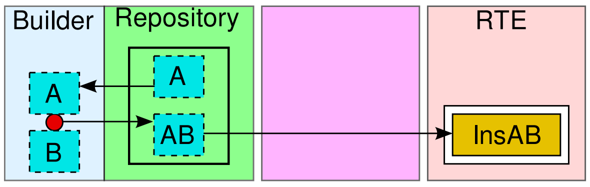


Figure 4.8: Category 4: design with repository.

In category 4 (*design with repository* – Fig. 4.8), components constructed in design phase can be deposited in and retrieved from a repository. Retrieved components can be composed. Resulting composites can be deposited and further composed (only in the design phase). SOFA 2, KobrA, Palladio, SCA and ProCom belong to this category.

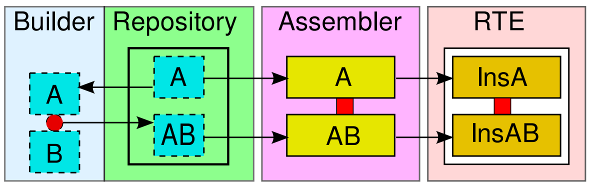


Figure 4.9: Category 5: design and deployment with repository.

In category 5 (*design and deployment with repository* – Fig. 4.9), components can be retrieved, composed and deposited in a repository at design time. Moreover, components can be further composed in the deployment phase. The only model belonging to this category is X-MAN.

It is interesting to note that X-MAN meets all the requirements of the idealised life cycle (Fig. 4.10), hence the CBD desiderata. Indeed, models in category 1 are focused on designing components from scratch, rather than reusing existing ones. Models in categories 2 and 3 use repositories, but they behave differently from those in category 4, in that the former stores binaries, whereas the latter stores units of design.

Moreover, X-MAN is the only model that uses encapsulated components and exogenous connectors as composition mechanism. Encapsulation facilitates reuse as it removes coupling between components. Exogenous connectors enable systematic hierarchical composition by preserving encapsulation.

Category	**Component Models**	**Design**				**Deploy**
		Deposit-N	Retrieve	Compose	Deposit-C	Compose
Design without Repository	Acme-like ADLs UML 2.0, PECOS	×	×	✓	×	×
Design with Deposit-only Repository	EJB, OSGi, Fractal COM, .NET, CCM	✓	×	✓	×	×
Deployment with Repository	JavaBeans, Web Services	✓	×	×	×	✓
Design with Repository	SOFA 2, Kobra SCA, Palladio, ProCom	✓	✓	✓	✓	×
Design & Deployment with Repository	X-MAN	✓	✓	✓	✓	✓

Figure 4.10: Categories based on composition [211].

By contrast, models based on objects and architectural units are both lacking in encapsulation and compositionality. Objects encapsulate data, but not control or computation. They are not compositional as the 'composition' of two objects by direct message passing is not supported by a composition theory: the composition result is not a single object, but just the two objects calling each other's methods. Architectural units are compositional and can encapsulate data, but they do not encapsulate control or computation. Architectural units compose via their ports by indirect message passing and have a simple composition theory, which does not support systematic composition. In particular, it is usually defined at the level of ports and used for type checking connected ports [212], rather than at the level of whole components. Message passing relies on, and induces, close coupling between components. This hampers their reuse, in particular the reuse of composite components.

Finally, encapsulation has the potential to counter *complexity.* In X-MAN encapsulation occurs at every level of composition, implying that larger and larger composites can be composed regardless of their size or complexity.

For those reasons and due to the high level of complexity and scalability required by a software product line, X-MAN has been chosen as the candidate component model in this research.

At its current stage, X-MAN does not include the concepts of variability, therefore it can only deal with the construction of one system at the time. A component model for product family based on X-MAN must therefore introduce such concept along with new exogenous connectors to deal with it.

4.5 Summary

The cornerstone of any component-based software modelling approach is a component model, as it defines syntax and semantics of components and their composition. Current component can be categorised according to their compliance to the idealised component life cycle [23].

The resulting taxonomy shows that X-MAN is the only component models that fully satisfies the idealised component life cycle desiderata, and posses the characteristics (encapsulation and exogenous composition) to potentially deal with the complexity and scalability required when modelling a software product family.

Chapter 5

A Component-based Approach to Modelling Software Product Families

"Never think that lack of variability is stability. Don't confuse lack of volatility with stability, ever."
— Nassim Nicholas Taleb

5.1 Introduction

A component model for constructing product families must define a family of architectures by incorporating variation points, as well as composition mechanisms for combining (sub)families of architectures into larger ones. In this chapter we introduce FX-MAN, a component model based on X-MAN to modelling software product families.

The basic idea of FX-MAN is that it defines: (i) basic component-based architectures that correspond to features; (ii) variations of sets of basic architectures; (iii) composition of sets of basic architectures into a product family. This is illustrated by the three levels in Fig. 5.1.

Section 5.2 describes X-MAN [208, 209], its new *aggregator* connector and how it is used to construct basic component-based architectures, which are intended to implement features in the final products.

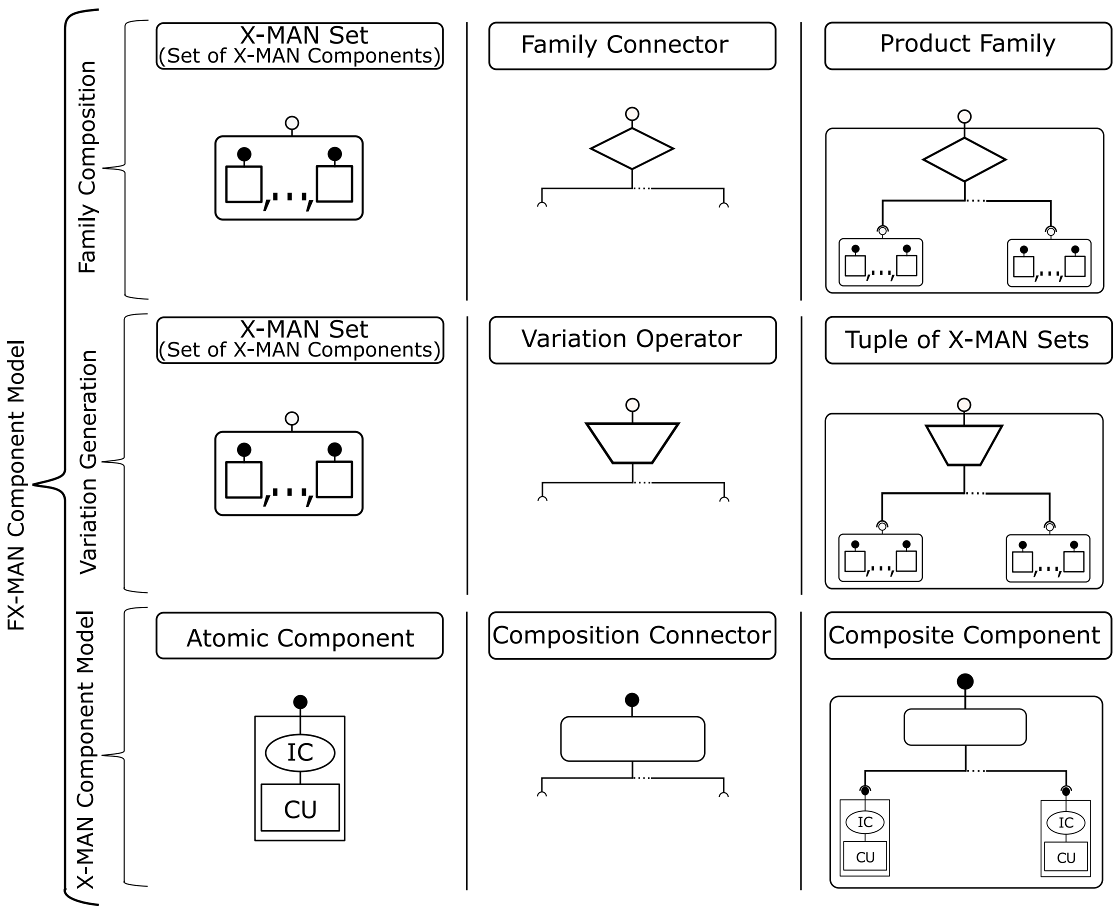


Figure 5.1: FX-MAN overview.

A set of X-MAN architectures is then a family of product parts. We call such a set an *X-MAN set*. Variations of X-MAN sets are constructed by *variation operators* that correspond to standard variation points in feature models, namely *OPT* (optional), *ALT* (alternative, or exclusive *or*), and *OR* (inclusive *or*). Their semantics is defined in section 5.3.

Tuples of X-MAN sets that represent variations generated by variation operators can be composed into a product family by *family connectors*. As detailed in section 5.4, such a family contains all the possible products, containing all possible variations as defined in the feature model.

Finally, section 5.5 details how FX-MAN can be used to model product families architectures, which are isomorphic to the related feature model, with explicit variability and that can be recursively composed to model families of families.

A key property of FX-MAN is that any architecture it defines can be mapped (automatically) to a state chart, alongside the associated activity chart. This means that to define a functional model, it is sufficient to construct an FX-MAN model of the product family according to the feature model for the given domain. To show how FX-MAN architectures define state and activity charts, each section also details how the respective level in Fig. 5.1 defines these charts.

5.2 X-MAN component model

As introduced in section 4.3.3, an X-MAN *component* can be either `atomic` or `composite`. An `atomic` component (A in Fig. 5.2) is a unit of composition and computation, which exposes a set of `services`. Its `computation unit` (CU) fully encapsulates the

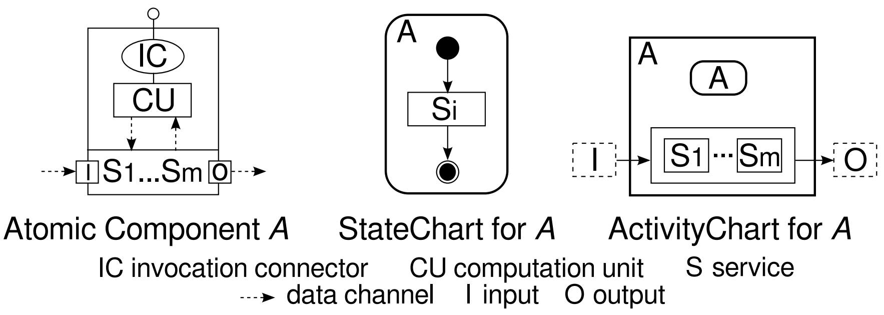


Figure 5.2: An atomic component and its functional model.

(functional) implementation of the services ($S_1, \ldots, S_m$) it exposes via the `invocation connector` (IC). Its behaviour can be specified in the language of state charts, as shown in the state chart[1] for A in Fig. 5.2: when a service $S_{i|1\leq i\leq m}$ is invoked, a transition from the initial state to the one in which S_i is computed takes place. Once computation ends, the end state is reached. Data to and from the CU is provided and retrieved via `service` parameters, as shown by the activity chart[2] for A, where parameters are represented as external activities, while services are represented as activities. The state chart for A appears as a control activity for the activities in the activity chart for A.

Atomic components are composed into `composite` components by means of `composition connectors`. Composition connectors coordinate the execution of the components they compose: a `sequencer` (*SEQ*) provides sequencing, while a `selector` (*SEL*) provides branching.

[1]We denote a state chart by a rounded box with a bold outline.

[2]We denote an activity chart by a box with a bold outline.

Figs. 5.3 and 5.4 show two composite components Q and B built using *SEQ* and *SEL* respectively to compose n atomic components $A_1 \ldots A_n$. The state chart for

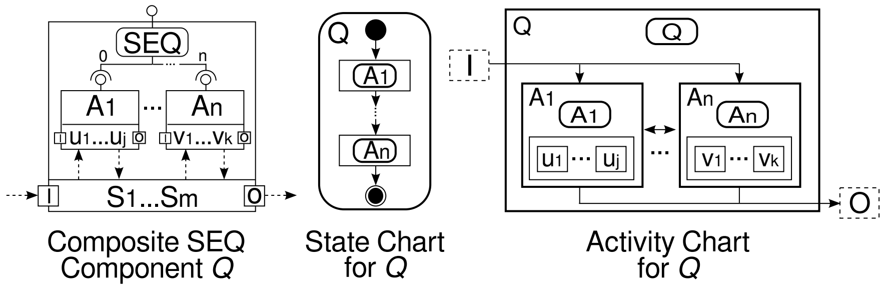


Figure 5.3: A composite component with sequencer and its functional model.

Q is composed from the state charts for $A_1 \ldots A_n$ by sequencing them in the order specified in *SEQ*. Similarly, the state chart for B is composed from the state charts for $A_1 \ldots A_n$ by branching according to the conditions in *SEL*.

The activity charts for Q and B are composed from those of $A_1 \ldots A_n$. Data flow among activities mirrors the data flow among the corresponding services. The control activity B receives a control flow input needed to perform branching decisions.

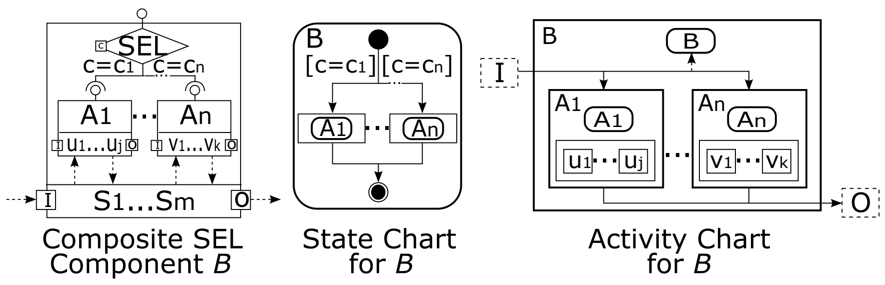


Figure 5.4: A composite component with selector and its functional model.

In order to deal with the variability introduced by the *OR* operator at the *variation generation* level (Fig. 5.1), we have defined an `aggregator` connector (*AGG*), which aggregates in a new composite component the services exposed by its sub-components. An aggregated component effectively provides a *façade* to the aggregated services.

In Fig. 5.5, the component G is built by aggregating the services exposed by components $A_1 \ldots A_n$. Like B in Fig. 5.4, the state chart for G is composed by branching among the state charts for $A_1 \ldots A_n$, but with a condition on the choice of service. Its activity chart is composed from the activity charts for $A_1 \ldots A_n$.

Single components in X-MAN can be adapted by `adapters` such as `loop` (L) and `guard` (G). The former provides looping, while the latter gating. Fig. 5.6 shows a

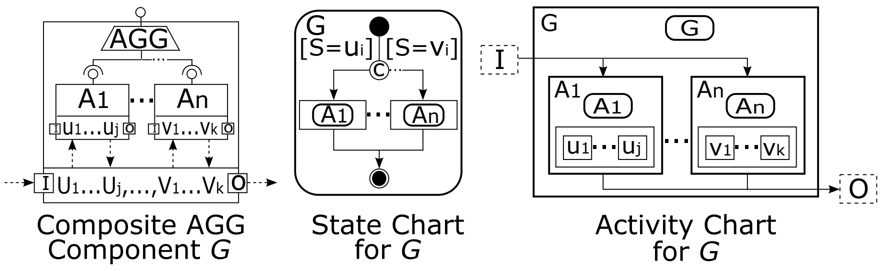


Figure 5.5: A composite component with aggregator and its functional model.

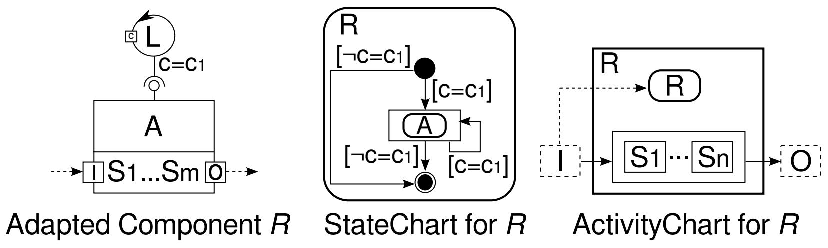


Figure 5.6: A component with loop and its functional model.

component A adapted by L into R. The state chart for R is composed from the state chart for A by looping the latter until condition c is verified; failing that, the end state is reached. In its activity chart, the loop condition is shown as a control flow coming from the external activity I.

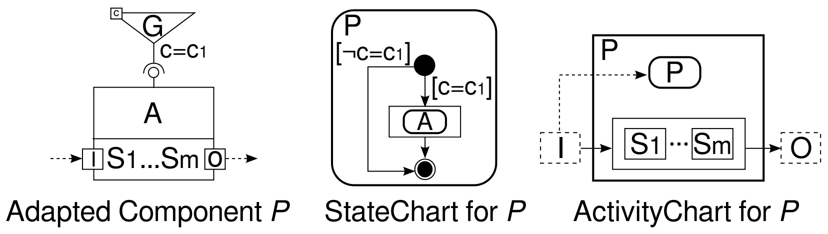


Figure 5.7: A component with guard and its functional model.

Finally, Fig. 5.7 shows a component A adapted by G into P. As for R, the state chart for P is composed by the state chart for A by entering its initial state only if the condition c is satisfied; if not, the end state is reached. In its activity chart, the guard condition is depicted as control data flow coming from the external activity I.

5.3 Variation generation

We can use X-MAN to build a set of X-MAN architectures (an X-MAN set), store it in the repository of our tool and retrieve it whenever we need it. An X-MAN set is any set of (well-formed) X-MAN architectures. More precisely, an X-MAN set F is such

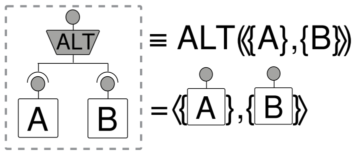


Figure 5.8: Example for *ALT*.

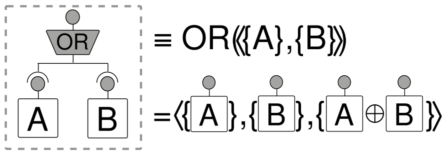


Figure 5.9: Example for *OR*.

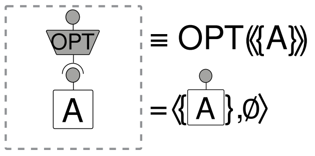


Figure 5.10: Example for *OPT*.

that:

$$F \in \mathcal{P}(\mathcal{S})$$

where $\mathcal{S}$ is the set of all possible well-formed X-MAN architectures, and $\mathcal{P}(\mathcal{S})$ is the power set of $\mathcal{S}$. Note that because each element of an X-MAN set is an individual X-MAN architecture, a set of X-MAN sets may not be well-formed: an X-MAN set F inside another X-MAN set will define a valid X-MAN architecture only if F is a singleton set. For this reason, we use tuples of X-MAN sets.

To generate variations of X-MAN sets, we have defined three *variation operators* (middle level, Fig. 5.1), which are functions that apply the variability expressed in a feature diagram to an input tuple of X-MAN sets (implementation in appendix E). In terms of functional model, variation operators are applied to state charts corresponding to non mandatory features to generate permutations, which are aggregated into a set of state charts for their parent feature.

The language of our variation operators is defined by the context free grammar $G = (V, \Sigma, R, S)$:

$$
\begin{array}{ll}
V = \{S, NVAR, UVAR, N, +, F\} & \\
\Sigma = \{alt, or, opt, x\} & \\
S = \{S\} & \\
R = \{S \rightarrow NVAR|UVAR & NVAR \rightarrow N(\langle T, T+\rangle) \\
\quad UVAR \rightarrow opt(\langle T\rangle) & N \rightarrow alt|or \\
\quad + \rightarrow, T + |\epsilon & T \rightarrow S|F \\
\quad F \rightarrow \{xX\} & X \rightarrow, xX|\epsilon \\
\} &
\end{array}
$$

where the terminal x is a single X-MAN architecture, $\{x_1, \ldots, x_n\}$ is an X-MAN set, and the non-terminal T is either a single X-MAN set, or a recursive application of a variation operator to an X-MAN set.

An example of a sentence produced by this grammar is the following:

$$or(\langle alt(\langle opt(\langle\{x\}\rangle), \{x\}\rangle), \{x, x, x, x\}\rangle)$$

showing the nesting of variation operators.

The `ALT` variation operator is a function that takes a tuple of at least two T's as input, and returns each input set as a possible alternative. It is defined by:

$$ALT(\langle T_1, \ldots, T_n\rangle) = \langle T_1, \ldots, T_n\rangle, \text{ for } n \geq 2$$

An instance of this is shown in Fig. 5.8, where $T_1 = \{A\}, T_2 = \{B\}$ and A, B are single X-MAN architectures.

The `OR` variation operator also takes as input a tuple of at least two Ts, and returns all possible combinations (without repetition) of its input. It is defined by:

$$\begin{aligned}\texttt{OR}(\langle T_1, \ldots, T_n\rangle) = \langle & T_1, \ldots, T_n,\ T_1 \oplus T_2, \ldots, T_1 \oplus T_n, \\ & \vdots \\ & T_{n-1} \oplus T_n, \\ & T_1 \oplus T_2 \oplus T_3, \ldots, T_1 \oplus T_2 \oplus T_n, \\ & \vdots \\ & T_{n-2} \oplus T_{n-1} \oplus T_n, \\ & \vdots \\ & T_1 \oplus T_2 \oplus \cdots \oplus T_n\rangle, \text{ for } n \geq 2\end{aligned}$$

where $\oplus$ is a combinator function for T's which (recursively) aggregates their elements. For example, for two X-MAN sets T_1, and T_2, $OR(\langle T_1, T_2\rangle)$ would be the tuple $\langle T_1, T_2, T_1 \oplus T_2\rangle$. This is shown in Fig. 5.9, where $T_1 = \{A\}, T_2 = \{B\}$ and A, B are single X-MAN architectures.

The `OPT` operator makes a single T optional. It is defined in terms of ALT as follows:

$$OPT(\langle T\rangle) = ALT(\langle \emptyset, T\rangle) = \langle \emptyset, T\rangle$$

An example is shown in Fig. 5.10, where $T = \{A\}$ and A is a single X-MAN architecture.

Variation operators can be nested, since they all return tuples of X-MAN sets. This is in keeping with the hierarchical nature of variation points in a feature model. ALT and OR can be nested in any order. However, the order of nesting is significant, since they do not distribute over each other; so care is required when evaluating nested *ALT*s and *OR*s.

Fig. 5.11 shows an example of nesting an *OR* inside an *ALT*, while Fig. 5.12 shows an example of nesting an *ALT* inside an *OR*. These two examples show clearly how the resulting X-MAN sets differ.

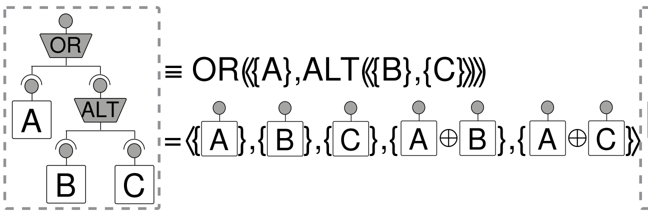


Figure 5.11: Nesting *OR* inside *ALT*.

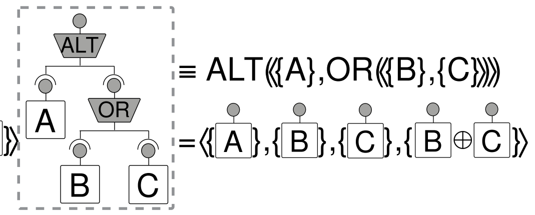


Figure 5.12: Nesting *ALT* inside *OR*.

5.4 Family composition

Once variations of X-MAN sets have been generated, the X-MAN architectures in these sets can be composed together into a family of products, which is another tuple of one X-MAN set. The composition of these sets can be defined in terms of X-MAN composition connectors, since it is ultimately X-MAN architectures that are being composed. However, for any set composition, there are many possible combinations of the members of the input sets. In order not to lose any potential products (as specified by the feature model), we need to keep all possible combinations, and so we have defined *family connectors* accordingly (implementation in appendix F) to perform these set compositions (top level Fig. 5.1).

A *family connector* is defined as an n-ary function that takes a tuple of at least two X-MAN sets and returns a product family, which is a tuple of an X-MAN set. More precisely, a *family connector* *F-Conn* is defined in terms of the corresponding X-MAN composition connector *Conn* as follows:

$$\begin{aligned}&F\text{-}Conn(\langle F_1,\ldots,F_n\rangle)\\&=\langle\{Conn(f_i,\ldots,f_j)|(f_i,\ldots,f_j)\in F_1\times\cdots\times F_n\}\rangle,\ \text{for } n\geq 2\end{aligned}$$

where $F_1,\ldots,F_n$ are X-MAN sets, and $1\leq i,j\leq n$. The Cartesian product $F_1\times\cdots\times F_n$ ensures that all combinations of X-MAN architectures in $F_1,\ldots,F_n$ are kept.

The result of the composition performed by *F-Conn* is a family of fully formed, executable products, each one in the form of an X-MAN architecture.

The two *F-Conn* connectors are *F-SEQ* and *F-SEL* corresponding to the X-MAN composition connectors *SEQ* and *SEL* respectively.

An example of *F-Conn* is shown in Fig. 5.13, where a *F-SEQ* is applied to two tuples, each containing a set of single X-MAN architecture $\{A, B\}$, and $\{C, D\}$ respectively.

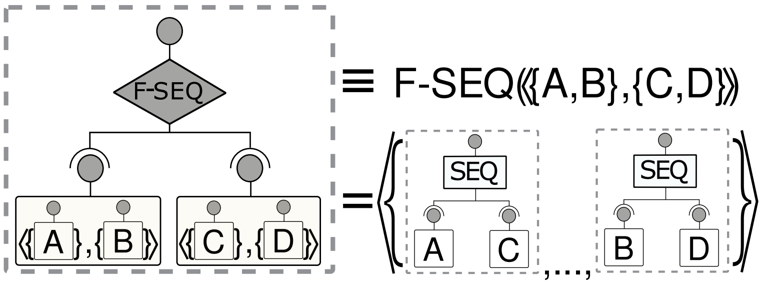


Figure 5.13: Product family resulting from *F-SEQ*.

As in X-MAN, our new component model FX-MAN also has *adapters*: *F-Loop*, and *F-Guard*. An adapter in FX-MAN is a unary connector which applies the corresponding X-MAN adapter (*Guard*, or *Loop*) to each member of the X-MAN set. Its behaviour can be described as follows:

$$\begin{aligned}&F\text{-}Adapter(\langle F\rangle)\\&= \langle\{Adapter(f_1)\ \ldots, Adapter(f_n)\}\rangle | f_i \in F, 1 \leq i \leq n\}\end{aligned}$$

An example of *F-Adapter* is depicted in Fig. 5.14, where a *F-LOOP* is applied to a tuple containing a set of single X-MAN architecture $\{A, B\}$.

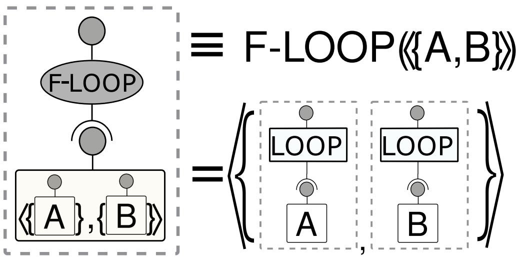


Figure 5.14: Product family resulting from *F-LOOP*

As for X-MAN architectures, well-formed FX-MAN ones can be deposited in the repository of our tool and retrieved whenever needed. This allows an hierarchical composition of families of families.

A family connector thus creates a family of products. It also creates a state chart for each member of the family. Fig. 5.15 shows the state charts created by F-SEQ,

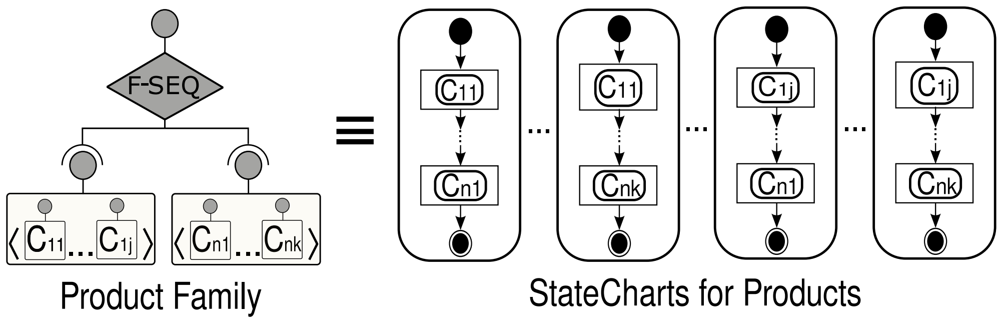


Figure 5.15: State charts created by F-SEQ.

based on SEQ. Activity charts for the family members can also be generated (we omit them here). Similarly, F-SEL, based on SEL, can produce a family and generate a state chart and an activity chart for each member of the family. The same holds for *F-Adapters*.

5.4.1 Family filters

So far we have described the ingredients necessary for defining a product family. However, we have not taken into account *composition rules*, or *constraints*, that may be present in a feature model. Such rules disallow certain combinations of features, and thereby reduce the total number of legitimate products in a family.

In order to handle these rules in the construction of product families in FX-MAN, we define a *family filter* as an operator on components composed by a family connector. A family filter removes products containing illegal combinations of components from the family constructed by the family connector. The language of a family operator is defined by the context free grammar $G = (V, \Sigma, R, S)$:

$V = \{S, COMP, CONSTR, OP, AND\}$

$\Sigma = \{x, requires, excludes, \&\&\}$

$S = \{S\}$

$R = \{S \rightarrow CONSTR$

$\quad CONSTR \rightarrow COMP\ OP\ COMP\ AND$

$\quad COMP \rightarrow x \qquad OP \rightarrow requires | excludes$

$\quad AND \rightarrow \&\&\ CONSTR | \epsilon$

$\}$

An example of sentence produced by this grammar is the following:

$$x_i\ \textit{requires}\ x_j\ \&\&\ x_k\ \textit{excludes}\ x_v\ \&\&\ x_n\ \textit{requires}\ x_m$$

where x is a single X-MAN architecture.

Listing 5.1 shows the filter function invoked to remove elements of a *product family* in input according to the provided list of *Constraint* (more details in appendix G).

```
public static ProductFamily filterProducts(ProductFamily productFamily, EList<
    Constraint> fullConstraint) {
      for (Iterator<Product> iter = productFamily.getProducts().iterator(); iter.
          hasNext();) {
            Product element = iter.next();
            if (!filterTranslator(element.toString(), fullConstraint)) iter.remove
                ();
      }
      return productFamily;
}
```

Listing 5.1: Detail of the family filter implementation.

Note that this grammar not only can remove products with illegal combinations of components, but it can also define filters for keeping only products with desired components. This is a useful property as it allows us to construct only products with desired features.

5.5 Constructing a Product Family

By itself, FX-MAN only provides the building blocks for constructing product families. However, the nature of these building blocks leads itself to the construction of product families architectures structurally isomorphic to a given feature diagram.

As Fig. 5.1 illustrates, composition of the building blocks is rigorously algebraical, as the type system defined by FX-MAN allows the modelling process to flow both vertically (from X-MAN component model to family composition) and horizontally (at each level). Algebraic composition mechanisms are fundamental for hierarchical composition (and therefore systematic construction), since each composition is carried out in the same manner regardless of the level of the construction hierarchy.

As a direct consequence, in FX-MAN the architecture of any product family is a tree. This means that a variant is an hierarchical composition of components. Therefore, if we use components to implement features in a feature model and construct a PFA

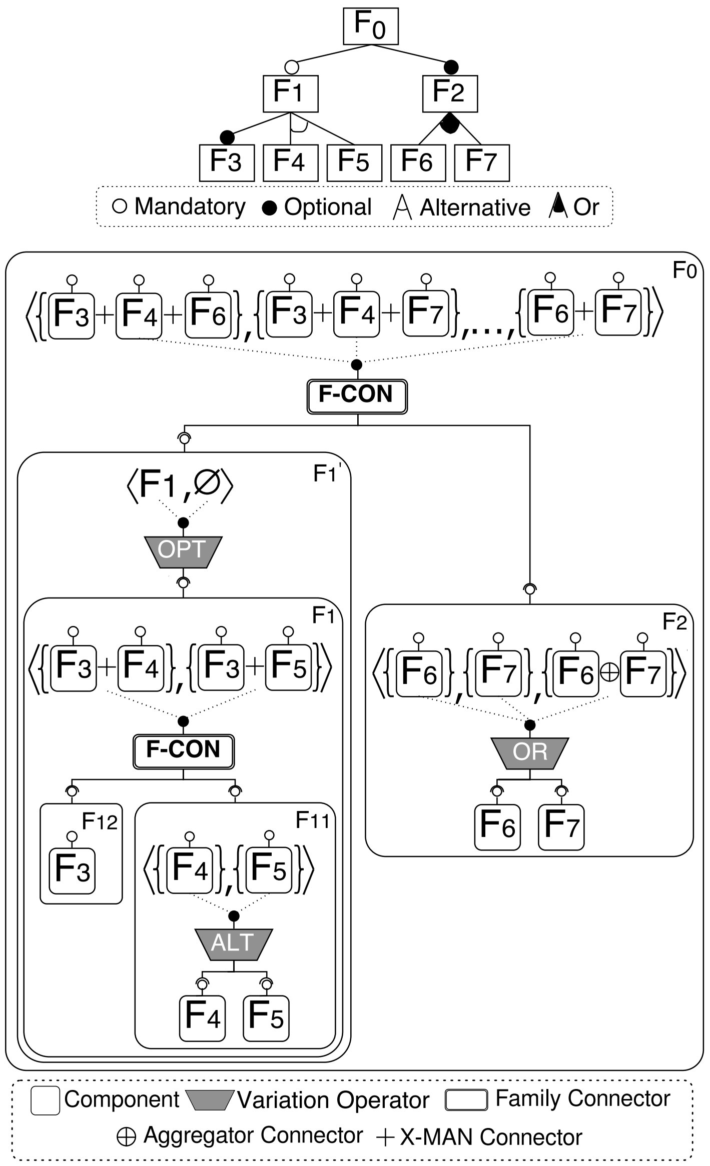


Figure 5.16: Constructing a product family architecture from the feature model.

from these components, the result is a family architecture structurally isomorphic to the feature diagram with no variability mismatch and explicit variation points.

Moreover, the architecture of every product in FX-MAN (*i.e.*an X-MAN component, atomic or composite) has a corresponding functional model, which can be automatically generated. As a result, the functional model we define for a given domain is realised by a set of state charts and activity charts. Since it is feature-oriented, it is fully equivalent to a functional model defined in FODA. This is the basis of our approach to constructing product families in FX-MAN.

We construct components to implement leaf features. As in [8], we distinguish between abstract and concrete features: abstract features are used for structuring, while leaf features are bound to concrete implementation artefacts. Parent-child features relationship is realised by either recursively applying variation operators as in the feature model, or in the presence of mandatory features, by means of family connectors. The latter ensures that the resulting permutation contains the components implementing mandatory features.

Our approach is illustrated on a generic example in Figure 5.16. We start by constructing components (atomic or composite) for each leaf feature (F_4-F_9). Components corresponding to mandatory features (e.g. F_6 and F_7) are composed by connectors into composite components (e.g. F_2 composed from F_6 and F_7). Following the feature model, variation operators are applied to components corresponding to non-mandatory features in order to generate permutations which are aggregated into a tuple of X-MAN sets for their parent feature. For example, for the optional features $F4$ and $F5$, the permutations $\langle\{F_4\}, \emptyset\rangle$ and $\langle\{F_5\}, \emptyset\rangle$ respectively are generated, and aggregated into the component for F_1. As F_1 is itself optional, the variant $\langle F_1, \emptyset\rangle$ is generated and aggregated (into F_1'). Finally, the tuples of X-MAN sets generated by variation operators are composed by family connectors into a family. The top-level family connector produces the family and all associated charts for F_0.

Our 1-to-1 mapping between features and components is possible due to component encapsulation, i.e. the absence of external functional dependencies in our components. To be more precise, the mapping occurs between features and services exposed by components. Therefore, in theory it would be possible to have a component exposing more than one service, which maps to more than one feature. However, it could incur duplication and redundancy in component code as in the presence of cross-cutting features. A *log* feature is a typical example of such.

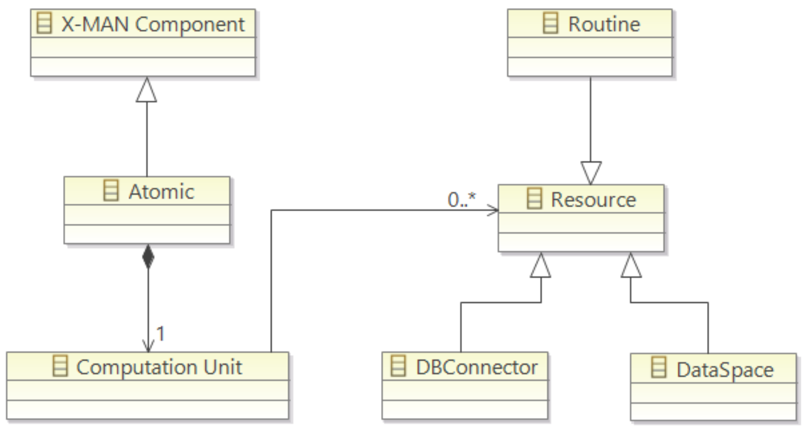


Figure 5.17: Detail of the FX-MAN meta-model to deal with shared resources.

In order to avoid such redundancy, we have extended the FX-MAN meta-model (described in appendix D) in order to manage shared resources. Fig. 5.17 shows the main entities of such extension and their relationships. The computation unit of an X-MAN component can refer to one or more `Resources`, which can be a `Routine`, a `DBConnector`, or a `Dataspace`. In order to preserve composition encapsulation, we require a resource to be self contained. Therefore, a hypothetical routine implementing a log feature may not invoke other routines, and must return control flow as soon its computation ends.

5.6 Summary

FX-MAN is a component model designed for modelling and constructing software product families by providing facilities for defining and building a functional model, and hence a PFA, based on a feature model (the most widely used kind of variability model) as well as components that result from domain analysis. An innovative aspect of FX-MAN is its algebraical type system, which allows to systematically compose families into bigger ones. This is possible because variation operators and family connectors can be applied at any level of composition on X-MAN sets, as every product family is a tuple of an X-MAN set.

Chapter 6

Tool Support

" I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

— Abraham H. Maslow

6.1 Introduction

In order to experiment with the modelling of software families, we have implemented a tool-set [213] , which is composed by the X-MAN tool, the functional model tool and the FX-MAN tool. In this chapter we explain the implementation of the tool-set and its usage in constructing a family of Vehicle Control Systems (VCSs).

The tool-set is based on Eclipse and it is implemented using a model driven development approach, which is described in section 6.2.

Since FX-MAN relies on X-MAN as its underlying component model, and the existing X-MAN tool [214] was based on a discontinued platform, the implementation of the FX-MAN tool-set has required a change in the underlying platform (section 6.3) and therefore a complete re-implementation of the X-MAN tool (section 6.4).

The functional model tool and the FX-MAN tool are respectively illustrated in sections 6.5 and 6.6

6.2 Model Driven Engineering

In software engineering, as in any traditional engineering discipline, modelling has a rich tradition in designing complex systems [215, 216]. However, due to the intangible nature of software and the relative ease of writing code, as depicted in Fig. 6.1 models' role ranges from being secondary (*code-only*) to primary (*model-only*) [217].

A code-only approach is only suitable for small size projects [218]. Indeed, its limitations are evident during the evolution and maintenance of a system due to an increase in complexity, or to the inability to access the original designers.

A possible solution is to consider code as a model, and then provide its graphical representation using tools (*code visualisation* [219, 220] in Fig. 6.1). Therefore, as code is written, models are automatically synchronised. Nevertheless, inconsistencies across models may occur when a piece of information, shared across several artefacts, is deleted in just one of them and not consistently updated in the others.

To avoid such inconsistencies, tools can automate the model-to-code and code-to-model transformation (*round-trip engineering* [221] in Fig. 6.1). This can be achieved by adding meta-data in the source code in order to distinguish between generated, and not generated code. However, without considerable discipline, models and implementation can quickly end up out of step.

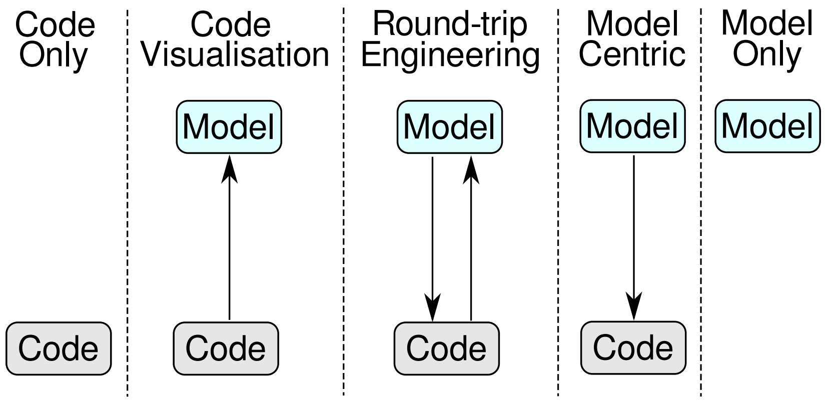


Figure 6.1: The modelling spectrum in software engineering.

In a Model Driven Development (MDD) approach (*model centric* in [222–224] Fig. 6.1), code is automatically generated from platform independent models. Mostly used in highly specialised domains [225, 226], MDD is increasingly gaining attention from both industry and academia [227, 228]. This is due to two main reasons: (i) the necessary technologies have matured; (ii) industry-wide standards have emerged. Indeed, the Object Management Group (OMG)[1] is advocating a model-driven approach through its Model Driven Architecture (MDA)[2] initiative [218, 229, 230] and its supporting standards, such as UML [231], Meta Object Facility (MOF) [232] and XML Metadata Interchange (XMI) [233].

Finally, a *model-only* approach (right-hand-side, Fig. 6.1) is used in situations where implementation may be disconnected from models. Perhaps, the most common example is represented by the growing number of companies that, still remaining in control of their overall enterprise architecture, outsource its implementation and maintenance.

Existing implementation of the X-MAN tool [209] follows a MDD approach. Indeed, it is based on GME [234], a configurable tool-set for the creation of generic modelling environments. However, rather than conforming to standards such as MOF and XMI, GME uses proprietary formats for meta-modelling (MetaGME, a graphical UML-like meta-modelling language) and models exchange (UML Model Transformer). Moreover, GME is currently discontinued and its lack of support makes it quite difficult to extend.

The new tool-set is based on Eclipse and it is also implemented using a model driven approach. The advantages of using this new platform are twofold: (i) Eclipse uses the OMG MDA standards and technologies; tools created using this platform can be easily transferred to any other platform that uses the same standards; (ii) the Eclipse Modelling Project[3] is quite mature, and well supported by a large community.

6.3 Technologies

As depicted in Fig. 6.2, the FX-MAN tool-set leverages a powerful stack of model driven technologies such as EMF [235], Xcore [236], Graphiti [237], Spray [238], Xtend [239] and CDO [240].

The tool-set meta-models (described in appendices B to D) are defined using EMF and its extended syntax Xcore; the latter being designed to overcome the inability of

[1]`http://www.omg.org/`

[2]The term ‘architecture’ refers to the various standard technologies that constitute the MDA foundation.

[3]`https://eclipse.org/modeling/`

the EMF editor to implement behaviour and to deal with complex meta-models. Meta-models instances are created and edited using Graphiti, a modelling infrastructure for EMF. Due to its verbosity, Graphiti code was generated using Spray. A repository for depositing and retrieving X-MAN components, X-MAN sets, and whole product families is realised using CDO on a shared remote server. In addition, component and product implementations are generated via a built-in code generator implemented using Xtend. The following subsections give an overview of the aforementioned technologies.

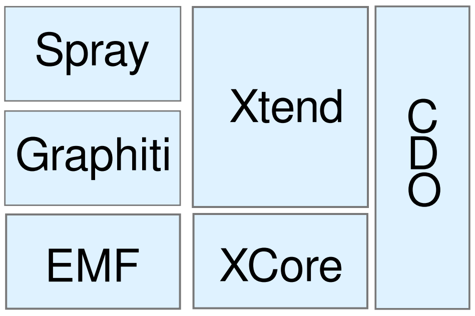


Figure 6.2: FX-MANtool-set technology stack.

6.3.1 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) is both a sophisticated modelling framework and a code generator facility for Eclipse. Albeit it was released as an Eclipse public sub-project only in 2003, EMF has a long heritage as a model-driven, meta-data management engine in IBM Visual Age IDE [230].

EMF distinguishes between the concepts of meta-model and model: the former represents a model structure, while the latter its instance. EMF itself is based on two meta-models, called respectively *Ecore* and *GenModel*. Ecore contains data about the defined classes and their relationships, whereas GenModel encompasses the data needed for code-generation like control parameters and output paths. The generated code, referred to as *core model*, can be used as the foundation for constructing the modelled system.

6.3.2 XCore

EMF provides a basic tree editor to create Ecore meta-models. However, as the meta-model increases in complexity, it becomes cumbersome to use. For example, tracing relationships between classes it is not immediate. It follows that debugging an Ecore meta-model requires a lot of effort.

To overcome this limitation, Eclipse Modelling Tools[4] provides an Ecore Graphical Modeller (EGM) tool that allows manipulating an Ecore meta-model in the form of a UML class diagram. Nevertheless, despite being developed for several years, EGM still presents several bugs (*e.g.* synchronisation between UML and Ecore models), making it unusable when a meta-model increases in complexity.

A solution for these problems is represented by Xcore, a syntax for EMF that, in combination with Xbase[5], transforms it in a fully-fledged programming language similar to Java. Xcore represents a single source of information, as it combines the strength of modelling and programming.

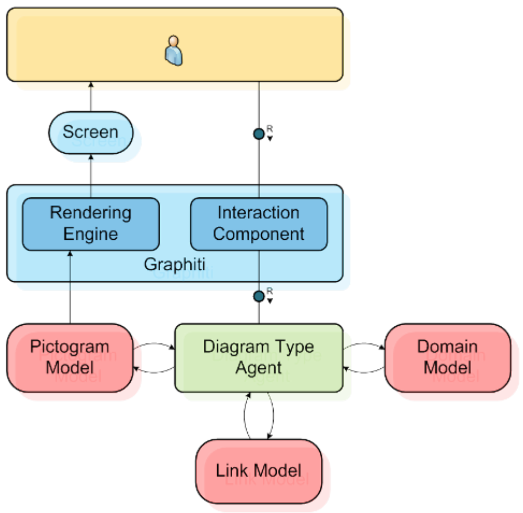
6.3.3 Graphiti

Evolving around EMF, Graphiti is a modelling framework that enables rapid development of diagram editors for EMF-based domain models. It represents a valid alternative to the well known, but also more complex, Eclipse Graphical Modelling Framework (GMF)[6]. Indeed, it lowers the entry barriers as it hides underlying platform-specific technologies such as GEF[7] and Draw2D[8]. Moreover, unlike GMF that uses a generative approach, Graphiti uses a run-time-oriented approach to adapt the editor behaviour. A generative approach, or rather an approach where code is generated starting from a meta-model, implies that any custom behaviour must be implemented by changing the generated source code. This involves a variety of issues related to both evolution and maintenance of the generated tool [241]. Using a run-time oriented approach, or rather an approach where a programmer can adapt an editor by overriding/extending methods exposed by interfaces, guarantees that no generated code has to be

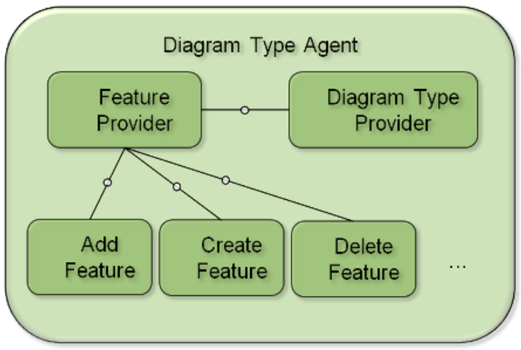
[4]`http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/marsr`
[5]`http://wiki.eclipse.org/Xbase`
[6]`http://www.eclipse.org/gmf-tooling/`
[7]`https://eclipse.org/gef/`
[8]`https://www.eclipse.org/gef/draw2d/`

manipulated.

Fig. 6.3(a) illustrates the Graphiti architecture. An *interception component* captures users requests and forwards them to the *diagram type agent* component, which architecture is detailed in Fig. 6.3(b). It consists of two main modules: a *feature provider* and a *diagram type provider*. The feature provider supplies the needed features according to the given context. Typical examples of Graphiti features are *add*, *create*, *remove*, and *delete*. The diagram type provider supplies the needed tool specific requests to the interaction component. This implies, for example, the choice of a suitable update strategy when the underlying *domain model* changes.



(a) Overall architecture



(b) Detail of the diagram type agent.

Figure 6.3: Graphiti architecture.

The linkage between instantiated domain model objects and their relative graphical representation in a *Pictogram Model* is maintained by the *Link Model*, which architecture is illustrated in Fig. 6.4).

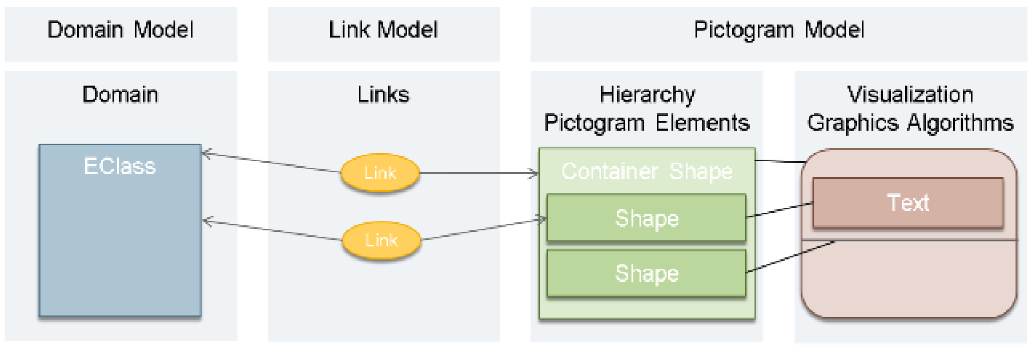


Figure 6.4: Link model architecture.

Finally, domain objects are rendered on the screen via the *rendering engine* (Fig.6.3(a)). The latter, together with the interaction component, form the actual Graphiti run-time based on GEF and Draw2D.

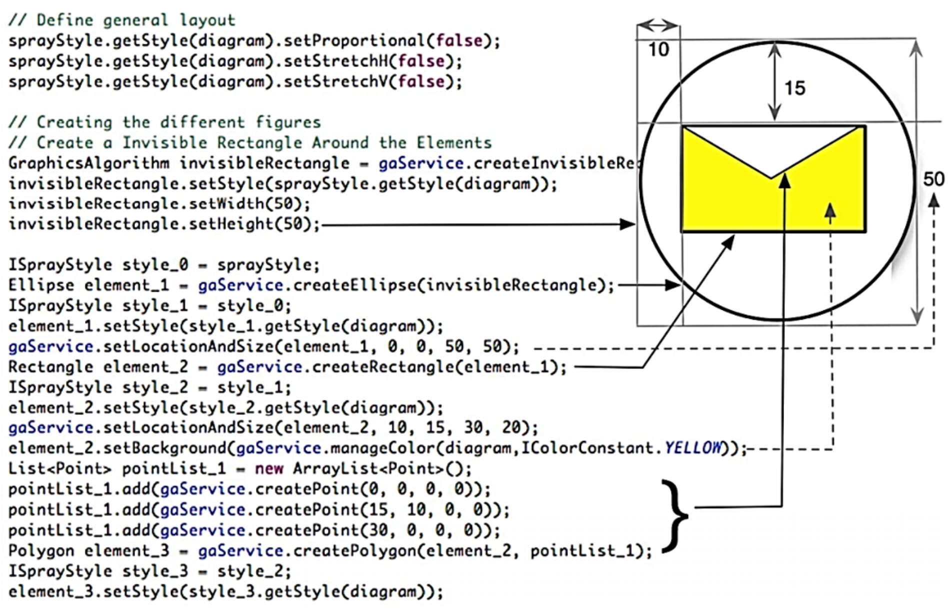
6.3.4 Spray

Programming a visual tool against Graphiti framework APIs is a fairly simple task. However, it soon becomes repetitive and error prone. For instance, for each Ecore class, one has to program at least four features: add, create, remove, and delete. In terms of Graphiti, these features have the following behaviour.

- Add: attaches a graphical representation to an instantiated meta-model class (referred as *business object*).
- Create: instantiates a business object and invokes the respective *add* feature to generate its graphical representation.
- Remove: opposite to *add*, it removes the graphical representation of a business object.
- Delete: opposite to *create*, it deletes a business object and invokes the relative *remove* feature to eliminate its graphical representation.

Once implemented, these features need to be manually registered by the *feature provider* (Fig. 6.3(b)), and then added to the configuration palette of the *tool behaviour provider*.

The repetitiveness of such approach does not scale to large Ecore meta-models. Spray solves this problem by providing Domain Specific Languages (DSLs) to describe visual DSL editors against the Graphiti runtime, along with a code generator to produce their boilerplate implementation code against the Graphiti APIs. It follows that Spray provides a faster and more reliable way to create Graphiti editors.

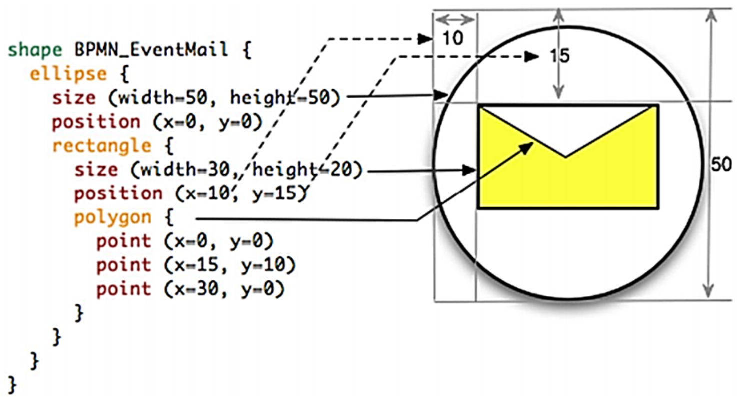


```
// Define general layout
sprayStyle.getStyle(diagram).setProportional(false);
sprayStyle.getStyle(diagram).setStretchH(false);
sprayStyle.getStyle(diagram).setStretchV(false);

// Creating the different figures
// Create a Invisible Rectangle Around the Elements
GraphicsAlgorithm invisibleRectangle = gaService.createInvisibleRe
invisibleRectangle.setStyle(sprayStyle.getStyle(diagram));
invisibleRectangle.setWidth(50);
invisibleRectangle.setHeight(50);

ISprayStyle style_0 = sprayStyle;
Ellipse element_1 = gaService.createEllipse(invisibleRectangle);
ISprayStyle style_1 = style_0;
element_1.setStyle(style_1.getStyle(diagram));
gaService.setLocationAndSize(element_1, 0, 0, 50, 50);
Rectangle element_2 = gaService.createRectangle(element_1);
ISprayStyle style_2 = style_1;
element_2.setStyle(style_2.getStyle(diagram));
gaService.setLocationAndSize(element_2, 10, 15, 30, 20);
element_2.setBackground(gaService.manageColor(diagram,IColorConstant.YELLOW));
List<Point> pointList_1 = new ArrayList<Point>();
pointList_1.add(gaService.createPoint(0, 0, 0, 0));
pointList_1.add(gaService.createPoint(15, 10, 0, 0));
pointList_1.add(gaService.createPoint(30, 0, 0, 0));
Polygon element_3 = gaService.createPolygon(element_2, pointList_1);
ISprayStyle style_3 = style_2;
element_3.setStyle(style_3.getStyle(diagram));
```

(a) Graphiti



```
shape BPMN_EventMail {
  ellipse {
    size (width=50, height=50)
    position (x=0, y=0)
    rectangle {
      size (width=30, height=20)
      position (x=10, y=15)
      polygon {
        point (x=0, y=0)
        point (x=15, y=10)
        point (x=30, y=0)
      }
    }
  }
}
```

(b) Spray

Figure 6.5: Example of code needed to draw a complex picture in Graphiti and Spray.

As an example, Fig. 6.5 shows the amount of code needed to create a quite complex picture in Graphiti (Fig. 6.5(a)) and in Spray (Fig. 6.5(b)). It is easy to notice how the use of a DSL to represent the shape of a graphical element drastically reduces the amount of code needed and increases its expressiveness, making it easy to debug.

6.3.5 CDO

The Connected Data Objects (CDO) is a shared model framework for EMF-based applications that works as a repository at development-time, and as a persistence framework at run-time. In contrast to Teneo[9], which allows several clients to access a database (realizing in this way a client/server two-tier architecture), CDO presents a three-tier architecture (Fig. 6.6) featuring a central model repository server. The server architecture allows different types of pluggable data storage back-ends. Communication between clients and server is implemented using Net4j Signalling Platform[10], an extensible client/server communications framework. Moreover, CDO has a built-in versioning facility, as well as pluggable security services to regulate access to shared objects.

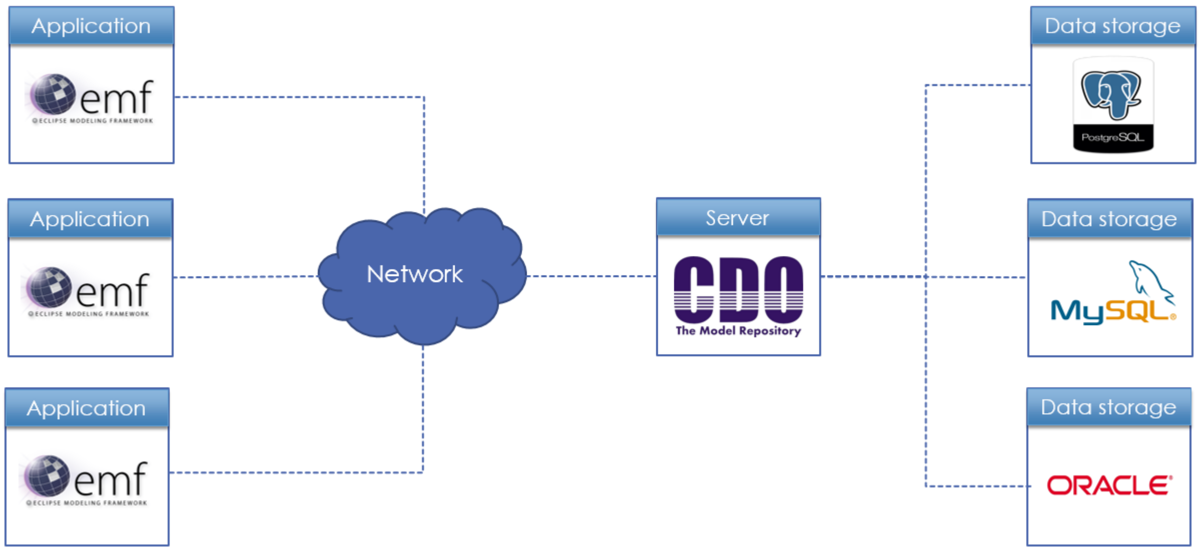


Figure 6.6: CDO overview.

6.3.6 Xtend

Xtend is a statically typed language that uses the Java type system to infer EMF models in order to generate readable Java 5 compatible source code. Model-to-text transformation, or rather the generation of textual artefacts from models (*e.g.* Ecore), is a fundamental part of any MDD approach. Indeed, albeit it is possible to manually transform models into code, the undeniable power of MDD comes from the automation of this process. Such transformations accelerate the MDD process, and can enforce the adoption of 'best practises' to improve code quality.

[9]`https://wiki.eclipse.org/Teneo`
[10]`https://wiki.eclipse.org/Net4j`

Xtend, in contrast to other model-to-text technologies such as Acceleo[11] and JET[12] relies on EMF to automatically create Abstract Syntax Trees (ASTs) to traverse model structure. Once the AST is stored in memory, Xtend uses it to generate code without the need of re-analysing the whole model. The compiled output is readable and pretty-printed and tends to run as fast as the equivalent handwritten Java code [239].

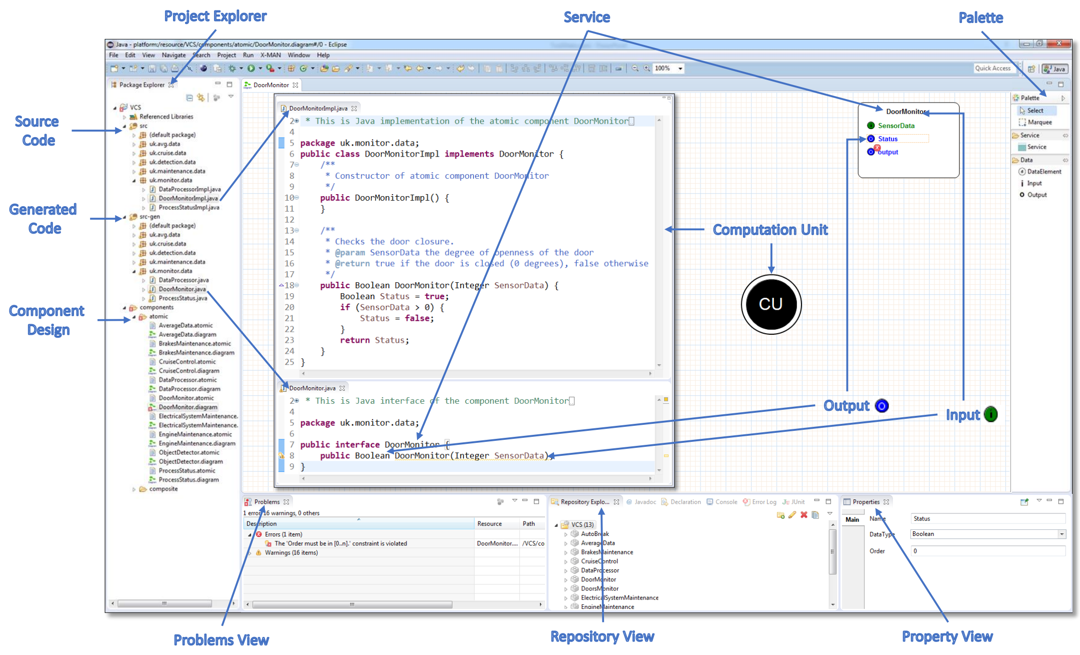
6.4 X-MAN Tool

The X-MAN tool supports the component-based system development and its associated life cycle [242] (described in section 4.4). The latter consists of: (i) a component development phase; and (ii) a system development phase. In (i) components are designed, built, and deposited in a repository. In (ii) components are retrieved and deployed into the system under construction. Figs. 6.7 and 6.8 show our Eclipse workbench for (i) and (ii) respectively.

For each development phase, the tool provides a canvas as a design space, as well as a palette of pre-defined building blocks. Properties of each element, *e.g.* sequencing order and branch conditions, are set by means of the Eclipse *Property* view. During the whole design process the tool continuously validates the diagram, showing errors both graphically in the canvas and textually in the Eclipse *Problem* view. Once validated, component source code (that can be deployed as a stand-alone application) can be generated. In particular, for atomic components, code skeleton of computation units is generated. For composite components and systems, source code is entirely generated including connector coordination logics, data passing logics, data initialisation, as well as constituent component implementations. The latter makes use of previously generated source code. It is worth noting that we currently support Java and experimentally C on x86 PC platform.

This new implementation is based on an extended version of the X-MAN component model [208]. Its meta-model, described in appendix B and depicted in Fig. B.1, shows the relationships between the main entities of the X-MAN component model: *components*, *connectors*, and *services*.

[11]`https://eclipse.org/acceleo/`
[12]`https://eclipse.org/modeling/m2t/?project=jet`



```
 * This is Java implementation of the atomic component DoorMonitor

package uk.monitor.data;
public class DoorMonitorImpl implements DoorMonitor {
    /**
     * Constructor of atomic component DoorMonitor
     */
    public DoorMonitorImpl() {
    }

    /**
     * Checks the door closure.
     * @param SensorData the degree of openness of the door
     * @return true if the door is closed (0 degrees), false otherwise
     */
    public Boolean DoorMonitor(Integer SensorData) {
        Boolean Status = true;
        if (SensorData > 0) {
            Status = false;
        }
        return Status;
    }
}
```

```
 * This is Java interface of the component DoorMonitor

package uk.monitor.data;

public interface DoorMonitor {
    public Boolean DoorMonitor(Integer SensorData);
}
```

Figure 6.7: Eclipse workbench for component development.

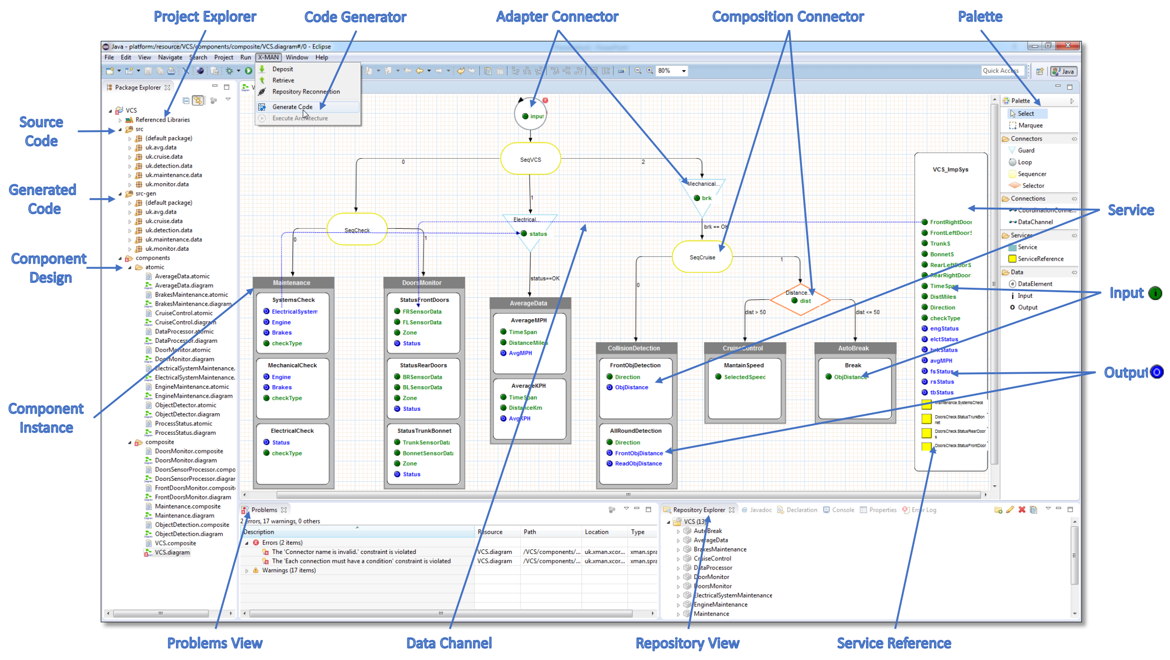
Components

X-MAN has two types of components: *atomic* and *composite*. They are both fully encapsulated, *i.e.* they have no external functional dependencies and contain only *provided services*.[13]

An *atomic component* is a unit of computation. Its *computation unit* (CU) contains the implementation of the services it exposes. As shown in Fig. 6.7, according to the dragged service(s), the tool generates an interface and an empty implementation. Therefore, in terms of object-oriented programming, a service is an interface, while the CU is its implementation.

Valid components are deposited in a standalone or collaborative repository. In our tool, a repository is displayed in the *X-MAN Repository Explorer* view, at the bottom of Figures 6.7 and 6.8. From the *X-MAN Repository Explorer* view pre-built components can be retrieved and their instances (represented as grey rectangles in Fig. 6.8) can be composed into *composite components* by dragging and dropping elements (*e.g.* composition connectors) from the palette on the right-hand-side panel.

[13]This is in sharp contrast to objects or architectural units, both of which have external dependencies.



```
Problems
2 errors, 17 warnings, 0 others
Description                                                    Resource      Path                 Location          Type
Errors (2 items)
  The 'Connector name is invalid.' constraint is violated      VCS.diagram   /VCS/components/...  uk.xman.xcor...   xman.spra
  The 'Each connection must have a condition' constraint is violated  VCS.diagram   /VCS/components/...  uk.xman.xcor...   xman.spra
Warnings (17 items)
```

Figure 6.8: Eclipse workbench for system development.

Connectors

Composition connectors are (exogenous) control structures that coordinate the execution of the components they compose [243]. They are *Sequencer* and *Selector*, which provide sequencing and branching respectively. In Fig. 6.8 a sequencer is shaped as an ellipse and a selector as a rhombus.

In addition, *adapter* connectors such as *Guard*, and *Loop* provide gating, and looping respectively. In Fig. 6.8 a loop is shaped as a circle, whereas a guard as a triangle.

While connectors control execution among component instances, data between components flows through *data channels* (dotted arrow in Fig. 6.8).

Finally, the new meta-model, and hence the tool, includes a new connector called *Aggregator*, which aggregates in a composite component the services exposed by the components it composes. Therefore, it provides a façade to their services.

Services

A *service* represents an operation exposed by a component. It contains two main entities: parameters, and service references. *Parameters* are inputs and outputs, while *service references* specify services in sub-components that contribute to the provided operation. In Fig. 6.8, a service reference is represented as a square, while a parameter as an ellipse.

6.4.1 Example

Fig. 6.9 shows the environment in which a Vehicle Control System (VCS) (adapted from [65]) has been developed. A VCS is a real-time, on-board system for supervising a vehicle. It manages several routine services and tasks, including:

- statistical data calculation, *e.g.* of fuel consumption and of average speed;
- observation or monitoring of the vehicle's internal state, *e.g.* maintenance status;
- cruise control, *i.e.* automatically controlling the vehicle's speed in such a way that a steady (cruise) speed can be set (by the driver) and then maintained by taking over control of the throttle whenever necessary;
- collision detection, which is useful in its own right to ensure safety, can enable automatic driving (while cruising).

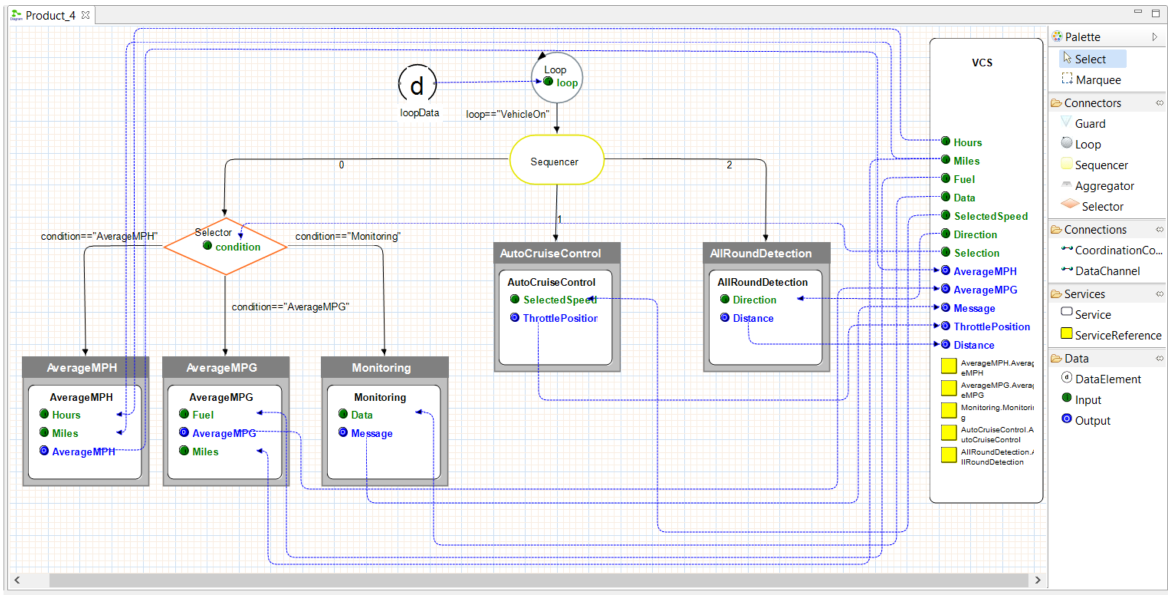
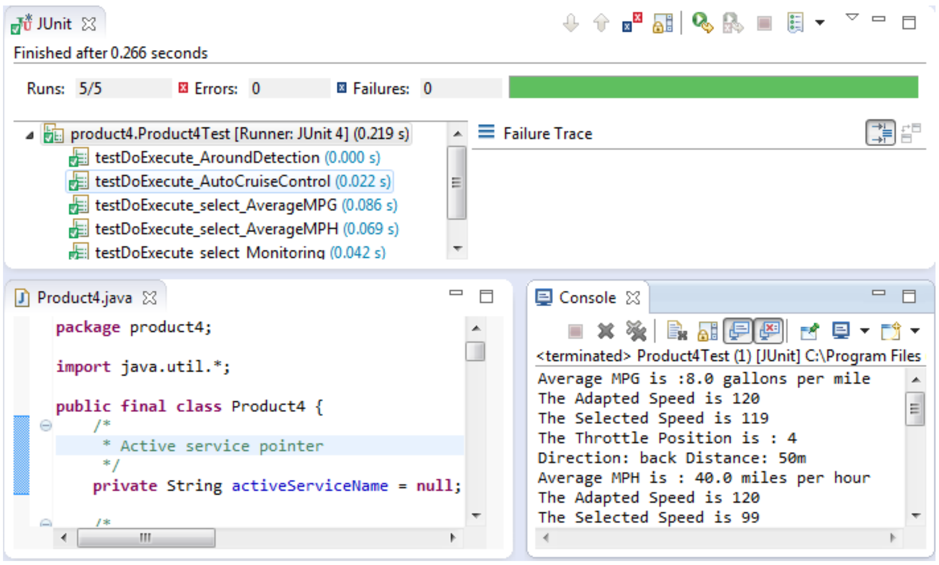


Figure 6.9: Example of a Vehicle Control System in X-MAN.

Our simplified VCS results from the composition of five component instances (*i.e.* `AverageMPH`, `AverageMPG`, `Monitoring`, `AutoCruiseControl` and `AllRoundDetection`), and it exposes the *VCS* service, which is composed from services exposed by the component instances. The behaviour of the VCS in Fig. 6.9 can be described by its automatically generated functional model in the notation of state chart and activity chart.

Moreover, as illustrated in Fig. 6.10, its generated implementation can be validated by a set of JUnit tests.



```
JUnit
Finished after 0.266 seconds
Runs: 5/5    Errors: 0    Failures: 0

product4.Product4Test [Runner: JUnit 4] (0.219 s)
    testDoExecute_AroundDetection (0.000 s)
    testDoExecute_AutoCruiseControl (0.022 s)
    testDoExecute_select_AverageMPG (0.086 s)
    testDoExecute_select_AverageMPH (0.069 s)
    testDoExecute_select_Monitoring (0.042 s)
```

```java
package product4;

import java.util.*;

public final class Product4 {
    /*
     * Active service pointer
     */
    private String activeServiceName = null;

    /*
```

```
<terminated> Product4Test (1) [JUnit] C:\Program Files
Average MPG is :8.0 gallons per mile
The Adapted Speed is 120
The Selected Speed is 119
The Throttle Position is : 4
Direction: back Distance: 50m
Average MPH is : 40.0 miles per hour
The Adapted Speed is 120
The Selected Speed is 99
```

Figure 6.10: Execution of the VCS example in Fig. 6.9

6.5 Functional Model Tool

The functional model tool, which meta-model is described in appendix C, allows the automatic creation of functional and behavioural views of an X-MAN architecture (built using the X-MAN tool) in the form of activity-chart and state-chart respectively. As depicted in Fig. 6.11, this tool can be invoked within the context of an X-MAN environment[14]. The tool automatically validates the currently open X-MAN architecture[15] and provides details (in *Problem View*) in case validation fails. Following the hierarchical structure of the current X-MAN architecture, the tool recursively generates linked state-charts and activity-charts in separate folders. This implies that a user can analyse a functional model by drilling down activities and states as well as by checking for each control activity the corresponding state-chart.

6.5.1 Example

The generated functional model for the VCS in Fig. 6.9 is shown by the activity and state charts of Figs. 6.12(a) and 6.12(b) respectively.

[14]The menu is dynamically enabled when suitable X-MAN artefacts are available and valid.
[15]This is achieved by invoking the validation routines provided by EMF.

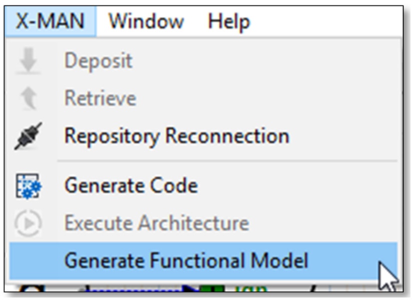
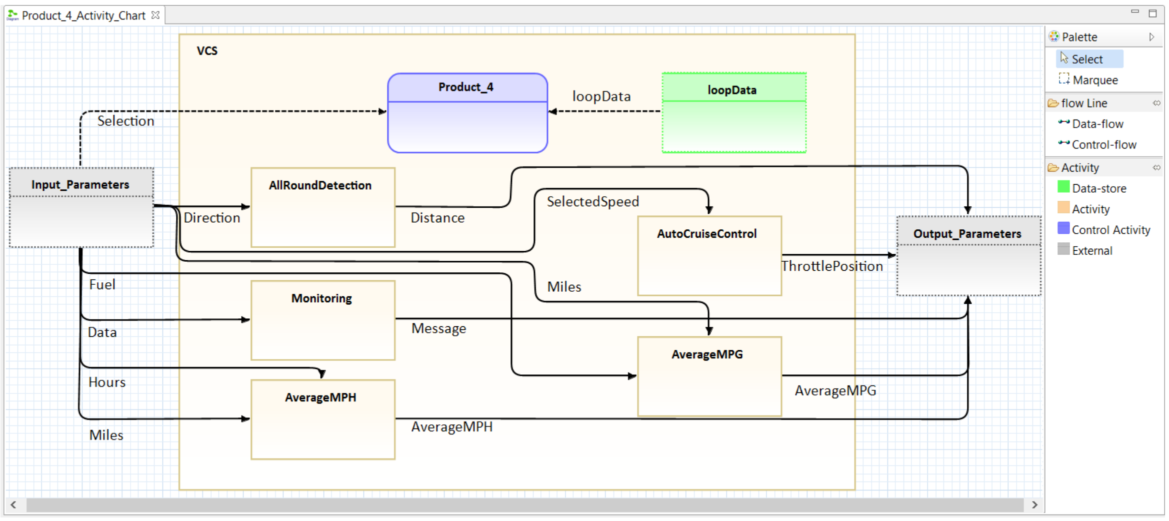


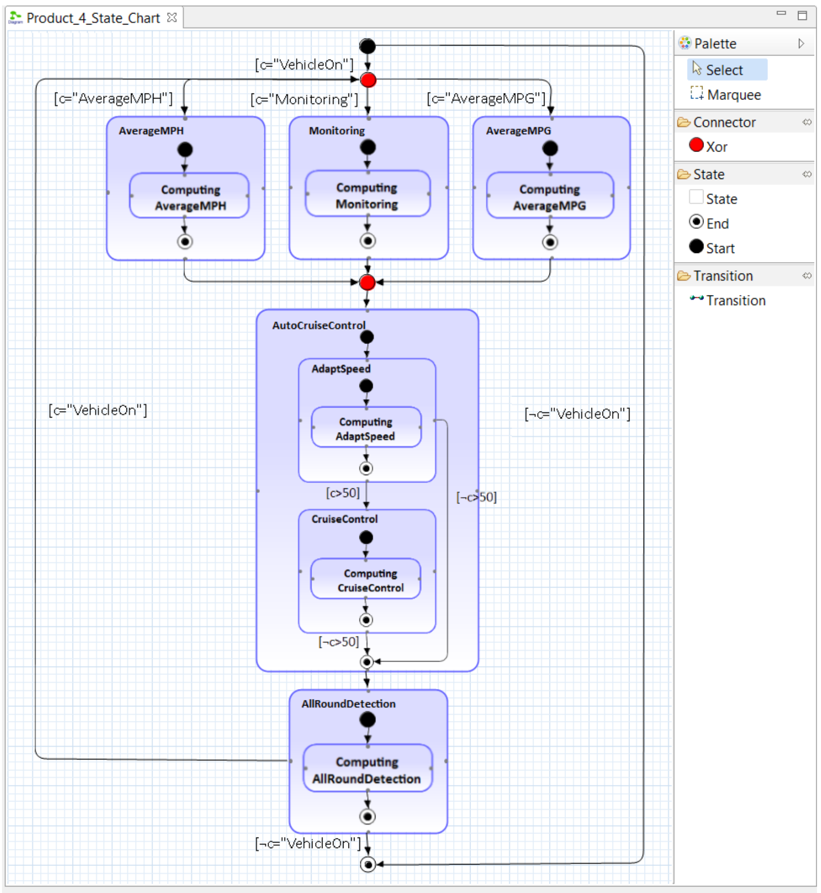
Figure 6.11: Detail of the X-MAN menu.

In the activity chart of Fig. 6.12(a), the top activity is the service `VCS` that the product exposes. Its sub-activities, `AllRoundDetection`, `Monitoring`, `AverageMPH`, `AutoCruiseControl` and `AverageMPG` are linked to the activity charts generated for the corresponding components. Data flowing between parameters are depicted as data flow lines. Input and output parameters contained in the exposed service are represented as data flow lines coming from, or going to the external activites `Input_Parameters` and `Output_Parameters`. Reading from top to bottom, left to right, `Selection`, `Fuel`, `Data`. `Hours`, `Miles`, `Direction`, and `SelectedSpeed` are data provided as input; while `Distance`, `Message`, `AverageMPH`, `ThrottlePosition`, and `AverageMPG` as output. The data element `LoopData`, which supplies condition data to the loop, is represented as a data store with the same name. Indeed, semantically they both define a temporary storage of data. Data supplied to the loop is depicted as a control flow line going to the control activity `Product_4`,[16] which behaviour is defined by the state-chart in Fig. 6.12(b).

[16]The name is derived from the X-MAN architecture file-name



(a) Generated VCS activity-chart.



(b) Generated VCS state-chart.

According to the value provided by the data store `LoopData`, the machine depicted in Fig. 6.12(b) transits from the *Start* state directly to the *End* state if the condition [*c*=*VehicleON*] is not met. Otherwise, it reaches an *XOR* (exclusive or) connector, which leads to three states, namely *AverageMPH*, *Monitoring*, and *AverageMPG*, each containing the state chart generated for the corresponding component. The machine transits from the only state (at the time) that meets the condition, to the state *AutoCruiseControl*, which contains the state chart of the composite component `AutoCruiseControl`. In here, a transition from the state *AdaptSpeed* to the state *CruiseControl* takes place only if the condition $[c > 50]$ is satisfied. On leaving *AutoCruiseControl*, the machine transits to the state *AllRoundDetection*, which consists of the state chart generated for the atomic component `AllRoundDetection`. Finally, if the vehicle is switched off, the condition $[\neg\ c = \mathit{VehicleOn}]$ becomes true and consequently the machine reaches its *End* state. Alternatively, it will start again by reaching the initial *XOR* connector.

6.6 FX-MAN Tool

Our FX-MAN tool, as described in chapter 5, supports the construction of software product line architectures with explicit variability through three stages:[17] (i) constructing components (based on the X-MAN component model and tool), (ii) adding variability onto components to create variations, and (iii) composing variations into one PFA capturing all possible variants. Moreover, as a result of an on-going collaboration with pure-systems GmbH,[18] our tool is interoperable with their tool pure::variants, current market leader in SPLE tools [13] (see section 6.6.1).

The screenshot in Fig. 6.12 gives a view of the tool (in which the example of section 6.6.2 was constructed). As for the X-MAN tool, the FX-MAN one provides a canvas as a design space, as well as a palette of pre-defined design blocks. During the whole design process, the tool continuously validates the diagram, showing errors both graphically in the canvas and textually in the Eclipse *Problem* view. A diagram context menu gives access to an additional set of features such as automatic layout and diagram export; while the FX-MAN menu in the top bar allows product extraction (from a product family that has been constructed).

[17]It is important to emphasize that FX-MAN is only a tool for constructing a product family once it has been defined. Nevertheless, the compositional nature of FX-MAN does seem to suggest a systematic way to define and construct a product family by following the structure of the feature model, but we do not claim that we have a general methodology.

[18]`https://www.pure-systems.com/`

In our tool, variation operators can be dragged from the palette, and connected to instances of pre-built X-MAN components (in X-MAN sets) or to other variation operators. The tool automatically checks the validity of every new dragged connection. Fig. 6.12 shows two *Alternative* and two *Optional* operators connected to four X-MAN components. It also shows an *Or* operator nested within an *Alternative* operator.

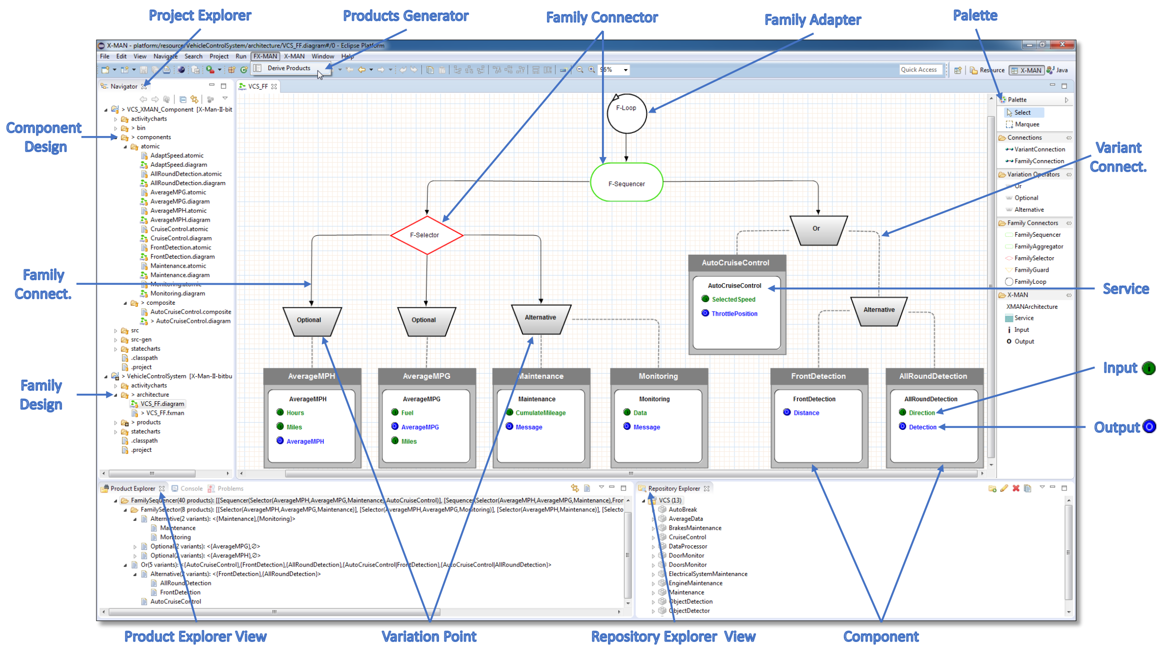


Figure 6.12: Eclipse workbench for constructing an FX-MAN architecture.

Variation generation results in sets of X-MAN architectures. Family connectors compose X-MAN sets into a PFA, which is an architecture containing the architectures of all the members. A product family can be adapted by a family adapter[19]. Family connectors and adapters are implemented in a palette in our tool as shown in Fig. 6.12. To apply a family connector we drag it onto the canvas and make connections from it to X-MAN components (in X-MAN sets) or variation operators. In Fig. 6.12, the *F-Selector* composes three sets of variations produced by the two *Optional* and the *Alternative* variation operators into a single architecture. An *F-Sequencer* composes the previous architecture with another set of variations created by the variation operator *Or* to yield a larger architecture. Finally, a *F-Loop* is connected at the top.

Variability introduced by a variation operator, as well as the family constructed by a family connector, can be analysed at design time by means of the *Product Explorer* view (bottom, Fig. 6.12). Architectures of individual members (as well as the whole

[19]The family connectors and adapters specify coordination and adaptation applied to all variations.

product family) can be directly extracted and executed using the menu[20] depicted in Fig. 6.13. Moreover, the constructed PFA can be deposited in the repository, and subsequently retrieved to be further composed.

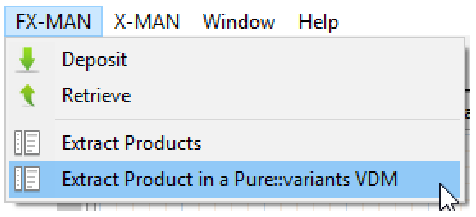


Figure 6.13: Detail of the FX-MAN menu.

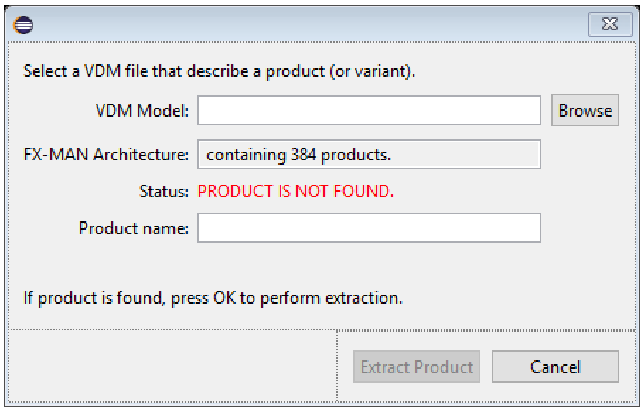
6.6.1 Pure::variants Plug-in Connector

By leveraging the modular architecture of tool, it can be extended with new functionalities by means of plug-ins.

In particular, we have developed a plug-in (depicted in Fig. 6.14) that allows the interoperability with pure::variants for variant derivation. In pure::variants, a stakeholder can specify a (valid) feature configuration and generate a Variant Description Model (VDM) [244]. The created variant model is then provided as an input into our plug-in, which interprets and validates it against a chosen PFA to ascertain the product variant exists.[21] The validation makes use of the unique names of features in a VDM model and matches them against component names in the PFA. If a match is found, the plug-in extracts the desired product by traversing the PFA, retrieving components from repository, replacing family connectors with their counterpart X-MAN ones and constructing the product model as X-MAN architecture. Moreover, our plug-in also offers a dialog to specify the name of the extracted product and its location on the file system.

[20]Like the X-MAN menu, FX-MAN menu is also enabled only when suitable models are present and valid.

[21]This is technically achieved by generating a family filter that reflects the selected features relationships.



Select a VDM file that describe a product (or variant).

VDM Model:

FX-MAN Architecture: containing 384 products.

Status: PRODUCT IS NOT FOUND.

Product name:

If product is found, press OK to perform extraction.

Figure 6.14: Pure::variants plug-in visual interface.

6.6.2 Example

Returning to the example of section 6.4.1, let us consider a family of Vehicle Control Systems (VCSs) as defined by the feature model in Fig. 6.15.

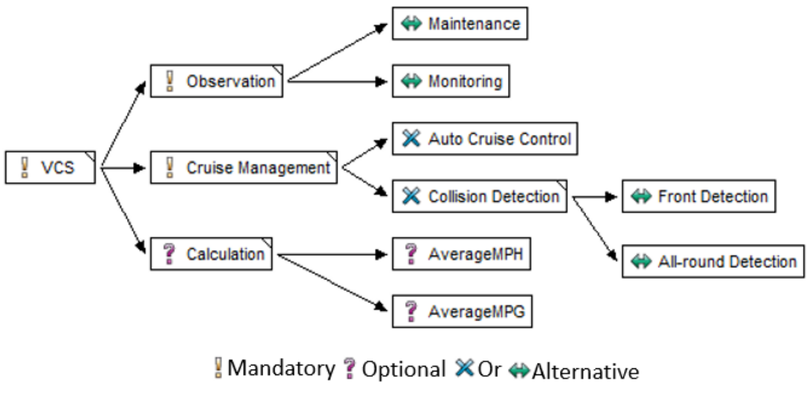


Figure 6.15: VCS feature model.

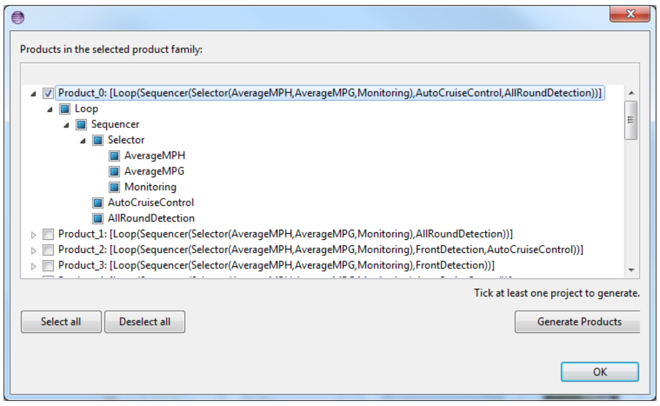
This describes 40 product variants which can:

- Optionally have the capability to calculate statistical data (*Calculation*) about average speed (*AverageMPH*), or average fuel consumption (*AverageMPG*).
- Mandatorily observe the vehicle's status (*Observation*) by signalling the need to carry out ordinary maintenance (*Maintenance*), or by warning the driver about internal malfunctions (*Monitoring*).
- Mandatorily provide a cruise management facility (*CruiseManagement*) by providing *AutoCruiseControl*, *CollisionDetection*, or both. *AutoCruiseControl* allows the vehicle to automatically adjust its speed

in accordance with the driver's desired pace. *CollisionDetection* enables the system to calculate and show the distance between the vehicle, and the nearest objects in different directions. If present, it can either detect objects in front of the vehicle (*FrontDetection*), or all-round it (*All-roundDetection*).

Our repository already contains five of the seven X-MAN components corresponding to seven leaf features (implemented in section 6.4.1): `AverageMPH`, `AverageMPG`, `Monitoring`, `AutoCruiseControl` and `AllRoundDetection`. Therefore, using the X-MAN tool we build (and deposit) the remaining components `FrontDetection` and `Maintenance`.

Subsequently, we retrieve those components using the *Repository Explorer* view and apply them the variation operators as defined in the feature model. In Fig. 6.12, we apply *Optional* to `AverageMPH` and `AverageMPG`; *Alternative* to `Maintenance`, `Monitoring`, and to `FrontDetection` and `AllRoundDetection`; *Or* to the latter and `AutoCruiseControl`.

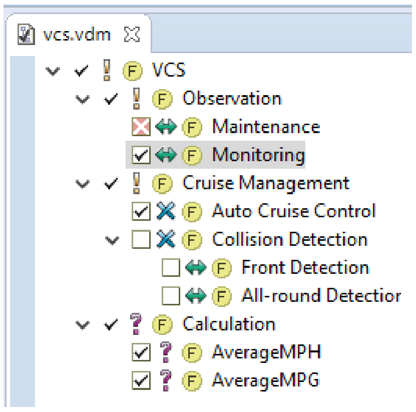


Products in the selected product family:

```
▲ ☑ Product_0: [Loop(Sequencer(Selector(AverageMPH,AverageMPG,Monitoring),AutoCruiseControl,AllRoundDetection))]
    ▲ ■ Loop
        ▲ ■ Sequencer
            ▲ ■ Selector
                  ■ AverageMPH
                  ■ AverageMPG
                  ■ Monitoring
              ■ AutoCruiseControl
              ■ AllRoundDetection
▷ ☐ Product_1: [Loop(Sequencer(Selector(AverageMPH,AverageMPG,Monitoring),AllRoundDetection))]
▷ ☐ Product_2: [Loop(Sequencer(Selector(AverageMPH,AverageMPG,Monitoring),FrontDetection,AutoCruiseControl))]
▷ ☐ Product_3: [Loop(Sequencer(Selector(AverageMPH,AverageMPG,Monitoring),FrontDetection))]
```

Tick at least one project to generate.

(a) Built-in product extraction interface.



vcs.vdm

```
VCS
    Observation
        Maintenance
        Monitoring
    Cruise Management
        Auto Cruise Control
        Collision Detection
            Front Detection
            All-round Detection
    Calculation
        AverageMPH
        AverageMPG
```

(b) A pure::variants VDM.

Figure 6.16: Product variant extraction.

Finally, the tuples of X-MAN sets generated by all the variation operators are composed by means of family connectors. In the VCS architecture in Fig. 6.12, the family of 40 product variants specified by the feature model is modelled by composing the tuples of X-MAN sets by means of two family connectors (*F-Sequencer*, *F-Selector*), and then adapted by a *F-loop*. We use *F-Sequencer* to allow a driver to choose any subset of the features *Calculation* and *Observation*; and *F-Sequencer* to combine the driver's choice with the *Cruise Management* feature. The pick of family connectors is a design decision. However, it will not affect the total of number of products in the family. The resulting PFA contains a total of 40 products, which can be inspected (using the *Product Explorer* view), and extracted either using the extraction dialogue box in Fig. 6.16(a) or a pure::variants VDM as in Fig. 6.16(b).

Using the VDM in Fig. 6.16(b) we can extract the same VCS constructed in section 6.4.1[22]. It represents a premium version of VCS which is capable of displaying the result calculated by `AverageMPH`, `AverageMPG`, or `Monitoring` (chosen by the user). Then `AutoCruiseControl` is invoked with the aim of maintaining the speed (selected by the user). Finally, the system shows the distance from the vehicle to the nearest obstacle in a specified direction.

6.7 Summary

The implementation of the FX-MAN tool-set follows a model driven development approach and it is based on a powerful and mature stack of technologies such as EMF, Xcore, Graphiti, Spray and CDO.

For each stage of a family modelling process, the tool-set provides a canvas as a design space, as well as a palette of pre-defined design blocks. It also performs continuous validation, seamlessly integrates with pure::variants, and enables collaborative modelling thanks to a shared repository.

Finally, the tool-set is available via the University of Manchester Intellectual Property (UMIP) platform at `http://www.click2go.umip.com/i/soft-ware/x_man.html`.

[22]This implies that we can generate its functional model and deployable source code, which can be tested as shown in Fig. 6.10

Chapter 7

Use Case: External Front Car Light Family

"Claiming truth in the absence of evidence is prejudice."

— Joseph Rain

7.1 Introduction

The lighting system of a vehicle consists of lighting and signalling devices which allow other drivers and pedestrians to see its presence, as well as its actual and intended direction of travel. As depicted in Fig. fig. 7.1 forward illumination is provided by high and low beam headlights, which may be augmented by auxiliary fog lamps, driving lamps, or cornering lamps. A high beam headlight produces a centre-weighted, intense distribution of luminosity with limited control of glare. Consequently, a high beam can be used only when the road is empty, as the glare it produces may blind other drivers. On the contrary, a low beam headlight produces a light distribution to permit an adequate forward and lateral illumination without blinding other drivers with excessive glare. Therefore, it can be used whenever other road users are present ahead.

In this industrial use case provided by *pure::systems GmbH*, maker of the market leader tool for variability management pure::variants, we detail the construction of a family of 28688 external front car light (ECL) systems in FX-MAN. Section 7.2 defines the ECL family requirements, categorised in the feature model of Fig. 7.2, whereas

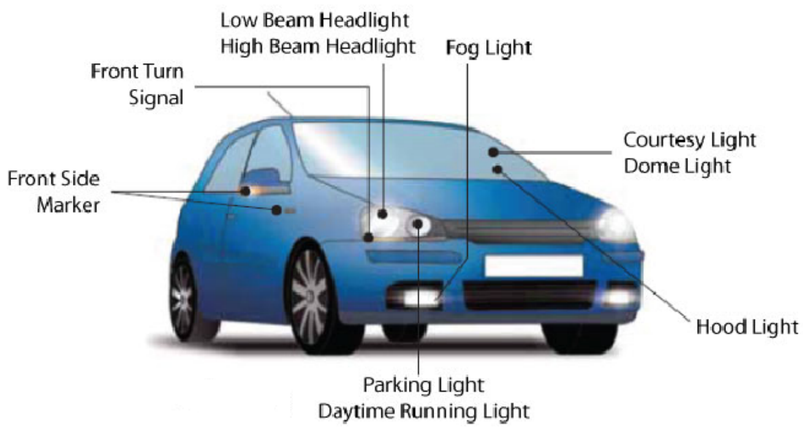


Figure 7.1: An external car light system.

section 7.3 details the steps involved in its modelling. Finally, section 7.4 summarises the chapter and describes the challenges one would face if not adopting the FX-MAN approach.

7.2 Requirements

The feature model in Fig. 7.2 contains 26 features distributed across 4 levels. Starting from the children of the root node 'External Car Lights' we now details the requirements of each feature.

- *Fog Lights* produce a wide, bar-shaped beam of light with a sharp cut-off at the top. They are manually activated by the driver.
- An ECL system must support *Beam Configuration* for both *Low Beam* and *High Beam*. A beam can either be *Xenon* or *Halogen* and activated both manually and automatically.
- A *Daytime Running Light* is an emitting light that is automatically switched on when the vehicle is moving forward, thus to increase its conspicuity during daylight conditions. It is realised by using either a (*Reduced Low Beam*) lamp or a dedicated one. The latter can be either *LED* or a *Standard Bulb*.
- *Driver Assistance* are systems aimed at increasing driver safety. It is realised by at least one (or any combination) of the following sub-systems: *Automatic Light*, *Automatic High/Low Beam Detection* and *Cornering Lights*.

- *Automatic Light* makes use of an optical sensor positioned on the inside of the vehicle's windscreen to detect the environment light level. When the light level drops to less than 1000 lux, the electronic control unit automatically activates vehicle's *Low Beam* and *Cornering Lights*. When the light level returns to 3000 lux, the lights are automatically switched off.
- *Automatic High/Low Beam* supports driver by automatically switching headlights accordingly to ahead traffic conditions. A camera integrated in the rear-view mirror monitors the headlights of approaching vehicles, as well as the rear lights of vehicles ahead. The system automatically switches the headlights from high beam to low beam, returning to high beam as soon as other drivers are no longer in danger of being dazzled.
- *Cornering Lights* is realised by at least one (or any combination) of the following sub-systems: *Static Cornering Lights* and *Adaptive Forward Light*.
  * *Static Cornering Light* provide an extra light source behind the headlight reflector. When the car is moving at a speed of at least 10 m/s and the steering wheel is above 15 degree the system automatically activates (if present) the daytime running lights. Furthermore, If the steering wheel angle goes above 25 degrees it activates (if present) the fog lights. As a result, driver gets a better view of potential hazards.
  * *Adaptive Forward Light* allows to modify beam direction and shape according to road geometry. In entering a corner, the system steers the light cone inside the curve. This provides a better view into the corner, allowing earlier identification of possible dangers ahead. It is activated only when high or low beam is operating in full light mode.

It is important to notice that the ECL feature model in Fig. 7.2 expresses a 'requires' constraint between the features *Static Corner Lights* and *Fog Lights* that reduces the number of valid ECL systems to 386. This is an important constraints that needs to be taken into account when modelling the ECL family in FX-MANas avoiding it would lead to the construction of invalid products.

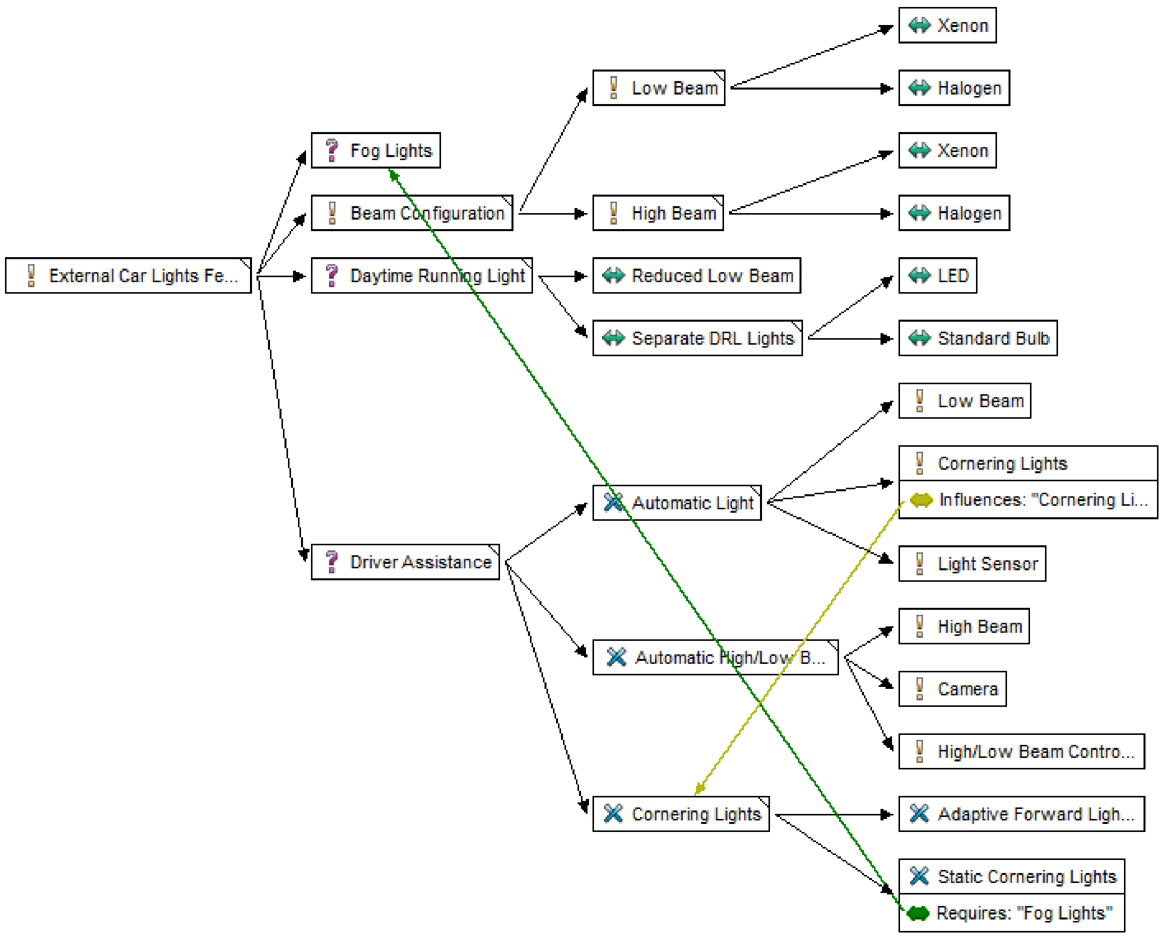


Figure 7.2: Automotive front light controller feature model

7.3 Building the ECL Product Family

Starting from the four-levels feature model in Fig. 7.2, we describe the steps needed to construct the ECL PFA.

7.3.1 Step 1 - Implement leaves features

The first step is to implement the leaves features (nodes at the lowest level of each sub-tree) as X-MAN components. The ECL feature model has 14 leaf features, each of which implemented by an encapsulated component as listed in Table 7.1. Features whose behaviour is simple enough to not require further decomposition are realised as atomic components. That is, behaviours such as toggling a bulb (features *Reduced Low Beam*, *LED* and *Standard Bulb*), interrogating a sensor status (features *Light Sensor*, *Camera*), or processing sensor data to control beams status (feature *High/Low Beam Controller*), can be directly implemented inside computation units.

Feature Name	**Component Name**	**Component Type**
Fog Lights	FogLights	Composite
Xenon	LowBeamXenon	Composite
Halogen	LowBeamHalogen	Composite
Xenon	HightBeamXenon	Composite
Halogen	HightBeamHalogen	Composite
Reduced Low Beam	DRL_LowBeam	Atomic
LED	DRL_LED	Atomic
Standard Bulb	DRL_Bulb	Atomic
Adaptive Forward Light	AdaptiveForwardLight	Composite
Static Cornering Light	StaticCornerLight	Composite
Light Sensor	LightSensor	Atomic
Camera	CameraFeed	Atomic
High/Low Beam Controller	HighLowBeamController	Atomic

Table 7.1: Implementation of leaves features as X-MAN components.

Fig. 7.3(a) shows the architecture of the `DRL_LED` atomic component, which implements the feature *LED*. `DRL_LED` provides a toggleLight service, which has two (boolean) parameters, *currentStatus* in input and *status* in output, to keep track of the bulb status (false off, true on). Fig. 7.3(b) shows the `DRL_LED` generated activity chart, where the service toggleLight is represented as an activity, *currentStatus* and *status* are represented as data flow coming from and to the external activities *Input Parameter* and *Output Parameter* respectively. The toggleLight activity is controlled by the control activity DRL_LED (rounded rectangle), which behaviour is specified by the state chart in Fig. 7.3(c). From the initial state, the machine transits to the state *Computing toggleLight* in which the beam toggle is computed. Once computation ends, the machine transits to the final state.

Behaviour of the remaining features listed in Table 7.1 are implemented by X-MAN composite components. *Xenon* and *Halogen* features (children of the feature *Low Beam*) are implemented by the components `LowBeamXenon` and `LowBeamHalogen` respectively. Similarly, the behaviour of the features *Xenon* and *Halogen* (children of the *High Beam* feature) is encapsulated by the components `HighBeamHalogen` and `HighBeamXenon` respectively.

Fig. 7.4 depicts the architecture of the latter. As already described at the beginning of this section, the high beam is activated when the high beam lever is pulled by the driver and the light mode switch is set to full light mode. To realise such behaviour, a sequencer firstly invokes the service CheckLightMode exposed by the component `LighModeSensor`, secondly it invokes the service getLeverPosition provided by the component `LeverPositionSensor`, thirdly it invokes a guard adapter, which checks that the light mode status is equal to 1 (full mode). If so, the control goes to a second guard which checks that the lever position is equal to 1 (high beam). If it evaluates true, than the service toggleLight exposed by the component `XenonActuator` toggle the light only if the bulb *currentStatus* is false (off).

The `HighBeamXenon` component generated functional model is depicted in Figs. 7.5. The activity chart shows the activity *toggleHighBeam* containing three sub-activities *CheckLightMode*, *toogleLight*, *getLeverPosition* corresponding to the aforementioned services. The activity *toggleLight* receives an inbound data flow **highBeamStatus**, and provides a **status** data flow as output. *CheckLightMode* and *getLeverPosition* instead provide a control flow data (*status* and *position*) to the control activity HighBeamXenon. Such data is then used by the state machine to evaluate the transition to the composite state *XenonActuator*, which compute the beam activation.

The behaviour of the remaining leaf features, or rather, *Standard Bulb*, *Adaptive Forward Light* and *Static Corner Light* has been implemented by the corresponding X-MAN components `DRL_Bulb`, `AdaptiveForwardLight` and `CornerStaticLights`. Their source code, along with their corresponding functional models can be downloaded at `https://goo.gl/jY1lLA`.

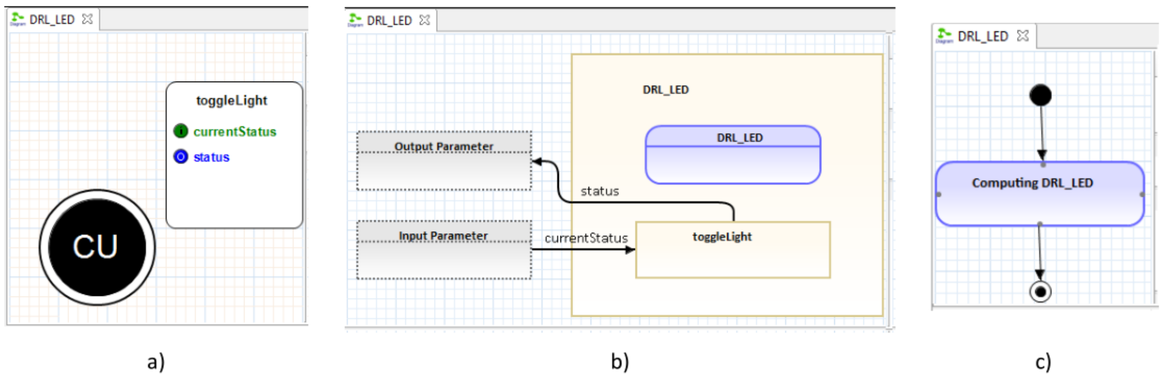


Figure 7.3: Architecture of DRL_LED component and relative functional model.

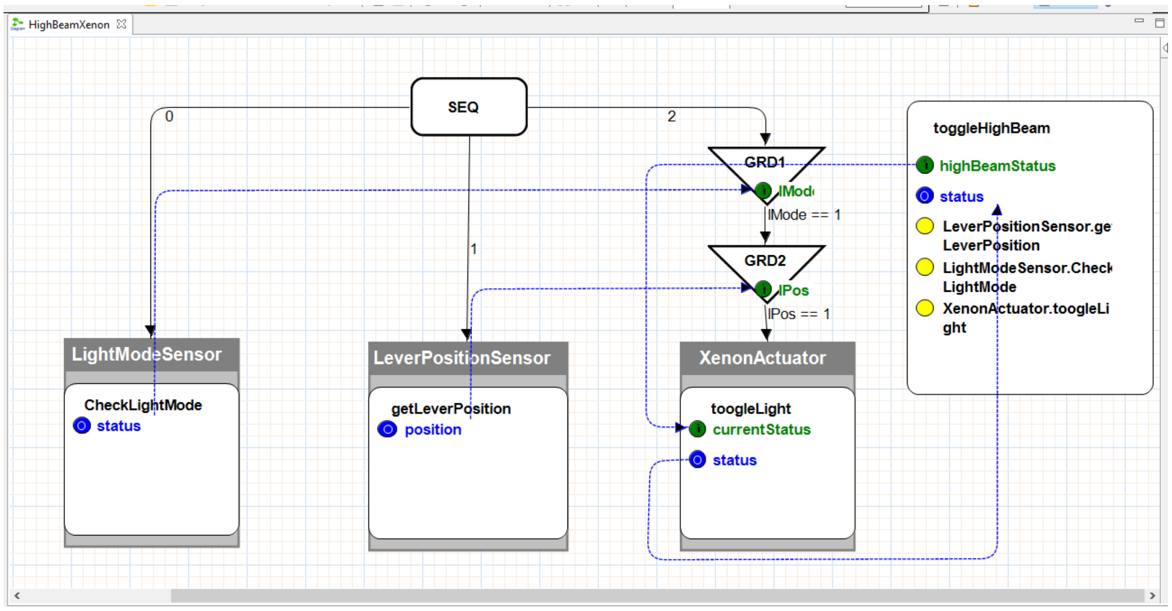


Figure 7.4: Architecture of the `HighBeamXenon` component.

7.3.2 Step 2 - Add Variation Operators

The second step is to add variation operators as defined by the ECL feature model in Fig. 7.2. To this end, we retrieve the components created in Step 1 and apply the specified operators to them. Fig. 7.6 illustrates the families of X-MAN sets constructed in this phase.

An *OPT* operator applied to an instance of the `FogLights` components yields the tuple $F_0 = \langle\{FogLights\}, \emptyset\rangle$. A family of *Low Beam* is created by applying an *ALT* operator to the components `LowBeamXenon` and `LowBeamHalogen`. The result is the tuple $F_1 = \langle\{LowBeamXenon\}, \{LowBeamHalogen\}\rangle$. Similarly, an *ALT* operator applied to the components `HighBeamXenon` and `HighBeamHalogen` gives rise to the tuple $F_2 = \langle\{HighBeamXenon\}, \{HighBeamHalogen\}\rangle$. An additional *ALT* operator applied to the components `DRL_LED` and `DRL_Bulb` realises the tuple $F_3 = \langle\{DRL\_LED\}, \{DRL\_Bulb\}\rangle$.

As Fig. 7.2 illustrates, *Automatic Light* has a sub-feature *Light Sensor*, and it demands features *Low Beam* and *Cornering Lights* as well. As sub-families for `Low Beam` and `Cornering Lights` were already developed (and deposited in our tool repository), the only component that needs to be built is the one for `Light Sensor`, which senses and calculates the external light condition. Since *Automatic Light* sub-feature and demanded features are all mandatory, we need to compose them into a new tuple

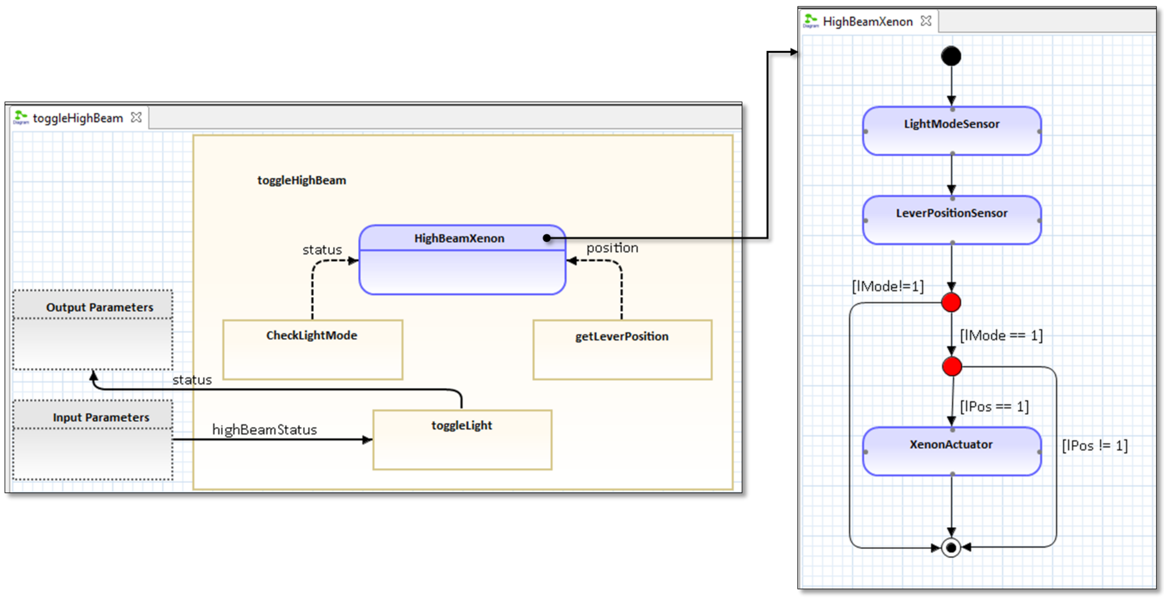


Figure 7.5: `HighBeamXenon` component corresponding functional model.

by mean of family connectors. Fig. 7.7(a) shows the architecture of the `Automatic Light` sub-family.[1] A family sequencer *F-SEQ* firstly invokes `LightSensor` to process the light conditions. Secondly, according to the calculated light conditions, it toggles low beams and then the cornering lights. The 6 variants it realises are listed in the product explorer view at the bottom of Fig. 7.7(a)). They are identified by the tuple

$F_4 = \langle\{Sequencer(LightSensor, AutoLightController, HalogenLowBeam, AutomaticForwardLight), Sequencer(LightSensor, AutoLightController, HalogenLowBeam, StaticCornerLight), Sequencer(LightSensor, AutoLightController, HalogenLowBeam, Aggregator(AutomaticForwardLight, StaticCornerLight)), Sequencer(LightSensor, AutoLightController, XenonLowBeam, AutomaticForwardLight), Sequencer(LightSensor, AutoLightController, XenonLowBeam, StaticCornerLight), Sequencer(LightSensor, AutoLightController, XenonLowBeam, Aggregator(AutomaticForwardLight, StaticCornerLight))\}\rangle$

In order to realise the refined *Automatic High/Low Beam* sub-family of Fig. 7.2, we reuse the sub-family `High Beam` (realised by the alternative components `XenonHighBeam` and `HalogenHighBeam`), and implement new components for *CameraFeed* and *High-/Low Beam Controller*. Fig. 7.7(b) depicts the PFA of the new sub-family. A family sequencer (*F-SEQ*) firstly invokes `CemeraFeed` to check the status of the incoming traffic. Secondly, it invokes `HighLowBeamController` to process the camera data

[1]Dashed arrows represent data flow.

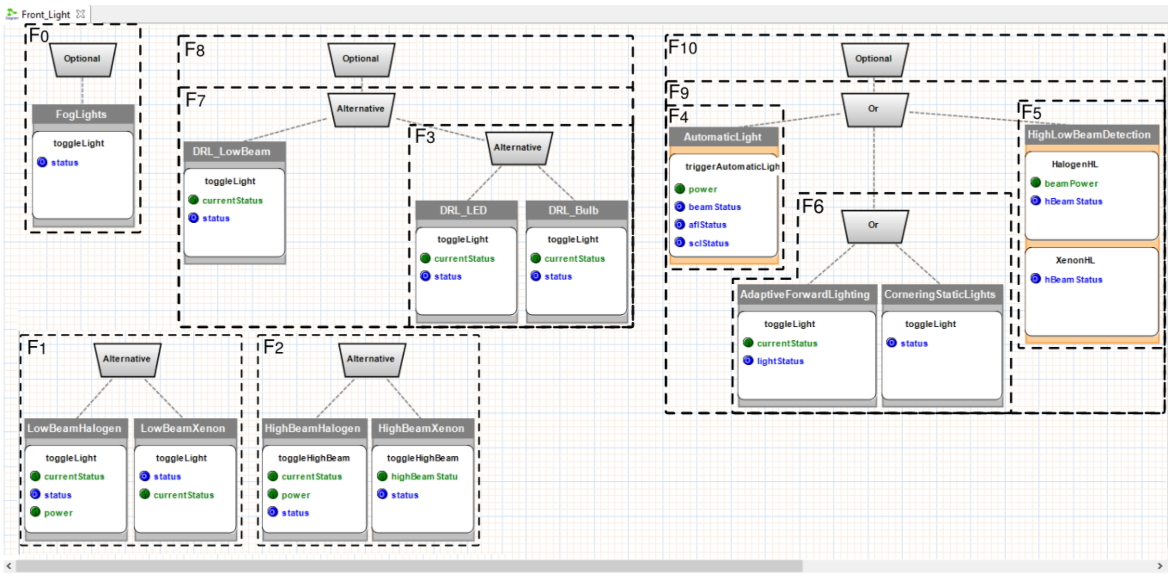


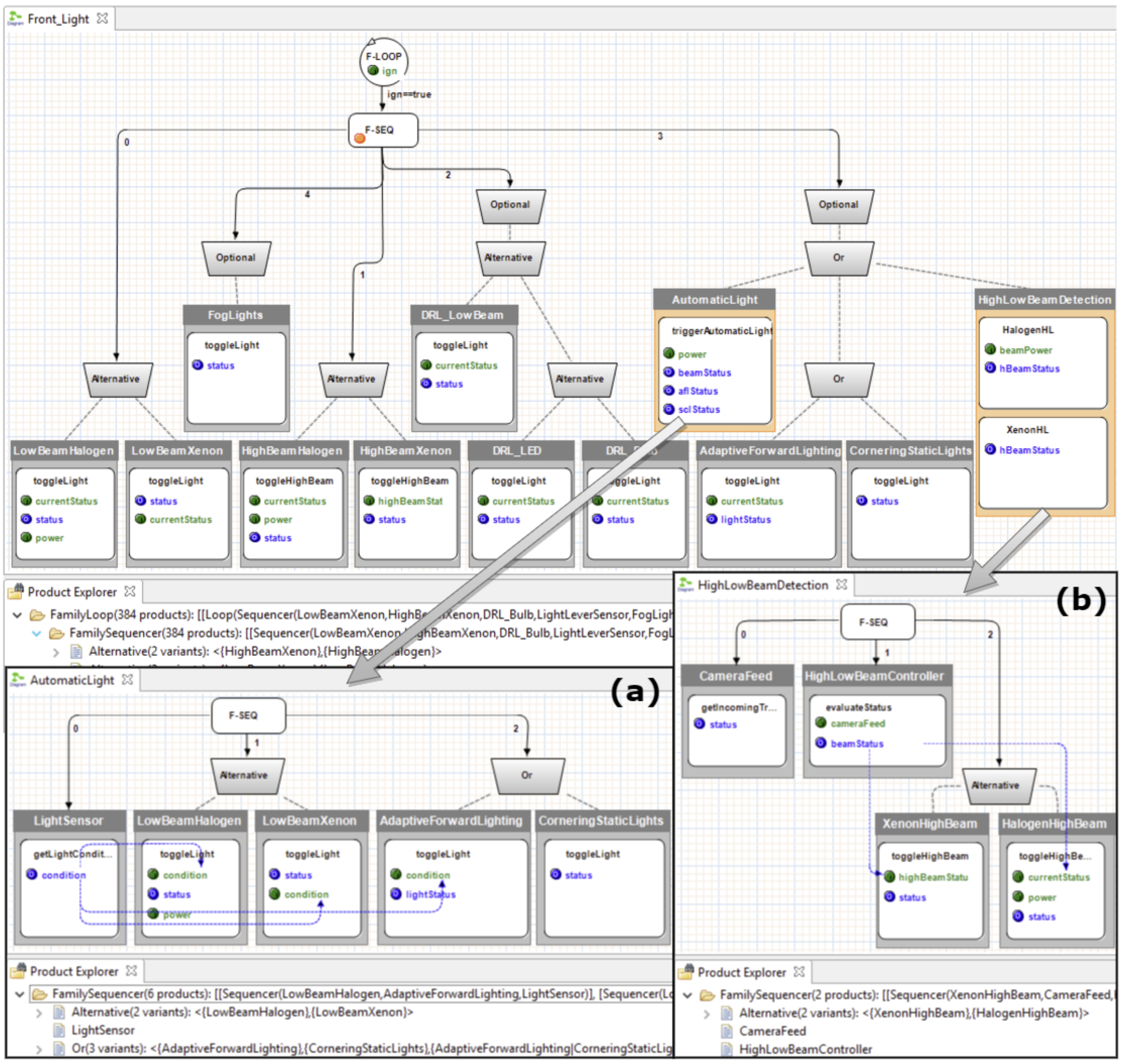
Figure 7.6: Families of X-MAN sets constructed during step 2.

and to evaluate the new beam status. Thirdly, *F-SEQ* invokes either *XenonHighBeam* or *HalogenHighBeam* to (if required) toggle their status. The result is the tuple of two variants $F_5 = \langle\{Sequencer(CameraFeed, HighLowBeamController, XenonHighBeam)\}, \{Sequencer(CameraFeed, HighLowBeamController, HalogenHighBeam)\}\rangle$, as depicted by the product explorer view in (bottom of) Fig. 7.7(b).

An *OR* operator applied to the components `AdaptiveForwardLight` and `CornerStaticLights` yields the tuple $F_6 = \langle\{AdaptiveForwardLight\}, \{CornerStaticLights\}, \{AdaptiveForwardLight \oplus CornerStaticLights\}\rangle$, which realise the sub-family of *Cornering Lights*.

The optional feature *Daytime Running Light* is realised by applying an *ALT* operator to F_3 and to the component `DRL_LowBeam`, which implements the leaf feature *Reduced Low Beam*, and thereafter an *OPT* variation operator to the resulting tuple. The result is the sub-family $F_8 = \langle F_7, \emptyset\rangle$, where $F_7 = \langle F_3, \{DRL\_LowBeam\}\rangle$.

Finally, the variability defined by the optional feature *Driver Assistance* is obtained by applying an *OR* operator to the tuples F_4, F_5, F_6, and subsequently an *OPT* operator to the produced tuple. The result is the tuple $F_{10} = \langle F_9, \emptyset\rangle$, wherein $F_9 = \langle\{F_4 \oplus F_5\}, \{F_4 \oplus F_6\}, \{F_5 \oplus F_6\}\rangle$.



Product Explorer
- FamilyLoop(384 products): [[Loop(Sequencer(LowBeamXenon,HighBeamXenon,DRL_Bulb,LightLeverSensor,FogLigh
  - FamilySequencer(384 products): [[Sequencer(LowBeamXenon,HighBeamXenon,DRL_Bulb,LightLeverSensor,Fogl
    - Alternative(2 variants): <{HighBeamXenon},{HighBeamHalogen}>

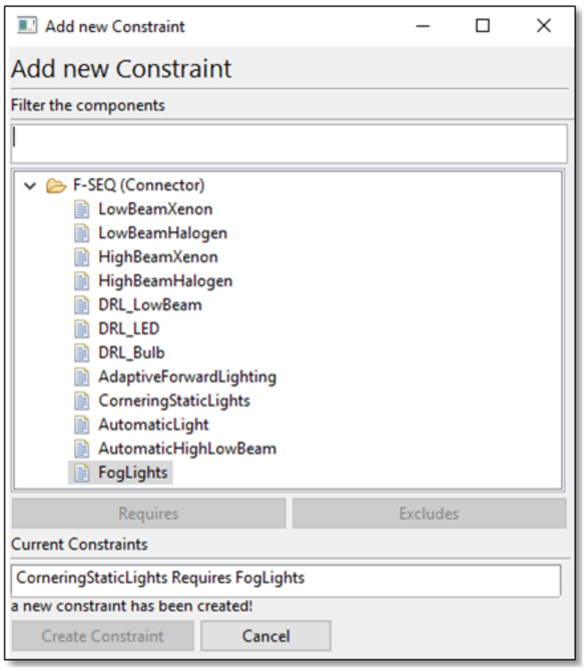
Product Explorer
- FamilySequencer(6 products): [[Sequencer(LowBeamHalogen,AdaptiveForwardLighting,LightSensor)], [Sequencer(L
  - Alternative(2 variants): <{LowBeamHalogen},{LowBeamXenon}>
  - LightSensor
  - Or(3 variants): <{AdaptiveForwardLighting},{CorneringStaticLights},{AdaptiveForwardLighting|CorneringStaticLig

Product Explorer
- FamilySequencer(2 products): [[Sequencer(XenonHighBeam,CameraFeed,
  - Alternative(2 variants): <{XenonHighBeam},{HalogenHighBeam}>
  - CameraFeed
  - HighLowBeamController

Figure 7.7: ECL product family architecture (and its sub-families).

7.3.3 Step 3 - Compose X-MAN Sets

At this stage, variability expressed in the feature model of Fig. 7.2 has been captured by tuples of X-MAN sets. It remains to compose them in order to model the ECL product family. The choice of family connectors to adopt is a design decision guided by requirements. However, it will not affect the total number of products in the family as family-connectors always ensure that all combinations of X-MAN architectures are kept (see section 5.4). We choose to compose the tuples F_0, F_1, F_8 and F_{10} resulting from the previous step, by means of a family sequencer *F-SEQ*, as we want to allow the driver to choose any subset of the services provided by the ECL family components. The resulting tuple F_{11} is further adapted by a family loop *F-LOOP* to ensure that products will loop until the value of the parameter *ign* is evaluated true (i.e. ignition



CorneringStaticLights Requires FogLights

a new constraint has been created!

Figure 7.8: Family-filter dialogue.

is on).

The resulting ECL PFA depicted in Fig. 7.7 models 28688 products, reduced to 386 by filtering out (during composition) products that contain `StaticCornerLights` but not `FogLights` components. Fig. 7.8 shows the dialogue in which the aforementioned constraint has been created in the tool.

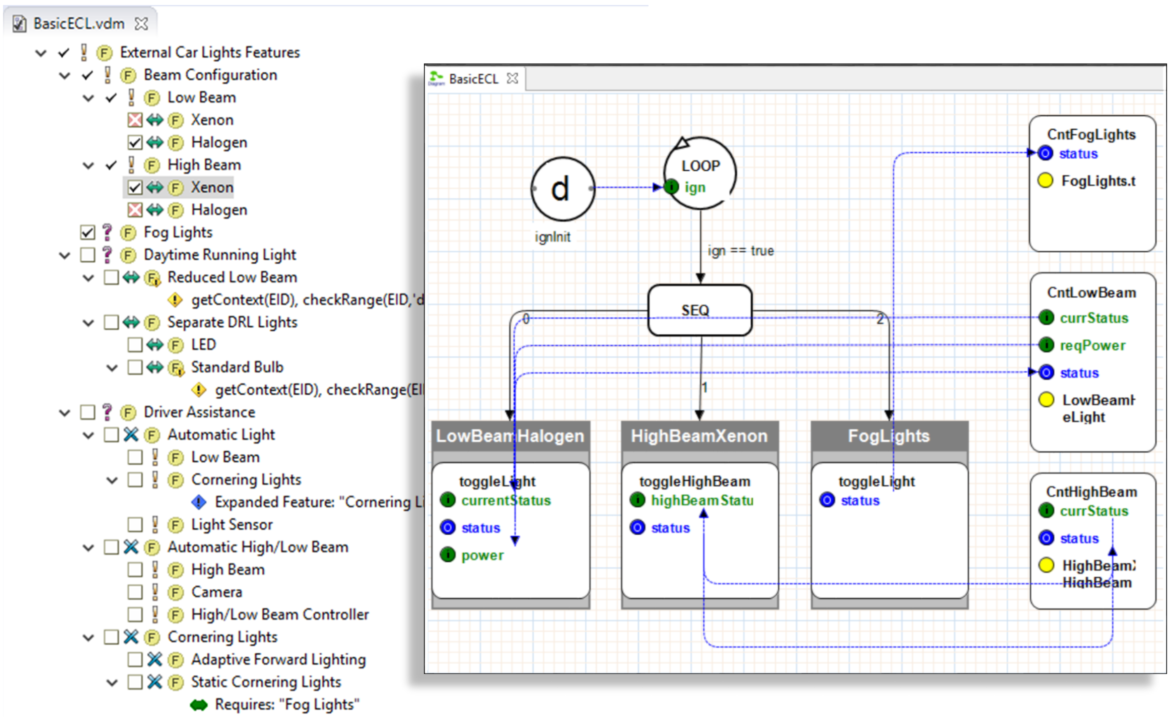
It is important to highlight the fact that as for X-MAN architectures, an FX-MAN one may contain exposed services and data-channels (for the sake of clarity they are not shown in Fig. 7.7). During product extraction, they become part of the variant architecture only if the following conditions are satisfied:

- all the services referenced by an exposed service are present;
- source and target parameters of a data-channel are present.

In case such conditions are not met, the extracted product needs to be manually customised before its functional model and code can be automatically generated.

7.3.4 Step 4 - Extract variants

Finally, the complete product family, or a single variant can be extracted. In order to decide which family member needs to be obtained, a feature configuration that



```
BasicECL.vdm
✓ External Car Lights Features
  ✓ Beam Configuration
    ✓ Low Beam
      ☒ Xenon
      ☑ Halogen
    ✓ High Beam
      ☑ Xenon
      ☒ Halogen
  ☑ Fog Lights
  ☐ Daytime Running Light
    ☐ Reduced Low Beam
        getContext(EID), checkRange(EID,'d
    ☐ Separate DRL Lights
      ☐ LED
      ☐ Standard Bulb
          getContext(EID), checkRange(EI
  ☐ Driver Assistance
    ☐ Automatic Light
      ☐ Low Beam
      ☐ Cornering Lights
          Expanded Feature: "Cornering Li
      ☐ Light Sensor
    ☐ Automatic High/Low Beam
      ☐ High Beam
      ☐ Camera
      ☐ High/Low Beam Controller
    ☐ Cornering Lights
      ☐ Adaptive Forward Lighting
      ☐ Static Cornering Lights
          Requires: "Fog Lights"
```

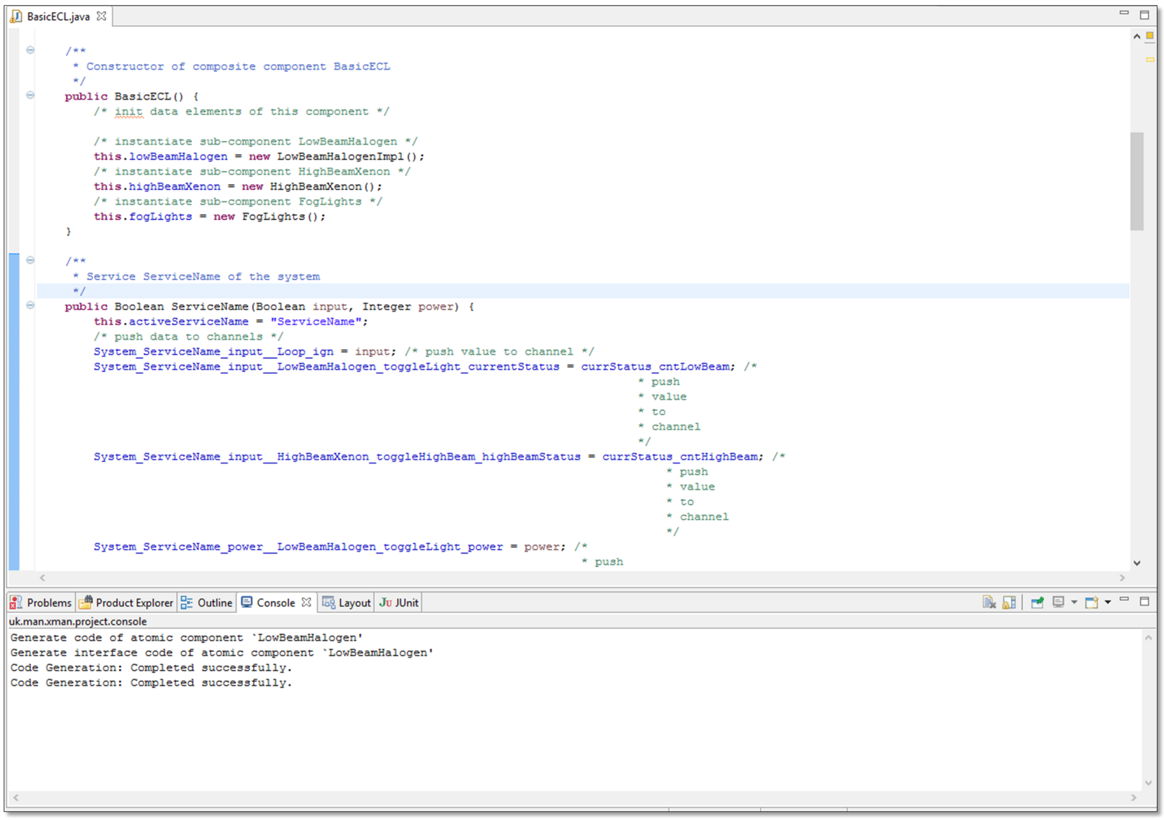
Figure 7.9: Configuration of a *basic ECL* in pure::variants VDM and the resulted product.

fulfils the stakeholder's needs has to be carried out. To do so, a pure::variants' variant description model (VDM) is created and the desired features are selected. Using this VDM and our pure::variants plug-in (presented in section 6.6.1), a family filter is generated allowing the extraction of the demanded product.

Fig. 7.9 shows VDM and architecture of one of the extracted products. It is a basic ECL (BasicECL), which supports *Halogen* low beams, *Xenon* high beams and *Fog Lights*. The X-MAN architecture on the right contains the components matching the selected features. They are `LowBeamHalogen`, `HighBeamXenon` and `FogLights`, coordinated by a sequencer `SEQ`, and adapted by a loop `LOOP`. Fig. 7.10 depicts a detail of the code automatically generated for the derived product, in which components and data channels are initialised.

Additionally, for each service the product variant exposes (i.e. `CntLowBeam`, `CntHigh-Beam` and `CntFogLights`) the tool automatically generates its activity chart (Fig. 7.11) and its relative state chart (Fig. 7.12).

`CntLowBeam` (top, Fig. 7.11) contains a sub-activity `toogleLight` (described by the `LowBeamHalogen`' activity chart). It receives two inbound data flows (*currStatus* and *reqPower*) and provides a *status* data flow as output.



```
    /**
     * Constructor of composite component BasicECL
     */
    public BasicECL() {
        /* init data elements of this component */

        /* instantiate sub-component LowBeamHalogen */
        this.lowBeamHalogen = new LowBeamHalogenImpl();
        /* instantiate sub-component HighBeamXenon */
        this.highBeamXenon = new HighBeamXenon();
        /* instantiate sub-component FogLights */
        this.fogLights = new FogLights();
    }

    /**
     * Service ServiceName of the system
     */
    public Boolean ServiceName(Boolean input, Integer power) {
        this.activeServiceName = "ServiceName";
        /* push data to channels */
        System_ServiceName_input__Loop_ign = input; /* push value to channel */
        System_ServiceName_input__LowBeamHalogen_toggleLight_currentStatus = currStatus_cntLowBeam; /*
                                                                                 * push
                                                                                 * value
                                                                                 * to
                                                                                 * channel
                                                                                 */
        System_ServiceName_input__HighBeamXenon_toggleHighBeam_highBeamStatus = currStatus_cntHighBeam; /*
                                                                                     * push
                                                                                     * value
                                                                                     * to
                                                                                     * channel
                                                                                     */
        System_ServiceName_power__LowBeamHalogen_toggleLight_power = power; /*
                                                                     * push
```

```
uk.man.xman.project.console
Generate code of atomic component `LowBeamHalogen'
Generate interface code of atomic component `LowBeamHalogen'
Code Generation: Completed successfully.
Code Generation: Completed successfully.
```

Figure 7.10: Details of the *basic ECL* generated code.

`CntFogLight` (middle, Fig. 7.11) contains the sub-activity *toggleLights* (described by the `FogLights`' activity chart), which computed data (*status*) is provided as output parameter.

`CntHighBeam` (bottom, Fig. 7.11) contains the sub-activity *toogleHighBeam* (described in Step 1 of this section), which receives the beam current status (*currStatus*) as inbound data and returns the computed *status* as output parameter.

In all the three activity charts, the data store *ingInit* provides control flow to the control activity `BasicECL`, whose behaviour is described by the generated state-chart in Fig. 7.12. If the value of the provided is false *[ign == false]* the machine goes directly to the end state. On the contrary, the machine transits to the states *LowBeamHalogen*, *HighBeamXenon* and *FogLights* (all specified by the compound state-charts as in the Figure) to return to the *Xor* connector.

7.4 Summary

In this chapter we have detailed the modelling of a family of 28688 external car light (ECL) systems. This industrial use case provided by *pure::systems GmbH* has demonstrated the key features of FX-MAN described in chapter 4: (i) algebraical modelling; (ii) design time evaluation; (iii) automatic functional model translation; (iv) complete family extraction without configuration.

As illustrated by the comparison framework presented in chapter 3, current SPLE modelling approaches construct a template that needs to be configured one at the time according to the specific family member one wants to build. For instance in annotation-based SPLE approaches such as *pure::variants*, family members are obtained by removing parts of the ECL code base, whereas in weaving-based SPLE approaches variants are derived by adding parts to a base-model. This implies that evaluating the ECL family amounts to the configuration of all its members. This is particularly true in absence of a functional model.

Moreover, the lack of algebraical composition operators does not allow current SPLE modelling approaches to construct systems of systems in a systematic way. This implies that constructing the ECL family using an ADL-based SPLE modelling approach, would have scattered the instantiation of the same components in several composite-components, with evident problems for features localisation and family evolution.

As demonstrated in this use case, the nature of the FX-MAN building blocks lends itself to the construction of product family architectures structurally isomorphic to a given feature diagram. Indeed, in FX-MAN, composition of the building blocks is strictly hierarchical, hence the architecture of any product family is a tree. This means that a variant is a hierarchical composition of components. Therefore, we use components to implement features in a feature model and construct a PFA from these components. The result will be an architecture that naturally maps to a feature diagram as there is no variability mismatch.

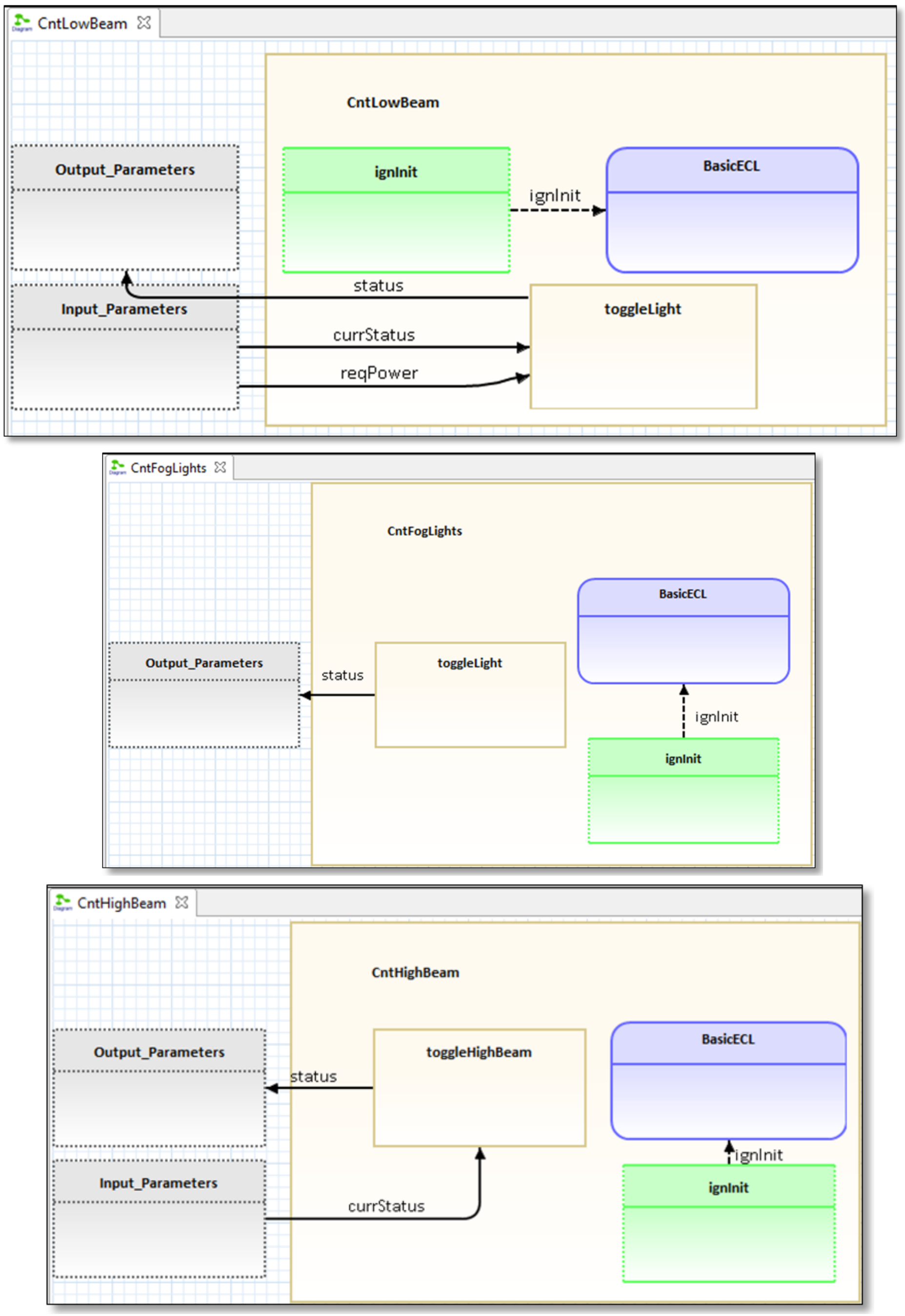


Figure 7.11: Generated activity-charts for the *BasicECL* product.

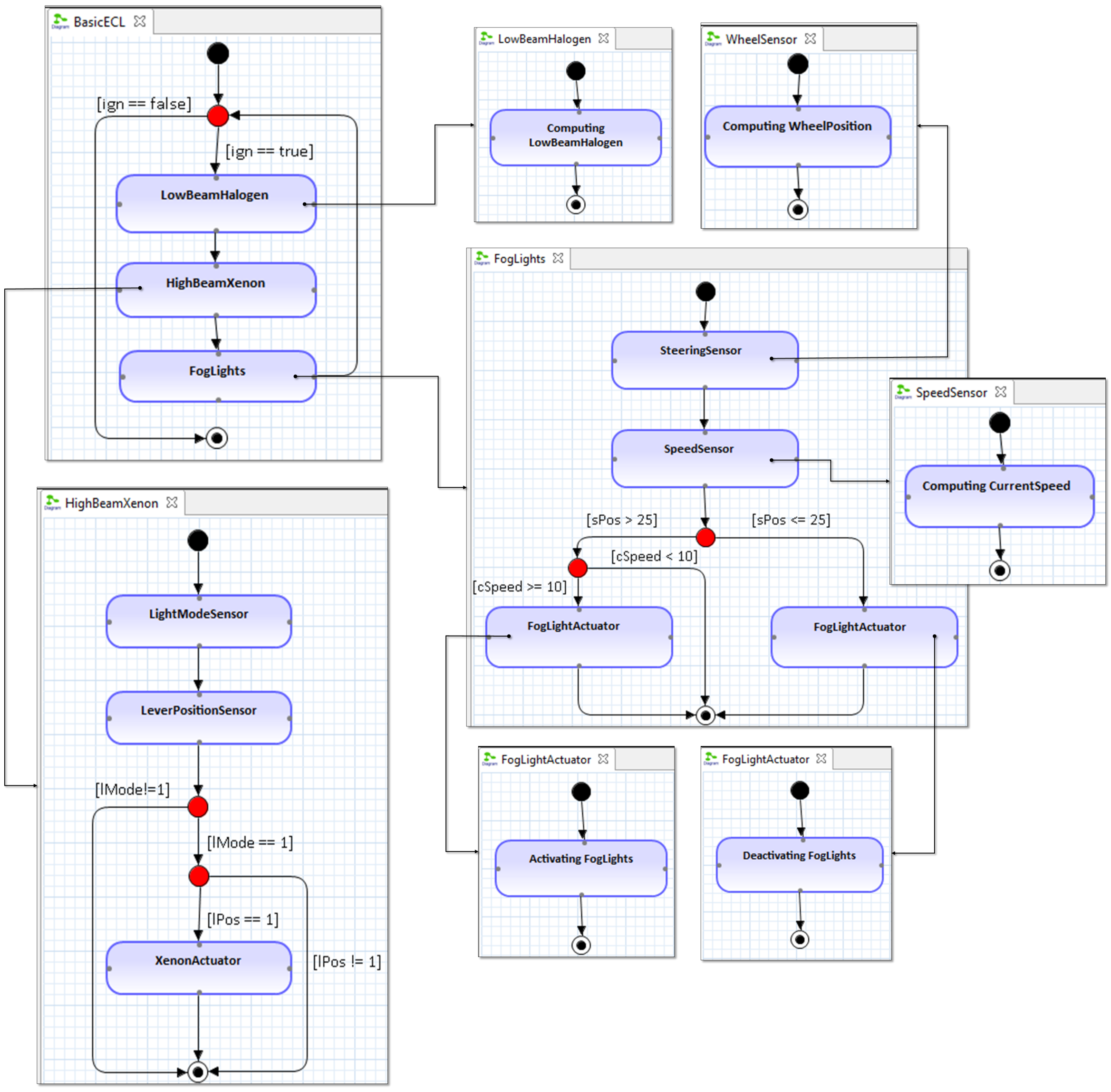


Figure 7.12: Generated state-charts for the *BasicECL* product.

Chapter 8

Evaluation

"In chess, knowledge is a very transient thing. It changes so fast that even a single mouse-slip sometimes changes the evaluation."
— Viswanathan Anand

8.1 Introduction

In order to gauge the potential of FX-MAN in enabling organisations to achieve the execution of both domain and application engineering together with a comprehensive variability's management, we use the Family Evaluation Framework (FEF)[28] as benchmark tool.

Section 8.2 introduces the evaluation framework, details its relevant part and justifies its relevance to the prosed modelling approach.

Section 8.3 applies the framework to the evaluate the maturity level of FX-MAN against the product family modelling approaches analysed in chapter 3.

Finally, section 8.4 analyses the result of the evaluation from two aspects: (i) execution of domain and application engineering; (ii) variability management.

8.2 Evaluation Framework

FEF is a consolidated result of three European co-operation projects (ESAPS [245, 246], CAFÉ [247] and FAMILIES [4]) with industry and academia, successfully applied in many case studies [34, 248–250]. It is based on the best industrial practises [28] and well established capability models as the Capability Maturity Model Integration (CMMI) [251], SEI Product Line Technical Probe (PLTP) [252] and BAPO [253]. Specifically, CMMI is used as basis for the process dimension within FEF; PLTP, which is based on the SEI's Framework for Software for Product Line Practice [254], is used to examine an organisation's compliance to adopt a software product family approach; BAPO provides the four dimensions used by the FEF, or rather Business, Architecture, Process and Organisation.

As depicted in Fig. 8.1, FEF evaluates each of the BAPO concerns independently, each of which leads to its own evaluation value. Each dimension of the framework has a collection of aspects that are to be considered in the evaluation. Dependent on the evaluation of these aspects, a maturity level raging from 1 to 5 can be obtained. Business, process and organisation dimensions consider the non-technical aspects related to a product family such as the ability of an organisation to manage, predict and steer its development costs and profitability.

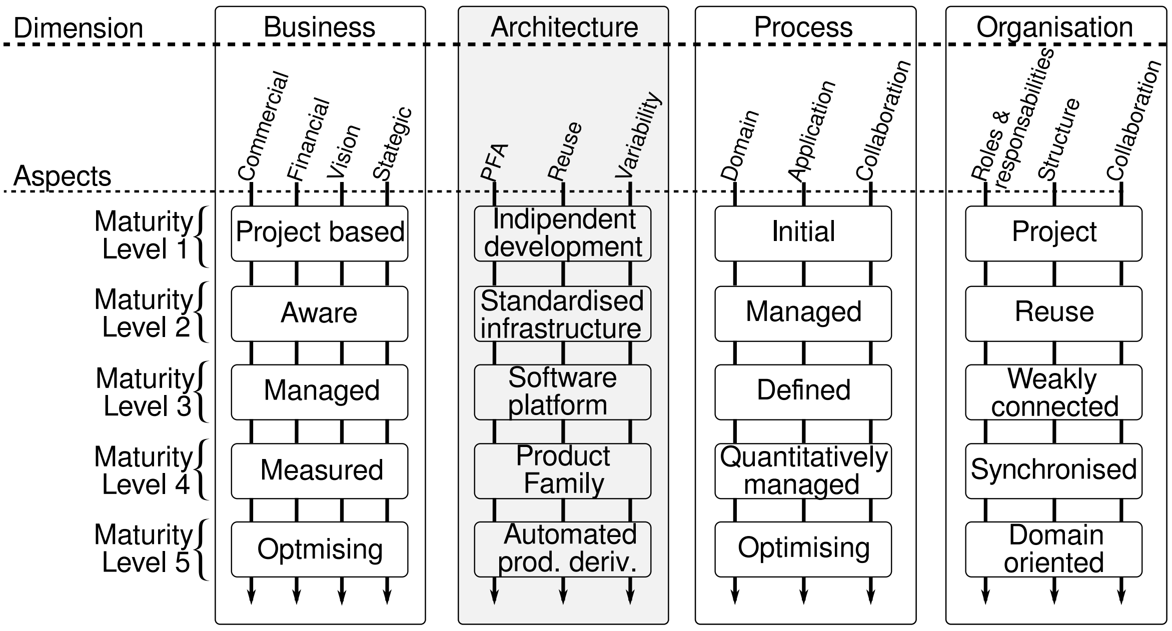


Figure 8.1: The Family Evaluation Framework (based on [253])

Since FX-MAN leaves out these concerns while focusing on the technical aspects of modelling a product family, we assess the potential of our approach using according to the FEF architecture dimension (FEF-A).

Highlighted in Fig. 8.1, the architecture dimension relates to the technical realisation of products in a scoped product family, with a focus on PFA. This dimension mainly focuses on the relationships between PFA and variants architectures, taking into account how variability is modelled in a PFA. The evaluation for this dimension is categorised by five maturity levels across three aspects:

- *PFA*: the extent to which the product family architecture (in domain engineering) determines variants architectures (in application engineering).
- *Asset reuse level*: the extent of the reuse of domain assets in application engineering.
- *Variability management*: the explicit use of variation points and supporting variability mechanisms.

In the following sections, each FEF-A maturity levels is discussed in detail.

8.2.1 Maturity Level 1: Independent Development

At this level, each product gets its own architecture as reuse is not visible at architectural level, and variability is not managed. In terms of the FEF-A aspects, we have the following situation:

- *PFA*: there is no available PFA.
- *Asset reuse level*: reuse is either not present or it is ad-hoc.
- *Variability management*: variability is unmanaged.

8.2.2 Maturity Level 2: Standardises Infrastructure

At this level, a common third-party software infrastructure is defined. There is no formal reuse of domain-specific assets, and variability is determined by the third-party infrastructure. Related to the FEF-A aspects, we have the following situation:

- *PFA*: there is a common third-party infrastructure (*e.g.* middleware) defined and in use.
- *Asset reuse level*: reuse is ad-hoc, mainly based on the repository of the third-party artefacts.

- *Variability management*: variability is limited to the one offered by the standardised infrastructure and it is open to be determined by the application architecture.

8.2.3 Maturity Level 3: Software Platform

At this level, a configurable software platform, built as a collection of common assets in a domain repository, captures domain commonalities. Reuse is determined by the platform and there is no support for its configuration. In terms of FEF-A aspects, we have the following situation:

- *PFA*: domain commonality is captured and implemented in a software platform using proprietary components. It contains rules that determine its use in order to evaluate the validity of a configuration.
- *Asset reuse level*: reuse is confined to the software platform and restricted by architectural constraints.
- *Variability management*: explicit variation points determine where application-specific variants may be bound. However, there is no variability support for product derivation.

8.2.4 Maturity Level 4: Product Family

At this level, a PFA specifies the complete software product line. Variability is explicitly defined and managed by the PFA, and it also include support for variants derivation.

- *PFA*: there is a family architecture that explicitly specifies domain commonality and variability.
- *Asset reuse level*: reuse is systematic and based on an asset repository. Variability is explicitly defined within the assets.
- *Variability management*: it is explicitly addressed in the PFA, which determines rules that application-specific variants have to obey.

8.2.5 Maturity Level 5: Automated Product Derivation

At this level, the PFA is enforced and determines the application architectures completely, with automated configuration support to derive specific applications. This implies that application engineering plays a marginal role.

- *PFA*: it is dominant and variants' architecture divert only marginally from it.
- *Asset reuse level*: reuse is systematic and based on a shared repository.
- *Variability management*: variability is fully integrated into the architecture. Variants are automatically derived.

8.3 Applying FEF-A to SPLE modelling approaches

To summarise the FEF architecture dimension, Table 8.1 illustrates that: (i) from an initial level in which no *PFA* is established, this aspect grows to a PFA that governs the whole product family; (ii) from an initial level of unsystematic or no *reuse*, this aspect grows to a level in which reuse is systematically managed through explicit variation points; (iii) from an initial level in which *variability management* is absent, this aspect grows to a level where variability is fully-integrated in a PFA and product variants are automatically derived.

Lv	**PFA**	**Asset reuse level**	**Variability management**
1	not established	no reuse / ad-hoc	absent
2	third-party components	ad-hoc	limited
3	common platform	limited to platform	determined by platform
4	fully specified	systematic	determined by PFA
5	enforced	systematic, self-configurable	fully-integrated in PFA

Table 8.1: FEF architecture dimension maturity levels.

By assigning a level for each aspect in this dimension we evaluate our SPLE modelling approach against the ones presented in chapter 3.

Annotation-based approach

- *PFA*: not established (**maturity level 1**).
- *Asset reuse level*: assets are systematically reused by removing unselected parts from the domain artefacts (i.e. 150% models) (**maturity level 4**).
- *Variability management*: variability is expressed within domain artefacts by explicit variation points defined by naming the features under which each configuration applies (**maturity level 3**).

Overall: **maturity level 3**.

Weaving-based approach

- *PFA*: not established (**maturity level 1**).
- *Asset reuse level*: as aspects apply advice at specific pointcuts, their reuse is restricted to the target base model (**maturity level 3**).
- *Variability management*: pointcuts in the base model determines where variant-specific aspects may be bound (**maturity level 3**).

Overall: **level 2**.

Superimposition-based approach

- *PFA*: not established (**maturity level 1**).
- *Asset reuse level*: reuse is limited by structural and nominal similarities between artefact fragments (**maturity level 3**).
- *Variability management*: domain artefact fragments are composed by merging their sub-structures on demand according to the provided feature configuration (**maturity level 3**).

Overall: **maturity level 2**.

Δ-based approach

- *PFA*: not established (**maturity level 1**).
- *Asset reuse level*: as for aspects, reuse of deltas is restricted by the target based model (**maturity level 3**).
- *Variability management*: pointcuts in the base model determines where variant-specific delta may be bound (**maturity level 3**).

Overall: **maturity level 2**.

ADL-based approach

- *PFA*: fully specified, but with limited explicit variability (**maturity level 4**).
- *Asset reuse level*: the composition mechanisms (i.e. port connection) allows a systematic reuse of components within the PFA (**maturity level 4**).
- *Variability management*: the PFA determines the variation points as well as their usage constraints (**maturity level 4**).

Overall: **level 4**.

Our approach

- *PFA*: fully specified, with automated configuration support to derive product variants (**maturity level 5**).
- *Asset reuse level*: the exogenous composition mechanisms and the explicit variation points allow a systematic reuse of components within the PFA (**maturity level 4**).
- *Variability management*: variability is fully integrated in the PFA by explicit variation operators with fixed semantic. Variants are derived automatically (**maturity level 5**).

Overall **maturity level 5**.

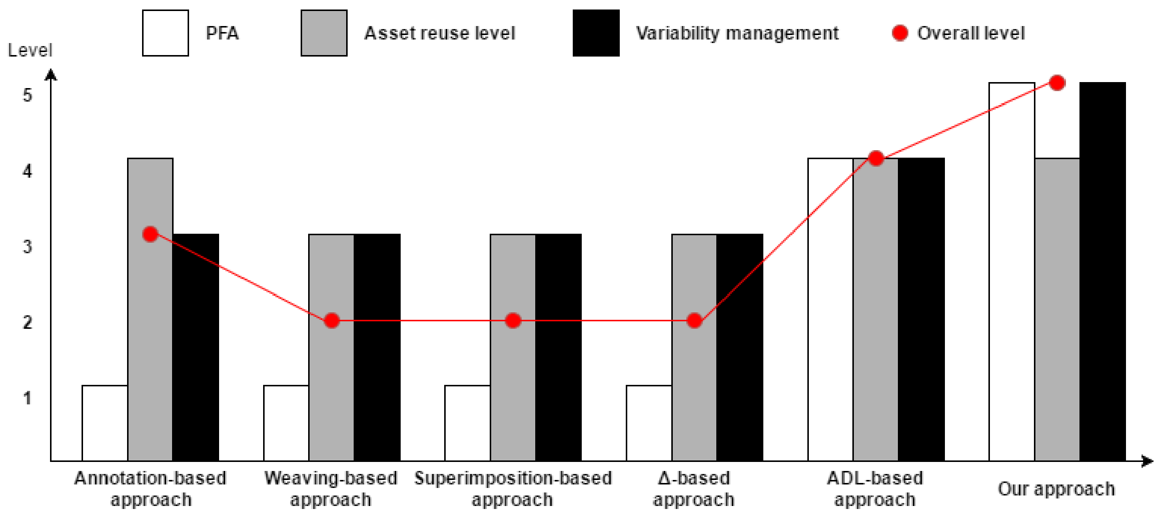


Figure 8.2: SPLE modelling approaches evaluation (based on FEF-A metrics).

8.4 Analysis of Results

Fig. 8.2 depicts for each SPL modelling approach (described in chapter 3) the maturity level reached in each FEF-A aspect along with their overall score (red dot). As the Figure shows, ours is the only modelling approach that reaches an overall 5 maturity level. This implies that FX-MAN enables *automated product derivation* by means of a PFA with explicit variation points that determines the application architectures completely, thus minimising the role played by the application engineering phase.

In general, the level of automation realised by SPLE modelling approaches with an overall FEF-A maturity level of at least 4 characterises what is defined by Krueger *et al.* [255, 256] as Second Generation Product Line Engineering (2GPLE) approaches.

1GPLE approaches, or rather SPLE modelling approaches with an overall FEF-A maturity level 2 and 3, have never enforced the concept of automation. However, large-scale software product lines are so complex that demand it. To give a yardstick, the feature space of the industrial case presented by Flores *et al.* in [255] is so complex that the number of possible variants exceeds the number of atoms in the visible universe.

1GPLE (depicted in Fig. 2.1) emphasise on a clear separation of the activities carried out by the domain and application engineering phases: core assets developed during the domain engineering phase are used in the application engineering one to create products "with reuse". Undeniably, this is the natural evolution from component-based software reuse. For instance, in weaving-based SPLE modelling approaches, base and aspect models constitute the core assets reused to derive product variants. At first glance, everything seems to go smoothly. However, three problems arise once the SPL is evolved and maintained over the time.

Firstly, during the application engineering phase, developers are focused on creating product-specific configurations and " variability" in each product. This results in unique systems, that often require a dedicated team of engineers to produce them. By increasing the number of products, the number of required engineers increases as a consequence.

Secondly, each product has its own context in which core assets are reused. This makes it difficult to enhance the core assets in a non-trivial way; changes that have occurred in an SPL asset need to be merged into the specific product.

Thirdly, there could be conflicts between teams working in domain engineering phase and application engineering one, leading to a culture of "us-against-them".

As a result, 1GPLE approaches questioned about the right balance of effort that should have been spent in domain and application engineering. Fig. 8.3 shows the impact of these phases on a SPL costs. The Y-axis represents the accumulated overall cost for developing six products, while the bar on the X-axis shows the Y-cost for any given ratio of domain engineering and application engineering. It is possible to observe that a pure domain engineering approach is far more advantageous.

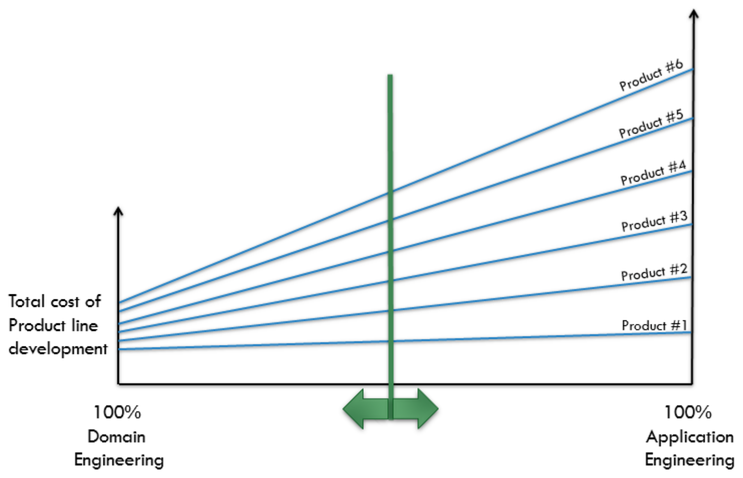


Figure 8.3: Impact of domain and application engineering in SPL costs [257].

It follows that 2GPLE paradigms consider the application engineering phase as harmful, therefore it shrinks to almost nothing [5]. Products are built via a product configurator, which automatically composes core-assets accordingly to the required features (Fig. 8.4). As explained in chapter 3, this approach is widely used in industry as supported by the market leader tools pure::variants and Gears. Due to the absence of the application engineering phase, all developers and engineers are focused on developing core assets. This implies that personnel can be organised around core assets, eliminating the need to add a development team each time a product is added. Therefore, core assets and product variants evolution coincide: any change in an asset can be followed by automated products re-instantiation without manual merging. Hence, 2GPLE approaches only manage core-asset versions.

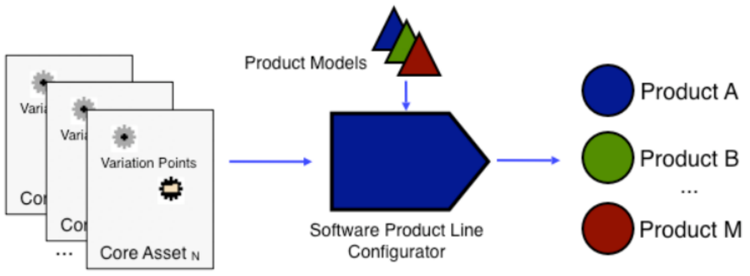


Figure 8.4: Software product line configurator [257].

2GPLE approaches aim to deliver an essential mean for large-scale product lines that was stated, but never achieved, by 1GPLE ones [5]. However, annotation-based and ADL-based SPLE modelling approaches lack of compositional semantics. As a consequence, they largely fail to manage both scalability and complexity even for normal size SPLs [152]. On the contrary, our approach described in chapter 5 and demonstrated in the industrial use case of chapter 7, delivers a systematic and strictly hierarchical compositional approach that scales from systems-of-systems to families-of-families. This has several advantages. The composition hierarchy is really a nested product line hierarchy; each nested product line can be reused in different product lines. Furthermore, decomposing a large product line into smaller product lines allows allocating smaller development teams to each of them.

From the point of view of the *variability management* aspect, Table 8.2(a) shows a comparison between FX-MAN and the component models presented in section 3.7. Some of these models do not define variation points explicitly and express variability by other means. For example, MontiArch^{HV} uses presence conditions, Δ-MontiArc[142] uses architectural deltas, while xADL2 [258] defines conditions in XML schemas. Other component models do define some variation points explicitly. For example, Koala defines the *Alt* variation point explicitly (as a *switch* between components), but not *Opt* and *Or* (these can be simulated by parameters in the *diversity interface* of a component to change its internal structure). By contrast, FX-MAN explicitly defines the full standard set of variation points that appears in feature models: *Opt*, *Alt* and *Or*. However, as previously mentioned, the key difference between FX-MAN and the other component models in Table 8.2(a) is that in FX-MAN an architecture is defined for a product family containing all the products, whereas in the other models individual products have to be derived one at a time as instances of the templates. For example, in Koala a parametrised architectural template includes a configurator (*module*), *switch* connectors and *diversity interfaces*. For each setting of the configurator, the Koala complier removes from the architectural template the undesired components, as well as the unreachable code.

(a) Component models.

Component Model	**Explicit Variation Points**			**Product Template/ Family**
	Alt	Opt	Or	
ADLARS	×	×	×	Template
MontiArc^{HV} ΔMontiArc	×	×	×	Template
KobrA	×	×	×	Template
Mae	×	×	×	Template
Plastic Partial Components	×	×	×	Template
xADL2	×	×	×	Template
LightPL-ACME	×	×	×	Template
Koala Koalish	✓	×	×	Template
COSMOS*-VP	✓	×	✓	Template
Com^2	✓	✓	×	Template
Kumbang	✓	✓	×	Template
Our model: FX-MAN	✓	✓	✓	Family

(b) Variability handling approaches.

Variability Handling Approach	**Mandatory Features**	**Variant Features**	**Configuration Points**	**Product Template/ Family**
`Annotating' e.g. cpp, FArM	Code Base	Code Base	Annotations	Template
`Weaving' e.g. XWeave, Lee	Code Base	Aspects	Pointcuts	Template
`Superimposition' e.g. Czarnecki, Apel	Artefact Fragment	Artefact Fragment	Presence Condition	Template
`Delta' e.g. Haber,Behringer	Code Base	Deltas	Pointcuts	Template
FX-MAN	Components	Components	Variation Points	Family

Table 8.2: Variability representation in current SPLE modelling approaches.

In a wider context, SPLE methods and tools that do not construct architectures (or use a component model), rely on variability handling mechanisms. Table 8.2(b) shows a comparison between FX-MAN and existing approaches to handle variability. There are four main categories of such approaches: (i) annotation-based (section 3.3) (ii) weaving-based (section 3.4) (iii) superimposition-based (section 3.5) (vi) delta-based (section 3.6).

Annotation-based approaches are widely used in industry [13] as they are very well supported through the commercial tools Gears [5] and pure::variants [71]. On the low level side the c-preprocessor (cpp), or FArM [259] are examples of such approaches. Here, artefacts as fragments of a code-base are annotated with statements like *#ifdef*. Product derivation is achieved by removing fragments that do not reflect feature selection.

Weaving-based approaches (*e.g.* XWeave [98] and AFM [113]) manage variability by applying the principles of aspect-oriented programming [89] at the meta-level. Base models are varied by pointcuts and advices: the former define where to affect the base model, while the latter specify how to modify it. Product derivation is achieved by weaving the set of aspect models corresponding to a particular feature configuration.

Superimposition is the process of composing fragments of software artefacts (e.g. code, UML diagrams) by merging their corresponding substructures on the basis of nominal and structural similarity. In approaches based on this technique (*e.g* Czarnecki [109] and Apel [107]) products are derived by merging only the fragments that satisfy their presence condition.

Delta-based approaches (*e.g.* Haber [260] and Behringer [261]) relies on the notion of program deltas (introduced in [262]) to model variability. Delta modules specify where to modify the core module (by adding, removing or modifying its parts) to implement products.

Like the component models in Table 8.2(a), the key difference between all these variability handling approaches and FX-MAN is that they define a template for a product family, and not an architecture for a product family as in FX-MAN. Individual products have to be configured one at a time using the template.

8.5 Summary

In order to evaluate the potential of our approach we have used the family evaluation framework (FEF) as benchmark tool. Based on the best industrial practices and well established capability models, FEF focuses on the two main aspects of modelling a product family: (i) the execution of both domain and application engineering; (ii) the comprehensive management of variability.

Compared to existing SPLE modelling approaches, ours is the only one that reaches the maximum overall maturity level (automated product derivation). This implies that the PFA is enforced and determines the application architectures completely, with automated configuration support to derive specific family members. Consequently, the application engineering phase plays a marginal role.

Moreover, reuse is systematic and based on a shared repository, with variability management fully integrated in the architecture via explicit variation points with fixed semantics. As a result, FX-MAN is the only modelling approach that instead of realising a template (architecture with place-holders) models the complete family.

Chapter 9

Conclusions and Future Work

"The thing is, it's very dangerous to have a fixed idea. A person with a fixed idea will always find some way of convincing himself in the end that he is right."

— Atle Selberg

The distinguishing characteristic of FX-MAN is its applicability to the construction of the architecture of a complete family of executable software products, together with the key advantage that variants can be analysed at design time without the need to be extracted.

Indeed, where FX-MAN constructs a PFA, other approaches construct templates of families, which have to be configured into products one at a time (table 8.2). For instance, modelling the same ECL family in Koala would require constructing a Koala architecture first, and then compiling it once for each family member. As far as we know, variation operators and family connectors for sets of architectures do not exist in other ADL. Annotation-based approaches use 150% models as templates, where products are configured by removing fragments that do not reflect feature selections. Feature-oriented modelling approaches gradually refine a base model with aspects (as for weaving-based approaches) or similar (sub)structures of artefacts (as for superimposition-based approaches) corresponding to a particular features selection. Similarly, Δ-based SPLE modelling approaches refine a base model by means of delta models according to the selected features.

However, enumerating a complete product family is an NP-hard problem: for large-scale families with a high degree of variability, enumeration and extraction of a complete family is costly both in terms of computation time and memory. As an example, in a product line with 216 simple boolean features, the number of possible products is comparable to the estimated 10^{65} numbers of atom in the entire visible universe. With only 33 boolean features, the combinatorics is comparable to the human population of our planet.

In order to limit the feature space and ensure efficiency of the modelling process, a divide-and-conquer strategy might be necessary to handle a large product family by decomposing it into sub-families. Happily, this is possible in FX-MAN thanks to its compositional nature, and its associated algebraical type system. The latter allows the modelling process to flow both vertically (from X-MAN component model to family composition) and horizontally (at each level of Fig. 5.1). Algebraic composition mechanisms are fundamental for systematic construction, since each composition is carried out in the same manner regardless of the level of the construction hierarchy.

We have demonstrated this in the industrial use case presented in chapter 7, where we have modelled a family of 28688 external car light systems by composing the two sub-families `AutomaticLight` and for `HighLowBeamDetection` (Fig. 7.6) into a bigger one.

Having the full set of variation points explicitly enables FX-MAN to be used to define architectures structurally isomorphic to the feature model in all cases. As a consequence, the mapping between artefacts in problem and solution space is facilitated: in our approach, one feature is instantiated by only one component (exposing the service defined by the feature) without any interaction. In case of cross-cutting features, they are encapsulated into shared resources invoked by the computation unit of each component (section 5.5).

The challenge of mapping features to components is well known in the literature [263]. It can be of two types: 1-to-1 and n-to-m.

Precondition of a 1-to-1 mapping is the presence of a feature model built in terms of components, achievable either by design, or through iterative refinements. FArM [259] and [264] are two methodologies, evaluated against real industrial case studies, that realise such a mapping.

In FArM, a feature model is iteratively refined until it exclusively contains functional features. Business logic of each feature is implemented by an architectural component, whose interface reflects the feature relationships.

In [264], starting from the leafs of a feature model, core components are mapped to mandatory features. Non-mandatory features are also implemented by single components. The latter are composed in a derived product via interfaces required by *relationship components*, also part of the core architecture.

While both approaches construct a product line architecture following the structure of a feature model, they lack a strict hierarchical and compositional construction.

An n-to-m mapping makes no assumption on feature models. However, relationships between features and artefacts have to be traced. In [265] the authors propose an approach to map features to UML fragments. Each fragment is composed by parts, and each part defines the precondition (i.e. the specification) a component must comply in order to be in that part. A feature can then be linked to n fragments and m fragments can be linked to a feature. In addition a feature can be directly linked to n components, which in turn may be linked to a part (as long it is compliant with it). Approaches based on n-to-m mapping do not scale as a large number of features and components lead to an explosion on the number of traces.

Therefore, a strong link with features combined with an algebraical composition mechanisms, allows FX-MAN to be an effective way to model product families; whereas, its capability of saving on resources utilisation by reusing components as well as entire families enables efficiency.

9.1 Achievements

This work has defined a new component-model called FX-MAN (chapter 5) that enables the systematic modelling of product families as architectures with explicit variation points.

We have developed a tool-set (chapter 6), and empirically evaluated our component-model on an industrial use case (chapter 7).

Our work represents an initial attempt at providing a methodology and tool support for product line engineering. In this regard, FX-MAN provides a good starting point, since we have evidence that X-MAN can be used to define and construct functional models (section 5.2), while FX-MAN can be used to construct and implement product family architectures.

9.2 Future Work

At this stage, FX-MAN provides an efficient and effective component-model to modelling product families. Moreover, it can provides the cornerstone for further work that can expand its scope of application, thus overcome its limitations. In the following sections we describes such applications, detailing at which level the current implementation of FX-MAN supports them.

9.2.1 Support for extractive SPLE

According to Krueger *et al.* [266], a company that wants to adopt a product line approach can follow three possible techniques: *proactive*, *reactive* and *extractive*. A *proactive*, or revolutionary, approach implies that a product line is modelled from scratch by carefully applying analysis and design methods. On the contrary, a *reactive* approach begins with a small, easy to handle product line, which is incrementally extended with new features and artefacts so as to expand its scope. Finally, an *extractive* approach starts with a portfolio of existing products and gradually refractor them to form a product line. At present, FX-MAN supports a proactive and (partially) a reactive approach, but has no support for an extractive one. Reactive approach is partially supported as the initial product either is constructed as an X-MAN architecture, or it can be transformed into its X-MAN equivalent. In order to support an extractive approach, we are currently working on a systematic way to model a family in FX-MAN by mining components and their variability from a portfolio of related products.

9.2.2 Support for load-time variability

Variability offers choices. Indeed, in deriving a product a client decides which features are included and which are not. It can be also said that a client binds a decision. Binding can happen at different stages; in particular in our discussion we differentiate between *compile-time*, and *load-time* variability. The former is resolved before a program starts (*i.e.* at compile-time), the latter while it is running (*i.e.* at load-time). As usual, each binding time has its own pros and cons. Since variability is resolved before a program starts, compile-time variability allows more room for optimization. That is, unused code can be removed from the final binaries, reducing in this way the run-time overhead in terms of memory consumption and execution time. As a consequence, once the program is generated and deployed is impossible to change. Load-time variability fills this gap, but at the price of memory and performance overhead, as all binaries are compiled and variability must be resolved and checked at run-time. At present, FX-MAN offers no support for load-time variability. Nevertheless, the FX-MAN meta-model can be extended to support run-time binding, a late form of load-time variability, which allows dynamic reconfiguration of components during system execution.

9.2.3 Support for fine-grained variability

Features are seldom implemented alone. Indeed, their value arise when they collaborate with each other to achieve the desired behaviour. A feature introduces then a change in behaviour, and depending on the implementation technique, this change can take place at different levels of granularity. Level of granularity here refers to the hierarchical structure of artefact implementation. Changes at the top of hierarchy are called *coarse-grained*, while those at the lower levels are defined *fine-grained*. Annotation-based approaches support more fine-grained changes than composition-based approaches. Undoubtedly, annotations can be applied almost everywhere across the common code-base, while composition-approaches rely on interfaces, typically defined at classes or methods level. Although FX-MAN only supports coarse-grained variability, its integration with pure::variants may allow to exploit fine-grained variability provided by annotation.

9.2.4 Support for versioning

In software engineering it is an undeniable fact that artefacts evolve over time. For instance, a technological evolution subsumes an evolution of many artefacts. The presence of different versions of an artefact valid at different times is denoted as *variability* *in time.* Complementarily, the presence of different shapes of an artefact at distinct time, with the restriction that they pertain to the same variation point, is denoted as *variability in space.* The composition mechanism of FX-MAN supports variability in space, but at the moment FX-MAN has a limited support for variability in time. Despite the fact that our tool uses CDO as a shared repository, which offers versioning support out of the box, our repository explorer only shows for each artefact the last deposited version. Up to today, one has to manually do that. A new release of our tool, which will be based on Eclipse Sirius[1] rather than on Graphiti, will deal with these technical aspects.

9.2.5 Support for non-functional features

In SPLE, a non-functional feature is a characteristic that can be used to judge the operation of a family member, not its behaviour. Currently FX-MAN does not offer any support to model non-functional features (*e.g.* reliability, performance and response time). Since in domains as cyber-physical-systems such features are as important as the functional ones, we are planning to investigate this aspect. Thanks to its rigorous algebraical type system and its strictly encapsulated components, FX-MAN should allow the integration with a methodology developed by the University of Salento [267], for managing Service Level Agreement (SLA) information hierarchically in service-based environments.

9.2.6 Support for non functional artefacts

Although a PFA and its constituent components (along with their implementation and functional specification) are the key artefact of a SPL, they are not the only ones. For instance, tests and documentation are two important artefacts that we do not model in our approach. However, being part of the pure::variants ecosystem allows FX-MAN to leverage its interoperability with other tools (*e.g.* IBM Rational DOORS and Microsoft Office). Indeed, pure::variants uses the variability mechanisms existing for the artefact type/tool and controls the derivation process (transformation) of variants. That is, pure::variants is a variability management tool that focuses on variability only, hence

[1]Unlike Graphiti which is still in incubation phase, Sirius in the last year has become more and more mature as it is supported by Thales and Obeo.

it is agnostic to the actual implementation of product families.

Bibliography

[1] H. Ford and S. Crowther. *My life and work.* Cosimo, Inc., 2005.

[2] M. Addis and M. B. Holbrook. “On the conceptual link between mass customisation and experiential consumption: an explosion of subjectivity”. *Wiley Journal of consumer behaviour* 1.1 (2001).

[3] F. S. Fogliatto, G. J. da Silveira, and D. Borenstein. “The mass customization decade: An updated review of the literature”. *Elsevier International Journal of Production Economics* 138.1 (2012).

[4] K. Pohl, G. Böckle, and F. van Der Linden. *Software product line engineering: foundations, principles, and techniques.* Springer, 2005.

[5] C. Krueger and P. Clements. “Systems and software product line engineering with BigLever software Gears”. In: *Proceedings of the 17th International Software Product Line Conference co-located workshops.* ACM. 2013.

[6] F. van der Linden, K. Schmid, and E. Rommes. “The product line engineering approach”. In: *Software Product Lines in Action.* Springer, 2007.

[7] J. Bosch. “From software product lines to software ecosystems”. In: *Proceedings of the 13th International Software Product Line Conference.* ACM. 2009.

[8] S. Apel et al. *Feature-oriented software product lines.* Springer, 2013.

[9] R. Love. *Linux Kernel Development.* Novell Press, 2005.

[10] R. Tartler et al. “Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem”. In: *Proceedings of the 6th Conference on Computer Systems.* ACM. 2011.

[11] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice.* SEI Series in Software Engineering. Addison-Wesley, 2012.

[12] J. Bosch et al. “Variability issues in software product lines”. In: *Proceedings of the 4th International Workshop on Software Product-Family Engineering.* Springer. 2002.

[13] T. Berger et al. “A survey of variability modeling in industrial practice”. In: *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems.* ACM. 2013.

[14] E. Alaña and A. Rodriguez. “Domain engineering methodologies survey”. *GMV Inovvating Solutions* (2007).

[15] L. Chen, M. Ali Babar, and N. Ali. “Variability management in software product lines: a systematic review”. In: *Proceedings of the 13th International Software Product Line Conference.* ACM. 2009.

[16] K. Kang et al. *Feature-oriented domain analysis (FODA) feasibility study.* Tech. rep. CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie-Mellon University, 1990.

[17] K. C. Kang, J. Lee, and P. Donohoe. “Feature-oriented product line engineering”. *IEEE Software* 19.4 (2002).

[18] M. Svahnberg and J. Bosch. “Issues concerning variability in software product lines”. In: *Proceedings of the International Workshop on Software Architectures for Product Families.* Springer. 2000.

[19] A. Alayed et al. “Towards component-based domain engineering”. In: *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications.* IEEE, 2013.

[20] I. Groher and R. Weinreich. “Integrating variability management and software architecture”. In: *Proceedings of the 10th European Joint Working IEEE/IFIP Conference on Software Architecture.* IEEE. 2012.

[21] G. Heineman and W. T. Councill. *Component-based software engineering: putting the pieces together.* Addison Wesley, 2001.

[22] T. Vale et al. “Twenty-eight years of component-based Software Engineering”. *Elsevier Journal of Systems and Software* 111 (2016).

[23] K.-K. Lau and Z. Wang. “Software Component Models”. *IEEE Transactions on Software Engineering* 33.10 (2007).

[24] K.-K. Lau et al. “Incremental Construction of Component-based Systems”. In: *Proceedings of the 15th International Symposium on Component-based Software Engineering.* ACM, 2012.

[25] S. Kumar and P. Phrommathed. *Research methodology.* Springer, 2005.

[26] M. D. McIlroy et al. “Mass-produced software components”. In: *Proceedings of the 1st International Conference on Software Engineering.* Springer-Verlag, 1968.

[27] P. Clements and L. Northrop. *Software product lines: practices and patterns.* Addison-Wesley Reading, 2002.

[28] F. J. van der Linden, K. Schmid, and E. Rommes. *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media, 2007.

[29] P. Donohoe. *Software product lines: experience and research directions.* Springer Science & Business Media, 2012.

[30] L. M. Northrop. “SEI’s software product line tenets.” *IEEE Software* 19.4 (2002).

[31] A. Metzger and K. Pohl. “Software product line engineering and variability management: achievements and challenges”. In: *Proceedings of the Symposium on Future of Software Engineering.* ACM. 2014.

[32] P. Toft, D. Coleman, and J. Ohta. “A cooperative model for cross-divisional product development for a software product line”. In: *Software Product Lines.* Springer, 2000.

[33] M. Coriat, J. Jourdan, and F. Boisbourdin. “The SPLIT method”. In: *Software Product Lines.* Springer, 2000.

[34] P. America et al. “COPAM: a component-oriented platform architecting method family for product family engineering”. In: *Software Product Lines.* Springer, 2000.

[35] D. C. Sharp. “Component based product line development of avionics software.” *Microprocessors and Microsystems - Embedded Hardware Design* 23.7 (1999).

[36] S. Thiel and F. Peruzzi. “Starting a product line approach for an envisioned market”. In: *Software Product Lines.* Springer, 2000.

[37] J. Bayer, D. Muthig, and B. Göpfert. “The library system product line - a Kobra case study”. *Fraunhofer Institute for Experimental Software Engineering, Tech. Rep. IESE-Report No* 24 (2001).

[38] K. Schmid, R. Rabiser, and P. Grünbacher. “A comparison of decision modeling approaches in product lines”. In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. ACM. 2011.

[39] Ø. Haugen. “CVL: common variability language or chaos, vanity and limitations?” In: *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems*. ACM. 2013.

[40] K. Czarnecki et al. “Cool features and tough decisions: a comparison of variability modeling approaches”. In: *Proceedings of the 6th International workshop on variability modeling of software-intensive systems*. ACM. 2012.

[41] T. Berger et al. “What is a feature?: a qualitative study of features in industrial software product lines”. In: *Proceedings of the 19th International Conference on Software Product Line*. ACM. 2015.

[42] L. Geyer and M. Becker. “On the influence of variabilities on the application-engineering process of a product family”. In: *Software Product Lines*. Springer, 2002.

[43] K. Czarnecki and A. Wasowski. “Feature diagrams and logics: there and back again”. In: *Proceedings of the 11th International Software Product Line Conference*. IEEE. 2007.

[44] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. “Feature diagrams: a survey and a formal semantics”. In: *Proceedings of the 14th International Requirements Engineering Conference*. IEEE. 2006.

[45] E. Y. Nakagawa, P. O. Antonino, and M. Becker. “Reference architecture and product line architecture: a subtle but critical difference”. In: *Proceedings of the 9th European Conference on Software Architecture*. 2011.

[46] S. Angelov, P. Grefen, and D. Greefhorst. “A classification of software reference architectures: analyzing their success and effectiveness”. In: *Proceedings of the 3rd European Joint Working IEEE/IFIP Conference on Software Architecture*. IEEE. 2009.

[47] M. Galster. “Software reference architectures: related architectural concepts and challenges”. In: *Proceedings of the 1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures*. ACM. 2015.

[48] H. Gomaa. *Designing software product lines with UML*. IEEE, 2005.

[49] P. A. da Mota Silveira Neto et al. “Testing software product lines”. *IEEE Transactions on Software Engineering* 28.5 (2011).

[50] E. Engström and P. Runeson. “Software product line testing – a systematic mapping study”. *Information and Software Technology* 53.1 (2011).

[51] K. Pohl and A. Metzger. “Software product line testing”. *Communications of the ACM* 49.12 (2006).

[52] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach.* Pearson Education, 2000.

[53] M. Dalgarno, D. Beuche, et al. “Variant management”. In: *Proceedings of the 3rd British Computer Society Configuration Management Specialist Group Conference.* Vol. 1. 2007.

[54] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice.* Wiley Publishing, 2009.

[55] K. C. Kang et al. “FORM: a feature-oriented reuse method with domain-specific reference architectures”. *Annals of Software Engineering* 5.1 (1998).

[56] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co., 2000.

[57] K. Chen et al. “An approach to constructing feature models based on requirements clustering”. In: *Proceedings of the 13th IEEE International Conference on Requirements Engineering.* IEEE. 2005.

[58] D. Batory, J. Sarvela, and A. Rauschmayer. “Scaling step-wise refinement”. *IEEE Transactions on Software Engineering* 30.6 (2004).

[59] A. Classen, P. Heymans, and P.-Y. Schobbens. “What’s in a feature: a requirements engineering perspective”. In: *Fundamental Approaches to Software Engineering.* Springer, 2008.

[60] P. Zave. “An experiment in feature engineering”. In: *Programming methodology.* Springer, 2003.

[61] D. Batory. *Feature models, grammars, and propositional formulas.* Springer, 2005.

[62] S. Apel et al. “An algebra for features and feature composition”. In: *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology.* Springer. 2008.

[63] E. Yourdon. *Modern structured analysis.* Prentice Hall PTR, 2000.

[64] J. E. Cooling. *Software engineering for real-time systems.* Pearson Education, 2003.

[65] D. Hatley and I. Pirbhai. *Strategies for real-time system specification.* Addison-Wesley, 2013.

[66] D. Harel and M. Politi. *Modeling reactive systems with statecharts: the STATEMATE approach.* McGraw-Hill, Inc., 1998.

[67] D. Harel et al. "Statemate: a working environment for the development of complex reactive systems". *IEEE Transactions on Software Engineering* 16.4 (1990).

[68] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified modeling language reference manual.* Pearson Higher Education, 2004.

[69] A. K. Tyagi. *MATLAB and SIMULINK for engineers.* Oxford University Press, 2012.

[70] A. Angermann et al. *MATLAB-Simulink-Stateflow.* 2008.

[71] D. Beuche. "Modeling and building software product lines with pure::variants". In: *Proceedings of the 16th International Software Product Line Conference-Volume 2.* ACM. 2012.

[72] K. Czarnecki, S. Helsen, and U. Eisenecker. "Staged configuration using feature models". In: *Proceedings of the 3rd International Conference on Software Product Lines.* Springer. 2004.

[73] A. Polzer et al. "Managing complexity and variability of a model-based embedded software product line". *Innovations in Systems and Software Engineering* 8.1 (2012).

[74] H. Gronninger et al. "Modeling variants of automotive systems using views". *arXiv* (2014).

[75] J. O. Coplien. "Multiparadigm design and implementation in C++". In: *Proceedings of the Technology of Object-Oriented Languages and Systems.* IEEE. 1999.

[76] J. Liebig et al. "An analysis of the variability in forty preprocessor-based software product lines". In: *Proceedings of the 32nd International Conference on Software Engineering.* Vol. 1. IEEE. 2010.

[77] S. Apel et al. "Classic, tool-driven variability mechanisms". In: *Feature-Oriented Software Product Lines.* Springer, 2013.

[78] D. Muthig and C. Atkinson. "Model-driven product line architectures". In: *Proceedings of the 2nd International Conference on Software Product Lines.* Springer-Verlag, 2002.

[79] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. “Towards a UML profile for software product lines”. In: *Software Product-Family Engineering*. Springer, 2004.

[80] S. Salicki and N. Farcet. “Expression and usage of the variability in the software product lines”. In: *Software Product-Family Engineering*. Springer, 2002.

[81] H. Spencer and G. Collyer. “# ifdef considered harmful, or portability experience with C News”. *USENIX* (1992).

[82] D. Lohmann et al. “A quantitative analysis of aspects in the eCos kernel”. In: *Proceedings of the 1st European Conference on Computer Systems*. ACM. 2006.

[83] D. Le, E. Walkingshaw, and M. Erwig. “# ifdef confirmed harmful: promoting understandable software variation”. In: *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE. 2011.

[84] C. Kästner and S. Apel. “Virtual separation of concerns-a second chance for preprocessors”. *Journal of Object Technology* 8.6 (2009).

[85] C. Kastner et al. “Type checking annotation-based product lines”. *ACM Transactions on Software Engineering and Methodology* 21.3 (2012).

[86] J. Meinicke et al. “FeatureIDE: taming the preprocessor wilderness”. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM. 2016.

[87] J. Bosch, S. Deelstra, and M. Sinnema. “COVAMOF”. In: *Systems and Software Variability Management*. Springer, 2013.

[88] L. Moonen. “Towards evidence-based recommendations to guide the evolution of component-based product families”. *Elsevier Science of Computer Programming* 97 (2015).

[89] S. Clarke and E. Baniassad. *Aspect-oriented analysis and design*. Addison-Wesley Professional, 2005.

[90] G. Kiczales et al. “Aspect-oriented programming”. In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. Springer Berlin Heidelberg. 1997.

[91] M. Voelter and I. Groher. “Product line implementation using aspect-oriented and model-driven software development”. In: *Proceedings of the 11th International Software Product Line Conference*. IEEE. 2007.

[92] J. Klein, L. Hélouët, and J.-M. Jézéquel. "Semantic-based weaving of scenarios". In: *Proceedings of the 5th International Conference on Aspect-oriented software development.* ACM. 2006.

[93] P. Jayaraman et al. "Model composition in product lines and feature interaction detection using critical pair analysis". In: *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems.* Springer, 2007.

[94] P. Lahire et al. "Introducing variability into aspect-oriented modeling approaches". In: *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems.* Springer, 2007.

[95] Y. R. Reddy et al. "Directives for composing aspect-oriented design class models". In: *Transactions on Aspect-Oriented Software Development.* Springer, 2006.

[96] T. J. Brown et al. "Weaving behavior into feature models for embedded system families". In: *Proceedings of the 10th International Software Product Line Conference.* IEEE. 2006.

[97] G. Perrouin et al. "Weaving variability into domain metamodels". *Software & Systems Modeling* 11.3 (2012).

[98] I. Groher and M. Voelter. "XWeave: models and aspects in concert". In: *Proceedings of the 10th International Workshop on Aspect-oriented Modeling.* ACM. 2007.

[99] S. Apel, T. Leich, and G. Saake. "Aspectual feature modules". *IEEE Transactions on Software Engineering* 34.2 (2008).

[100] L. Fuentes and P. Sánchez. "Designing and weaving aspect-oriented executable UML models." *Journal of Object Technology* 6.7 (2007).

[101] A. Colyer, A. Rashid, and G. Blair. *On the separation of concerns in program families.* Tech. rep. Lancaster University, 2004.

[102] C. Koppen and M. Störzer. "PCDiff: attacking the fragile pointcut problem". In: *Proceedings of the 3rd European Interactive Workshop on Aspects in Software.* Vol. 7. IEEE. 2004.

[103] G. Kniesel. "Detection and resolution of weaving interactions". In: *Transactions on Aspect-Oriented Software Development.* Springer, 2009.

[104] W. Havinga, I. Nagy, and L. Bergmans. "An analysis of aspect composition problems" (2006).

[105] M. Störzer, F. Forster, and R. Sterr. "Detecting precedence-related advice interference". In: *ASE*. Vol. 6. 2006.

[106] K. Tian, K. Cooper, and K. Zhang. "A framework based approach for unified detection of Aspect Weaving Problems". In: *Proceedings of the 11th International Conference on Information Reuse and Integration*. IEEE. 2010.

[107] S. Apel et al. "Model superimposition in software product lines". In: *Theory and Practice of Model Transformations*. Springer, 2009.

[108] S. Apel and C. Lengauer. "Superimposition: a language-independent approach to software composition". In: *Proceedings of the 7th International Conference on Software Composition*. Springer. 2008.

[109] K. Czarnecki and M. Antkiewicz. "Mapping features to models: a template approach based on superimposed variants". In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Springer. 2005.

[110] J. Bosch. "Superimposition: a component adaptation technique". *Elsevier Information and Software Technology* 41.5 (1999).

[111] S. Apel et al. "FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming". In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Springer. 2005.

[112] M. Mezini and K. Ostermann. "Variability management with feature-oriented programming and aspects". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 29. 6. ACM. 2004.

[113] S. Apel, T. Leich, and G. Saake. "Aspectual mixin layers: aspects and features in concert". In: *Proceedings of the 28th International Conference on Software Engineering*. ACM. 2006.

[114] G. Bracha and W. Cook. "Mixin-based inheritance". *ACM Sigplan Notices* 25.10 (1990).

[115] D. Batory et al. "The GenVoca model of software-system generators". *IEEE Software* 11.5 (1994).

[116] D. Batory et al. "Design wizards and visual programming environments for GenVoca generators". *IEEE Transactions on Software Engineering* 26.5 (2000).

[117] S. Apel, C. Kastner, and C. Lengauer. “FEATUREHOUSE: language-independent, automated software composition”. In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE. 2009.

[118] S. Apel, C. Kästner, and C. Lengauer. “Language-independent and automated software composition: the FeatureHouse experience”. *IEEE Transactions on Software Engineering* 39.1 (2013).

[119] I. Schaefer, A. Worret, and A. Poetzsch-Heffter. “A model-based framework for automated product derivation”. In: *Proceedings of the 1st International Workshop on Model-Driven Approaches in Software Product Line Engineering*. ACM. 2009.

[120] I. Schaefer et al. “Delta-oriented programming of software product lines”. In: *Software Product Lines: Going Beyond*. Springer, 2010.

[121] H. Sabouri and R. Khosravi. “Delta modeling and model checking of product families”. In: *Fundamentals of software engineering*. Springer, 2013.

[122] I. Schaefer and F. Damiani. “Pure delta-oriented programming”. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. ACM. 2010.

[123] M. Helvensteijn, R. Muschevici, and P. Y. Wong. “Delta modeling in practice: a Fredhopper case study”. In: *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*. ACM. 2012.

[124] P. C. Clements. “A survey of architecture description languages”. In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE. 1996.

[125] N. Medvidovic and R. N. Taylor. “A classification and comparison framework for software architecture description languages”. *IEEE Transactions on Software Engineering* 26.1 (2000).

[126] D. Garlan, R. Monroe, and D. Wile. “Acme: an architecture description interchange language”. In: *CASCON First Decade High Impact Papers*. IBM Corp. 2010.

[127] R. Allen and D. Garlan. “Formalizing architectural connection”. In: *Proceedings of the 16th International Conference on Software Engineering*. IEEE. 1994.

[128] D. Garlan, R. Allen, and J. Ockerbloom. “Exploiting style in architectural design environments”. *ACM SIGSOFT Software Engineering Notes* 19.5 (1994).

[129] N. Medvidovic et al. “Using object-oriented typing to support architectural design in the C2 style”. *ACM SIGSOFT Software Engineering Notes* 21.6 (1996).

[130] D. C. Luckham et al. "Specification and analysis of system architecture using Rapide". *IEEE Transactions on Software Engineering* 21.4 (1995).

[131] M. Moriconi, X. Qian, and R. A. Riemenschneider. "Correct architecture refinement". *IEEE Transactions on Software Engineering* 21.4 (1995).

[132] R. Roshandel et al. "Mae-a system model and environment for managing architectural evolution". *ACM Transactions on Software Engineering and Methodology* 13.2 (2004).

[133] E. Dashofy, A. van der Hoek, and R. Taylor. "A highly-extensible, XML-based architecture description language". In: *Proceedings of the European Joint Working IEEE/IFIP Conference on Software Architecture.* IEEE, 2001.

[134] D. Gurov, B. M. Østvold, and I. Schaefer. "A hierarchical variability model for software product lines". In: *Leveraging Applications of Formal Methods, Verification, and Validation.* Springer, 2012.

[135] C.-M. Park et al. "A component model supporting decomposition and composition of consumer electronics software product lines". In: *Proceedings of the 11th International Software Product Line Conference.* IEEE. 2007.

[136] R. van Ommering et al. "The Koala component model for consumer electronics software". *IEEE Transactions on Software Engineering* 33.3 (2000).

[137] F. van der Linden, K. Schmid, and E. Rommes. "Philips consumer electronics software for televisions". In: *Software Product Lines in Action.* Springer, 2007.

[138] T. Asikainen, T. Soininen, and T. Männistö. "A Koala-based approach for modelling and deploying configurable software product families". In: *Software Product-Family Engineering.* Springer, 2003.

[139] A. Haber, J. O. Ringert, and B. Rumpe. "Montiarc-architectural modeling of interactive distributed and cyber-physical systems". *arXiv* (2014).

[140] H. Krahn, B. Rumpe, and S. Völkel. "MontiCore: a framework for compositional development of domain specific languages". *Springer International Journal on Software Tools for Technology Transfer* 12.5 (2010).

[141] D. Sabin and R. Weigel. "Product configuration frameworks-a survey". *IEEE Intelligent Systems* 13.4 (1998).

[142] A. Haber et al. "Delta-oriented architectural variability using MontiCore". In: *Proceedings of the 5th European Conference on Software Architecture: Companion Volume.* ACM. 2011.

[143] T. Asikainen, T. Männistö, and T. Soininen. “Kumbang: a domain ontology for modelling variability in software product families”. *Elsevier Advanced Engineering Informatics* 21.1 (2007).

[144] E. Silva et al. “A lightweight language for software product lines architecture description”. In: *Software Architecture*. Springer, 2013.

[145] M. O. Dias et al. “Leveraging aspect-connectors to improve stability of product-line variabilities”. In: *Proceedings of the 4th International Conference on Performance Engineering*. Springer. 2010.

[146] J. Pérez et al. “Plastic partial components: a solution to support variability in architectural components”. In: *Proceedings of the 3rd Joint Working IEEE/IFIP European Conference on Software Architecture*. IEEE. 2009.

[147] R. Bashroush et al. “Adlars: an architecture description language for software product lines”. In: *Proceedings of the 29th annual IEEE/NASA Software Engineering workshop*. IEEE. 2005.

[148] T. Asikainen, T. Mannisto, and T. Soininen. “A unified conceptual foundation for feature modelling”. In: *Proceedings of the 10th International Software Product Line Conference*. IEEE. 2006.

[149] A. L. Medeiros et al. “ArchSPL-MDD: An ADL-based Model-Driven Strategy for Automatic Variability Management”. In: *Components, Architectures and Reuse Software (SBCARS), 2015 IX Brazilian Symposium on*. IEEE. 2015.

[150] L. A. Gayard, C. M. F. Rubira, and P. A. de Castro Guerra. “Cosmos*: a component system model for software architectures”. *Institute of Computing, University of Campinas, Tech. Rep. IC-08-04* (2008).

[151] U. Aßmann. *Invasive software composition*. Springer, 2003.

[152] S. Di Cola et al. “A component model for defining software product families with explicit variation points”. In: *Proceedings of the 19th International symposium on component-based Software Engineering*. ACM, 2016.

[153] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. 2nd. Addison-Wesley, 2002.

[154] B. Meyer. “The grand challenge of trusted components”. In: *Proceedings of the 25th International Conference on Software Engineering*. IEEE. 2003.

[155] G. Coulson et al. “A generic component model for building systems software”. *ACM Transactions on Computer Systems* 26.1 (2008).

[156] B. Ding et al. “Component model supporting trustworthiness-oriented software evolution”. *Journal of Software* 22.1 (2011).

[157] R. K. Keller and R. Schauer. “Design components: toward software composition at the design level”. In: *Proceedings of the 20th International Conference on Software Engineering*. IEEE. 1998.

[158] I. Crnkovic et al. “A classification framework for software component models”. *IEEE Transactions on Software Engineering* 37.5 (2011).

[159] C. Lüer and A. Hoek. *Composition environments for deployable software components – Tech. Rep. UCI-ICS-02-18*. University of California, Irvine, 2002.

[160] F. Bachmann et al. *Technical concepts of component-based software engineering*. Tech. Rep. CMU/SEI-2000-TR-008. Carnegie Melon Software Engineering Institute, 2000.

[161] R. Englander. *Developing Java Beans*. O’Reilly & Associates, 1997.

[162] B. Burke and R. Monson-Haefel. *Enterprise JavaBeans 3.0*. " O’Reilly Media, Inc.", 2006.

[163] S. Weerawarana et al. “Bean markup language: a composition language for JavaBeans components”. In: *Proceedings of the 6th Conference on Object-Oriented Technologies and Systems*. USENIX Association, 2001.

[164] E. Roman, R. P. Sriganesh, and G. Brose. *Mastering enterprise JavaBeans*. John Wiley & Sons, 2005.

[165] V. Matena, B. Stearns, and L. Demichiel. *Applying enterprise JavaBeans: component-based development for the J2EE platform*. Pearson Education, 2003.

[166] S. Goebel and M. Nestler. “Composite component support for EJB”. In: *Proceedings of the Winter International Symposium on Information and Communication Technologies*. Trinity College Dublin. 2004.

[167] R. Haefel. *Enterprise Java Beans*. 4th. O’Reilly, 2004.

[168] D. Box. *Essential COM*. Addison-Wesley, 1998.

[169] A. Major. *COM IDL and Interface Design*. John Wiley & Sons, 1999.

[170] T. Pattison. *Programming distributed applications with COM+ and Microsoft Visual Basic 6.0*. Microsoft Press, 2000.

[171] J. Siegel. *CORBA 3 fundamentals and programming.* Vol. 2. John Wiley & Sons New York, 2000.

[172] N. Wang, D. C. Schmidt, and C. O'Ryan. "Overview of the CORBA component model". In: *Proceedings of the 4th International Conference on Component-Based Software Engineering.* Addison-Wesley Longman Publishing Co., Inc. 2001.

[173] A. Wigley et al. *Microsoft .NET compact framework (Core Reference).* Microsoft Press, 2003.

[174] D. S. Platt. *Introducing Microsoft .NET.* 3rd. Microsoft Press, 2003.

[175] G. Alonso et al. *Web Services: concepts, architectures and applications.* Springer-Verlag, 2004.

[176] D. K. Barry. *Web services, service-oriented architectures, and cloud computing.* Morgan Kaufmann, 2012.

[177] C. Peltz. "Web services orchestration and choreography". *IEEE* 36.10 (2003).

[178] O. Alliance. "Osgi service platform, core specification, release 4, version 4.1". *OSGi Specification* (2007).

[179] T. Gu, H. K. Pung, and D. Q. Zhang. "Toward an OSGi-based infrastructure for context-aware applications". *IEEE Pervasive Computing* 3.4 (2004).

[180] R. Hall et al. *OSGi in action: creating modular applications in Java.* Manning Publications Co., 2011.

[181] D. H. Lorenz and P. Petkovic. "Design-time assembly of runtime containment components". In: *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems.* IEEE. 2000.

[182] R. G. Cattell and J. Inscore. *J2EE in practice: building business applications with the Java 2 Platform Enterprise.* Addison-Wesley Longman Publishing Co., Inc., 2001.

[183] R. Marvie and P. Merle. "CORBA component model: discussion and use with OpenCCM" (2001).

[184] K. P. Birman. "Corba: The common object request broker architecture". In: *Guide to Reliable Distributed Systems.* Springer, 2012.

[185] A. L. Lemos, F. Daniel, and B. Benatallah. "Web service composition: a survey of techniques and tools". *ACM Computing Surveys (CSUR)* 48.3 (2016).

[186] S. Weerawarana et al. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more.* Prentice Hall PTR, 2005.

[187] L. Richardson and S. Ruby. *RESTful web services.* O'Reilly Media, Inc., 2008.

[188] T. Bray. "The javascript object notation (JSon) data interchange format". *IETF* (2014).

[189] D. Garlan, R. Monroe, and D. Wile. "Acme: architectural description of component-based systems". In: *Foundations of Component-Based Systems.* Cambridge University Press, 2000.

[190] J. Ivers et al. *Documenting component and connector views with UML 2.0.* Tech. rep. CMU/SEI-2004-TR-008. The Software Engineering Institute, 2004.

[191] OMG. *UML 2.0 superstructure specification.* Tech. rep. ptc/04-10-02. Object Management Group, 2004.

[192] S. Roh, K. Kim, and T. Jeon. "Architecture modeling language based on UML 2.0". In: *Proceedings od the 11th International Conference on Software Engineering.* IEEE. 2004.

[193] F. Plášil, D. Bálek, and R. Janeček. "SOFA/DCUP: architecture for component trading and dynamic updating". In: *Proceedings of the 4th International Conference on Configurable Distributed Systems.* IEEE, 1998.

[194] T. Bures, P. Hnetynka, and F. Plasil. "Sofa 2.0: Balancing Advanced Features in a Hierarchical Component Model". In: *Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications.* IEEE. 2006.

[195] C. Atkinson et al. *Component-based product line engineering with UML.* Addison-Wesley, 2001.

[196] C. Atkinson et al. "Modeling components and component-based systems in KobrA". In: *The Common Component Modeling Example.* Springer, 2008.

[197] S. Becker, H. Koziolek, and R. Reussner. "The Palladio component model for model-driven performance prediction". *Elsevier Journal of Systems and Software* 82.1 (2009).

[198] F. Brosch et al. "Architecture-based reliability prediction with the palladio component model". *IEEE Transactions on Software Engineering* 38.6 (2012).

[199] T. Bures et al. *ProCom – the progress component model.* Malardalen University. Vasteras, Sweden, 2010.

[200] S. Sentilles et al. “A component model for control-intensive distributed embedded systems”. In: *Proceedings of the 11th International Symposium on Component-Based Software Engineering*. Springer. 2008.

[201] E. Bruneton et al. “An open component model and its support in Java”. In: *Proceedings of the 7th International Symposium on Component-based Software Engineering*. Springer, 2004.

[202] E. Bruneton et al. “The fractal component model and its support in Java”. *Wiley & Sons, Software: Practice and Experience* 36.11-12 (2006).

[203] H. Gomaa and G. Farrukh. “Composition of software architectures from reusable architecture patterns”. In: *Proceedings of the 3rd International Workshop on Software Architecture*. ACM. 1998.

[204] I. Georgiadis, J. Magee, and J. Kramer. “Self-organising software architectures for distributed systems”. In: *Proceedings of the 1st Workshop on Self-healing Systems*. ACM. 2002.

[205] M. Shaw et al. “Abstractions for software architecture and tools to support them”. *IEEE Transactions on Software Engineering* 21.4 (1995).

[206] J. Marino and M. Rowley. *Understanding SCA (service component architecture)*. Pearson Education, 2009.

[207] M. Abi-Antoun et al. “Modeling and implementing software architecture with Acme and ArchJava”. In: *Proceedings of the 27th International Conference on Software Engineering*. ACM. 2005.

[208] N. He et al. “Component-based design and verification in X-MAN”. In: *Proceedings of the 4th European Congress on Embedded Real Time Software and Systems*. ACM. 2012.

[209] K.-K. Lau and C. Tran. “X-MAN: an MDE tool for component-based system development”. In: *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2012.

[210] M. Broy et al. “What characterizes a (software) component?” *Springer, Software-Concepts & Tools* 19.1 (1998).

[211] K.-K. Lau and Z. Wang. “A Taxonomy of software component models”. In: *Proceedings of the 31st Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2005.

[212] J. Aldrich, C. Chambers, and D. Notkin. “Architectural reasoning in ArchJava”. In: *Proceedings of the 16th European Conference on Object-Oriented Programming.* Springer-Verlag, 2002.

[213] S. Di Cola et al. “An MDE tool for defining software product families with explicit variation points”. In: *Proceedings of the 19th International Conference on Software Product Line.* ACM. 2015.

[214] S. D. Cola, C. Tran, and K.-K. Lau. “A Graphical tool for model-driven development using components and services”. In: *Proceedings of 41st Euromicro Conference on Software Engineering and Advanced Applications.* 2015.

[215] M. Torchiano et al. “Relevance, benefits, and problems of software modelling and model driven techniques - a survey in the Italian industry”. *Journal of Systems and Software* 86.8 (2013).

[216] R. F. Paige, N. Matragkas, and L. M. Rose. “Evolving models in model-driven engineering: State-of-the-art and future challenges”. *Elsevier, Journal of Systems and Software* 111 (2016).

[217] J. Whittle, J. Hutchinson, and M. Rouncefield. “The state of practice in model-driven engineering”. *IEEE, Software* 31.3 (2014).

[218] A. W. Brown. “Model driven architecture: principles and practice”. *Springer Software and Systems Modeling* 3.4 (2004).

[219] J. Sorva, V. Karavirta, and L. Malmi. “A review of generic program visualization systems for introductory programming education”. *ACM Transactions on Computing Education* 13.4 (2013).

[220] B. A. Price, R. M. Baecker, and I. S. Small. “A principled taxonomy of software visualization”. *Elsevier Journal of Visual Languages & Computing* 4.3 (1993).

[221] A. Henriksson and H. Larsson. “A definition of round-trip engineering”. *University of Linköping, Sweden* (2003).

[222] C. Atkinson and T. Kuhne. “Model-driven development: a metamodeling foundation”. *IEEE Software* 20.5 (2003).

[223] D. C. Schmidt. “Model-driven engineering”. *IEEE Transactions on Software Engineering* 39.2 (2006).

[224] B. Selic. “The pragmatics of model-driven development”. *IEEE Software* 20.5 (2003).

[225] J. Hutchinson, M. Rouncefield, and J. Whittle. “Model-driven engineering practices in industry”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011.

[226] J. Hutchinson et al. “Empirical assessment of MDE in industry”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011.

[227] B. Schatz. “10 years model-driven–What did we achieve?” In: *Proceedings of the 2nd European Regional Conference on the Engineering of Computer Based Systems*. IEEE. 2011.

[228] J. Hutchinson, J. Whittle, and M. Rouncefield. “Model-driven engineering practices in industry: social, organizational and managerial factors that lead to success or failure”. *Elsevier Science of Computer Programming* 89 (2014).

[229] D. S. Frankel. *Model driven architecture applying MDA*. John Wiley & Sons, 2003.

[230] L. Zhu. “Model-driven architecture”. In: *Essential software architecture*. Springer, 2011.

[231] A. Dennis, B. H. Wixom, and D. Tegarden. *Systems analysis and design: an object-oriented approach with UML*. John Wiley & Sons, 2015.

[232] W. Tang. “Meta object facility”. In: *Encyclopedia of database systems*. Springer, 2009.

[233] M. Weiss. “Xml metadata interchange”. In: *Encyclopedia of Database Systems*. Springer, 2009.

[234] J. Davis. “GME: the generic modeling environment”. In: *Companion of the 18th Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM. 2003.

[235] D. Steinberg et al. *EMF: Eclipse modeling framework*. 2nd ed. Pearson Education, 2009.

[236] The Eclipse Foundation. *Xcore*. `https://wiki.eclipse.org/Xcore`. Accessed: 26-04-2015.

[237] The Eclipse Foundation. *Graphiti*. `https://www.eclipse.org/graphiti/`. Accessed: 12-03-2015.

[238] J. R. Jos W. Karsten T. *Spray - A quick way of creating Graphiti*. `http://goo.gl/qULZny`. Accessed: 08-05-2015.

[239] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd, 2013.

[240] E. Stepper. *CDO model repository.* `https://eclipse.org/cdo/`. Accessed: 16-06-2015. 2016.

[241] D. S. Kolovos et al. "Taming EMF and GMF using model transformation". In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems.* Springer. 2010.

[242] K.-K. Lau, F. Taweel, and C. Tran. "The W Model for component-based software development". In: *Proceedings of the 37th Euromicro Conference on Software Engineering and Advanced Applications.* IEEE, 2011.

[243] K.-K. Lau, L. Ling, and Z. Wang. "Composing components in design phase using exogenous connectors". In: *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications.* IEEE, 2006.

[244] D. Beuche and M. Dalgarno. "Software product line engineering with feature models". *ACCU Overload Journal* 78 (2007).

[245] F. van der Linden and H. Obbink. "ESAPS–Engineering Software Architectures, Processes and Platforms for System Families". In: *Software Architectures for Product Families.* Springer, 2000.

[246] M. Jazayeri, A. Ran, and F. van Der Linden. *Software architecture for product families: principles and practice.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[247] F. van der Linden. "Software product families in Europe: the Esaps & Cafe projects". *IEEE Software* 19.4 (2002).

[248] H. Obbink et al. "COPA: a component-oriented platform architecting method for families of software-intensive electronic products". In: *Proceedings of the 1st Conference on Software Product Line Engineering.* 2000.

[249] J. G. Wijnstra. "Critical factors for a successful platform-based product family approach". In: *Software Product Lines.* Springer, 2002.

[250] D. Oliveira and N. Rosa. "Evaluating product line architecture for grid computing middleware systems: Ubá experience". In: *Proceedings of the 24th International Conference on Advanced Information Networking and Applications Workshops.* IEEE. 2010.

[251] C. P. Team. “CMMI® for development, version 1.3, improving processes for developing better products and services”. *Software Engineering Institute* (2010).

[252] L. Northrop, L. Jones, and P. Donohoe. *Examining product line readiness: experiences with the SEI product line technical probe.* Carnegie Mellon University. 2005.

[253] F. van Der Linden et al. “Software product family evaluation”. In: *Software Product Lines.* Springer, 2004.

[254] L. Northrop et al. “A framework for software product line practice, version 5.0”. *SEI Interactive* (2007).

[255] R. Flores, C. Krueger, and P. Clements. “Mega-scale product line engineering at General Motors”. In: *Proceedings of the 16th International Software Product Line Conference-Volume 1.* ACM. 2012.

[256] J. Lanman et al. “Employing the second generation software product-line for live training transformation”. In: *Proceedings of the 16th Interservice/Industry Training, Simulation, and Education Conference.* NTSA. 2011.

[257] C. W. Krueger. “New methods in software product line practice”. *ACM Communications* 49.12 (2006).

[258] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor. “A comprehensive approach for the development of modular software architecture description languages”. *ACM Transactions on Software Engineering and Methodology* 14.2 (2005).

[259] P. Sochos, M. Riebisch, and I. Philippow. “The feature-architecture mapping (FArM) method for feature-oriented development of software product lines”. In: *Proceedings of the 13th International Symposium and Workshop on Engineering of Computer Based Systems.* IEEE. 2006.

[260] A. Haber et al. “Hierarchical variability modeling for software architectures”. In: *Proceedings of the 15th International Conference on Software Product Line.* IEEE. 2011.

[261] B. Behringer and S. Rothkugel. “Integrating feature-based implementation approaches using a common graph-based representation”. In: *Proceedings of the 31st Annual Symposium on Applied Computing.* ACM. 2016.

[262] R. E. Lopez-Herrejon, D. Batory, and W. Cook. “Evaluating support for features in advanced modularization technologies”. In: *Proceedings of the 20th European Conference on Object-Oriented Programming.* Springer. 2005.

[263] K. Berg, J. Bishop, and D. Muthig. “Tracing software product line variability: from problem to solution space”. In: *Proceedings of the 2nd Annual Research Conference on IT Research in Developing Countries*. South African Institute for Computer Scientists and Information Technologists. 2005.

[264] P. Trinidad et al. “Mapping feature models onto component models to build dynamic software product lines.” In: *Proceedings of the International Workshop on Dynamic Software Product Lines*. IEEE. 2007.

[265] M. Korner, S. Herold, and A. Rausch. “Composition of applications based on software product lines using architecture fragments and component sets”. In: *Companion Volume of the 11th Working IEEE/IFIP Conference on Software Architecture*. ACM, 2014.

[266] C. Krueger. “Easing the transition to software mass customization”. In: *Software Product-Family Engineering*. Springer, 2002.

[267] A. Longo, M. Zappatore, and M. A. Bochicchio. “Service and contract composition: a model and a tool”. In: *Proceedings of the 14th International Symposium on Integrated Network Management*. IEEE. 2015.

[268] T. Berger et al. “Variability mechanisms in software ecosystems”. *Elsevier Information and Software Technology* 56.11 (2014).

[269] F. P. Miller, A. F. Vandome, and J. McBrewster. *Apache Maven*. Alpha Press, 2010.

[270] R. Mecklenburg. *Managing projects with GNU make*. "O’Reilly Media, Inc.", 2004.

[271] B. Muschko. *Gradle in action*. Manning, 2014.

[272] Y. Brun et al. “Early detection of collaboration conflicts and risks”. *IEEE Transactions on Software Engineering* 39.10 (2013).

[273] E. Gamma et al. “Design patterns: abstraction and reuse of object-oriented design”. In: *Software pioneers*. Springer, 2002.

[274] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Vol. 1. Prentice Hall Englewood Cliffs, 1996.

[275] M. Fowler. “Inversion of control containers and the dependency injection pattern” (2004).

[276] E. Ernst. “Separation of concerns”. In: *Proceedings of the 5th Workshop on Software-Engineering Properties of Languages for Aspect Technologies*. ACM. 2003.

[277] K. Sullivan et al. “Information Hiding Interfaces for Aspect-oriented Design”. In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th International Symposium on Foundations of Software Engineering.* ACM, 2005.

[278] S. Apel and C. Kästner. “An overview of feature-oriented software development”. *Journal of Object Technology* 8.5 (2009).

[279] E. Gamma et al. *Design patterns – elements of reusable object-oriented design.* Addison-Wesley, 1995.

Appendix A

Variability Mechanisms

Building for mass customization implies the idea of managed variability. As software can be easily adapted to suit new purposes, managed variability has a great impact on how software is developed, maintained and extended.

The great variety of techniques for adapting a piece of software can be classified according to the underling variability mechanism . As depicted in Fig. A.1, three basic mechanisms may be used: adaptation, replacement and extension [268].

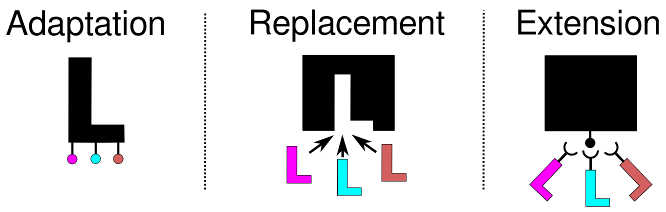


Figure A.1: Basic variability mechanisms.

Adaptation mechanisms allow adapting the behaviour of a component by exploiting its interfaces. Parameters and configurator represent two of the most used variability mechanisms belonging to this category.

Replacement mechanisms allow the presence of several implementations of a component. As the provided interfaces fully specify the required behaviour, components differ from each other only in how this behaviour is implemented. Based on the requested features, during the application engineering phase one component is chosen, and integrated with the rest of the architecture. Two concrete mechanisms of replacement are code generators and components replacement.

Extension mechanisms provide interfaces that allow components to be connected

Variability mechanism	Concrete techniques
Adaptation	Parameters, Build systems, Pre-processors, Design patterns
Replacement	Components
Extensions	Frameworks

Table A.1: Classification of concrete variation mechanisms.

with the rest of the architecture. As architecture requirements are less stringent than in a replacement mechanism, different type of components can be added to an architecture; this implies that a component may not be product-specific.

In the next sub-sections, the concrete variability mechanisms in Table A.1 are explained in more details.

A.1 Mechanisms based on adaptation

A.1.1 Parameters

Parameters embody the simplest, yet the most widely used mechanism for implementing variability. Indeed, parametrisation represents a flexible and fine-grained tool for adapting a system's behaviour at run-time. For instance, different configuration parameters lead to different configurations of the same product.

Nevertheless, unless a compiler performs optimisations, a deployed product contains all functionalities, even though it is known a priori that some of them will be never used. This entails a series of problems in terms of performance, security and resources consumption.

Moreover, dependencies between features are hardly systematically checked, as parameters make it quite difficult to guarantee that at run-time two alternative features are not active. In fact, the use of parameters often leads to poor code quality. On the one hand, global parameters reduce modularity as they violate the principles of separation of concerns and information-hiding. On the other hand, propagating method arguments introduce scattering and tangling of configuration knowledge. Undoubtedly, due to the scattering and lack of cohesion, features code is disseminated across several files and modules. Therefore, it is difficult to trace a feature to all code fragments implementing it.

In conclusion, unless specific conventions are in place, this variability mechanism leads to an undisciplined ad-hoc implementation that is difficult to analyse, maintain and evolve.

A.1.2 Build systems

Build systems (e.g *Apache Maven* [269], *GNU Make* [270] and *Gradle* [271]) automate software building related tasks. Such tasks vary from compiling, assembling and testing source files, to creating and copying deliverable units. Therefore, build systems are convenient to manage variability at files level.

As an example, the Linux kernel is built using *Kbuild*[1]. Linux is probably the only operative system that has a unified code base used for a very broad range of computing systems, ranging from supercomputers to very tiny embedded devices. According to the selected features, Kbuild determines which of the several thousand of C files should be compiled and linked together when building a Linux kernel.

As long as features can be linked to files, build systems are well-suited for feature-oriented product lines. Indeed, being language-agnostic allows building systems to consistently manage variability for both code and non-code artefacts.

Nonetheless, this granularity can be problematic. Firstly, there is no notion of modularity. Secondly, in the case of small changes, one needs to replicate the whole files. In order to solve the last problem, build systems typically orchestrate other variability mechanisms, like pre-processor, version control systems or parametrisation.

Build systems resolve variability at compile time. This implies that, as for parameters, static analysis tasks are undecidable since a build system may execute arbitrary shell scripts. It follows that rigorous and exhaustive analysis is achievable only for restricted languages, or for specific patterns.

A.1.3 Pre-processors

A pre-processor is a tool that, invoked by the compiler as the first part of the translation, shapes source code before being compiled. As an example, the popular C pre-processor *cpp* [2], provides directives that allow inclusion of header files (#include),

[1]`http://kbuild.sourceforge.net/`
[2]`http://gcc.gnu.org/onlinedocs/cpp/`

line controls (#line), macros (#define), as well as conditional compilations (#ifdef, #endif).

Conditional compilation is the prevailing mechanism for modelling variability in product lines [77] since it is natively supported by many programming languages. Indeed, as in Listing A.1, a strategy is to annotate a code fragment with a feature name and conditionally add or remove it before compilation, according to its selection.

This strategy makes quite easy to identify files, or code fragments, belonging to a particular feature.

```
#ifdef Feature_Name  //conditional group begins
    #define Feature_Name //controlled source code
#endif               //conditional group ends
```

Listing A.1: An example of a conditional compilation macro

Pre-processor annotations can be uniformly used to any kind of textual and non-textual artefacts. Indeed, approaches like [78–80] and Eclipse projects like fmp2rsm[3] and FeatureMapper[4] allow annotations and pre-processing of static and behavioural models using UML stereotypes.

Despite their undoubted simplicity, the use of pre-processor annotation has been heavily criticised [81–83] as its undisciplined use may compromise artefact quality and maintainability. In fact, pre-processor directives are usually not confined by mechanisms of the host language[5] and they can be applied with any arbitrary level of granularity. This is especially true when directives are applied at a fine grain level. Indeed, as Listing A.2 shows, scattered annotations used in undisciplined ways make the code difficult to follow. Moreover, considering that many pre-processors lack of proper diagnostic tools [83], pre-processor directives can easily introduce errors difficult to detect.

[3]http://gp.uwaterloo.ca/fmp2rsm
[4]http://featuremapper.org/
[5]For instance, *cpp* works on the basis of directives that control syntactic program transformation. Therefore it is not limited to *C* code.

```
class Stack
{
  void push(Object o
                  #ifdef SECURE
                   ,Hash sha1
                  #endif
            )
  {
       if (o == null
          #ifdef SECURE
           || sha1 == null
          #endif
           )
       [...]
 }
  [...]
}
```

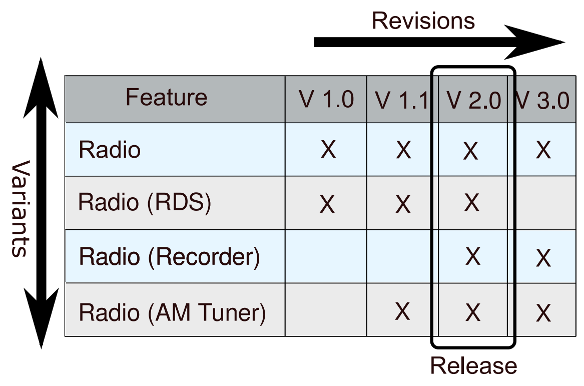
Listing A.2: An example of cpp directives in a Java class.

Scattered annotations used in undisciplined ways may jeopardise separation of concerns, i.e. encapsulation. As a consequence, code implementing a feature could be disseminated across the code base, and intermixed with the code of other features. Moreover, as pre-processor annotations can be defined, and redefined, in different places within the code, even in the case of feature code partitioned into distinct files, it is hard to understand when a determined code fragment will or will not be compiled. Although disciplined annotation [84, 85] and tools support [86, 87] mitigate these weaknesses, they do not solve all of them.

On the whole, pre-processors are a simple, but insidious, means to implement variability. Judging their advantages and weakness depends considerably on their usage.

A.1.4 Version control systems

Version control systems trace changes in both code and non-code artefacts to ease collaborative development. In particular, a version control system allows to manage variation in time (revisions) and variation in space (variants). As depicted in Figure A.2, revisions represent ordered variations over time. A selected set of revisions is called release. A release, which typically has a name or a number[6] , may refer to specific or to all variants. A variant express intended variations that co-exist in parallel. In our example, different variants of radio can co-exist as they are intentionally developed in different branches. The concept of version embraces both revision and variant.



Feature	V 1.0	V 1.1	V 2.0	V 3.0
Radio	X	X	X	X
Radio (RDS)	X	X	X	
Radio (Recorder)			X	X
Radio (AM Tuner)		X	X	X

Figure A.2: Revision and variant in a product line of radio tuners

It is certainly appealing to implement a software product line via multiple branches in a version control system. A simple approach would be to develop each feature in a separate branch, and create a product by merging the corresponding feature branches. Figure A.3 shows an example of this process. Note that each branch does not only contain the code concerning the implemented feature but the entire code-base. It is clear that the link between features and relative implementation it is not so obvious once the final product is created.

Version control systems can be applied uniformly to all artefacts, and changes can be applied at an arbitrary level of granularity. This allows developers to easily manage cross-cutting features, as they can simply change any code fragment without upfront pre-planning. Especially at the beginning of a project, developers can quick respond to a particular customer request, by simply creating a branch and implementing the required feature. Nevertheless, as the product line evolves, problems start to arise. Indeed, using a version control system for feature variability has more disadvantages than advantages.

[6]For example V 2.0 in Figure A.2.

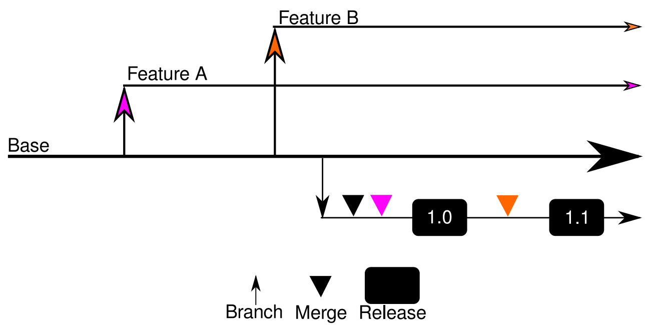


Figure A.3: SPLE by merging per-feature branches

Firstly version control systems encourage development of variants, not features.

Secondly, many customer-specific variants, or many unrelated features branches, make a product family difficult to evolve. In effect, branching does not provide support for modularity, as branches are essentially copies. This implies that even a simple bug fixing in one branch must be merged into all relevant branches. Since it is quite easy to forget such merges, the delta between branches increases more than expected.

Finally, since each branch contains the entire code-base, it is not trivial to solve conflicts while merging branches. Conflicts are usually text-based and are reported by tools as differences between lines of text. Solving conflicts is an error-prone task as usually developers have little or no support from tools. Moreover, considering that merge tools are based on heuristics[272], they may miss conflicts and assemble unreliable code.

A.1.5 Design patterns

By definition, design patterns offer general solutions to recurring design problems [273, 274]. As implementing variability is a recurring problem, design patterns provide the building blocks for realizing it. *Observer*, *template*, *strategy*, and *decorator* are design patterns that can be used in order to decouple and encapsulate variability.

In the observer pattern an entity, called *subject*, notifies registered *observers* when a relevant event has occurred. Assuming that an optional feature can be implemented as an observer, variability is then achieved by its registration or de-registration.

The template pattern defines the structure of an algorithm as an abstract class, which will be further specialised by its subclasses. Therefore, alternative implementations of an abstract feature can be realised by different subclasses. However, due to the limitations of inheritance, this pattern is not suited for combining multiple features (inclusive OR).

The strategy pattern enables an algorithm's behaviour to be selected at runtime using delegation instead of inheritance. That is, instead of an abstract class, a developer specifies a strategy interface, which is implemented by its clients. As a variability mechanism, this pattern is well suited for alternative and optional features, as different clients implement different features.

The decorator pattern specifies a delegation-based mechanism to expand the behaviour of an object at run-time, elegantly solving some composition problems faced with interfaces. This pattern is well-suited to implement multiple and optional features.

Design patterns and parameters share the same advantages and drawbacks. However, design patterns represent a well established, good-practice guidelines for a disciplined implementation of variability. Indeed, in contrast with parameters, design patterns enable separation of concern and information hiding. Moreover, they allow a non-invasive extension, providing a clear separation between base code and features implementation. Therefore, the tracing between features and their implementation code is more explicit.

A.2 Mechanisms based on extension

A.2.1 Frameworks

According to the definition provided by Apel *et al.* [77], a framework is a set of collaboration classes that embodies an abstract solution for a specific problem. It is open for extensions at well-defined points called hot spots, thus supporting reuse at a larger granularity than classes. Likewise the strategy and template design patterns, frameworks use the inversion of control technique [275] to manage the control flow and ask extensions for specialised behaviour at run-time.

We can distinguish between two kinds of frameworks: white-box and black-box. The former consists of a set of concrete and abstract classes, overridden or expanded in order to achieve the desired behaviour. The latter separate framework code and its extensions, typically referred as plug-ins, via interfaces.

White-box frameworks are well suited for implementing alternative features and optional features. Nevertheless, combining multiple features can be problematic due the limitations of sub-classing. On the contrary, in black-box frameworks features are usually implemented as plug-ins composed both at design and run-time. Ideally, this allows a 1-to-1 mapping between features and plug-ins which greatly facilitates traceability and product derivation.

As long they adhere to the common interfaces, both frameworks and plug-ins can be changed independently. However, once defined, hot-spots and interfaces are quite hard to evolve without implying a deep and complex code re-factoring. In fact, plug-ins can be reused only within the context of the framework they have been developed for. Sure enough, due to the fact that every framework encodes specific design, architectural structure and implementation details, it is quite improbable that a plug-in can be shared between several frameworks without being modified.

Although frameworks provide reuse of design, they induce both development and run-time overhead. For example, even if a hot-spot is not used, it will be deployed along with the rest of the code. This results in an increase of system binary size, and eventually a degradation in performance.

Overall, frameworks are best suited for coarse extensions. Conversely, they are unsuited for fine-grained and cross-cutting features. Indeed, fine-grained and cross-cutting features require many hot-spots, greatly increasing the complexity of the final system.

A.3 Mechanisms based on replacement

A.3.1 Components

Components and plug-ins are similar in many respects. Both of them are modular, use interfaces as means to provide access to their encapsulated implementation, and both require a certain amount of pre-planning effort.

By contrast, they differ on reuse beyond product-lines and on their automation potential. While plug-ins are built to fit a particular framework, components are built to fit a particular composition model. Moreover, plug-ins can be loaded automatically, while composing components usually require additional glue-code.

Components and plug-ins share similar advantages and limitations. Both allow compile-time product derivation, deploying only the required (selected) features. At the same time, both share the same limitation for fine-grained and cross-cutting extensions.

A.4 Comparison

The variation mechanisms presented so far can be analysed taking into account the criteria summarised in Table A.2.

Variation mechanisms diverge in how they *represent variability* in the code base, and the way they generate products from user's features selection. In particular, we can distinguish between *annotation*-based and *composition*-based mechanisms. In annotation-based mechanisms (i.e. parameters, build systems and pre-processors), features implementations are merged in a common code-base; code belonging to a specific feature, or to a combination of them, is annotated accordingly. Therefore, annotation can be thought as a function that maps code to the features it belongs to. During product derivation, deselected, or invalid features combinations are removed at compile-time, or ignored at run-time. Composition-based mechanisms (i.e. version control systems, design patterns, frameworks and components) do not share a common code-base, but rather implement features in dedicated, composable units. During product derivation, units corresponding to a valid selection of features are composed to construct the desired variant. Another way to look at the differences between annotation-based and composition-based mechanisms is that the former supports negative variability, while the latter supports positive variability. Negative variability implies that code is removed on demand, while positive variability implies the opposite.

Features are seldom implemented alone. Indeed, their value arises when they collaborate with other features to achieve the desired behaviour. A feature introduces then a change in behaviour, and depending on the implementation technique, this change can take place at a different level of *granularity*. The Level of granularity here refers to the hierarchical structure of artefact implementation. Changes at the top of hierarchy are called *coarse-grained*, while those at the lower levels are defined *fine-grained.* Annotation-based mechanisms support fine-grained changes than composition-based ones. Undoubtedly, annotations can be applied almost everywhere across the common code-base, while composition-approaches rely on interfaces, typically defined at classes or methods level. It is interesting to note that due to their nature, version control systems and pre-processors allow an arbitrary level of granularity.

Variability always involves a certain amount of *pre-planning effort.* As not all variations are predictable, we can distinguish between variability mechanisms that facilitate or impede this task. As shown in Table A.2, composition-based mechanisms (excluding version control systems) require a high pre-planning effort compared to the annotation-based one. For instance, a framework needs to foresee all the possible hot spots in order to provide the variability needed. On the contrary, pre-processor directives can be applied any time variability is required.

Feature-oriented product derivation depends on the capability of establishing and managing a link between features and the relative core-assets. Variability mechanisms support different levels of *feature traceability.* Due to their granularity, annotation-based mechanisms allow a lower level of feature traceability when compared to component-based ones. Indeed, while annotation-based mechanisms realise a n-to-m mapping between features and their implementation, component-based ones enable an ideal 1-to-1 mapping.

Two fundamental principles of software engineering are *separation of concerns*[276] and *information hiding*[277]. Separation of concerns is a design principle for dividing a program into modules so that each module addresses a separate concern. A concern is an area of interest of a system, and features are the primary concern in SPLE. Many approaches, such as procedures and classes, have been developed to implement this design principle. However, during the 90's [90], a particular class of concerns, denominated *cross-cutting concerns*, started to emerge. Cross-cutting concerns are aspects of a system that affect other concerns. These concerns are often hard to decompose [90], resulting in either scattering[7], tangling[8], or both. As features are sometimes

[7]Code duplication.

[8]Significant dependencies between part of a system.

cross-cutting concerns, the capability of a variation mechanism to manage them into a cohesive implementation is an important criterion to consider. Separation of concerns alone is not enough. Indeed, other design principles are needed to guarantee an appropriate level of abstraction. Information hiding allows a dichotomy between interface and implementation, or rather between external (visible), and internal (hidden) parts. In so doing, developers can reason about a module without the burden of understanding its implementation. This implies that a module implementation can be freely changed, so long as it preserves the behaviour specified in the interface. Information hiding is an important quality criterion in software product lines [278], because once features are separated into high cohesive units, we would like to hide their internal implementations, and at the same time make all their communications explicit through interfaces. Doing this, teams can be allocated to single modules, rather than to single products. As depicted in Table A.2, component-based mechanisms are characterised by a high level of separation of concerns and information hiding, but they are not suited to realise cross-cutting features due to their coarse granularity. On the contrary, annotation-based mechanisms are suited to realise cross-cutting features but are characterised by a low level of separation of concerns and information hiding.

Variability offers choices. Indeed, in deriving a product a stakeholder decides which features are included and which are not. It can be also said that a stakeholder binds a decision. *Binding* can happen at different stages; in particular in our discussion we differentiate between *compile-time*, and *load-time* variability. The former is decided before a program starts, while the latter after its compilation when it is actually started. Since variability is resolved before a program starts, compile-time variability allows more room for optimisation. That is, unused code can be removed from the final binaries, reducing the run-time overhead in terms of memory consumption and execution time. As a consequence, once a deployed program is impossible to change. Load-time variability fills this gap, but at the price of memory and performance overhead, as all binaries are compiled and variability must be resolved and checked at run-time. Parameters and frameworks realise binding at loading time, while build systems, version control systems and pre-processors at compile time. Components and design patterns however, can realise both types of binding.

A product family includes code and non-code artefacts. The principle of *uniformity*

[58], states that "features are implemented by a diverse selection of software artefacts and any kind of software artefacts can be subject to subsequent changes and extensions. Conceptually, all artefacts (annotated or composed) should be encoded and synthesised in a similar manner." This principle denotes an important characteristic that a variation mechanism should hold, or rather it should uniformly scale to any kind of assets. Indeed, such a mechanism is far more desirable than many variability mechanisms specific to the languages in which artefacts are created. Build systems, version control systems and pre-processors fully support uniformity, while the other mechanisms are restricted to code-only artefacts.

Finally, composition mechanisms can be compared in the way they support adoption of a product line approach. According to Krueger *et al.* [266] such adoption can follow a *proactive*, *reactive* and *extractive* approach. A proactive, or revolutionary, approach implies that a product line is developed from scratch by carefully applying analysis and design methods. In the opposite way, a reactive approach begins with a small, easy to handle product line which is incrementally extended with new features and artefacts, so as to expand its scope. Finally, an extractive approach starts with a portfolio of existing products and gradually refactor them to form a product line. As described in Table A.2, composition-based mechanisms, due to their high pre-planning effort, are suitable for a proactive approach. Whereas, parameters and version control systems are suitable to realise a reactive approach. In the presence of existing artefacts, pre-processors and building systems are mechanisms perfectly suitable to support an extractive approach.

Variability Mechanism	**Repres.**	**Granularity**	**Pre-planning effort**	**Feature Traceability**	**Separation of Concerns**	**Information Hiding**	**Cross-cutting Features**	**Binding-time**	**Uniformity**	**Adoption Approach**
Parameters	Annotation	Fine-grained	Low	Low	Low	Low	Suited	Load-time	Code-only	Reactive
Build Systems	Annotation	Coarse-grained[a]	Low	Low	Low	Low	Unsuited	Compile-time	Full	Extractive
Version control systems	Composition	Arbitrary	Low	Low	Low	Low	Suited	Compile-time	Full	Reactive
Pre-processors	Annotation	Arbitrary	Low	Low	Low	Low	Unsuited	Compile-time	Full	Extractive
Design-patterns	Composition	Fine-grained	High	High	High	High	Unsuited	Compile-time - Load-time	Code-only	Proactive
Frameworks	Composition	Coarse-grained	High	High	High	High	Unsuited	Load-time	Code-only	Proactive
Components	Composition	Coarse-grained	High	High	High	High	Unsuited	Compile-time - Load-time	Code-only	Proactive

Table A.2: Comparison of concrete variation mechanisms.

[a]File level

Appendix B

X-MAN Meta-model

Listed in E.1 and depicted in Fig. B.1, the new X-MAN meta-model blends component and system development processes [242] into one model. This is in contrast with the previous version of X-MAN [209] where the two processes, although sharing a lot of concepts, were specified by two distinct meta-models.

The component development process is abstracted by the interfaces[1] *Component, Connector, Connection and Data.* Taken together, they form the backbone of the X-MAN meta-model as they protect it from future extensions, thus enhancing its maintainability.

The *Component* interface abstracts the concept of encapsulated components. As described in section 4.3.3, an encapsulated component can either be atomic or composite and can only provide services without requiring any. Hence, *Component* is inherited by the classes *AtomicComponent* and *CompositeComponent*, whereas through the interface *Provider*, it contains one or more *Service* and zero or more *DataElement*. *AtomicComponent* has one *ComputationUnit*, which in turn contains one or more objects of type *Method* and can invoke zero or more *Resource*, or rather *DBConnector*, *Routine* and *DataSpace*. *CompositeComponent* inherits from the interface *Composable* in order to contain one or more *Composable* and *Connection*, as well as zero or more *DataChannel*.

Connector abstracts the concept of exogenous connectors. Therefore, it is inherited by the classes *CompositeConnector* and *AdapterConnector*. *CompositeConnector* it is

[1]Xcore makes no distinction between interfaces and abstract classes: any interface in Xcore is translated to an abstract class once code is generated. Therefore, in this context we can use the two terms interchangeably.

specialised by the classes *Sequencer*, *Selector* and *Aggregator*, while *AdapterConnector* by the classes *Loop* and *Guard*.

In X-MAN control and data flow are separated. Control flow is abstracted by the interface *Connection*, which is extended by the class *CoordinationConnection* that refers to a *Connector* as source and to a *Composable* as connection target. Similarly, data flow is abstracted by the interface *Data*, which is extended by the classes *DataChannel* and *Parameter*. Data flowing among *Input* and *Output* parameters is carried by data channels (*DataChannel*), which implements two *ChannelPolicy*: cache-less or cache-full. A cache-less policy determines a channel that implements a "destructive" read and write. A destructive read means that data is destroyed once read. Similarly, a destructive write entails that a data channel will have its content overwritten when its source produces data items. In contrast, a cache-full data channel implements a non-destructive read, but still implements a destructive write in order to privilege updated data.

Finally, the interface *Composable* provides the link between component and system development processes as it abstracts reuse and composition of pre-built components since it is inherited by *Connector* and *ComponentInstance*. The former models a *Component* (along with its selected services via the class *ServiceReference*) retrieved, and subsequently deployed, from the X-MAN repository.

```
package uk.xman.xcore

enum ChannelPolicy{
 DESTRUCTIVE_READ = 0
 NONE_DESTRUCTIVE_READ = 1
}
enum Language{
 Java = 0
 CPlusPlus = 1
}
enum DataType{
 Integer = 0
 Float = 1
 String = 2
 Boolean = 3
 List = 4
 Set = 5
 Map = 6
 IntegerList = 7
 IntegerSet = 8
 FloatList = 9
 FloatSet = 10
 StringList = 11
 StringSet = 12
 BooleanList = 13
 BooleanSet = 14
}
enum LoopType{
 DoWhile = 0
 WhileDo = 1
}
interface Composable{
 String name = ""
}
interface Connector extends Composable{
 String showedName
}
abstract class CompositionConnector extends Connector{
 refers Connection[2..*] connections
}
class Aggregator extends CompositionConnector {
}
class Sequencer extends CompositionConnector{
}
class Selector extends CompositionConnector{
 contains Input[1..*] input
}
abstract class AdapterConnector extends Connector{
 contains Input[1..*] input
 refers Connection[1..1] connection
}
class Loop extends AdapterConnector{
 LoopType loopType
}
class Guard extends AdapterConnector{
}
interface Connection{
}
class CoordinationConnection extends Connection{
 String condition = ""
 refers Connector source
 refers Composable  target
}
abstract class Component extends Provider {
 String name = "ComponentName"
 String author = "Unspecified"
 String comment = ""
 Boolean verified = "false"
 Boolean valid = "false"
}
class AtomicComponent extends Component{
 contains ComputationUnit[1..1] computationUnit opposite atomicComponent
}
@Ecore(constraints="MustContainAtLeastOneMethod")
class ComputationUnit{
 Language implementationLanguage
 String packageName = ""
 String interfaceCode = ""
 String sourceCode = ""
 contains Method[1..*] methods
 container AtomicComponent[1] atomicComponent opposite computationUnit
}
class Method {
 String name = "MethodName"
 String comment
 contains Parameter[1..*] parameters
}
class CompositeComponent extends Component, Composite {
 boolean isSystem = "false"
 op void addAll(Composable[] data) {
 composables.addAll(data)
 }
}
interface Data{
 String name = "paramName"
 DataType dataType
 refers  DataChannel[0..*] dataChannels
}
@Ecore(constraints = "OrderMustBeSpecified")
abstract class Parameter extends Data{
 int order = "-1"
 container Service service opposite parameters
}
class Input extends Parameter{
}
abstract class Resource{}
 class Output extends Parameter{
}
@Ecore(constraints = "ValueCannotBeNull NameConnotBeNull DefaultValueConnotBeNull")
class DataElement extends Data{
 String range = ""
 String value = ""
 Object oValue
 container Provider component opposite dataElements
}
class DataChannel {
 ChannelPolicy policy
 refers Data  source
 refers Data target
}
class ServiceReference {
 String  name = "Notset"
 refers Service service
}
class Service {
 String name = "ServiceName"
 String comment = ""
 contains Parameter[0..*] parameters opposite service
 contains Contract[0..*] contracts
 contains ServiceReference[0..*] serviceReferences
}
class Contract {
 String name = "ContractName"
 String preCondition = ""
 String postCondition = ""
}
class ComponentInstance extends Composable{
 String componentType = "ComponentType"
 contains Service[1..*]  selectedServices
 contains DataElement[0..*] dataElements
 contains Component[1..1] componentReference
}
abstract class Provider {
 contains Service[1..*] services
 contains DataElement[0..*] dataElements opposite component
}
abstract class Composite {
 contains Composable[1..*] composables
 contains Connection[1..*] connections
 contains DataChannel[0..*] dataChannels
}
```

Listing B.1: Xcore implementation of the X-MAN meta-model.

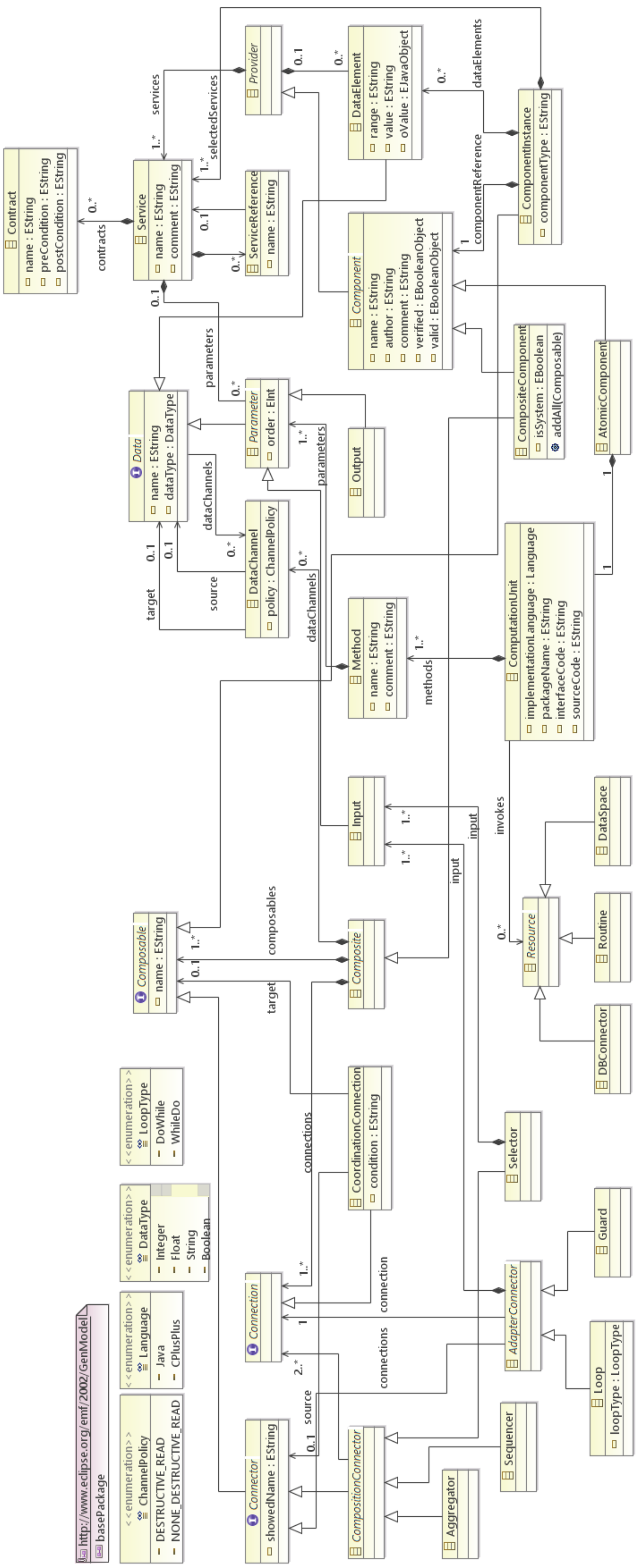


Figure B.1: The X-MAN meta-model.

Appendix C

Functional Model Meta-model

Functional analysis aims at identifying operative commonalities and variabilities among applications in a defined domain. A functional model resulting from this analysis describes domain capabilities from a structural and a behavioural perspective. A structural perspective (captured by the meta-model in appendix C.1) details functional components, or *activities*, and how data flows among them. Complementary to this perspective, the behavioural one (captured by the meta-model in appendix C.2) details in terms of states and transitions between them when, and under what circumstances, functional components are triggered.

C.1 Activity-chart Meta-model

Depicted in Fig. C.1, the main elements of the activity-chart meta-model (listed in G.1) are abstracted by the interfaces *IActivity*, *IFlowLine* and *IConnector* (contained by the *ActivityChart* class).

IActivity abstracts the concept of activity. It is therefore extended by the classes *External*, *DataStoreActivity*, *ControlActivity* and *Activity*. The hierarchical structure of an *Activity* is constructed adopting the composition design pattern [279].

IFlowLine abstracts the flow of information among activities. Since two types of flow-lines are allowed in an activity-chart, *IFlowline* is inherited by the classes *DataFlowLine* and *ControlFlowLine*. The single information element carried by a flow-line is modelled by the class *Element*. It can refer to a *Condition*, an *Event*, a *DataItem* or a group of them. A flow-line originates from the *source* activity (*Vertex*) that produces the information elements and it leads to its *target* activity (*Vertex*) that consumes them.

In order to economise in the number of arrows, flow-lines can be combined using various types of connectors. Such connectors, abstracted by the interface *IConnector*, are modelled by the classes *JunctionConnector* and *JointConnector*. The former models a flow-lines merger, while the latter a flow-line multiplier.

```
package xman.xcore.activitychart

class ActivityChart {
 String chartName
 contains IActivity [2..*] activities
 contains IFlowLine [1..*] flowlines
 contains IConnector [0..*] connectors
}
interface Vertex {
}
interface IActivity extends Vertex {
 String name
}
interface IConnector extends Vertex {
 String name
}
interface IFlowLine {
 String expression
 refers Element [1..1] element
 refers Vertex [1..1] source
 refers Vertex [1..1] target
}
class Activity extends IActivity {
 contains IActivity [0..*] activities
}
class DataStoreActivity extends IActivity {
}
class ControlActivity extends IActivity {
}
class External extends IActivity {
}
class DataFlowLine extends IFlowLine {
}
class ControlFlowLine extends IFlowLine {
}
class JunctionConnector extends IConnector {
}
class JointConnector extends IConnector {
}
```

```
class Element {
 refers DataItem[0..*] dataItems
 refers Condition [0..*] conditions
 refers Event [0..*] events
}
class DataItem {
}
class Condition {
}
class Event {
}
```

Listing C.1: Xcore implementation of the activity-chart meta-model

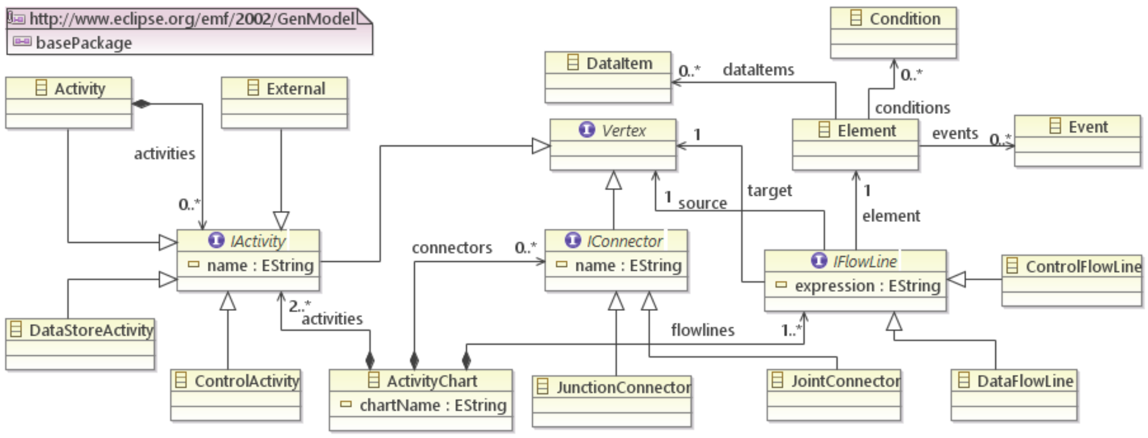


Figure C.1: The X-MAN meta-model.

C.2 State-chart Meta-model

The behavioural description of a system is modelled by the meta-model illustrated in Fig. C.2, which Xcore implementation is listed in Lis. C.2. Its main elements, contained by the class *StateChart*, are abstracted by the interfaces *IState* and *IConnector*, as well as by the class *Transition*.

IState abstracts the concept of state, which can be a *MandatoryState* (i.e. *Start* and *End*), or a concrete *State* which can contain zero or more sub-states.

Transition models the change of state triggered by an event. The trigger of a transition may be an expression that, in a form of a triple (*Triple*), can combine events (*Event*), trigger conditions (*Condition*) and actions (*Action*). Source and target states of a transition are defined by the direct associations *from* and *to* with the *Vertex* interface, which in turn is inherited by *IState*.

Finally, *IConnector* abstract the concept of condition connectors, known as *Xor* connectors, which model transition branching according to a mutual exclusive condition.

```
package uk.xcore.statechart

class StateChart {
 contains IState[2..*] states
 contains IConnector[0..*] connector
 contains Transition[1..*] transition
}

interface IState extends Vertex {
}
interface MandatoryState extends IState {
}
class End extends MandatoryState {
}
class Start extends MandatoryState {
}
class State extends IState {
 String name
 refers State[0..*] states
}
interface Vertex {
}
interface IConnector extends Vertex {
}
class Xor extends IConnector {
 String name
}
class Transition {
 String expression
 refers Triple triple
 refers Vertex [1..1] to
 refers Vertex [1..1] from
}
class Triple {
 refers Event[0..*] events
 refers Condition [0..*] conditions
 refers Action [0..*] actions
}
class Event {
 String name
}
class Condition {
 String conditionCase
}
class Action {
 String name
}
```

Listing C.2: Xcore implementation of the state-chart meta-model

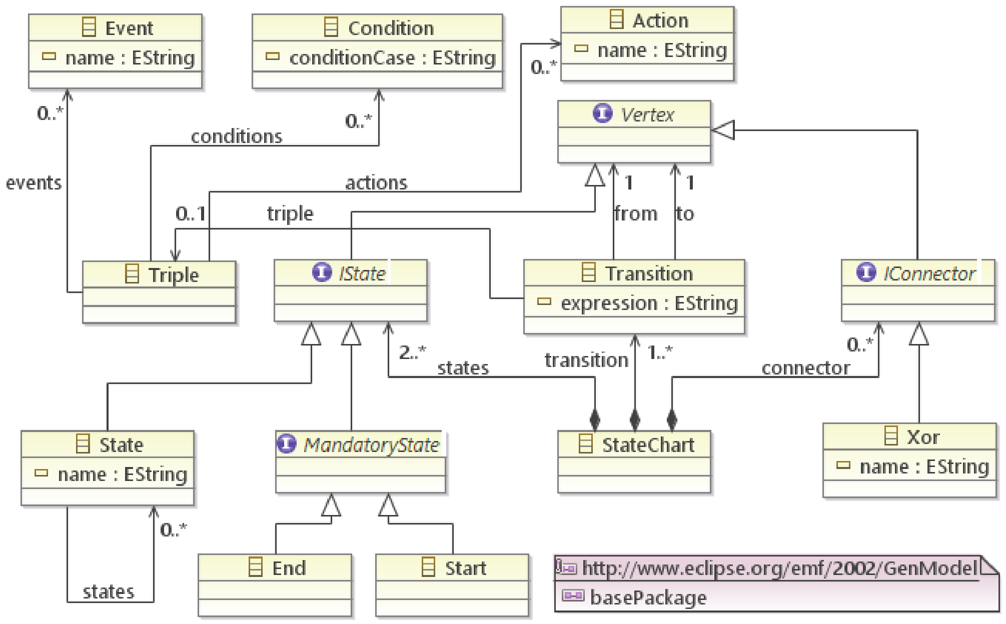


Figure C.2: The state-chart meta-model.

Appendix D

FX-MAN Meta-model

As discussed in chapter 5 and depicted in Fig. 5.1, FX-MAN can be seen a layered component-model. Consequently, its meta-model depicted in Fig. D.1[1] (listed in Lis. D.1) can be described accordingly.

At the lowest level, the class *XMANArchitecture* models instances of X-MAN architectures (retrieved from the shared repository) contained in a *FXMANArchitecture*.

Variability is abstracted by the interface *VariationPoint* and inherited by the classes *Or*, *Alternative* (both extending *NaryVariationPoint*) and *Optional* (which extends *UnaryVariationPoint*). Resulting variability is abstracted by the interface *Variant* and modelled in terms of tuple of X-MAN sets (*TupleXMANSet*) and its related X-MAN sets (*XMANSet*). The latter may contain aggregations of X-MAN architecture (*Aggregation*) resulting from the application of an *Or* variation point.

The interface *FamilyConnector* abstracts the family connectors needed to compose tuples of X-MAN sets. As in X-MAN meta-model, we use two abstract classes to distinguish between *FamilyComposer* and *FamilyAdapter*. The former is extended by the classes *FamilySequencer*, *FamilyAggregator* and *FamilySelector*; the latter is extended by the classes *FamilyGuard* and *FamilyLoop*. In order to filter out unwanted products (or not valid according to the feature model), a *FamilyComposer* can optionally contain a *FamilyFilter*, which semantics is defined by one or more objects of type *Constraint*. The resulting products (*Product*) are referred by a *ProductFamily* contained by an *FXMANArchitecture*.

[1]In order to simplify the diagram, references to the X-MAN meta-models are not shown.

```
package uk.fxman

import org.eclipse.emf.common.util.BasicEList
import uk.xman.xcore.CoordinationConnection
import uk.xman.xcore.Connection
import uk.xman.xcore.DataChannel
import uk.xman.xcore.Service
import uk.xman.xcore.ServiceReference
import uk.xman.xcore.DataElement
import uk.xman.xcore.Composable
import uk.xman.xcore.Connector
import uk.xman.xcore.Input
import uk.xman.xcore.ComponentInstance

enum FamilyLoopType{
 DoWhile = 0
 WhileDo = 1
}
enum ConstraintType{
 Requires = 0
 Excludes = 1
}
class FXMANArchitecture {
 contains FamilyConnector[0..*] familyConnector
 contains XMANArchitecture[1..*] xmanArchitectures
 contains FXMANArchitectureInstance[0..*] fxManArchitectures
 contains DataChannel[1..*] dataChannels
 contains Connection[1..*] connections
 contains Service[1..*] services
 contains VariationPoint[0..*] variationPoint
 contains ProductFamily[0..1] productFamily
 String name = ""
 String author = ""
 String description = ""
}
interface IComponent {
 String name = ""
}
class ProductFamily {
 refers unordered Product[1..*] products
 op void add(Product product) {
  products.add(product)
 }
 op void remove(Product product) {
  products.remove(product)
 }
 op boolean contain(Product product) {
  for (Product p : products) {
   if (p.equals(product))
```

```
    return true
   }
   return false
  }
  op String toString() {
   return products.toString()
  }
 }
 interface Variant {}
 class TupleXMANSet extends Variant {
  refers XMANSet[1..*] sets
  op void add(XMANSet element) {
   sets.add(element)
 }
 op String toString() {
  var s = '<'
  for (m: sets) {
   s = s + m + if (m != sets.last) ',' else ''
  }
  s  = s + '>'
  return s
  }
 }
 class XMANSet extends Variant {
  refers XMANArchitecture[0..*] members
  op void add(XMANArchitecture member) {
   members.add(member)
  }
  op String toString() {
   var s = ''
   if (members.size > 0) {
    s = s + '{'
    for (m: members) {
     s = s + m + if (m != members.last) ',' else ''
    }
    s  = s + '}'
   }
   else {
    s = s + '\u2205'
  }
   return s
  }
  op boolean isEmpty() {
   return members.size === 0
  }
 }
 class Aggregation extends XMANArchitecture {
  refers XMANArchitecture[2..*] members
  refers Service[1..*]  services
  op void add(XMANArchitecture member) {
   members.add(member)
  }
  op String toString() {
   var s = ''
   for (m: members) {
    s = s + m + if (m != members.last) '|' else ''
   }
   s  = s + ''
   return s
  }
 }

 @Ecore(constraints = "NameCannotBeNull")
 class Product {
  String name
  refers unordered ComponentInstance[0..*] xmanArchitectures
  refers Service[1..*]  selectedServices
  refers DataElement[0..*] dataElements
  refers Composable[1..*] composables
  refers Connection[1..*] connections
  refers DataChannel[0..*] dataChannels
  op boolean isEmpty() {
   xmanArchitectures.empty
  }
  op void add(ComponentInstance xman) {
   xmanArchitectures.add(xman)
   sort()
  }
  op void addAll(ComponentInstance[] xman) {
   xmanArchitectures.addAll(xman)
   sort()
  }
  op void remove(ComponentInstance xman) {
   xmanArchitectures.remove(xman)
   sort()
  }
  op void sort() {
   xmanArchitectures.sortInplaceBy[ComponentInstance xman1|xman1.hashCode]
  }
  op Composable[] getConnectedEntities(Connector conn) {
   var vals = new BasicEList<Composable>
   var coords =  connections.filter(CoordinationConnection).filter(coord | coord.
       source === conn)
   for (coord : coords){
    vals.add(coord.target)
   }
  return vals.toEList
  }
```

```
op Composable getRoot() {
 for(composable : composables) {
 var coords =  connections.filter(CoordinationConnection).filter(coord | coord.
     target === composable)
 if (coords.size == 0)
  return composable
}
 return null;
}
op String toString() {
 var s = '['
 if (getRoot != null)
  s = s + visit(getRoot)
 else if (xmanArchitectures.size > 0) {
  for( xa : xmanArchitectures) {
   s = s + xa.name
   s = s + if (xa != xmanArchitectures.last) ',' else ''
  }
 }
 else
  s = s + 'empty'
 s = s + ']'
 return s
}
op String visit(Composable composable) {
 var s = ''
 if (composable instanceof Connector) {
  s = s + composable.class.simpleName.replace("Impl","") + '('
  val conns = connections.filter(CoordinationConnection).filter(
  connection| connection.source === composable)
  for (conn : conns) {
   if (conn.target instanceof Connector)
    s = s + visit(conn.target)
   else
    s = s + (conn.target as XMANArchitecture).name
   s = s + if (conn != conns.last) ',' else ''
  }
  s = s + ')'
 }
 return s
}
op int hashCode() {
 this.xmanArchitectures.hashCode()
}
op boolean equals(Object object) {
 if (object instanceof Product) {
  (object as Product).xmanArchitectures.equals(this.xmanArchitectures)
 }
 else {
   false
  }
 }
}
interface FamilyComposable {
}
class FamilyFilter{
 contains Constraint [1..*] constraints
 op void removeConstraint(String op1, String op2, ConstraintType cons) {
  for(c : constraints){
   if(c.firstOperand.equals(op1) && c.secondOperand.equals(op2) && c.constraint.
       equals(cons)) {
    constraints.remove(c)
    return
   }
  }
 }
}
class Constraint{
 String firstOperand
 String secondOperand
 ConstraintType constraint
 op String toString() {
  if(constraint.equals(ConstraintType.REQUIRES)){
    return firstOperand +" Requires "+ secondOperand;
  }
  else return firstOperand +" Excludes "+ secondOperand;
 }
}
@Ecore(constraints = "TerminalCannotBeNull")
interface FamilyConnector extends FamilyConnectorTerminal {
 String name
 refers FamilyConnector parent
 refers FamilyConnectorTerminal[2..*] productFamilies
}
abstract class FamilyComposer extends FamilyConnector {
 refers FamilyConnection [2..*] connections
 contains FamilyFilter[0..1] filter
}
@Ecore(constraints = "NameCannotBeNull")
class FamilySequencer extends FamilyComposer {
}
class FamilyAggregator extends FamilyComposer {
}
@Ecore(constraints = "InputCannotBeNull")
class FamilySelector extends FamilyComposer {
 contains Input[1..*] input
}
abstract class FamilyAdapter extends FamilyConnector{
 contains Input[1..*] input
 refers FamilyConnection[1..1] connection
}
@Ecore(constraints = "ConnectionCannotBeNull InputCannotBeNull")
class FamilyGuard extends FamilyAdapter{
}
@Ecore(constraints = "ConnectionCannotBeNull InputCannotBeNull")
class FamilyLoop extends FamilyAdapter{
  FamilyLoopType loopType
}
interface VariationPoint extends FamilyConnectorTerminal, VariationPointTerminal {
}
interface NaryVariationPoint{
 refers VariationPointTerminal[2..*] familyMembers
}
interface UnaryVariationPoint {
 refers VariationPointTerminal[1..1] familyMember
}
@Ecore(constraints = "VariationTerminalCannotBeNull")
class Alternative extends VariationPoint, NaryVariationPoint {
}
@Ecore(constraints = "VariationTerminalCannotBeNull")
class Or extends VariationPoint, NaryVariationPoint {
}
@Ecore(constraints = "VariationTerminalCannotBeNull")
class Optional extends VariationPoint, UnaryVariationPoint {
}
interface ConnectionTerminal{
}
interface FamilyConnectorTerminal extends ConnectionTerminal {
}
interface VariationPointTerminal extends ConnectionTerminal {
}
class XMANArchitecture extends FamilyConnectorTerminal, VariationPointTerminal,
    FamilyComposable, ComponentInstance {
 op boolean isEmpty() {
  return componentReference != null
 }
 op String toString() {
  return name
 }
}
class FamilyConnection extends Connection {
 refers FamilyConnector[1..1] source
 refers FamilyConnectorTerminal[1..1] target
 String condition = ""
}
class VariantConnection extends Connection {
 refers VariationPoint[1..1] source
 refers VariationPointTerminal[1..1] target
}
class FXMANArchitectureInstance  extends FamilyConnectorTerminal,
    VariationPointTerminal,  FamilyComposable {
 String name = ""
 contains Service[1..*]  services
 contains FXMANArchitecture refFXMANArchitecture
}
```

Listing D.1: Xcore implementation of the X-MAN meta-model

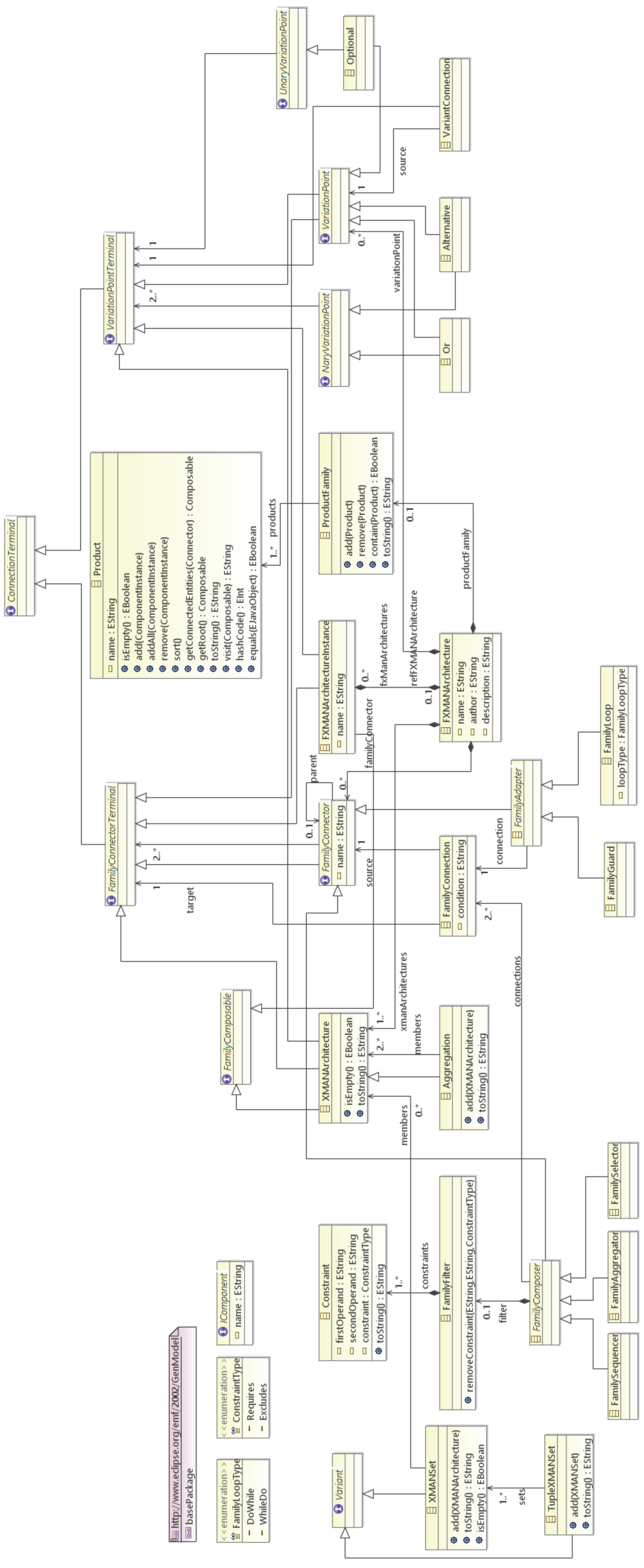


Figure D.1: The FX-MAN meta-model.

Appendix E

Variation Operators Implementation

```
package fxman.metamodel.helper;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import org.paukov.combinatorics.Factory;
import org.paukov.combinatorics.Generator;
import org.paukov.combinatorics.ICombinatoricsVector;
import uk.fxman.Aggregation;
import uk.fxman.FxmanFactory;
import uk.fxman.TupleXMANSet;
import uk.fxman.Variant;
import uk.fxman.XMANArchitecture;
import uk.fxman.XMANSet;
import com.google.common.collect.ImmutableList;
import com.google.common.collect.ImmutableSet;
import com.google.common.collect.Sets;



public class VariationGenerator {

        public static List<List<XMANSet>> distribute(List<List<XMANSet>> input) {
                List<List<XMANSet>> res = new ArrayList<List<XMANSet>>();

                ImmutableList.Builder<ImmutableSet<XMANSet>> listBuilder =
                new ImmutableList.Builder<ImmutableSet<XMANSet>>();
                for (List<XMANSet> lxset : input) {
                        ImmutableSet<XMANSet> isxset = ImmutableSet.copyOf(lxset);
                        listBuilder.add(isxset);
                }
                // generate list of set of XMANSet
                final ImmutableList<ImmutableSet<XMANSet>> listOfSetOfXMANSets =
                    listBuilder.build();

                // perform cartesian products on this list
                Set<List<XMANSet>> products = Sets.cartesianProduct(
                    listOfSetOfXMANSets);

                res.addAll(products);

                return res;
        }


        public static TupleXMANSet GenericOR(List<Variant> variants) {
                TupleXMANSet xmanSetTuple = FxmanFactory.eINSTANCE.createTupleXMANSet
                    ();

                // Create the initial vector
                ICombinatoricsVector<Variant> inputTuple =
                Factory.createVector(variants.toArray(new Variant[0]));

                // create a subsets generator
                Generator<Variant> generator = Factory.createSubSetGenerator(
                    inputTuple);
                List<ICombinatoricsVector<Variant>> combis = generator.
                    generateAllObjects();

                for (ICombinatoricsVector<Variant> combination : combis) {
                        // work around to filter out the empty family set returned
                            automatically
                        // by the combinatoric library
                        if (combination.getSize() == 0)
                        continue;

                        // add straight away into X-MAN Set combination of 1 element
                        if (combination.getSize() == 1) {
                                Variant var = combination.getValue(0);
                                if (var instanceof XMANSet)
                                xmanSetTuple.add((XMANSet) var);
                                else {
                                        for (XMANSet set : ((TupleXMANSet)var).
                                            getSets()) {
                                                xmanSetTuple.add(set);
                                        }
                                }
                        }
```

```
else {
        ImmutableList.Builder<ImmutableSet<XMANSet>>
            listBuilder =
        new ImmutableList.Builder<ImmutableSet<XMANSet>>();

        // populate list builder with elements from
            combination
        for(Variant variant: combination) {
                if (variant instanceof XMANSet) {
                        ImmutableSet<XMANSet> setOfXMANSet =
                            ImmutableSet.of((XMANSet)variant)
                            ;
                        listBuilder.add(setOfXMANSet);
                }
                else {
                        ImmutableSet<XMANSet> setOfXMANSet =
                            ImmutableSet.copyOf(((
                            TupleXMANSet)variant).getSets());
                        listBuilder.add(setOfXMANSet);
                }
        }

        // generate list of sets of XMANSet
        final ImmutableList<ImmutableSet<XMANSet>>
            listOfSetOfXMANSets = listBuilder.build();

        // perform  cartesian products on this list
        Set<List<XMANSet>> products = Sets.cartesianProduct(
            listOfSetOfXMANSets);

        // go through the products and perform 2nd cartesian
            product
        for (List<XMANSet> listOfXMANSet: products) {
                ImmutableList.Builder<ImmutableSet<
                    XMANArchitecture>> listBuilder1 =
                new ImmutableList.Builder<ImmutableSet<
                    XMANArchitecture>>();
                // populate list builder with elements from
                    combination
                for (XMANSet set: listOfXMANSet) {
                        ImmutableSet<XMANArchitecture>
                            setOfXMANArchitecture =
                            ImmutableSet.copyOf (set.
                            getMembers());
                        listBuilder1.add(
                            setOfXMANArchitecture);
                }
                // generate list of XMANSet
```

```
                                        final ImmutableList<ImmutableSet<
                                            XMANArchitecture>> listOfXMANSets =
                                            listBuilder1.build();

                                        // create an XMANSet
                                        XMANSet set1 = FxmanFactory.eINSTANCE.
                                            createXMANSet();

                                        // perform aggregation composition on
                                            cartesian products on this list
                                        Set<List<XMANArchitecture>> products1 = Sets.
                                            cartesianProduct(listOfXMANSets);
                                        for (List<XMANArchitecture> product:
                                            products1) {
                                                // create an Aggregation set per
                                                    combination with 2 or more
                                                    elements
                                                Aggregation aggr = FxmanFactory.
                                                    eINSTANCE.createAggregation();
                                                for (XMANArchitecture arch : product)
                                                aggr.add(arch);

                                                set1.add(aggr);
                                        }

                                        xmanSetTuple.add(set1);
                                }
                        }
                }

                return xmanSetTuple;
        }


        public static TupleXMANSet GenericOR(TupleXMANSet sets) {
                TupleXMANSet xmanSetTuple = FxmanFactory.eINSTANCE.createTupleXMANSet
                    ();

                // Create the initial vector
                ICombinatoricsVector<XMANSet> inputTuple =
                Factory.createVector(sets.getSets().toArray(new XMANSet[0]));

                // create a subsets generator
                Generator<XMANSet> generator = Factory.createSubSetGenerator(
                    inputTuple);
                List<ICombinatoricsVector<XMANSet>> combis = generator.
                    generateAllObjects();

                for (ICombinatoricsVector<XMANSet> combi : combis) {
```

```
        // work around to filter out the empty family set returned
            automatically
        // by the combinatoric library
        if (combi.getSize() == 0)
        continue;

        // add straight away into X-MAN Set combination of 1 element
        if (combi.getSize() == 1)
        xmanSetTuple.add(combi.getValue(0));
        else {
                ImmutableList.Builder<ImmutableSet<XMANArchitecture>>
                     listBuilder =
                new ImmutableList.Builder<ImmutableSet<
                    XMANArchitecture>>();

                // populate list builder with elements from
                    combination
                for(XMANSet set: combi) {
                        ImmutableSet<XMANArchitecture>
                            setOfXMANArchitecture = ImmutableSet.
                            copyOf (set.getMembers());
                        listBuilder.add(setOfXMANArchitecture);
                }
                // generate list of XMANSet
                final ImmutableList<ImmutableSet<XMANArchitecture>>
                    listOfXMANSets = listBuilder.build();

                // create an XMANSet
                XMANSet set = FxmanFactory.eINSTANCE.createXMANSet();

                // perform aggregation composition on cartesian
                    products on this list
                Set<List<XMANArchitecture>> products = Sets.
                    cartesianProduct(listOfXMANSets);
                for (List<XMANArchitecture> product:  products) {
                        // create an Aggregation set per combination
                            with 2 or more elements
                        Aggregation aggr = FxmanFactory.eINSTANCE.
                            createAggregation();
                        for (XMANArchitecture arch : product)
                        aggr.add(arch);

                        set.add(aggr);
                }

                // then add the X-MANSet into the resulted tuple
                xmanSetTuple.add(set);
        }
}
```

```
        return xmanSetTuple;
}


public static TupleXMANSet GenericALT(List<Variant> sets) {
        TupleXMANSet tuple = FxmanFactory.eINSTANCE.createTupleXMANSet();
        for (Variant set : sets) {
                if (set instanceof XMANSet)
                tuple.add((XMANSet) set);
                else if (set instanceof TupleXMANSet) {
                        for (XMANSet subset : ((TupleXMANSet) set).getSets())
                        tuple.add(subset);
                }
        }

        return tuple;
}


public static TupleXMANSet OPT(XMANSet set) {
        TupleXMANSet xmanSetTuple = FxmanFactory.eINSTANCE.createTupleXMANSet
            ();
        xmanSetTuple.add(set);
        // create an empty family set
        XMANSet empty = FxmanFactory.eINSTANCE.createXMANSet();

        xmanSetTuple.add(empty);

        return xmanSetTuple;
}


public static TupleXMANSet OPT(TupleXMANSet sets) {
        // check if there is no empty set yet
        if (!Util.containsEmptyXMANSet(sets)) {
                // create an empty tuple
                XMANSet empty = FxmanFactory.eINSTANCE.createXMANSet();
                // add it
                sets.add(empty);
        }

        return sets;
}

public static TupleXMANSet GenericOPT(Variant variant) {
        if (variant instanceof XMANSet)
        return OPT((XMANSet) variant);
        else
```

```
              return OPT((TupleXMANSet)variant);
    }


}
```

Listing E.1: Xcore implementation of the X-MAN meta-model.

Appendix F

Product Family Architecture Composer Implementation

```
package fxman.spray.architecture.compiler;

[...]

public final class ArchitectureComposer {

	private static final int    UNARY  = 1;
	private static final int    BINARY = 2;
	private int   level = 0;
	private List<String> pfList = new ArrayList<>();
	private Logger LOGGER = Logger.getLogger(ArchitectureComposer.class);
	private FXMANArchitecture fxmanArch;

	public ArchitectureComposer(FXMANArchitecture arch) {
		this.fxmanArch = arch;
	}

	/**
	* Quietly generate products (product families)
	*/
	public void doGenerate() {
		doGenerate(false, false);
	}

	public void doGenerate(ProductFamily pf, Product prd, String name) {
		addExposedServices(pf);
		addDataChannels(pf);

		for (final Product p : pf.getProducts()) {
```

```
                if (prd == p) {
                        p.setName(name);
                        // Create a new diagram
                        final IDiagramContainer diagramContainer = GraphitiUtils.
                            createDiagram(GraphitiConstants.REFERENCE_DESIGN, p.
                            getName());
                        final IFeatureProvider fp = diagramContainer.
                            getDiagramTypeProvider().getFeatureProvider();
                        final TransactionalEditingDomain editingDomain =
                            diagramContainer.getDiagramBehavior().
                            getEditingDomain();
                        editingDomain.getCommandStack().execute(new
                            RecordingCommand(editingDomain) {
                               @Override
                               protected void doExecute() {
                                       Map<EObject, EObject> componentInstancesMap
                                            = createAllComponentInstances(fp, p.
                                           getXmanArchitectures());
                                       Map<EObject, EObject> servicesMap =
                                           createAllExposedService(fp, p.
                                           getSelectedServices());
                                       Map<EObject, EObject> connectorsMap =
                                           createAllConnectors(fp, p.
                                           getComposables());
                                       createAllCoordinationConnections(fp, Util.
                                           getCoordConnections(p),
                                           componentInstancesMap, connectorsMap);
                                       createAllDataChannels(fp, p, p.
                                           getDataChannels(),
                                           componentInstancesMap, servicesMap,
                                           connectorsMap);
                                       createAllServiceReferences(fp, p, p.
                                           getDataChannels(),
                                           componentInstancesMap, servicesMap);
                               }
                        });
                        break;
                }
        }
}


/**
* Generate products and product model files
*
* @param debug
*            print out more details
* @param createFile
*            create product model files
```

```
*/
public void doGenerate(boolean debug, boolean createFile) {
        int counter = 0;
        FamilyConnector rootFConnector = (FamilyConnector) Util.getRoot(
            fxmanArch);

        if (rootFConnector != null) {
                ProductFamily pFamily;
                try {
                        pFamily = (ProductFamily) composeArchitecture(
                            rootFConnector);
                        addExposedServices(pFamily);
                        addDataChannels(pFamily);
                        if (debug) {
                                for (int i = pfList.size() - 1; i >= 0; i--) {
                                        ConsoleManager.println(pfList.get(i));
                                }
                        }

                        if (createFile) {
                                for (final Product p : pFamily.getProducts()) {
                                        p.setName("Product_" + ++counter);
                                        // Create a new diagram
                                        final IDiagramContainer diagramContainer =
                                            GraphitiUtils.createDiagram(
                                            GraphitiConstants.REFERENCE_DESIGN, p.
                                            getName());
                                        final IFeatureProvider fp =
                                            diagramContainer.getDiagramTypeProvider
                                            ().getFeatureProvider();
                                        final TransactionalEditingDomain
                                            editingDomain = diagramContainer.
                                            getDiagramBehavior().getEditingDomain()
                                            ;
                                        editingDomain.getCommandStack().execute(new
                                             RecordingCommand(editingDomain) {
                                                @Override
                                                protected void doExecute() {
                                                        Map<EObject, EObject>
                                                            componentInstancesMap =
                                                            createAllComponentInstances
                                                            (fp, p.
                                                            getXmanArchitectures());
                                                        Map<EObject, EObject>
                                                            servicesMap =
                                                            createAllExposedService(
                                                            fp, p.getSelectedServices
                                                            ());
                                                        Map<EObject, EObject>
                                                            connectorsMap =
                                                            createAllConnectors(fp, p
                                                            .getComposables());
                                                        createAllCoordinationConnections
                                                            (fp, Util.
                                                            getCoordConnections(p),
                                                            componentInstancesMap,
                                                            connectorsMap);
                                                        createAllDataChannels(fp, p,
                                                            p.getDataChannels(),
                                                            componentInstancesMap,
                                                            servicesMap,
                                                            connectorsMap);
                                                        createAllServiceReferences(fp
                                                            , p, p.getDataChannels(),
                                                             componentInstancesMap,
                                                            servicesMap);
                                                }
                                        });

                                }
                        }
                } catch (Exception e) {
                 LOGGER.error("Error while creating the product family.", e);
        }
    }
}



/**
* Create model and diagram for a list of Products
*
* @param products
*/
public static void createModelAndDiagram(final List<Product> products) {
        int counter = 0;
        for (final Product p : products) {
                p.setName("Product_" + ++counter);

                // Create a new diagram
                final IDiagramContainer diagramContainer = GraphitiUtils.createDiagram(
                    GraphitiConstants.REFERENCE_DESIGN, p.getName());
                final IFeatureProvider fp = diagramContainer.getDiagramTypeProvider().
                    getFeatureProvider();
                final TransactionalEditingDomain editingDomain = diagramContainer.
                    getDiagramBehavior().getEditingDomain();
```

```
              editingDomain.getCommandStack().execute(new RecordingCommand(
                  editingDomain) {
                     @Override
                     protected void doExecute() {
                            Map<EObject, EObject> componentInstancesMap =
                                createAllComponentInstances(fp, p.
                                getXmanArchitectures());
                            Map<EObject, EObject> connectorsMap = createAllConnectors
                                (fp, p.getComposables());
                            Map<EObject, EObject> servicesMap =
                                createAllExposedService(fp, p.getSelectedServices());
                            createAllDataChannels(fp, p, p.getDataChannels(),
                                componentInstancesMap, servicesMap, connectorsMap);
                     }
              });

       }
}

private void createAllCoordinationConnections(IFeatureProvider fp, Iterable<
    CoordinationConnection> connections, Map<EObject, EObject> componentInstancesMap,
     Map<EObject, EObject> connectorsMap) {
       Validate.noNullElements(new Object[] { fp, connections, componentInstancesMap,
            connectorsMap });

       ICreateConnectionFeature createConnectionFeature = null;
       CreateConnectionContext createConnectionContext = new CreateConnectionContext
           ();
       Diagram currentDiagram = fp.getDiagramTypeProvider().getDiagram();

       for (CoordinationConnection connection : connections) {
              createConnectionFeature = GraphitiUtils.getCreateConnectionFeauture(fp.
                  getCreateConnectionFeatures(), "Coordination");

              // Set the connection SOURCE
              EObject bo = connectorsMap.get(connection.getSource());
              PictogramElement sourcePe = Graphiti.getLinkService().
                  getPictogramElements(currentDiagram, bo).get(GraphitiConstants.
                  FIRST_ELEMENT);
              createConnectionContext.setSourcePictogramElement(sourcePe);

              // Set the connection TARGET
              bo = getRightMap(connection, componentInstancesMap, connectorsMap).get(
                  connection.getTarget());
              PictogramElement targetPe = Graphiti.getLinkService().
                  getPictogramElements(currentDiagram, bo).get(GraphitiConstants.
                  FIRST_ELEMENT);
              createConnectionContext.setTargetPictogramElement(targetPe);
              org.eclipse.graphiti.mm.pictograms.Connection c =
                  createConnectionFeature.create(createConnectionContext);
              PictogramElement pe = c;
              Object a = Graphiti.getLinkService().
                  getBusinessObjectForLinkedPictogramElement(pe);
              CoordinationConnection res = (CoordinationConnection) a;
              if (connection.getCondition() != null) res.setCondition(setCondition(
                  connection, fxmanArch));
       }

}

private static Map<EObject, EObject> getRightMap(CoordinationConnection connection,
    Map<EObject, EObject> componentInstancesMap, Map<EObject, EObject> connectorMap)
    {
       return (connection.getTarget() instanceof Connector) ? connectorMap :
           componentInstancesMap;
}

private static Map<EObject, EObject> createAllConnectors(IFeatureProvider fp, EList<
    Composable> composables) {
       Validate.noNullElements(new Object[] { fp, composables });

       Map<EObject, EObject> ret = new HashMap<EObject, EObject>();

       ICreateFeature createFeature = null;
       CreateContext createContext = new CreateContext();
       createContext.setTargetContainer(fp.getDiagramTypeProvider().getDiagram());
       createContext.setLocation(0, 0);

       for (Composable composable : composables) {
           int i = 0;
              String createFeatureName = getComposableCreateFeatureName(composable);
              createFeature = GraphitiUtils.getCreateFeature(fp.getCreateFeatures(),
                  createFeatureName);
              Connector newConnector = (Connector) createFeature.create(createContext
                  )[0];
              newConnector.setName(getComposableName(composable, i++));
              newConnector.setShowedName(newConnector.getName());
              List<Input> listOfInputs = new ArrayList<Input>();
              if (newConnector instanceof Selector) {
                     listOfInputs = ((Selector) composable).getInput();
                     updateDefaultInput(((Selector) composable).getInput().get(0), ((
                         Selector) newConnector).getInput().get(0), fp);
              }
              else if (newConnector instanceof Loop) {
                     listOfInputs = ((Loop) composable).getInput();
                     updateDefaultInput(((Loop) composable).getInput().get(0), ((Loop
                         ) newConnector).getInput().get(0), fp);
              }
              else if (newConnector instanceof Guard) {
                     listOfInputs = ((Guard) composable).getInput();
                     updateDefaultInput(((Guard) composable).getInput().get(0), ((
                         Guard) newConnector).getInput().get(0), fp);
              }
              createInputs(listOfInputs, newConnector, fp);
              ret.put(composable, newConnector);
       }
       return ret;
}

private static void updateDefaultInput(Input inputReference, Input inputTarget,
    IFeatureProvider fp) {
       inputTarget.setName(inputReference.getName());
       inputTarget.setDataType(inputReference.getDataType());
       inputTarget.setOrder(inputReference.getOrder());

}

private static void createInputs(List<Input> listOfInputs, Connector connector,
    IFeatureProvider fp) {
       int order = 0;
       // Create inputs

       for (Input inp : listOfInputs) {
              if (order == 0) {
                     if (connector instanceof Selector) {
                         updateDefaultInput(inp, ((Selector) connector).getInput()
                             .get(0), fp);
                     }
                     else if (connector instanceof Loop) {
                         updateDefaultInput(inp, ((Loop) connector).getInput().get
                             (0), fp);
                     }
                     else if (connector instanceof Guard) {
                         updateDefaultInput(inp, ((Guard) connector).getInput().
                             get(0), fp);
                     }
                     order++;
                     continue;
              }
              Point inputPosition = new Point(GraphitiConstants.INITIAL_X_PARAMETER,
                     GraphitiConstants.INITIAL_Y_PARAMETER);
              ContainerShape selectorContainerShape = GraphitiUtils.
                     getTargetContainerShape(fp.getDiagramTypeProvider().getDiagram(),
                     connector);
              ContainerShape inputContainerShape = null;
              for (PictogramElement shape : selectorContainerShape.getChildren()) {
                     if (!shape.getGraphicsAlgorithm().getLineVisible()) {
                         inputContainerShape = (ContainerShape) shape;
                         break;
                     }
              }
              CreateContext createParameterContext = new CreateContext();
              createParameterContext.setTargetContainer(inputContainerShape);
              createParameterContext.setLocation(inputPosition.x, inputPosition.y);
              ICreateFeature iCreateFeature = GraphitiUtils.getCreateFeature(fp.
                     getCreateFeatures(), "Input");
              Input createdInput = (Input) iCreateFeature.create(
                     createParameterContext)[GraphitiConstants.FIRST];
              createdInput.setName(inp.getName());
              createdInput.setDataType(inp.getDataType());
              createdInput.setOrder(inp.getOrder());
              order++;
              inputPosition.y += GraphitiConstants.SPACING_PARAMETER;
       }
}

private static String getComposableCreateFeatureName(Composable composable) {
       String composableSimpleName = composable.getClass().getSimpleName();
       return StringUtils.removeEnd(composableSimpleName, "Impl");
}

private static Map<EObject, EObject> createAllComponentInstances(IFeatureProvider fp,
      EList<ComponentInstance> xmanArchitectures) {
       Validate.noNullElements(new Object[] { fp, xmanArchitectures });

       Map<EObject, EObject> ret = new HashMap<EObject, EObject>();

       for (ComponentInstance xmanArch : xmanArchitectures) {
              ICreateFeature createComponentInstance = GraphitiUtils.getCreateFeature
                     (fp.getCreateFeatures(), "ComponentInstance");

              CreateContext createContext = new CreateContext();
              createContext.setTargetContainer(fp.getDiagramTypeProvider().getDiagram
                     ());
              createContext.setLocation(0, 0);

              ComponentInstance newComponentInstance = (ComponentInstance)
                     createComponentInstance.create(createContext)[0];
              newComponentInstance.setName(xmanArch.getName());

              newComponentInstance.setComponentReference(xmanArch.
                     getComponentReference());

              for (Service service : xmanArch.getSelectedServices()) {
                    GraphitiUtils.createServiceWithParameters(GraphitiUtils.
                        getTargetContainerShape(fp.getDiagramTypeProvider().
                        getDiagram(), newComponentInstance), fp, service);
                }
                ret.put(xmanArch, newComponentInstance);
            }
            return ret;
        }

        public Object composeArchitecture(ConnectionTerminal node) throws Exception {
            level++;

            LOGGER.debug("Compiling FX-MAN archictecture level " + level + ".");

            Object ret = null;
            if (node instanceof FamilyConnector) {
                ret = FxmanFactory.eINSTANCE.createProductFamily();

                // container for variants at level below
                EList<Object> variants = new BasicEList<Object>();

                // recursively go to the next level and calculate variants from level
                // below
                // they can be either variation operators or other family connectors
                for (ConnectionTerminal child : Util.getConnectedEntities((
                    FamilyConnector) node, fxmanArch)) {
                    variants.add(composeArchitecture(child));
                }

                // process composition of variants into product family at the current
                // level
                // only two kinds of composition: unary or n-ary
                if (node instanceof FamilySequencer && variants.size() >= BINARY) {
                    Connector newConnector = XcoreFactory.eINSTANCE.createSequencer
                        ();
                    newConnector.setName(((FamilyConnector) node).getName());
                    ret = naryFamilyComposer(variants, newConnector);
                }
                else if (node instanceof FamilySelector && variants.size() >= BINARY) {
                    Connector newConnector = XcoreFactory.eINSTANCE.createSelector()
                        ;
                    newConnector.setName(((FamilyConnector) node).getName());
                    addInputToConnector(((FamilySelector) node).getInput(),
                        newConnector);
                    ret = naryFamilyComposer(variants, newConnector);
                }
                else if (node instanceof FamilyAggregator && variants.size() >= BINARY)
                     {
```

```
                Connector newConnector = XcoreFactory.eINSTANCE.createAggregator
                    ();
                newConnector.setName(((FamilyConnector) node).getName());
                ret = naryFamilyComposer(variants, newConnector);
        }
        else if (node instanceof FamilyLoop && variants.size() == UNARY) {
                AdapterConnector newConnector = XcoreFactory.eINSTANCE.
                    createLoop();
                newConnector.setName(((FamilyConnector) node).getName());
                addInputToConnector(((FamilyLoop) node).getInput(), newConnector
                    );
                ret = unaryFamilyComposer(variants.get(0), newConnector);
        }
        else if (node instanceof FamilyGuard && variants.size() == UNARY) {
                AdapterConnector newConnector = XcoreFactory.eINSTANCE.
                    createGuard();
                newConnector.setName(((FamilyConnector) node).getName());
                addInputToConnector(((FamilyGuard) node).getInput(),
                    newConnector);
                ret = unaryFamilyComposer(variants.get(0), newConnector);
        }

        // apply the filter
        if (node instanceof FamilyComposer) {
                if (ret != null && ((FamilyComposer) node).getFilter() != null
                    && !((FamilyComposer) node).getFilter().getConstraints().
                    isEmpty()) {
                        ret = filterProducts((ProductFamily) ret, ((
                            FamilyComposer) node).getFilter().gets());
                }

        }

}
else if (node instanceof VariationPoint) {
        // container for variants
        List<Variant> variants = new ArrayList<Variant>();

        // recursively go down and calculate the variants at the lower level
        List<VariationPointTerminal> objects = Util.getConnectedEntities((
            VariationPoint) node, fxmanArch);
        for (VariationPointTerminal object : objects) {
                Object obj = composeArchitecture(object);
                // if the calculation indeed returns variants and store them
                if (obj instanceof Variant)
                variants.add((Variant) obj);
                else ConsoleManager.println("Variation operator " + node.
                    getClass().getCanonicalName() + " encounters unsupported
                    type.");
                }

                if (variants.size() == 0) { throw new Exception("Empty set of variants
                    encountered."); }

                // process the variation operation to generate more variants at the
                // current level
                if (node instanceof Optional)
                ret = VariationGenerator.GenericOPT(variants.get(0));
                else if (node instanceof Alternative)
                ret = VariationGenerator.GenericALT(variants);
                else ret = VariationGenerator.GenericOR(variants);
        }
        else if (node instanceof XMANArchitecture) {
                ret = FxmanFactory.eINSTANCE.createXMANSet();
                ((XMANSet) ret).getMembers().add((XMANArchitecture) node);
        }
        else if (node instanceof FXMANArchitectureInstance) {
                FXMANArchitectureInstance familyInstance = (FXMANArchitectureInstance)
                    node;
                ret = new ArchitectureComposer(familyInstance.getRefFXMANArchitecture()
                    ).composeArchitecture(Util.getRoot(familyInstance.
                    getRefFXMANArchitecture()));
        }
        LOGGER.debug("Complete FX-MAN archictecture level " + level + ".");
        level--;

        return ret;
}



public Object composeArchitectureForProductExplorer(ConnectionTerminal node) throws
    Exception {
        level++;

        LOGGER.debug("Compiling FX-MAN archictecture level " + level + ".");

        Object ret = null;
        if (node instanceof FamilyConnector) {
                ret = FxmanFactory.eINSTANCE.createProductFamily();

                // container for variants at level below
                EList<Object> variants = new BasicEList<Object>();

                // recursively go to the next level and calculate variants from level
                // below
                // they can be either variation operators or other family connectors
```

```
for (ConnectionTerminal child : Util.getConnectedEntities((
    FamilyConnector) node, fxmanArch)) {
        variants.add(composeArchitectureForProductExplorer(child));
}

// process composition of variants into product family at the current
// level
// only two kinds of composition: unary or n-ary
if (node instanceof FamilySequencer && variants.size() >= BINARY) {
        Connector newConnector = XcoreFactory.eINSTANCE.createSequencer
            ();
        newConnector.setName(((FamilyConnector) node).getName());
        ret = naryFamilyComposer(variants, newConnector);
}
else if (node instanceof FamilySelector && variants.size() >= BINARY) {
        Connector newConnector = XcoreFactory.eINSTANCE.createSelector()
            ;
        newConnector.setName(((FamilyConnector) node).getName());
        ret = naryFamilyComposer(variants, newConnector);
}
else if (node instanceof FamilyAggregator && variants.size() >= BINARY)
     {
        Connector newConnector = XcoreFactory.eINSTANCE.createAggregator
            ();
        newConnector.setName(((FamilyConnector) node).getName());
        ret = naryFamilyComposer(variants, newConnector);
}
else if (node instanceof FamilyLoop && variants.size() == UNARY) {
        AdapterConnector newConnector = XcoreFactory.eINSTANCE.
            createLoop();
        newConnector.setName(((FamilyConnector) node).getName());
        ret = unaryFamilyComposer(variants.get(0), newConnector);
}
else if (node instanceof FamilyGuard && variants.size() == UNARY) {
        AdapterConnector newConnector = XcoreFactory.eINSTANCE.
            createGuard();
        newConnector.setName(((FamilyConnector) node).getName());
        ret = unaryFamilyComposer(variants.get(0), newConnector);
}

// apply the filter
if (node instanceof FamilyComposer) {
        if (ret != null && ((FamilyComposer) node).getFilter() != null
            && !((FamilyComposer) node).getFilter().getConstraints().
            isEmpty()) {
                ret = filterProducts((ProductFamily) ret, ((
                    FamilyComposer) node).getFilter().getConstraints());
        }
```

```
                }

        }
        else if (node instanceof VariationPoint) {
                // container for variants
                List<Variant> variants = new ArrayList<Variant>();

                // recursively go down and calculate the variants at the lower level
                List<VariationPointTerminal> objects = Util.getConnectedEntities((
                    VariationPoint) node, fxmanArch);
                for (VariationPointTerminal object : objects) {
                        Object obj = composeArchitectureForProductExplorer(object);
                        // if the calculation indeed returns variants and store them
                        if (obj instanceof Variant)
                        variants.add((Variant) obj);
                        else ConsoleManager.println("Variation operator " + node.
                            getClass().getCanonicalName() + " encounters unsupported
                            type.");
                }

                if (variants.size() == 0) { throw new Exception("Empty set of variants
                    encountered."); }

                // process the variation operation to generate more variants at the
                // current level
                if (node instanceof Optional)
                ret = VariationGenerator.GenericOPT(variants.get(0));
                else if (node instanceof Alternative)
                ret = VariationGenerator.GenericALT(variants);
                else ret = VariationGenerator.GenericOR(variants);
        }
        else if (node instanceof XMANArchitecture) {
                ret = FxmanFactory.eINSTANCE.createXMANSet();
                ((XMANSet) ret).getMembers().add((XMANArchitecture) node);
        }
        else if (node instanceof FXMANArchitectureInstance) {
                FXMANArchitectureInstance familyInstance = (FXMANArchitectureInstance)
                    node;
                ret = new ArchitectureComposer(familyInstance.getRefFXMANArchitecture()
                    ).composeArchitecture(Util.getRoot(familyInstance.
                    getRefFXMANArchitecture()));
        }
        LOGGER.debug("Complete FX-MAN archictecture level " + level + ".");
        level--;

        return ret;
}
```

```java
private Object unaryFamilyComposer(Object variant, AdapterConnector connector) {
        ProductFamily retPf = FxmanFactory.eINSTANCE.createProductFamily();

        EList<ProductFamily> pfs = new BasicEList<ProductFamily>();
        convert(pfs, variant);

        // call the family connectors to perform composition
        retPf = unaryFamilyComposer(pfs.get(0), connector);

        return retPf;
}



/**
* Compose variants into product family
*
* @param variants
* @param connector
* @return
*/
private ProductFamily naryFamilyComposer(EList<Object> variants, Connector connector)
     {
        ProductFamily retPf = FxmanFactory.eINSTANCE.createProductFamily();

        // convert list of variants into list of product family in order to be
        // composable
        // by the existing implementation
        EList<ProductFamily> pfs = new BasicEList<ProductFamily>();
        for (Object variant : variants) {
                convert(pfs, variant);
        }

        // call the family connectors to perform composition
        retPf = naryFamilyComposer(pfs, connector);

        return retPf;
}

/**
* @param pfs
* @param variant
*/
public void convert(EList<ProductFamily> pfs, Object variant) {
        if (variant instanceof ProductFamily)
        pfs.add((ProductFamily) variant);
        else if (variant instanceof TupleXMANSet) {
                ProductFamily pf = FxmanFactory.eINSTANCE.createProductFamily();

                // flatten all the XMANSet in the tuple
```

```
for (XMANSet xmanSet : ((TupleXMANSet) variant).getSets()) {
        if (xmanSet.isEmpty()) {
                Product p = FxmanFactory.eINSTANCE.createProduct();
                pf.add(p);
        }
        else {
                // create a product for each XMANArchitecture in the
                    XMANSet
                for (XMANArchitecture arch : xmanSet.getMembers()) {
                        // create a product and populate it with
                            Aggregator
                        Product p = FxmanFactory.eINSTANCE.createProduct()
                            ;
                        if (arch instanceof Aggregation) {
                                p.setName("Agg");
                                // add the Aggregator connector
                                Aggregator aggr = XcoreFactory.eINSTANCE.
                                    createAggregator();
                                p.getComposables().add(aggr);
                                for (XMANArchitecture childArch : ((
                                    Aggregation) arch).getMembers()) {
                                        // add the XMAN architecture
                                        p.add(childArch);

                                        // create a coordination connection
                                            linking Aggregator and the
                                        // architecture
                                        CoordinationConnection coord =
                                            XcoreFactory.eINSTANCE.
                                            createCoordinationConnection();
                                        coord.setSource(aggr);
                                        coord.setTarget(childArch);
                                        aggr.getConnections().add(coord);
                                        // add the coordination connection
                                        p.getConnections().add(coord);
                                }
                        }
                        else {
                                p.setName(arch.getName());
                                p.add(arch);
                        }

                        // add the product into the product family
                        pf.add(p);
                }
        }
}

pfs.add(pf);
```

```
}
else if (variant instanceof XMANSet) {
        ProductFamily pf = FxmanFactory.eINSTANCE.createProductFamily();

        XMANSet xmanSet = ((XMANSet) variant);
        // if the XMANSet is empty - as generated from Optional operator
        // create a respective Product
        if (xmanSet.isEmpty()) {
                Product p = FxmanFactory.eINSTANCE.createProduct();
                // add the product into the product family
                pf.add(p);
        }
        // otherwise create a Product per XMANArchitecture in non empty XMANSet
        else {
                // create a product for each XMANArchitecture in the XMANSet
                for (XMANArchitecture arch : xmanSet.getMembers()) {
                        // create a product and populate it with Aggregator
                        Product p = FxmanFactory.eINSTANCE.createProduct();
                        if (arch instanceof Aggregation) {
                                p.setName("Agg");
                                Aggregator aggr = XcoreFactory.eINSTANCE.
                                    createAggregator();
                                // add the Aggregator connector
                                p.getComposables().add(aggr);
                                for (XMANArchitecture childArch : ((Aggregation)
                                    arch).getMembers()) {
                                        // add the XMAN architecture
                                        p.add(childArch);
                                        // create a coordination connection linking
                                             Aggregator and the
                                        // architecture
                                        CoordinationConnection coord = XcoreFactory
                                            .eINSTANCE.createCoordinationConnection
                                            ();
                                        coord.setSource(aggr);
                                        coord.setTarget(childArch);
                                        aggr.getConnections().add(coord);
                                        // add the coordination connection
                                        p.getConnections().add(coord);
                                }
                        }
                        else {
                                p.setName(arch.getName());
                                p.add(arch);
                        }

                        // add the product into the product family
                        pf.add(p);
                }
                }

                pfs.add(pf);
        }
}

private ProductFamily unaryFamilyComposer(ProductFamily pf, AdapterConnector adapter)
     {
        ProductFamily pFamily = FxmanFactory.eINSTANCE.createProductFamily();

        for (Product product : pf.getProducts()) {
                Connector rootConnector = Util.getRoot(product);
                if (rootConnector != null) {
                        createConnection(product, adapter, rootConnector);
                }
                else {
                        createConnection(product, adapter, product.getXmanArchitectures
                            ().get(0));
                }

                product.getComposables().add(adapter);

                pFamily.add(product);
        }

        return pFamily;
}

/**
* Generate many products using a specified connector
*
* @param pflist
*           list of product families to be composed
* @param connector
*           X-MAN connector to be used for all products
* @return product family
*/
private ProductFamily naryFamilyComposer(List<ProductFamily> pflist, Connector
    connector) {

        ProductFamily res = binaryFamilyComposer(pflist.get(0), pflist.get(1),
            connector);
        for (int i = 2; i < pflist.size(); i++) {
                res = binaryFamilyComposer(res, pflist.get(i), connector);
        }
        return res;
}

/**
```

```
* Compose product family 1 and 2 using a connector
*
* @param productFamily1
* @param productFamily2
* @param conn
* @return the combined product family
*/
private ProductFamily binaryFamilyComposer(ProductFamily productFamily1,
    ProductFamily productFamily2, Connector conn) {
       ProductFamily pFamily = FxmanFactory.eINSTANCE.createProductFamily();

       for (Product product1 : productFamily1.getProducts()) {
              for (Product product2 : productFamily2.getProducts()) {
                     Product newProduct;
                     if (product1.isEmpty() || product2.isEmpty()) {
                            newProduct = product1.isEmpty() ? product2 : product1;
                     }
                     else {
                            newProduct = FxmanFactory.eINSTANCE.createProduct();
                            Connector rootConnector1 = Util.getRoot(product1);
                            if (rootConnector1 == null) {
                                   for (ComponentInstance xmanArch : product1.
                                       getXmanArchitectures()) {
                                          newProduct.add(xmanArch);
                                          createConnection(newProduct, conn, xmanArch
                                              );
                                   }
                            }
                            else {
                                   combineProducts(newProduct, product1);
                                   if (conn != rootConnector1) createConnection(
                                       newProduct, conn, rootConnector1);
                            }

                            Connector rootConnector2 = Util.getRoot(product2);
                            if (Util.getRoot(product2) == null) {
                                   for (ComponentInstance xmanArch : product2.
                                       getXmanArchitectures()) {
                                          newProduct.add(xmanArch);
                                          createConnection(newProduct, conn, xmanArch
                                              );
                                   }
                            }
                            else {
                                   combineProducts(newProduct, product2);
                                   if (conn != rootConnector2) createConnection(
                                       newProduct, conn, rootConnector2);
                            }
```

```
                        if (!newProduct.getComposables().contains(conn))
                            newProduct.getComposables().add(conn);
                    }

                    pFamily.add(newProduct);
                }
            }
            return pFamily;
        }

        /**
        * Combines the old product into the new product
        *
        * @param newProduct
        * @param oldProduct
        */
        private void combineProducts(Product newProduct, final Product oldProduct) {

            newProduct.addAll(oldProduct.getXmanArchitectures());
            newProduct.getComposables().addAll(oldProduct.getComposables());
            newProduct.getConnections().addAll(oldProduct.getConnections());
            newProduct.getDataChannels().addAll(oldProduct.getDataChannels());
            newProduct.getDataElements().addAll(oldProduct.getDataElements());
            newProduct.getSelectedServices().addAll(oldProduct.getSelectedServices());
        }

        private void createConnection(Product newProduct, Connector src, Composable trg) {
            CoordinationConnection coordinationConnection = XcoreFactory.eINSTANCE.
                createCoordinationConnection();
            coordinationConnection.setSource(src);
            coordinationConnection.setTarget(trg);
            newProduct.getConnections().add(coordinationConnection);
        }

        private static String getComposableName(Composable composable, int index) {
            return composable.getName().isEmpty() ? composable.getClass().getSimpleName().
                replaceAll("Impl", "") + index : composable.getName();
        }

        private void addDataChannels(ProductFamily pf) {
            for (Product product : pf.getProducts()) {
                for (DataChannel dataChannel : fxmanArch.getDataChannels()) {
                    Data source = dataChannel.getSource();
                    Data target = dataChannel.getTarget();
                    if (source != null && target != null) {
                        product.getDataChannels().add(dataChannel);
                    }
                }
            }
```

```
}

public void addExposedServices(ProductFamily pf) {
        for (Product product : pf.getProducts()) {
                for (Service service : fxmanArch.getServices()) {
                        if (isReferencedServiceExist(product, service)) {
                                product.getSelectedServices().add(service);
                        }
                }

        }
}

private boolean isReferencedServiceExist(Product product, Service service) {
        for (ServiceReference serviceReference : service.getServiceReferences()) {
                boolean flag = false;
                for (ComponentInstance component : product.getXmanArchitectures()) {
                        for (Service serviceIns : component.getSelectedServices()) {
                                String check = component.getName() + "." + serviceIns.
                                    getName();
                                if (check.equals(serviceReference.getName())) {
                                        flag = true;
                                        break;
                                }
                        }
                        if (flag == true) break;
                }
                if (flag == false) return false;
        }
        return true;
}

private static boolean isReferencedServiceExist(Service service, String
    serviceReferenceName) {
        for (ServiceReference serviceReference : service.getServiceReferences()) {
                if (serviceReference.getName().equals(serviceReferenceName)) return
                    true;
        }
        return false;
}

private static ComponentInstance findComponentStatic(Data param, Product product) {
    // service
        for (ComponentInstance xman : product.getXmanArchitectures()) {
                for (Service service : xman.getSelectedServices()) {
                        for (Parameter parameter : service.getParameters()) {
                                if (param.equals(parameter)) return xman;
                        }
                }
        }
        return null;
}

private static Service findServiceStatic(Data param, Product product) { // service
        for (Service service : product.getSelectedServices()) {
                for (Parameter parameter : service.getParameters()) {
                        if (param.equals(parameter)) return service;
                }
        }
        return null;
}

private static EObject findParameterStatic(Data param, Map<EObject, EObject> values,
    ComponentInstance compName) { // service
        ComponentInstance comp = (ComponentInstance) values.get(compName);
        if (comp != null) {
                for (Service service : comp.getSelectedServices()) {
                        for (Parameter parameter : service.getParameters()) {
                                if (param.getName().equals(parameter.getName())) return
                                    parameter;
                        }
                }
        }

        return null;
}




private static Composable findComponentConnectorStatic(Data param, Product product) {
     // service
        for (Composable composable : product.getComposables()) {
                List<Input> listOfInputs = new ArrayList<Input>();
                if (composable instanceof Selector) {
                        listOfInputs = ((Selector) composable).getInput();
                }
                else if (composable instanceof Loop) {
                        listOfInputs = ((Loop) composable).getInput();
                }
                else if (composable instanceof Guard) {
                        listOfInputs = ((Guard) composable).getInput();
                }
                for (Input input : listOfInputs) {
                        if (param.equals(input)) return composable;
                }
        }
        return null;
```

```
}

private static EObject findParameterServiceStatic(Data param, Map<EObject, EObject>
    values, Service serviceName) { // service
       Service service = (Service) values.get(serviceName);
       if (service != null) {
              for (Parameter parameter : service.getParameters()) {
                     if (param.getName().equals(parameter.getName())) return
                         parameter;
              }

       }

       return null;
}

private static EObject findParameterConnectorStatic(Data param, Map<EObject, EObject>
     values, Composable composable) { // service
       Composable comp = (Composable) values.get(composable);
       List<Input> listOfInputs = new ArrayList<Input>();
       if (comp instanceof Selector) {
              listOfInputs = ((Selector) comp).getInput();
       }
       else if (comp instanceof Loop) {
              listOfInputs = ((Loop) comp).getInput();
       }
       else if (comp instanceof Guard) {
              listOfInputs = ((Guard) comp).getInput();
       }
       if (comp != null) {
              for (Parameter parameter : listOfInputs) {
                     if (param.getName().equals(parameter.getName())) return
                         parameter;
              }

       }

       return null;
}

private static void createAllDataChannels(IFeatureProvider fp, Product product, EList
    <DataChannel> dataChannels, Map<EObject, EObject> componentInstancesMap, Map<
    EObject, EObject> servicesMap, Map<EObject, EObject> connectorsMap) {

       ICreateConnectionFeature createConnectionFeature = null;
       CreateConnectionContext createConnectionContext = new CreateConnectionContext
           ();
       Diagram currentDiagram = fp.getDiagramTypeProvider().getDiagram();
       for (DataChannel dataChannel : dataChannels) {
```

```
createConnectionFeature = GraphitiUtils.getCreateConnectionFeauture(fp.
    getCreateConnectionFeatures(), "Data Channel");

// Set the connection SOURCE
ComponentInstance compSource = findComponentStatic(dataChannel.
    getSource(), product);
EObject source = null;
if (compSource == null) {
       Service serviceSource = findServiceStatic(dataChannel.getSource
           (), product);
       source = findParameterServiceStatic(dataChannel.getSource(),
           servicesMap, serviceSource);
}
else {
       source = findParameterStatic(dataChannel.getSource(),
           componentInstancesMap, compSource);
}
// Set the connection TARGET
ComponentInstance compTarget = findComponentStatic(dataChannel.
    getTarget(), product);
EObject target = null;
if (compTarget == null) {
       Composable composableTarget = findComponentConnectorStatic(
           dataChannel.getTarget(), product);
       target = findParameterConnectorStatic(dataChannel.getTarget(),
           connectorsMap, composableTarget);
}
else {
       target = findParameterStatic(dataChannel.getTarget(),
           componentInstancesMap, compTarget);

}
if (target == null) {
       Service serviceTarget = findServiceStatic(dataChannel.getTarget
           (), product);
       target = findParameterServiceStatic(dataChannel.getTarget(),
           servicesMap, serviceTarget);
}

EObject bo = source;
EObject bo2 = target;
if (bo != null && bo2 != null) {
       PictogramElement sourcePe = Graphiti.getLinkService().
           getPictogramElements(currentDiagram, bo).get(
           GraphitiConstants.FIRST_ELEMENT);
       createConnectionContext.setSourcePictogramElement(sourcePe);
       PictogramElement targetPe = Graphiti.getLinkService().
           getPictogramElements(currentDiagram, bo2).get(
           GraphitiConstants.FIRST_ELEMENT);
```

```
                    createConnectionContext.setTargetPictogramElement(targetPe);
                    createConnectionFeature.create(createConnectionContext);
            }

        }

}

private static void createAllServiceReferences(IFeatureProvider fp, Product product,
    EList<DataChannel> dataChannels, Map<EObject, EObject> componentInstancesMap, Map
    <EObject, EObject> servicesMap) {
        for (Map.Entry<EObject, EObject> entryService : servicesMap.entrySet()) {
                Service service = (Service) entryService.getKey();
                for (Map.Entry<EObject, EObject> entryComponent : componentInstancesMap
                    .entrySet()) {
                        ComponentInstance compIns = (ComponentInstance) entryComponent.
                            getValue();

                        for (Service serv : compIns.getSelectedServices()) {
                                if (isReferencedServiceExist(service, compIns.getName() +
                                     "." + serv.getName())) {

                                        CreateContext createContext = new CreateContext();
                                        createContext.setTargetContainer(GraphitiUtils.
                                            getTargetContainerShape(fp.
                                            getDiagramTypeProvider().getDiagram(),
                                            entryService.getValue()));
                                        createContext.setLocation(GraphitiConstants.
                                            INITIAL_X_PARAMETER, GraphitiConstants.
                                            INITIAL_Y_PARAMETER);
                                        ServiceReference newServiceReference = (
                                            ServiceReference) GraphitiUtils.
                                            getCreateFeature(fp.getCreateFeatures(), "
                                            Service Reference").create(createContext)[0];

                                        // Set its properties
                                        newServiceReference.setName(compIns.getName() + ".
                                            " + serv.getName());
                                        newServiceReference.setService(serv);
                                }

                        }
                }
        }
        // Create the context

}
```

```
private static Map<EObject, EObject> createAllExposedService(IFeatureProvider fp,
    EList<Service> services) {
       Map<EObject, EObject> ret = new HashMap<EObject, EObject>();
       for (Service service : services) {
              Service newService = GraphitiUtils.createServiceWithParameters(fp.
                  getDiagramTypeProvider().getDiagram(), fp, service);
              ret.put(service, newService);
       }
       return ret;
}

private static String setCondition(CoordinationConnection connection,
    FXMANArchitecture fxmanArc) {
       List<Connection> connections = fxmanArc.getConnections();
       for (Connection conn : connections) {
              if (conn instanceof FamilyConnection) {
                     if (((FamilyConnection) conn).getSource().getName().equals(
                         connection.getSource().getName())) {
                            if (traverseTree(fxmanArc, ((FamilyConnection) conn).
                                getTarget(), connection)) { return ((FamilyConnection
                                ) conn).getCondition(); }
                     }
              }
       }
       return "";
}

public void addInputToConnector(List<Input> inputs, Connector connector) {
       if (connector instanceof Selector) {
              ((Selector) connector).getInput().addAll(EcoreUtil.copyAll(inputs));
       }
       else if (connector instanceof Loop) {
              ((Loop) connector).getInput().addAll(EcoreUtil.copyAll(inputs));
       }
       else if (connector instanceof Guard) {
              ((Guard) connector).getInput().addAll(EcoreUtil.copyAll(inputs));
       }
}

public static boolean traverseTree(FXMANArchitecture fxman, ConnectionTerminal
    current, CoordinationConnection connectionTarget) {
       if (current instanceof FamilySequencer) {
              if (((FamilySequencer) current).getName().equals(connectionTarget.
                  getTarget().getName())) {
                     return true;
              }
              else {
                     EList<FamilyConnection> listOfConnection = ((FamilySequencer)
                         current).getConnections();
```

```
                        for (FamilyConnection con : listOfConnection) {
                                boolean res = traverseTree(fxman, con.getTarget(),
                                    connectionTarget);
                                if (res) return true;
                        }
                }

        }
        else if (current instanceof FamilySelector) {
                if (((FamilySelector) current).getName().equals(connectionTarget.
                    getTarget().getName())) {
                        return true;
                }
                else {
                        EList<FamilyConnection> listOfConnection = ((FamilySelector)
                            current).getConnections();
                        for (FamilyConnection con : listOfConnection) {
                                boolean res = traverseTree(fxman, con.getTarget(),
                                    connectionTarget);
                                if (res) return true;
                        }
                }

        }
        else if (current instanceof FamilyAggregator) {
                if (((FamilyAggregator) current).getName().equals(connectionTarget.
                    getTarget().getName())) {
                        return true;
                }
                else {
                        EList<FamilyConnection> listOfConnection = ((FamilyAggregator)
                            current).getConnections();
                        for (FamilyConnection con : listOfConnection) {
                                boolean res = traverseTree(fxman, con.getTarget(),
                                    connectionTarget);
                                if (res) return true;
                        }
                }

        }
        else if (current instanceof FamilyLoop) {
                if (((FamilyLoop) current).getName().equals(connectionTarget.getTarget
                    ().getName())) {
                        return true;
                }
                else {
                        FamilyConnection connection = ((FamilyLoop) current).
                            getConnection();
                        boolean res = traverseTree(fxman, connection.getTarget(),
                            connectionTarget);
                        if (res) return true;
                }

        }
        else if (current instanceof FamilyGuard) {
                if (((FamilyGuard) current).getName().equals(connectionTarget.getTarget
                    ().getName())) {
                        return true;
                }
                else {
                        FamilyConnection connection = ((FamilyGuard) current).
                            getConnection();
                        boolean res = traverseTree(fxman, connection.getTarget(),
                            connectionTarget);
                        if (res) return true;
                }

        }
        else if (current instanceof Optional) {
                if (((Optional) current).getFamilyMember() instanceof XMANArchitecture)
                     {
                        XMANArchitecture optTarget = (XMANArchitecture) ((Optional)
                            current).getFamilyMember();
                        if (optTarget.equals(connectionTarget.getTarget())) { return
                            true; }
                }
        }
        else if (current instanceof Alternative) {
                if (((Alternative) current).getFamilyMembers().contains(
                    connectionTarget.getTarget())) return true;
        }
        else if (current instanceof Or) {
                boolean orChecker = true;
                boolean aggChecker = false;
                for (VariationPointTerminal vpt : ((Or) current).getFamilyMembers()) {
                        if (vpt instanceof XMANArchitecture) {
                                XMANArchitecture orTarget = (XMANArchitecture) vpt;
                                if (orTarget.equals(connectionTarget.getTarget())) {
                                        return true;
                                }
                                else if (connectionTarget.getTarget() instanceof
                                    Aggregator) {
                                        Aggregator agg = (Aggregator) connectionTarget.
                                            getTarget();
                                        List<Connection> aggCon = agg.getConnections();
                                        aggChecker = false;
                                        for (Connection ac : aggCon) {
                                                if (!((CoordinationConnection) ac).
                                                    getTarget().equals(vpt)) {
                                                        aggChecker = true;
                                                }
                                        }
                                        if (aggChecker == false) orChecker = false;
                                }
                        }
                }
                if (orChecker == true) return true;
        }

        else if (current instanceof XMANArchitecture) {
                if (((XMANArchitecture) current).equals(connectionTarget.getTarget()))
                    { return true; }
        }
        else if (current instanceof FXMANArchitectureInstance) {
                if (((FXMANArchitectureInstance) current).equals(connectionTarget.
                    getTarget())) { return true; }
        }
        return false;
        }
        }
```

Listing F.1: Xcore implementation of the activity-chart meta-model

Appendix G

Family Filter Implementation

```java
private static boolean requiresMatcher(String source, String subItem1, String
    subItem2) {
      if (source.matches(".*\\b" + subItem1 + "\\b.*") && source.matches(".*\\b" +
          subItem2 + "\\b.*"))
      return true;
      else if (!source.matches(".*\\b" + subItem1 + "\\b.*")) return true;

      return false;

}

private static boolean excludesMatcher(String source, String subItem1, String
    subItem2) {
      if (source.matches(".*\\b" + subItem1 + "\\b.*") && !source.matches(".*\\b" +
          subItem2 + "\\b.*")) return true;
      if (!source.matches(".*\\b" + subItem1 + "\\b.*")) return true;

      return false;
}

private static boolean filterTranslator(String source, EList<Constraint>
    fullConstraint) {
      boolean result = true;
      for (Constraint constraint : fullConstraint) {
            if (constraint.getConstraint().equals(ConstraintType.REQUIRES)) {
                  result &= requiresMatcher(source.toString(), constraint.
                      getFirstOperand(), constraint.getSecondOperand());
            }
            else {
                  result &= excludesMatcher(source, constraint.getFirstOperand(),
                      constraint.getSecondOperand());
            }
        }
        return result;
}

public static ProductFamily filterProducts(ProductFamily productFamily, EList<
    Constraint> fullConstraint) {
        for (Iterator<Product> iter = productFamily.getProducts().iterator(); iter.
            hasNext();) {
                Product element = iter.next();
                if (!filterTranslator(element.toString(), fullConstraint)) iter.remove
                    ();
        }
        return productFamily;
}
```

Listing G.1: Implementation detail of the family filter.

Appendix H

Pure-variants Research License Agreement

pure·systems

pure::variants Research License Terms And Conditions Agreement

between

pure-systems GmbH
Otto-von-Guericke-Str. 28
39104 Magdeburg
Germany
Phone: +49 391 5445690
Fax: +49 391 54456990
Email: sales@pure-systems.com
(pure-systems)

and

Customer:	The University of Manchester
Customer:	
Street:	Oxford Road
ZIP and City:	Manchester
Country:	England
Phone:	0161 275 8261
Fax:	
Email:	luke.a.kiernan@manchester.ac.uk
	~~(Licensee)~~

1. License Usage Restrictions

Research Licenses provided to the Licensee under the conditions of this agreement may be used for research and educational purposes in educational and academic organisations. Development of OpenSource applications by these organisations using pure::variants are treated as research activities.

2. License Transfer

The Licensee must not give other persons or organisations access to the Licenses without written approval by pure-systems. Persons in offical academic courses which are held by the Licensee are entitled to use pure::variants for the duration of the course.

3. License Renewal Requirements

pure-systems requires Licensee to provide a short report about the use of pure::variants in the organization and may publicize all or parts of this report on its web site or other media. All references to the report will mention the Licensee as source. Reports are requested at most twice in a 12 months period.

4. Termination

If the Licensee does not follow rules set out in this agreement, pure-systems may terminate the license agreement immediately.

This agreement is valid for 12 month and may be renewed by request of the Licensee. If it is not renewed, it terminates without further notices. There is no obligation of pure-systems to renew the license even if the Licensee fulfills the requirements defined in 3.

5. Other Terms

In addition to the terms set out above, the terms of the normal End User License agreement provided with pure::variants apply, unless they are in direct conflict with the terms set out above.

For pure-systems

For Licensee

[illegible]

(date, stamp and signature)

(date, stamp and signature)

Lisa Murphy
Solicitor
Head of Contracts
17 December 2015

Notice to User: This End User License Agreement (EULA) is a CONTRACT between you (either an individual or a single entity) and pure-systems GmbH ("pure-systems") which covers your use of

pure::variants

and related software components, which may include associated media, printed materials, and "online" or electronic documentation. All such software and materials are referred to here as the "Software Product." If you do not agree to the terms of this EULA, then do not install or use the Software Product. By explicitly accepting this EULA, however, or by installing, copying, downloading, accessing or otherwise using the Software Product, you are acknowledging and agreeing to be bound by the following terms:

1. **WARNING -- (for Evaluation Licenses only)**

   This Software Product can be used in conjunction with a free evaluation license. If you are using such an evaluation license, you may not use this Software Product in a live operating environment. Evaluation licenses have an expiration date, after which you understand and agree that you must stop using the Software Product. pure-systems bears no liability for any damages resulting from use or attempted use of the Software Product, and no duty to provide any support, after the expiration date.

2. **Grant of Non-Exclusive License**

   pure-systems grants you the right to install and use a single copy of the Software Product only on your computer. You may make copies of the Software Product as needed for backup and archival purposes.

3. **Intellectual Property Rights Reserved by pure-systems**

   The Software Product is protected by german and international copyright laws and treaties, as well as other intellectual property laws and treaties. You must not remove or alter any copyright notices on any copies of the Software Product. This Software Product is licensed, not sold. Furthermore, this EULA does not grant you any rights in connection with any trademarks or service marks of pure-systems. pure-systems reserves all intellectual property rights, including copyrights and trademark rights.

4. **No Right To Transfer**

   You may not rent, lease, lend or in any way distribute copies of or transfer any rights in the Software Product to third parties without pure-systems's written approval and subject to approval by the recipient of the terms of this EULA.

5. **Prohibition on Reverse Engineering, Decompilation, and Disassembly**

   You may not reverse engineer, decompile, or disassemble the Software Product, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.

6. **Third Party Rights**

   Any software provided along with the Software Product that is associated with a separate license agreement is licensed to you under the terms of that license agreement.

7. **Support Services**

   pure-systems may provide you with support services related to the Software Product. Use of any such support services is governed by the pure-systems polices and programs described in "on line" documentation and/or other pure-systems-provided materials. Any supplemental software code that pure-systems provides to you as part of the support services is to be considered part of the Software Product and is subject to the terms and conditions of this EULA. With respect to any technical information you provide to pure-systems as part of the support services, pure-systems may use such information for its business purposes, including for product support and development. pure-systems will not use such technical information in a form that personally identifies you.

8. **Compliance with Applicable Laws**

   You must comply with all applicable laws regarding use of the Software Product.

9. **Termination**

   Without prejudice to any other rights, pure-systems may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the Software Product.

10. **NO WARRANTIES**

    YOU ACCEPT THE SOFTWARE PRODUCT "AS IS" AND PURE-SYSTEMS MAKES NO WARRANTY AS TO ITS USE OR PERFORMANCE. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, PURE-SYSTEMS DISCLAIMS ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, WITH REGARD TO THE SOFTWARE PRODUCT, AND THE info@pure-systems.com.

PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE PRODUCT REMAINS WITH YOU.

11. **LIMITATION OF LIABILITY**

THE LIMITATION OF LIABILITY IS TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL PURE-SYSTEMS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF PURE-SYSTEMS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, PURE-SYSTEMS'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS EULA SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE PRODUCT OR U.S.$5.00; PROVIDED, HOWEVER, THAT IF YOU HAVE ENTERED INTO A PURE-SYSTEMS SUPPORT SERVICES AGREEMENT, PURE-SYSTEMS'S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT. BECAUSE SOME STATES/JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

12. **Governing Law**

This Agreement constitutes the complete agreement between the parties with respect to the Programs and is governed by the laws of the Federal Republic of Germany (other than its conflict of law provisions).

13. **Contact Info**

If you have any questions about this EULA, or if you want to contact pure-systems for any reason, please contact pure-systems:

pure-systems GmbH
Agnetenstr. 14
39106 Magdeburg
Germany

or email

Appendix I

FX-MAN Tool on the UMIP platform Click2Go

X-MAN II

Component-based Software Development Toolkit

PROVIDING SCIENTIST/AUTHOR(s)

Dr Cuong Tran
Dr Simone Di Cola

OPTIONS

Please check carefully that the terms you select correspond to your intended use of the product.

MAN_002-3494548-v2-UMIP Annual Research Licence C2G final 1.00

Annual Research Licence

View Terms

Further details
Term: 12 months
Seats: 1 seat(s)

Price excl. VAT: Free of charge

ADD TO BASKET

DESCRIPTION FILES (1) REFERENCES (0) SUPPORT

INTRODUCTION

X-MAN II toolset is developed by the Component-based Software Development group at the University of Manchester.

X-MAN II toolset consists of the X-MAN IDE and its extension FX-MAN. The X-MAN IDE provides a set of functionalities for component-based software development. It supports the W process which captures both component and system life cycles. The extension FX-MAN provides the capability of modelling variability in order to support constructing product families.

X-MAN II toolset is developed using Eclipse model-driven engineering. Essentially, it is EMF framework instantiated with meta-models of X-MAN component model and its extensions. Included plugins provide complete semantics of the X-MAN component model and component-based software development paradigm.

X-MAN II tool development is supported by the European funded EMC2 project and the Centre for Doctoral Training (CDT) programme at the University of Manchester.

This new tool will replace the old X-MAN tool (based on GME)here.

FEATURES

The W model for component-based software development should be one in the below figure. The model was defined in our publication back in 2011.

Essentially, the W model comprises of two life cycles:

- Component life cycle: identify components, construct components and build (domain specific) component repository.
- System life cycle: select components, instantiate (and adapt) components and construct systems.