# Feature Oriented Model Driven Product Lines

Salvador Trujillo Gonzalez

**Dissertation**
presented to
the Department of Computer Sciences of
the University of the Basque Country
in Partial Fulfillment of
the Requirements
for the Degree of

**Doctor of Philosophy**
("*doctor europeus*" mention)

**The University of the Basque Country**
Universidad del País Vasco / Euskal Herriko Unibertsitatea

San Sebastián, Spain, March 2007

# Summary

*Model Driven Development* (MDD) is an emerging paradigm for software construction that uses models to specify programs, and model transformations to synthesize executables. *Feature Oriented Programming* (FOP) is a paradigm for software product lines where programs are synthesized by composing features. *Feature Oriented Model Driven Development* (FOMDD) is a blend of FOP and MDD that shows how programs in a software product line can be synthesized in an MDD way by composing models from features, and then transforming these models into executables. A case study on a *product line of portlets*, which are components of web portals, is used to illustrate FOMDD. This case reveals mathematical properties (i.e., commuting diagrams) of portlet synthesis that helped us to validate the correctness of our approach (abstractions, tools and specifications), as well as optimize portlet synthesis. These properties expose the nature of metaprograms for program synthesis (i.e., synthesis is a metaprogram combining primitive operations that form a geometry). We exploit this to synthesize metaprograms, which when executed, will synthesize a target program of a product-line. Specifically, we elaborate on the *generation of metaprograms* (not programs) from abstract specifications. This is the core of GROVE: the *GeneRative metaprOgramming for Variable structurE* approach. Variability of synthesis is also considered. Nonetheless, the ultimate envision is a structural theory behind program synthesis.

# Contents

# Chapter 1

# Introduction

"A journey of a thousand miles begins with a single step".
– *Confucius.*

## 1.1 Overview

*Software artifacts* are arguably among the most complex products of human intellect. This complexity of software has driven both researchers and practitioners towards a number of general engineering challenges. So far, those efforts focused on the engineering of individual products.

As industrialization of the manufacturing processes (e.g., automobile) led to increased productivity and higher quality at lower costs, industrialization of software development process leads to the same advantages. The focus shifts from a stand-alone product to a family of similar products (a.k.a. mass customization). This body of research is known as *Software Product Line*s (SPL). In this scenario, a number of engineering challenges are being faced [CN01].

A key challenge is how prebuilt pieces are assembled together to synthesize a program product. Doing so, it is possible to synthesize automatically a customized program from customer *features* (i.e., increments of program functionality). That is *Feature Oriented Programming* [BSR04]. However, the focus so far was on how the feature realization code was assembled (i.e., composition of implementation artifacts), but not on how that code was obtained. There was no specific support to

obtain such code.

This dissertation shows a way to obtain feature realization code by raising the level of abstraction. To this end, the general *model-driven* paradigm is adjusted in the context of Feature Oriented Programming. Doing so, a feature realization scales from a set of implementation artifacts to a set of models. The variability realization techniques deal not only with implementation code, but directly with models.

Software engineering techniques are used not only in the process of assembling feature implementations, but in the process of modeling them (enabling an acceleration in the creation of feature implementations). Hence, it is not only possible to *assemble the artifacts* derived from models, but also to *assemble the models* themselves, and derive then code artifacts. The latter turns up as an unexpected way to synthesize products (a.k.a. synthesis paths).

This setting enables the traversal of two different paths to synthesize a product of a product line. This setting is where *commuting diagrams* appear in synthesis (i.e., the same implementation can be synthesized in multiple ways). These properties help us to validate the correctness of the abstractions, tools, and specifications, as well as optimize synthesis.

We implement by scripting synthesis metaprograms (i.e., a program that synthesizes a target program). This implies time-consuming, repetitive and cumbersome tasks. Our goal is to accelerate the development of metaprograms by generating them from abstract specifications (i.e., MDD is used to generate metaprograms). This work describes a way to synthesize metaprograms (not programs), which when executed, will synthesize a target program of a product-line. Specifically, we elaborate on the generation of metaprograms from abstract specifications. Variability of synthesis is also considered. The ultimate frontier is a structural theory for product synthesis. These and other insights are described in this work.

The work presented in this thesis poses an intellectual challenge combining Software Product Lines and Model Driven Development. It represents an advancement over previous techniques and succeeded in its application to the also challenging Portlet engineering domain.

## 1.2 Contributions

This thesis has been developed in the context of engineering. This pushes us to achieve not only an academic contribution but also to look at the broader applica-

bility of these ideas. This introduces a remarkable challenge throughout the thesis: the need to realize (implement) the ideas we describe. In our opinion, this thesis provides the following contributions:

**A PRODUCT LINE OF PORTLETS**

*Problem Statement*: Portlets need to be reusable services to accomplish the *Service Oriented Architecture* (SOA) vision.

*Contribution*: This work introduces a reusability approach based on SPL. It is one of the test cases of this work. Overall, building a product line of Portlets exposes many research issues.

**A MODEL DRIVEN APPROACH TO PORTLET DEVELOPMENT**

*Problem Statement*: Portlet development encompasses the implementation of repetitive and cumbersome code.

*Contribution*: We propose an approach to the generation of Portlets driven by model specifications. Doing so, starting from abstract models, repetitive and cumbersome implementation can be generated.

**FEATURE ORIENTED MODEL DRIVEN DEVELOPMENT**

*Problem Statement*: Model Driven Development promotes models to play a pivotal role in software development. However, product-line techniques (e.g., Feature Oriented Programming) focus on code artifacts.

*Contribution*: *FOMDD* combines *Feature Oriented* programming and *Model Driven Development* together where a new synthesis design space appears. This space shows interesting *commuting* properties that allow *(i)* to check the validity of the approach, and *(ii)* to optimize program synthesis. Our working case for Portlets is used to illustrate these findings.

**GENERATIVE METAPROGRAMMING FOR SYNTHESIS PROCESS**

*Problem Statement*: We implement synthesis metaprograms by scripting. This implies time-consuming, repetitive and cumbersome tasks. This work reveals also the nature of program synthesis in FOMDD. Commuting was symptomatic of some structure (and theory) behind our work.

*Contribution*: This work pioneers study on the primitives that form synthesis metaprograms. This is then exploited to synthesize metaprograms, which when executed, will synthesize a target program of a product-line. We generate the implementation code of the metaprograms (not programs) from abstract specifications. This is the core of the *GeneRative metaprOgramming for Variable structurE* approach (GROVE).

**VARIABILITY ON THE SYNTHESIS PROCESS**

*Problem Statement*: Synthesis process is a key factor in product-lines. The issue is how to accommodate synthesis to different production strategies in a product-line setting.

*Contribution*: A way to introduce variability into the synthesis process, and how to cope with such variability issues is described. More to the point, program features are separated from process features (i.e., it is possible to reuse the same synthesis process across different product-lines).

**FEATURE REFACTORING MULTIPLE-REPRESENTATIONS**

*Problem Statement*: The challenge is to mine a large-scale legacy program into a product-line family. This scale is almost 2 orders of magnitude larger than previous work. Additionally, this work not only refactors code, but multiple representations such as XML documents.

*Contribution*: It describes a large case study on feature refactoring where multiple and heterogeneous representations were refactored. As a result, a new product-line was refactored from an original program where new extensions are added in terms of features.

## 1.3 Outline

Our work starts by addressing web applications customizability. Shortly after, the well-known concept of variability arises. This leads us to *Software Product Lines*.

Web applications were steadily moving towards the realm of Portals (i.e., a website as an entry point for others). It was the birth of Portals and *Portlets* (i.e., basic building blocks of Portals and units for reuse). An SPL was then created to produce Portlet variants. Beyond the economical motivation inherent to SPL was the need to produce the same Portlet for different customer Portals. Portlet variants are needed to deal with those different requirements that Portal customers demand.

Our intention was to fully create a factory from which Portlet applications were later automatically produced. In so doing, the customer specifies the desired requirements within the available choices the SPL offers, and then produces the customized application. This approach provides a number of general benefits (e.g., time and cost reductions). However, these benefits always depend on the domain at hand.

Portlets were the inspiration to find a number of open issues. Being service-oriented, they provide new SPL issues. Overall, the Portlet domain was the setting

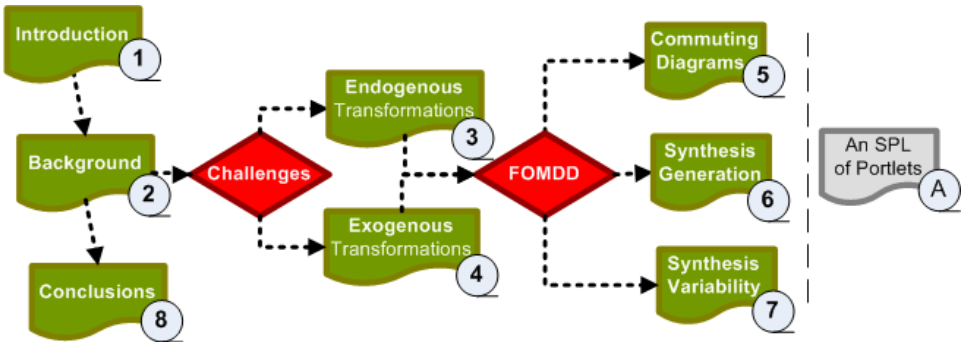



**Figure 1.1**: Map of Chapters

where *Feature Oriented Programming* and *Model Driven Development* were combined together to yield FOMDD. Their combination is the core of this thesis.

Figure 1.1 shows a chapter map with the major ideas covered by this thesis. It consists of 8 chapters (including this) and one appendix.

**Chapter 2** introduces the background (Software Product Lines, Model Driven Development and Portlets) on top of which the remaining chapters are developed.

**Chapter 3** discusses the endogenous transformations SPL need to realize feature variability. *AHEAD* is selected to deal with such transformations. We extended AHEAD Tool Suite (ATS) with further functionality (e.g., XAK is described as a solution for XML Refinement). A case study of feature refactoring a large program is then described. The target program is ATS which was refactored (from a single program) onto a product line.

**Chapter 4** discusses the exogenous transformations SPL need to abstract implementation using models. An example is shown to illustrate a model-driven approach to Portlet development.

**Chapter 5** combines Software Product Lines and Model Driven Development. *Feature Oriented Model Driven Development* arises as a blend of both. The new challenges to realize model increments are presented: model refinements and transformations between model refinements. Product synthesis exposes commuting diagrams, which suggest a structural theory behind synthesis process.

**Chapter 6** describes ideas to synthesize metaprograms, which when executed, will synthesize a target program of a product-line. Specifically, we elaborated on the generation of metaprograms from abstract specifications.

**Chapter 7** describes how to apply variability to the synthesis process. To attain this, it separates features that impact on the program and features that impact on how the program is built (i.e., the synthesis itself has features that change how it is done).

**Chapter 8** exposes the main contributions, results and publications. The entire work is evaluated with the benefit of hindsight. Here limitations of the work are discussed and future research directions are suggested.

**Appendix A** looks at Portlets as a family of products within an SPL (i.e., Portlets are synthesized from a product-line). Our test case is used to illustrate a general product-line approach. We explore as well the key issues that make Portlets worth from a product-line perspective, and which challenges should be faced to cater for product lines of Portlets.

# Chapter 2

# Background

"A child of five would understand this. Send someone to fetch a child of five."

*– Groucho Marx.*

## 2.1 Abstract

*Software Product Lines* (SPL) offer a paradigm to develop a family of software products. The focus shifts from the development of an individual program to the development of reusable assets that are used to develop a family of programs. Distinct approaches, methodologies and tools are proposed to realize SPL in a cost-effective way.

*Model Driven Development* (MDD) is a paradigm to develop programs based on modeling. Software models are specified, from which other models or even code are derived. This paradigm eases cumbersome and repetitive tasks, and achieves productivity gains.

*Portlets* are basic building blocks for portal construction. Their main benefit is that can be used in a variety of scenarios and by distinct portal customers (and end-users).

This chapter provides a *quick glance* at Software Product Lines, Model Driven Development, and Portlet (and Web) Engineering. Also, existing works bridging these fields are introduced. We invite the reader to skip this chapter if familiar with those concepts.

## 2.2 Software Product Lines

*Mass production* was popularized by Henry Ford in the early 20th Century. McIlroy coined the term *software* mass production in 1968 [McI68]. It was the beginning of *Software Product Lines*. In 1976, Parnas introduced the notion of software *program families* as a result of mass production [Par76]. The use of *features* (to drive mass production) was proposed by Kang in the early 1990s [Kea90]. Shortly, the first conferences appeared turning SPL into a new body of research [vdL02, Don00].

### 2.2.1 Definition

SPL are defined as "*a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [CN01].

We redefine SPL into five major issues:

1. **Products**: "*a set of software-intensive systems...*". SPL shift the focus from single software system development to SPL development. The development processes are not intended to build one application, but a number of them (e.g., 10, 100, 10,000, or more). This forces a change in the engineering processes where a distinction between domain engineering and application engineering is introduced. Doing so, the construction of the reusable assets (a.k.a., platform) and their variability is separated from production of the product-line applications[1].

2. **Features**: "*...sharing a common, managed set of features...*". Features are units (i.e., increments in application functionality) by which different products can be distinguished and defined within an SPL [BSR04][2].

3. **Domain**: "*...that satisfy the specific needs of a particular market segment or mission...*". An SPL is created within the scope of a domain. A domain is "*a specialized body of knowledge, an area of expertise, or a collection of related functionality*" [Nor02].

[1] Terminology: The terms system, application, variant, program and product are used interchangeably to refer to the outcome of a product-line.

[2] Other definitions of features include "*user-visible aspects or characteristics of the domain*" [Kea90], "*a logical unit of behavior that is specified by a set of functional and quality requirements*" [Bos00] or "*a recognizable characteristic of a system relevant to any stakeholder*" [CE00].

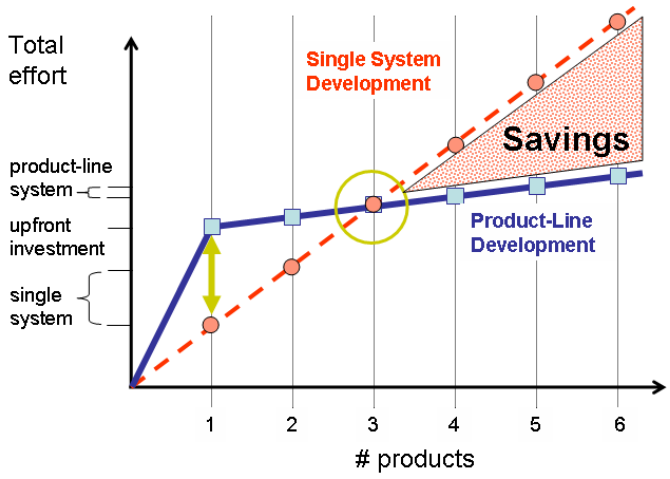



**Figure 2.1**: Software Product Lines versus Single Software Development

4. **Core Assets**: "*...are developed from a common set of core assets...*". A core asset is "*an artifact or resource that is used in the production of more than one product in a software product line*" [CN01].

5. **Production Plan**: "*...in a prescribed way*". It states how each product is produced. The production plan is "*a description of how core assets are to be used to develop a product in a product line and specifies how to use the production plan to build the end product*" [CM02]. The production plan ties together all the reusable assets to assemble (and build) end products. Synthesis is a part of the production plan.

### 2.2.2 Motivation

In general, SPL foster reuse in software development. Behind reuse, there are economic reasons.

SPL provide a number of general and potential benefits, namely, *(i)* productivity gains, *(ii)* improved product quality, *(iii)* faster time-to-market, and *(iv)* decreased labor needs [CN01, Coh01]. However, as in the real world, it depends on the specific case at hand.

An SPL approach is useful in many domains, but not for all domains. It depends not only on the number of products to be developed out of the SPL, but on the economic viability of the SPL. Hence, economic models are needed to study each case beforehand.

Most of the economic arguments are based on singular data points derived from case studies or convincing arguments based on reasonableness and simplistic cost curves. For instance, a general assumption is that a product line should consist at least of 3 products [BCM+04]. Figure 2.1 sketches this scenario where a graphic comparing single system versus software product line is presented [Beu03, Mut02]. Typically, an SPL has a higher (initial) upfront investment, but it makes the difference in the long-term compared to single software where profitability appears.

Existing models of conventional development costs in the context of reuse can only be applied in a restricted way as product line development involves some fundamental assumptions that are not reflected in these models.

Bockle et al. proposed an initial model for SPL economics [BCM+04]. Later, a comprehensive cost model was proposed. It was called the *Structured Intuitive Model for Product Line Economics* (SIMPLE[3]). This model enables to calculate the costs and benefits, and hence the return on investment (ROI), that we can expect to accrue from various product line development situations [CMC05]. Recently, some experience on industrial cases to predict the ROI beforehand was reported [GMY06].

### 2.2.3 Successful Case Studies

Several successful stories have been reported so far on organizations following SPL approaches. Next, a list of successful stories are presented.

Cummins Inc. was able to field more than 1.000 separate products based on just 20 software builds. They can build and integrate the software for a new diesel engine in about a week, whereas before, it took a year [CN01].

The U.S. National Reconnaissance Office commissioned a software product line of satellite ground control systems from Raytheon and reported a 10x quality improvement and a 7x productivity improvement as a result [CN01].

CelsiusTech Systems was able to decrease their software staff from 210 to around 30, while turning out more, larger, and more complex ship command and control system products. The product line approach let them change the hardware-to-software ratio for their systems from 35:65 to 80:20 [CN01].

Previous stories were on large organizations. Nonetheless, there are also reported cases for small organizations like Market Maker Software AG [KMSW00], or for embedded systems [Beu03]. Some stories on SPL for web applications are

[3] http://simple.sei.cmu.edu/

described shortly in section 2.5.2.

### 2.2.4 Software Product Line Engineering

In general, facing an SPL implies to distinguish between two separate processes, namely, the domain engineering process, and the application engineering process.

**Domain Engineering** is defined as "*the activity of collecting, organizing and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (e.g., architecture, models, code, and so on), as well as providing an adequate means for reusing these assets [...] when building new systems*" [CE00].

Using a "*design-for-reuse*" approach, domain engineering (a.k.a., core asset development [CN01]) is in charge of determining the commonality and the variability among product family members. In general, domain engineering is divided into domain analysis, domain design and domain implementation. However, this simple division hides a number of practices and activities. Refer to [CN01, PBvdL06] for a complete account.

**Application Engineering** is "*the process of building a particular system in the domain*" [CE00]. Application engineering (a.k.a., product Development [CN01]) is responsible for deriving a concrete product from the SPL using a "*design-with-reuse*" approach. To attain this, it reuses the reusable assets developed previously. This process is subdivided into application analysis, application design and application implementation (some other activities are omitted as well [CN01]).

Some authors introduce a separated process for **Management** where organizational issues are handled specifically [CN01].

For a broader review, we invite the reader to check comprehensive literature on software product line engineering [CN01, Gom04, PBvdL06].

### 2.2.5 Strategies

Clements and Krueger classify existing SPL strategies into *extractive*, *reactive* or *proactive* [CK02].

The **extractive** approach reuses one or more existing software products for the product line initial baseline. To be an effective choice, the extractive approach requires lightweight software product line technology and techniques that can reuse existing software without much re-engineering. This approach is very effective for an organization that wants to quickly transition from conventional to software

product line engineering, and brings the opportunity to start an SPL reusing existing code.

The **reactive** approach is like the spiral or extreme programming approach to conventional software. You analyze, architect, design, and implement one or several product variations on each development spiral. This approach works in situations where it is difficult to predict the requirements for product variations well in advance or where organizations must maintain aggressive production schedules with few additional resources during the transition to an SPL approach.

The **proactive** approach is like the waterfall approach to conventional software where all product variations on the foreseeable horizon are analyzed, architected, designed, and implemented upfront. This approach might suit organizations that can predict their SPL requirements well into the future and that have the time and resources for a waterfall development cycle. This approach is aimed to quickly manufacture products reducing production cost on a large scale production basis.

### 2.2.6 Existing Approaches

The literature shows a number of different approaches (following different strategies) to face software product lines. The aim of this section is to *briefly* introduce them. To this end, existing approaches are classified into 3 major concerns using an early classification proposed by Kang [Kea90]:

The **process** concern considers how the methodology will affect an organization (e.g., evolutionary vs. revolutionary); how to manage and maintain the products; how the producer gathers, organizes, validates, and maintains information; and how the users can effectively apply the products in the development. Several approaches fit in this area: *Feature Oriented Domain Analysis* (FODA) [Kea90], *Feature Oriented Reuse Method* (FORM) [KKL+98], FeatuRSEB [GFd98], *Family-oriented Abstraction Specification and Translation* (FAST) [WL99], *ProdUct Line Software Engineering* (PuLSE) [Bea99], KobrA [ABM00], SEI's *Product Line Practice Initiative* (PLPI) [CN01, SEI], *Feature Oriented Product Line Software Engineering* (FOPLE) [KLD02], *Quality-driven Architecture Design and quality Analysis* (QADA) [MND02, Mat04], *Evolutionary Software Product Line Engineering Process* (ESPLEP) [Gom04], *Product Line Use case modeling for Systems and Software engineering* (PLUSS) [EBB05], and PRIME [PBvdL06]. A complete survey with further approaches, practices and patterns involved in SPL are described at [CN01, CE00, PBvdL06].

The **product** concern considers the types of products that are generated by

the approach; how they are represented; and how applicable they are in application development. Regarding product-line architecture [Batb], it should be mentioned architecture-centered methodology [Bos00] and *Architecture Description Languages* (ADL) like KOALA [vOvdLKM00] or xADL [DvdHT05]. Regarding product synthesis or manufacturing, it should be stated AHEAD [BSR04], Generative Programming [CE00], CONSUL [BPSP04], Frames Programming [JBZZ03], and Feature Oriented Programming [BSR04, Pre97]. Implementation wise, a set of variability realization techniques are available [AG01].

The **tool** support concern considers the availability of tools and the extent to which the tools support the approach. It also looks at how well the tools are integrated, their ease of use, and their robustness. Some feature modeling tools are *GuiDSL* [Bat05], *FeaturePlugin* [CA05a] and *XFeature* [CPRS04]. Other tools support a complete approach, namely, *AHEAD Tool Suite* [Bata], *GEARS* [Kru02], CONSUL is supported by *pure::variants* [Sys], and XVCL supports *Customization Scripts* (frames) [JBZZ03].

### 2.2.7 Current Research Issues

The history of SPL development shows that it is a relatively new field in the broader area of Software Engineering. Though it is maturing, there are of course many opportunities for research. Next, some general trends in current research are discussed.

Early works on software product lines focused more on the practical cases that on how that cases could be generalized. Many of the works were pioneer in their respective areas [CN01]. The next challenge is to spread this strategy into regular companies (as it is today in other industrial production scenarios) [GMY06]. To attain this, further work on economic models, on product line quality, and on product line initiation activities is needed.

Another interesting field of research is that of extractive approaches to *refactor* existing legacy programs into a product line [LBL06, TBD06]. The interest of this work rests on the great amount of existing legacy software.

Traditional software is moving towards scenarios where functionality is provided by services. A number of challenges appear in service-oriented architectures, namely, how different products are produced (from different product lines) solving customization/privacy implementation [WKvdHW06], how variability is achieved in the production [DTA05], how distributed production is faced [Tea07], how availability is resolved, how a service is reconfigured [LK06]. Portlet development is

a service-oriented scenario. A small subset of the challenges presented previously were faced by this work.

SPL are not developed from scratch (proposing new methodologies), but a lot of *legacy* software engineering methodologies could be *reused* somehow. Model-driven paradigm is one of these methodologies that could be reused into SPL.

## 2.3 Model Driven Development

*Model Driven Development* (MDD) is a relatively new paradigm where models are central in the development. *Model Driven Architecture* (MDA) is a framework for software development proposed by the *Object Management Group* (OMG) in 2001 [OMG03] (i.e., MDA is a concrete realization of MDD). The notion of *Model Driven Engineering* (MDE) emerged later as a paradigm generalizing the MDA approach for software development [Ken02].

### 2.3.1 Definition

*Model-driven* is a paradigm where models are used to develop software. This process is driven by model specifications and by transformations among models. It is the ability to transform among different model representations that differentiates the use of models for sketching out a design from a more extensive model-driven software engineering process where models yield implementation artifacts.

*Model Driven Architecture* provides specific means for using models to accomplish the understanding, design, construction, deployment, maintenance and modification of software.

MDA's modeling techniques distinguish between business and technical aspects. This advocates that the designer must first capture the business concerns of the system in a model, called the *Platform-Independent Model* (PIM), while abstracting away technical details. Then, the PIM is transformed into a *Platform-Specific Model* (PSM) by introducing technical aspects of the target platform (in an MDA context). In general, a key challenge is the transformation of these models. This transformation is usually specified by a set of precise mapping rules (more shortly). Finally, the resulting PSM can be used to generate implementation code.

### 2.3.2 Motivation

In general, model-driven is a paradigm to reuse specific patterns or domains of software development. This emerges through the extensive use of models, which replaces cumbersome (and usually repetitive) implementation activities. In this way, model-driven approaches improve development practices by accelerating them.

According to [Kon], specific benefits of MDD are *(i)* productivity, *(ii)* reduced cost, *(iii)* portability, *(iv)* reduced development time, and *(v)* improved quality. Overall, the *main* economic reason behind model-driven is the *productivity* gain achieved, which is reported by some studies [HR03, OMG].

### 2.3.3 Successful Case Studies

OMG reports a number of success stories on the applications of MDD in well-known companies (e.g., ABB, Daimler Chrysler, Lockheed Martin) [OMG].

In general, these studies report an increment in productivity. Hence, time to market was shortened on these projects where MDD was used (although this does not imply to hold for all projects). Specifically, some particular experiences are reported: *(i)* fast and high ROI, *(ii)* quality improvements, *(iii)* effort reduction, *(iv)* reduce of development staff, and *(v)* reuse across multiple platforms.

### 2.3.4 Model Driven Engineering

Kent defines *Model Driven Engineering* (MDE) by extending MDA with the notion of software development process (i.e., MDE emerged later as a generalization of the MDA for software development) [Ken02]. MDE refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle[4]. Kurtev provides a discussion on existing MDE processes [Kur05] (refer to [BFJ+03, Béz04] for a specific approach). In general, these approaches introduce concepts, methods and tools [Sch06]. All of them are based on the concept of model, meta-model, and model transformation.

For a broader review, we invite the reader to check further literature on MDE [BBG05, Fra03, KWB03].

[4] http://en.wikipedia.org/wiki/Model_Driven_Engineering

### 2.3.5 Model Driven Architecture

*Model Driven Architecture* (MDA) is a concrete realization of MDD. As mentioned above, MDA classifies models into 2 classes: *Platform Independent Models* (PIMs) and *Platform Specific Models* (PSMs) [OMG03]. A PIM is "*a view of a system from a platform-independent viewpoint*". Likewise, a PSM is "*a view of a system from a platform-dependent viewpoint*" [OMG03]. Doing so, the definition of platform becomes fundamental.

This classification depends on the dependence from a given **platform**, which is defined as "*a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented*" [OMG03].

MDA integrates the existing technologies standardized by OMG. It defines a set of basic concepts such as model and modeling language, meta-model, and transformations [JBR98, RGJ04, OMG05b]. Models are expressed in *Unified Modeling Language* (UML) and UML profiles [OMG05b]. New modeling languages should be defined using the meta-modeling language of the *Meta Object Facility* (MOF) [OMG06]. Models are serialized to XML format using the *XML Metadata Interchange* (XMI) [OMG05c]. The transformations of models are described using *Queries/Views/Transformations* language (QVT) [OMG05a].

### 2.3.6 Existing Approaches

A number of tools are intended to support a model-driven approach (often based on MDA). Typically, they enable to define models in terms of UML, and to transform models using transformations. The following is a list of some existing approaches:

- IBM Rationale Software design is a well-known set of tools [IBM].
- Vanderbilt's Model Integrated Computing (MIC) [SK97].
- Eclipse Modeling Framework (EMF) [Ecl].
- ATL (ATLAS Transformation Language) framework [BDJ+03].
- Microsoft's Software Factories [GS04].
- MOMENT is a model management framework [BCR05, BCR06].
- RubyTL is a tool for transformations [CMT06].

Please, refer to [Mod] for a comprehensive list.

### 2.3.7 Current Research Issues

Several research directions are being explored nowadays in MDD. Selic (in his keynote at [RW06]) proposed these specific topics: *(i)* theory of modeling language design, *(ii)* theory of model transformations, *(iii)* model analysis, *(iv)* model synthesis, and *(v)* tools for automation support. He also detected some challenges and opportunities on automated model analysis, formal validation, and automatic code generation.

Languages to define model transformations [CMT06, Kle06, WvdS06] and model operations (e.g., model weaving [BJT05], model merging [BBdF+06] or model management [BCR05]) is a topic. Additionally, further work on how to make consistent models, and on how to implement constraints for those models is being carried out [dFBRG06, MCL06]. Refactoring techniques that enable existing software systems to be refactored into an MDA framework is the subject of ongoing work (a.k.a., harvesting [RGvD06]).

## 2.4 Portlet Engineering

The first node of ARPANET (the precursor network of today's Internet) went live in 1969. Fourteen years later (1983), *the* first TCP/IP wide area network (afterwards named NSFNet) interconnected some universities. This is nowadays considered the birth of *Internet*.

CERN publicized the new *World Wide Web* project in 1991 where Tim Berners-Lee developed *hypertext* technologies (e.g., HTTP, HTML). The *web* enabled to create and access (static content) web pages across the world. Shortly after, the words Internet and web became familiar to the public.

These technologies soon demonstrated their power. Hypertext was complemented with functionality. This was named a *web application*. Functionality was steadily more and more complex and diverse.

The use of Internet became common for many activities, and existing sites rocketed steadily. According to NetCraft[5], Internet sites were 18.000 in 1995, 50 millions in 2004, and 100 millions in 2006 (although only half of them are considered active). This growth fostered the emergence of search engines or central

[5] http://news.netcraft.com/


**Figure 2.2**: MyYahoo Portal

access points (e.g. Yahoo!, Altavista). Soon after, these sites were watered down as *Portals* (i.e., entry doors that provide access to diverse services and content). In this setting, each service was named a *Portlet*.

### 2.4.1 Definition

A portlet is a *presentation oriented* web service [OAS03]. Unlike web services, which offer only business logic methods, portlets additionally provide a web-GUI presentation interface. Hence, portlets not only return raw data but also renderable markup (e.g., XHTML) that can be displayed within a portal page in a way similar to Windows applications.

A portal is a web site that provides centralized access to a variety of services [DR04]. An increasing number of these services are realized as portlets. For instance, consider personalized portals as MyYahoo[6] (see Figure 2.2 ), where a variety of services are provided, some of which may be portlets.

Being presentation-oriented, portlets encapsulate an interface layer. portlets have multiple steps that provide the navigation logic that guides the user to accomplish the portlet aim (i.e., the service). Hence, portlet implementation is far from being straightforward as they encapsulate full-fledged applications. Therefore, portlet design should be made in terms of high-level models that abstract from concrete platforms and standards.

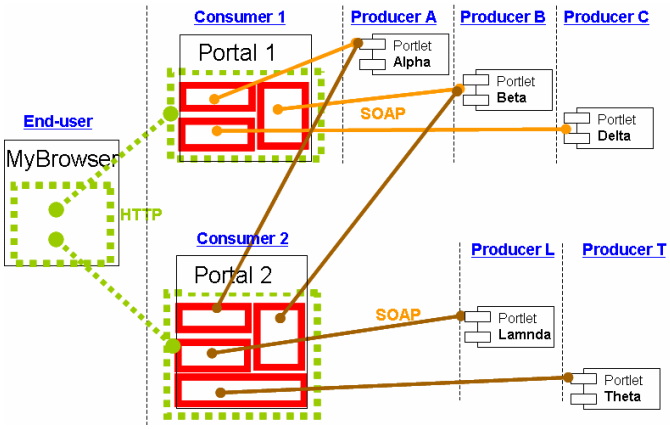



**Figure 2.3**: Portals and Portlets

### 2.4.2 Motivation

Until recently, portlet realization was dependent on the infrastructure of the service provider and the service container (e.g., the portal). This changed with the release of the *Web Services for Remote Portlets* (WSRP) [OAS03], which standardized the web service interface between the consumer (portal) and the producer (portlets), and the *Java Specification Request 168* (JSR 168) [JCP03], which defined how to implement portlets in Java (i.e., the interface between the producer and the portlet itself in J2EE [SSJ02]).

This accounts for portlet interoperability whereby portlets can be seamlessly deployed independently of the platform in which they were developed or accessed. Figure 2.3 depicts a 3-tier architecture for portlets, where an end-user's *MyBrowser* accesses the *Portal_1* page through HTTP. *Portal_1* is hosted by *Consumer_1* and consists of a layout aggregating (through SOAP [W3C03]) the *Alpha*, *Beta*, and *Delta* portlets that are hosted by different producers. Likewise, *Portal_2* is hosted by *Consumer_2* and aggregates the *Alpha* portlet used by *Portal_1* and additional portlets from third-party producers (e.g., *Lambda* portlet from *Producer_D*). Note that each portal uses a different product *variant* of the portlets customized to its requirements (next we will see that these variants can be developed as a product line of portlets).

Portlet *interoperability* fosters a market for portlets *à la* COTS. This implies that different portlets can be delivered to distinct customers which overlap in their

[6] http://my.yahoo.com/

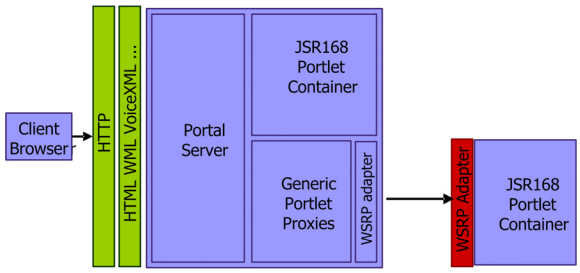



**Figure 2.4**: Portlet Architecture

functionality. Sharing a common platform and customizing assets to provide recurrent functionality have proven to be a cost-effective way to achieve mass customization within a given domain. Hence, it may be appropriate to support portlets as SPL products.

### 2.4.3 Successful Case Studies

There exists a number of *tools* available to construct portals based on portlets. Multiple vendors (e.g., IBM WebSphere Portal, BEA WebLogic Portal, Oracle Portal, Vignette Application Portal) provide tooling for this technology [DR04]. Likewise, some open source portals (e.g., eXo Portal platform, Apache WSRP4J, Liferay portal and uPortal) are available. We refer to the sites (of those tooling vendors) where some successful case studies are reported.

### 2.4.4 Portlet Web Engineering

There exist several methodologies to approach *web engineering*. When developing Web applications we need to face not only functionality but also content, navigation and presentation issues. Existing works focus on these last three issues (e.g., WebML [CFM02], W2000 [BGP00], UWE [NA02] or OO-HMethod [PGIP01]) from which applications are derived (manually or automatically) [PM00]. However, to the best of our knowledge, there is a *lack* of specific approaches (methodologies) for portlet engineering.

### 2.4.5 Existing Approaches

Portlets favor a distributed architecture (see figure 2.4) promoting the logical separation of portlets from portal servers (a.k.a., consumers) that use their services.

They are an example of *Service-Oriented Architectures* (SOA) [OAS06a].

In the traditional portal model, portlets run on the same J2EE application server the portal server does, interacting via simple J2EE interprocess communication. However, scalability considerations move the portlet to another piece of hardware. Frequently, departments within an organization decide to keep control over their own portlets. This task is hard to accomplish when portlets must be deployed to a centralized portal server.

WSRP was not only designed to allow *remote* portlet-to-portal communication. Even in an scenario where portlets are deployed locally, WSRP keeps its value as a platform-independent specification. Doing so, a J2EE-based portal server could interoperate with a portlet running on a *.NET* machine as long as it exposes its functionality via WSRP-compliant Web services.

Once in a portlet container (a.k.a., producer), portlets can be realized in distinct platforms (e.g., J2EE, .NET, etc). Here is when JSR168 becomes important. [JCP03]. This specification defines a standard set of APIs for portlets to be plugged into J2EE-based portal servers. In the same way that Servlets run into a Servlet container, JSR168 defines a portlet container that manages portlets. The container is the true interlocutor with the portlet consumer, and the responsible for mapping WSRP requests into JSR168 operations. This enables the portlet programmer to ignore the intricacies of WSRP. The container also offers infrastructure for personalization, presentation and security of portlets.

**The WSRP standard** WSRP was a joint effort of two OASIS technical committees (TC), namely, the *Web Services for Interactive Applications* (WSIA) and the *Web Service for Remote Portals* (WSRP) [OAS03]. The WSRP TC is formed by people from diverse companies (e.g., BEA Systems, Citrix Systems, IBM, and so on). WSRP layers on top of the existing web service stack, utilizing WSDL for defining a set of interfaces [W3C03, W3C01]. It standardizes the application programming interfaces (API) between consumers and producers of portlets, the communication protocol, and some aspects of the component model (e.g., modes, personalization descriptions, CSS terms and the like). WSRP is also extensible. The specification 1.0 contemplates four interfaces, namely, service description, markup, registration, and portlet management. According to the specification, producers are "*presentation-oriented Web Services that host Portlets*". Hence, a WSRP producer is modeled as a compound of portlets [DR04].

**The JSR168 standard** The *Java Standardization Request 168* (JSR168) is a *Java Community Process* (JCP) initiative that standardizes the way a portlet is developed in a J2EE framework [JCP03]. The JSR168 was co-led by IBM and Sun and had a large expert group (that includes Apache Software Foundation, BEA, Citrix Systems among others). It is defined as a set of extensions to the Java Servlets APIs [CY03]. Basically, three main actors are involved, namely, the portlet entity as a new web component, the portlet application, and the portlet container.

### 2.4.6 Current Research Issues

The first versions of portlet standardization efforts aimed to unify previously existing vendor-specific implementations. Hence, favoring this unification, a number of issues were postponed, some of which are subject of forthcoming work.

WSRP 1.0 is being revised nowadays. *WSRP 2.0* (revision of the specification) will add consumer managed coordination, additional lifecycle management and a set of related aggregation enhancements [OAS06b]. Consumer managed coordination (a.k.a., orchestration) enables inter-portlet communication. This is the subject of current research [DII05]. Nonetheless, the final draft is not yet scheduled (presumably along 2007).

*JSR-286* (the Portlet Specification 2.0 is the next version of JSR-168 [JCP03]) is now under review at the JCP [JCP06]. According to the proposal (at the time of this writing), the following areas are to be addressed: add access to Composite Capability/Preference Profiles (CC/PP) data via the JSR188 API, introduction of portlet filters, J2EE 1.4 support, enhance the portlet tag library, coordinate to better align JSF with portlets [Sun]. Some issues align with WSRP 2.0 [OAS06b], namely, inter-portlet communication, public render parameters, and enhance caching support. The final draft was initially scheduled to the end of 2006.

## 2.5 Related Work

### 2.5.1 Software Product Lines & Model Driven Development

The idea of merging MDD and product lines is not new [AFM05, AGESR06, CA05b, DSvGB03, GBLC05, SNW05]. The term *Model Driven Software Product Lines* were first coined by [CA05b], and we know of few examples. Batory provides a unification of the vocabularies used in FOP and MDD [Bat06].

There exists three major approaches: *(i)* Czarnecki's work on mapping features to models [CA05a], *(ii)* PuLSE-MDD that provides a process to drive pattern generation [AFM05], and *(iii) BoldStroke* that is a product-line written in several millions lines of C++ for supporting a family of mission computing avionics for military aircraft [Gea04]. Conversely, our work introduces *Feature Oriented Model Driven Development* (FOMDD) which scales AHEAD model of variability by introducing modeling, and provides some mathematical background (e.g., commuting diagrams).

In general, existing approaches differ in several ways: *(i)* the paradigm used to distinguish products is different (e.g., decisions vs. features), then *(ii)* the model to realize (implement) variability is also different, even *(iii)* the process to develop the product line reusable infrastructure differs, doing so *(iv)* the mechanisms used to synthesize (derive) a product (and its models) are also different. These differences call to a deeper analysis that was discussed in a workshop on the integration of SPL and MDD [SNW05].

### 2.5.2 Software Product Lines & Web

Jarzabek et al. presented a study on portal cloning (source code repeated from product to product). This study revealed that cloning rate ranges upwards to 50-60% [RJ05]. This cloning rate was resolved by means of a product-line using XVCL [JBZZ03]. In fact, this pioneering work reported a first case study on the use of XVCL for web applications. XVCL is a technique based on customized extensions where raw artifacts (i.e., frames in XVCL terminology) are instrumented for flexibility by inlaying variation points. A variation point denotes a particular place in a raw artifact where choices need to be made as to which variant to use. In this way, variations are completely detached from raw artifacts, and their effects are separately described through customization scripts. Scripts produce customized artifacts (i.e., artifacts that cater for certain variants of the feature model).

Capilla et al. proposed a lightweight approach to SPL web application [CD03]. It analyzed existing SPL assets using differentiation techniques (e.g., *diff*). *Diff*-detected common fragments were factored out into so-called core assets (i.e., the platform of the SPL). This approach was intended to build a few number of products where a number of benefits appear. The authors claim that the upfront investment is lower than in traditional (heavy) approaches. The setting up is quicker. Thus, the initial time-to-market is reduced. However, it might result in long-term drawbacks when the product line is created to build many (hundreds or thousands)

products. Core assets are not documented. The production plan is informal (i.e., not defined), so product production is a hand craft process where human intervention is required. In summary, this lightweight approach is powerful when the SPL consists of a few products, but it shows drawbacks scaling.

Balzerani et al. presented Koriandol as a product-line architecture designed specifically to develop web families [BdRPdA05]. It provides a common set of web modules and a way to configure them. Each module provides common web functionality, but does not provide means to deal with domain-specific variability when facing a web-specific SPL development. So, variability is achieved by just configuring such modules, developing domain-specific functionality, and integrating them together in built products.

Diaz et al. provided a first attempt to portlet syndication by means of variability [DR05]. Shortly, they focused on automatizing the production plan and bringing variability to it [DTA05]. The point was not only on how the core assets vary to create a feature-customized product, but on how the process to synthesize product varies (see chapter 7). To this end, a production plan capability with two levels of variability was created. The first affects to the product features (i.e., build process), while the second affects to the process features (i.e., synthesis process).

### 2.5.3 Model Driven Development & Web

Kurtev introduces XML transformations to develop XML applications [KvdB05]. Web applications face not only functionality but also content, navigation and presentation issues. Several approaches also introduce models in web engineering (e.g., WebML [CFM02], W2000 [BGP00], UWE [NA02] or OO-HMethod [PGIP01]).

*Web Software Architecture* (WebSA) approach [MG06a, Mel07] is based on the MDE paradigm [Béz04] and more specifically on the MDA [OMG03]. In general, WebSA offers the designer a set of architectural models and transformation models to specify a web application. Starting from these models, the designer can integrate the Web functional models (e.g., domain, navigation and presentation) with the architectural models. This is achieved applying a set of model transformations where QVT is used [OMG05a]. The result is the integration model, which is a platform independent model that can be transformed into the different platforms such as J2EE, .NET, etc.

Though the literature in web and model-driven is rich, to the best of our knowledge, there is a lack of specific approaches for portlets.

## 2.6 Conclusions

The purpose of this chapter was to provide a brief introduction to the existing background on top of which this work is built on.

- Software Product Lines
- Model Driven Development
- Portlet (and Web) Engineering

Some related work inter-connecting those presented fields was also discussed:

- Software Product Lines & Model Driven development
- Software Product Lines & Web
- Model Driven Development & Web

We invite the reader to check the bibliography section for a comprehensive account.

# Chapter 3

# SPL need *Endogenous* Transformations

"Every new beginning comes from some other beginning's end".

– *Seneca.*

## 3.1 Abstract

The previous chapter introduced *Software Product Lines* (SPL) whereby a family of programs can be created. Doing so, development shifts from individual programs to reusable artifacts that can be used in different programs. These programs are distinguished by different requirements, which typically involve the realization of different increments in functionality. Hence, SPL need techniques to realize these increments in functionality.

A key issue is how to realize this variability (a.k.a., *variability realization techniques*). There exist different techniques, and the selection of one technique imposes how the program synthesis is later realized. In general, the code of a reusable artifact is transformed during program synthesis. This transformation does not change the type of the artifact (e.g., code), but only increments its functionality. This type of transformation is known as *endogenous* and is fundamental to realize SPL variability.

In this chapter, we discuss *the need for endogenous transformations in SPL. AHEAD* is specifically selected as the variability realization technique to realize endogenous transformations during program synthesis. Hence, this chapter is about AHEAD, which is exploring the structure that increments in functionality (a.k.a., features) impose on programs. This *feature structure* is represented using mathematics. Its algebraic representation describes features and how programs are synthesized by composing features.

This chapter deals not only with AHEAD and endogenous transformations, but introduces as well some specific contributions, namely, *(i)* our extensions of *AHEAD Tool Suite* (ATS) to cope with variability of heterogeneous representations such as XML documents, and *(ii)* a case study for *feature refactoring* a large scale program with heterogeneous representations. The substantial program is ATS.

## 3.2 Rationale for *Endogenous* Transformations

In this section we discuss the rationale for using transformations between models expressed in the same language [MG06b]. Particularly, we are interested in those specific endogenous transformations that increment the functionality of programs.

Such transformations are commonly used to synthesize a set of different programs from an SPL (i.e., different programs demand different variations). Although it is not trivial to achieve such variability, there exist *variability realization techniques* to support the creation of different SPL programs.

### 3.2.1 Variability Realization Techniques

There are different SPL strategies, visions, and schools (section 2.2.6 introduces existing approaches). It is not our intention to describe them, but to argue **why the Feature Oriented Programming** (FOP) paradigm and its realizing model **AHEAD** is used in this work.

Existing techniques realize variability for different types of reusable artifacts (e.g., code, architecture, etc). This work concentrates initially on the code implementation level where distinct techniques have been proposed to realize variability, namely, aggregation/delegation, aspect-oriented programming, conditional compilation, dynamic class loading, dynamic link libraries, frames, inheritance, overloading, parameterization, configuration properties, static libraries, etc. See [AG01] for a discussion.

In general, all these techniques can be used to realize artifact variability (e.g., conditional compilation, parameterization, configuration properties), but some do not lend themselves to automatic program synthesis. This implies that program synthesis requires *human intervention* (e.g., configure certain parameters for conditional compilation or give some config properties).

Alternatively, some techniques support synthesis (e.g., frames, inheritance, etc). Hence, programs are synthesized automatically (i.e., without human intervention). Several approaches belong to this group, namely, CONSUL-pure::variants [Sys], XVCL [JBZZ03], and AHEAD [BSR04]. These techniques have been successfully applied to create SPL in different contexts.

The differences stem from the specific technique *(i)* to represent artifact variability, *(ii)* to realize synthesis, and *(iii)* to structure artifacts. Different techniques are used for *(i)*: CONSUL allows any type of transformation[1], XVCL uses frames[2], and AHEAD uses mixin-inheritance[3]. This implies that the mechanisms to realize synthesis also differs: CONSUL applies transformations, XVCL applies Customization Scripts into frames, and AHEAD applies feature composition[4].

AHEAD supports compositional programming as opposed to others. Compositional programming represents structure. By structure we mean the parts of an SPL and how these parts are composed to yield products. This structuring of artifacts differentiates AHEAD from other approaches. AHEAD imposes a general structure to arrange artifacts for synthesis. What is original of AHEAD is the way it expresses this structure via mathematics.

### 3.2.2 Mathematical Structure in AHEAD

Individual programs have inherently structure. Indeed, structuring programs and modularization are inherent to computer sciences.

SPL are not an exception. SPL programs are synthesized from structures containing artifacts. Doing so, a program is synthesized as a composition of structures of artifacts (i.e., synthesis composes units of structure). Frequently, features are

[1] This implies that not only endogenous, but also other types of transformations like exogenous can be used to increment a program in functionality. SPL need endogenous transformations just to increment functionality. The use of other transformations could alter this restricted goal because they are intended to realize other purposes (e.g., Model Driven Development).

[2] It is a template programming technique where variations (a.k.a., customization scripts) are applied on base templates. It is based on Bassett's frames [Bas97].

[3] Mixin inheritance details are introduced next. This technique is a specific endogenous transformation which only adds functionality.

[4] Refer to Blair et al. [BB04] for a comparative of generative approaches (XVCL vs. AHEAD).

those units (i.e., incrementing application functionality) by which different programs are not only distinguished and defined, but also SPL artifacts structured.

When features are used to impose that structure, it is known as *Feature Oriented Programming* (FOP) [BSR04]. This work focuses specifically on FOP and its realizing model AHEAD, which is a general structural model based on mathematics [BSR04]. It provides a particular case for endogenous transformations where structure of artifacts is central.

FOP structures code pieces into features. Doing so, a set of code pieces realizes the functionality addressed by such feature. AHEAD is a structural model realizing FOP that explores the structure that features impose on programs. To attain this, AHEAD studies feature modularity structure and its use in program synthesis [BSR04, LH06, Sma99].

The way AHEAD expresses structure is via mathematics. Indeed, the mathematical perspective is a search for structure (nothing more and nothing less). Mathematics is the science of structure. Those interested in understanding structure implications precisely are interested in mathematics to describe such structure [Len06]. Doing so, a scientific approach to programming methodology is pursued where algebra represents structure.

### 3.2.3 Rationale for AHEAD

FOP is driven by features [BSR04, Kea90]. The core idea behind FOP is not the intensive use of the term **feature**, but the idea that some unit is needed to distinguish among products within an SPL. This unit to distinguish products is fundamental to any SPL approach (e.g., decisions and features).

AHEAD fundamental idea is to structure artifacts based on features (i.e., the set of artifacts realizing a feature are grouped together). Doing so, the increment in functionality of each feature is separated into a feature module realizing it (a.k.a., feature layer). Features are not only used in domain analysis, but in the whole development of SPL by imposing a top-down use of features. AHEAD takes the definition of feature (i.e., encapsulates an increment of functionality) to the fullest extent of its consequences.

The artifacts that realize a feature have a dual nature in AHEAD. Some of them are **introductions** (i.e., newly introduced artifacts), whereas others are **refinements** (i.e., extensions at predefined points of previously introduced artifacts). This dual nature is similar in other SPL approaches, but is represented differently (e.g., variation points [Bos00, Gom04]).

Refinement is a central concept of AHEAD. It is a technique to realize variability where increments are realized using **mixin-inheritance**. Inheritance is broadly used, and mixin-inheritance is just a special case of regular inheritance [BC90] (detailed in section 3.4.1). Nonetheless, the term refinement could be misleading because it has different usages nowadays in computer science (explained shortly). However, what is common to any SPL approach is the need of some variability realization technique [AG01].

Overall, AHEAD structural model is intended to synthesize programs by composition. The **composition** of features implies the composition of their structure and artifacts. Composition of introductions is straightforward (i.e., just copy them). On the other hand, refinements must be composed with existing artifacts. Hence, a composition operator is necessary (i.e., mixin-inheritance composition). This operator enables to automate feature composition. This automation is the key that would eventually make compositional programming possible (i.e., the automatic composition of products from features) [Sma99].

The realization of a feature involves frequently not only source code artifacts, but other **multiple representations** (e.g., HTML pages, UML models, etc). To attain this, composition is a polymorphic operator (i.e., an specific composer tool for each type). This allows the composition of multiple and heterogeneous artifacts.

AHEAD has a companion suite of tools realizing described structural model. However, the availability of tools is not restricted to AHEAD. What is specific of AHEAD is that its usage is documented and that tool download is free [Bata].

## 3.3 AHEAD: A Model of Feature Oriented Programming

*Feature Oriented Programming* is a general paradigm of SPL program synthesis where feature units are the building blocks of programs. Each feature unit (a.k.a., feature layer or module*)* may include any number of artifacts (i.e., representations). GenVoca was an early model of FOP; *Algebraic Hierarchical Equations for Application Design* (AHEAD) is the current model [BSR04].

### 3.3.1 GenVoca

*GenVoca* is an algebra that offers a set of operations, where each operation realizes a feature. We write $M = \{f,\ h,\ i,\ j\}$ to mean model $M$ has operations or features $f$, $h$, $i$ and $j$. GenVoca distinguishes features as *constants* or *functions*. Constants (or constant functions) represent base programs. For example:

$f$ // a program with feature f
$h$ // a program with feature h

Functions represent program *refinements*[5] that extend a program that is received as input. For instance:

$i \bullet x$ // adds feature i to program x
$j \bullet x$ // adds feature j to program x

where $\bullet$ denotes function application. GenVoca is based on *step-wise refinement* paradigm for developing a complex program from a simple program by adding features incrementally [Dij76]. Endogenous transformations are used to map between models (e.g., programs) expressed in the same language, and GenVoca is a specific case to add functionality incrementally. Doing so, the design of a program is a named expression, e.g.:

$prog_1 = i \bullet f$ // program $prog_1$ has features f and i
$prog_2 = j \bullet h$ // program $prog_2$ has features h and j
$prog_3 = i \bullet j \bullet h$ // program $prog_3$ has features h, j, and i

The set of programs that can be created from a model is its product line[6]. Expression optimization corresponds to program design optimization, and expression evaluation corresponds to program synthesis [BCRW00, LB04, SAC+79]. Not all features are compatible. The use of one feature may preclude the use of some features or may demand the use of others. Tools that validate compositions of features are discussed in [Bat05, BRCT05, Ben07, BSTRC07].

### 3.3.2 AHEAD

*Algebraic Hierarchical Equations for Application Design (AHEAD)* extends GenVoca to express nested hierarchies of artifacts (i.e., files) and their composition [BSR04]. If feature $f$ encapsulates a set of artifacts $a_f$, $b_f$, and $d_f$ we write $f$=

[5] Note that *refinement* is used in the context of *FOP* to represent a refinement *function* that increments in feature functionality [BSR04]. It is inspired on Dijkstra's *Step-Wise Refinement* [Dij76]. Next chapter describes an alternative definition of refinement in the context of *MDD*, which is often used when a specification is gradually refined into a full-fledged implementation by means of successive concrete steps that *add more concrete details* [MG06b]. Although similar, each is intended for a different purpose. Refer to [Ape07] for more detail.

[6] Note: Although we write the composition of features *a* and *b* as $a \bullet b$, it really is an abbreviation of the expression *compose(a,b)*. We use $\bullet$ to simplify FOP expressions.

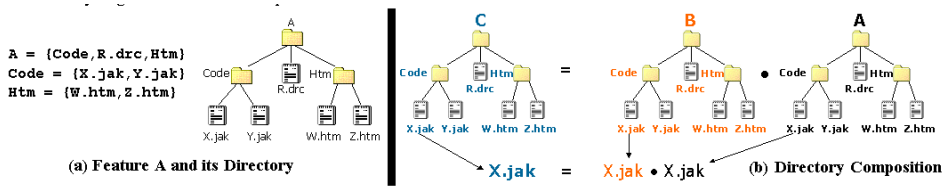



**Figure 3.1**: Features as Directories

$\{a_f, b_f, d_f\}$. Similarly, $i = \{a_i, b_i, c_i\}$ says that feature $i$ encapsulates artifacts $a_i$, $b_i$, and $c_i$ . As artifacts themselves may be sets, a feature is a nested set of artifacts. AHEAD uses directories to represent nested sets. Figure 3.1a shows an AHEAD feature and its corresponding directory.

The composition of features is governed by the rules of inheritance. In the composition $i \bullet f$, all artifacts of $f$ are inherited by $i$. Further, artifacts with the same name (ignoring subscripts) are composed pairwise. This is AHEAD's *Law of Composition*:

$$i \bullet f = \{a_i, b_i, c_i\} \bullet \{ a_f, b_f, d_f\} = \{ a_i \bullet a_f, b_i \bullet b_f, c_i, d_f\} \quad (1)$$

Features are composed by applying *(1)* recursively, where directories are folded together by composing corresponding artifacts in each directory. Figure 3.1b shows the composition of features A and B. The result is feature C, where artifact *X.jak* of C is synthesized by composing *X.jak* (from B) with *X.jak* (from A) [BSR04].

The polymorphism of the • operator is central to AHEAD. Artifacts of a given type (*.jak*, *.b*, etc.) and their refinements are defined in a type-specific language. That is, the definition and refinements of *.jak* files are expressed in the Jak(arta) language, a superset of Java. The definition and refinement of *.b* files are expressed as Bali grammars, which are annotated BNF files. And so on. One or more tools implement the • operator for each artifact type. The **jampack** and **mixin** tools implement the • operator for *.jak* files (i.e., *.jak* files are composed by either the **jampack** and **mixin** tools), and the **balicomposer** tool implements the • operator for *.b* files. **ATS** is the set of tools that compose artifacts, produce and analyze compositions, and derive artifacts of one type from others (e.g., the **jak2java** tool translates a *.jak* file to its *.java* counterpart).

```
// (a) Base
class Foo {
  void bar () { /* Base's content */ }
}
// (b) Feature1
refines class Foo {
  void bar () { Super().bar();
                /* Feature1's content */ }
}
// (d) Feature2
refines class Foo {
  void bar () { /* Feature2's content */ }
}
```

```
// (c) Feature1 (Base)
class Foo {
  void bar () { /* Base's content */
                /* Feature1's content */ }
}
// (e) Feature2 (Base)
class Foo {
  void bar () { /* Feature2's content */ }
}
```

**Figure 3.2**: *Jak* Refinement

### 3.3.3 Jak and Java Refinement

*Jak*(arta) is an extension of Java for FOP [Bata]. It supports special language constructs to express refinements of classes. Jak classes can be translated to standard Java classes using *jak2java* tool [Bata]. A refinement implies that there is something to be refined (i.e., the argument of the refinement). Indeed, we can think of refinement as a function (e.g., an endogenous transformation) that takes a base artifact as input, and returns another similar artifact which has been updated to support a given feature.

Jak refinement is not implemented by regular inheritance. It is implemented by *mixin-based inheritance* [BC90, SB02]. A mixin is an abstract subclass that can be applied to various classes to yield a new class (i.e., a class whose superclass is parameterized [BC90]). Composing a mixin and a class is called mixin composition. The relationship between mixin and superclass is called mixin-based inheritance, a form of inheritance that delays the link between subclass and superclass until composition time. The difference is that the link to the superclass is not fixed until composition time, whereas this link is fixed in regular inheritance. Figure 3.2 illustrates examples for Jak refinement.

Figure 3.2a shows a *Jak* artifact *Foo* defining a method *bar* realizing the feature *Base*. Now consider that the realization of *Feature1* implies changing the existing class *Foo* by extending the method *bar* with further functionality[7]. Figure 3.2b shows the definition of this refinement function in Jak [BSR04]. The expression *Feature1(Base)* returns a Jak artifact which refines *Base*. Figure 3.2c illustrates this composition. Similarly, 3.2d shows the realization of *Feature2* where method *bar*

[7] *Super().bar();* calls parent class. Note that *Super()* differs from common *super*.

is overridden (i.e., the previous functionality of this method is lost). Figure 3.2e illustrates the expression *Feature2(Base)*.

The order of feature composition can matter. Consider the expression *Feature1(Feature2(Base))* that stands for *Base* being composed with *Feature1* and *Feature2*. In this case, *Feature2* overrides *Base*, and *Feature1* extends the result. Conversely, the expression *Feature2(Feature1(Base))* produces a different result because *Feature2* overrides *Feature1* and *Base*. This yields a different *bar* method. Therefore, the result differs depending on the order of composition.

In general, when the artifact is source code, a *class* refinement[8] can introduce new data members, methods and constructors to a target class, as well as extend or override existing methods and constructors of that class. In addition, composition is polymorphic. AHEAD provides the *principle of uniformity* to generalize refinement [BSR04]. This implies that it is possible to refine other types of artifacts in addition to code. Each type of artifact has a language in which such artifacts can be refined. Also, there are type-specific composers that enable base artifacts to be composed with their refinements [BSR04]. There has been little work to show how an XML document can be refined. This topic is part of the next section.

## 3.4 Extensions of AHEAD Tool Suite

The *AHEAD Tool Suite* (ATS) is a set of tools supporting AHEAD. We developed a number of ATS extensions to *(i)* deal with heterogeneous artifacts such as XML documents, and *(ii)* to provide a web interface for ATS synthesis (a.k.a., product production).

### 3.4.1 AHEAD Tool Suite (ATS)

*ATS* is a collection of tools that were developed for feature-based program synthesis [Bata]. Over time, ATS grew to different tools such as: *(i) jampack* and *mixin* tools to compose *Jak* files; *(ii) jak2java* a *Jak* to *Java* file translator; *(iii) composer* a tool that composes features; *(iv) modelexplorer* is a GUI tool to browse feature realizations; and *(v) guidsl* a tool for feature modeling.

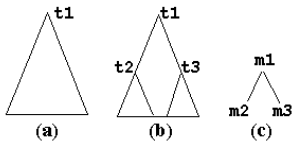



**Figure 3.3**: XAK Module Hierarchy

```
<bibliography xak:artifact="myBib">
    <paper xak:module="mATSRefac">
        <title>Feature Refactoring ..</title>
        <date>06-02-21 11:30:45</date>
        <authors xak:module="mATSAuthors">
            <author id="1">
                <name>Oscar</name>
            </author>
        </authors>
    </paper>
    <paper xak:module="mXAKRef">
        <!-- content collapsed -->
    </paper>
</bibliography>
```

(a)

(b)

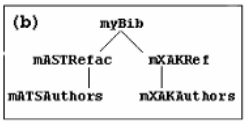



**Figure 3.4**: XAK Base Document Example

```
<xak:refines xak:artifact="myBib">
    <xak:extends xak:module="mATSAuthors">
        <xak:super xak:module="mATSAuthors"/>
        <author id="2">
            <name>Don</name>
        </author>
    </xak:extends>
</xak:refines>
```

(a)

```
<bibliography xak:artifact="myBib">
    <paper xak:module="mATSRefactor">
        <title>Feature Refactoring ..</title>
        <date>06-02-21 11:30:45</date>
        <authors xak:module="mATSAuthors">
            <author id="1">
                <name>Oscar</name>
            </author>
            <author id="2">
                <name>Don</name>
            </author>
        </authors>
    </paper>
    <!-- content collapsed -->
</bibliography>
```

(b)

**Figure 3.5**: XAK Composition Example

### 3.4.2 XAK and XML Refinement[9]

**XAK** (pronounced "sack") is a tool for composing base and refinement artifacts in XML format [Gro]. The need for **XAK** arose two years ago while developing SPL for Portlet applications using AHEAD [DTA05]. In general, an unusual characteristic of web applications is that a sizable fraction of their definition is not Java or Jak source, but rather XML specifications [RJ05] (e.g., JSP, HTML, and Struts control flow files [str]). Thus, it is common for a feature of a web application to contain XML artifacts (base or refinements) and Jak source (base or refinements). We began our work at a time when AHEAD did not have a language for XML artifact refinement and a tool for XML artifact composition. This lead to the creation of **XAK**.

**XAK** follows the AHEAD paradigm of module definition and refinement. An XML document is a tree rooted at node *t1* in Figure 3.3a. Its **XAK** counterpart is slightly different: it is a tree of trees (e.g., trees rooted at nodes *t1*, *t2*, and *t3* in Figure 3.3b), and each of these nodes is tagged with a *xak:module* attribute, so that the **XAK** module abstraction of the original tree is a tree of modules (module *m1* contains modules *m2* and *m3* in Figure 3.3a).

In general, a **XAK** module has a unique name and contains one or more consecutive subtrees. Each subtree may contain any number of modules. Note that the modules of an XML artifact reflect a natural hierarchical partitioning into semantic units that can be refined (more on this shortly).

As a concrete example, Figure 3.4a shows a **XAK** artifact that defines a bibliography. The artifact is partitioned into modules (see the *xak:module* attributes) which impose the module structure of Figure 3.4b. Also, the root of the artifact is labeled with the *xak:artifact* attribute. Note that all modules have unique names.

A refinement of a **XAK** module is defined similarly to method refinement in the *Jak* language [BSR04]. A refinement of the **XAK** artifact of Figure 3.4a is shown in Figure 3.5a that appends a new author to the *mATSAuthors* module. The *xak:super* node is a marker that indicates the place where the original module body is to be substituted. In general, a **XAK** refinement artifact can contain any number of *xak:module* refinements.

The result of composing the base artifact of Figure 3.4a and the refinement of Figure 3.5a is the artifact in Figure 3.5b. Note that it is possible for a **XAK**

[8] Beyond *Jak*, there are refinements for other code artifacts such as *C++*.

[9] This section is extracted from our GPCE 2006 paper [TBD06]. A draft describing XAK is at [ADT06a].

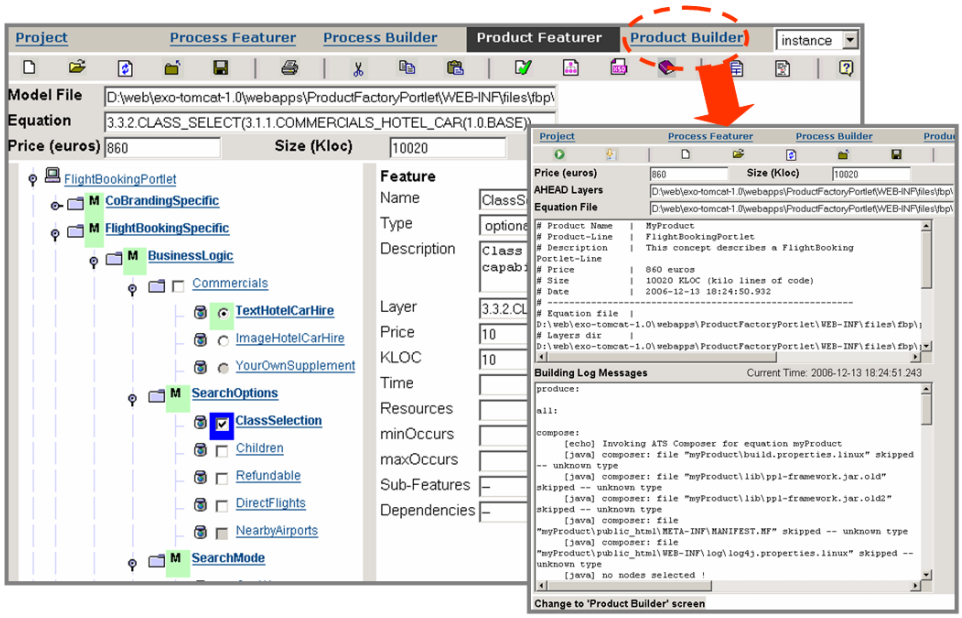



**Figure 3.6**: Feature Selection/Product Production screenshot

refinement to add new modules. In this example, only new content is added to an existing module. Thus, the modular structure of Figure 3.4b is unchanged.

The result of a XAK composition is a XAK artifact. The underlying XML artifact is the XAK artifact with the *xak:artifact* and *xak:module* attributes removed. XAK has other functionalities not needed for our work, such as interfaces, schema extensions and validation [ADT06a].

### 3.4.3 WebGUI Tooling

**WebGUI** is a web-interface (specifically Portlet) tool we created for product-line synthesis (i.e., to create individual programs). Its functionality is roughly *(i)* to configure a given factory project, *(ii)* to select features, and *(iii)* to invoke production processes. This tool uses feature composition by invoking ATS and our own refinement operators to synthesize a custom product. This tool is built on top of ATS tooling using *composer*. It is complementary to the GUI existing tools like *GuiDSL* or *modelexplorer*, but aimed for a Portlet setting.

Figure 3.6 shows 2 screenshots of this tool. On the left side, the (fourth) tab to select features is shown (see *Product Featurer*). This interface allows the browsing of feature model information and the selection of features. It displays a feature

model loaded from an XML specification in a tree-like fashion [BTT05]. Note that the tool provides 2 types of feature modeler: one for products (i.e., fourth tab named product featurer) and other for the production process (i.e., second tab named process featurer). The difference between both would become clearer in chapter 7. On the right side, a screenshot for builder is shown. Basically, this interface allows the invocation of processes (e.g., *ant* makefiles to synthesize products). The tool also provides 2 types of builder: one for products (i.e., fifth tab named product builder) and other for the production process (i.e., third tab named process builder). Both are similar.

## 3.5 A Case Study on Feature Oriented Refactoring[10]

*Feature Oriented Refactoring* (FOR) is the inverse of feature composition. Instead of starting with a base program $B$ and features $F$ and $G$, and composing them to build program $P = F \bullet G \bullet B$, feature refactoring starts with $P$ and refactors $P$ into an expression $F \bullet G \bullet B$. FOR is the process of decomposing a program into a sequence of *features*, that encapsulate increments in program functionality. Doing so, different compositions of features yield different programs. As programs are defined using multiple representations, such as code, makefiles, and documentation, feature refactoring requires *all* representations to be factored. Thus, composing features produces consistent representations of code, makefiles, documentation, etc. for a target program. We present a case study of feature refactoring a substantial tool suite that uses multiple representations. We describe the key technical problems encountered, and sketch the tool support needed for simplifying such refactorings in the future. First, we begin with a review of the rationale behind FOR.

### 3.5.1 Rationale for FOR

Program **evolution** can be described in a high-level and easy-to-understand way as the process of adding and removing features. Giving applications a feature-based design facilitates such extensibility. Designing an application from the ground using features is one approach [BSR04]; an alternative is to refactor a legacy application. FOR is about the latter. In general, the rationale behind FOR is to refactor an FOP product-line starting from an existing application.

[10] The core of this section comes from our GPCE 2006 paper [TBD06].

The challenge of FOR is two-fold. First, feature implementations often do not translate cleanly into traditional software modules, such as methods, classes, and packages. Refinements (e.g., fragments of methods, classes, and packages) are better suited [BSR04]. This requires a theory of program structure that is based on features [LBL06]. Second, what makes FOR unusual is that feature implementations are not monolithic: the implementation of a feature can vary from one program to another (i.e., interactions) [LBL06].

Features have a modular structure that we need to make explicit. To attain this, FOR manipulates program structure in a highly disciplined and sophisticated way. FOR is supported by a theory that relates code refactoring to algebraic factoring, defines relationships between features and their implementing modules, and why features in different programs of a product-line can have different implementations [BSR04, LBL06, ZJ04]. FOR provides a conceptual basis of program structure and manipulation in terms of FOP. Next, a case study on FOR is described.

### 3.5.2 A Case Study: ATS

A program has many representations. Beside source code, there may also be regression tests, documentation, makefiles, UML models, performance models, etc. When a program is refactored into features, *all* of its representations code, regression tests, documentation, etc. must be refactored as well. That is, a feature encapsulates all representations (or changes to existing representations) that define the features implementation. When a program is built by composing features, all relevant representations are synthesized.

In this section, we present a case study in feature refactoring that demonstrates these concepts. The program that we refactor is the *AHEAD Tool Suite (**ATS**)* which is a collection of tools that were developed for feature-based program synthesis [Bata]. Over time, **ATS** has grown to 24 different tools expressed in over 200 KLOC Java. In addition to code, there are makefiles, regression tests, documentation, and program specifications, all of which are intimately intertwined into an integrated whole. There has been an increasing need to customize **ATS** by removing or replacing certain tools. This motivated the feature refactoring of **ATS**, in order to create a product line of its variants.

What is new about our work is the scale of refactoring. Prior work on feature refactoring dealt with small programs under 5 KLOC and focused only on code refactoring [LBN05, LBL06]. In this case study, we scale features substantially in size (**ATS** is almost two orders of magnitude larger) and in the kinds of repre-

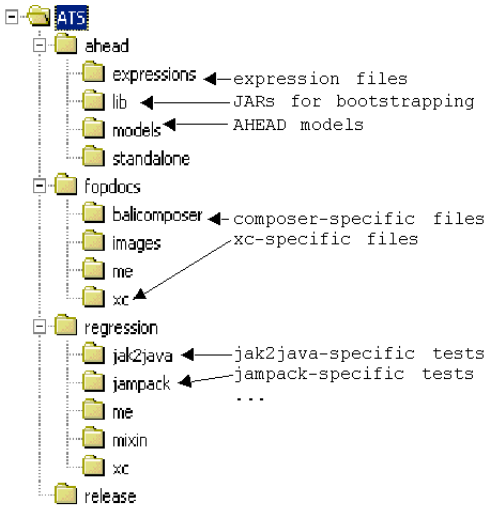



**Figure 3.7**: ATS Directory Structure

sentation (**ATS** required refactoring not only code, but documentation, makefiles, and regression tests as well). This wholistic approach to refactoring *information* is fundamental to feature orientation, and as such this work constitutes a valuable case study on the scalability of feature-based program refactoring and synthesis.

We describe the technical problems encountered in feature refactoring **ATS**, and the kinds of tool support needed for performing such refactorings in the future. We believe our experiences (except one) extrapolate to the feature refactoring of other tool suites. The exception is that **ATS** is bootstrapped, and it posed its own special conceptual and technical challenges.

### 3.5.3 Feature Refactoring and ATS

Figure 3.7 shows a part of the directory structure of **ATS**. There is an expression per tool in the *ahead/expressions* directory. *ahead/lib* contains *Java archive (JAR)* files for **ATS**, *ahead/models* contains directories of AHEAD models, *fopdocs* contains HTML tool documents, and *regression* is a directory of regression tests.

**ATS** is the baseline program for a product line of variants. We want to feature refactor **ATS** into a *core* (the kernel of **ATS**) and optional features, one per tool (e.g., *aj*, *cpp*, *drc*, *jedi*, etc.). By doing so, we create an AHEAD model of **ATS**, which we call the *ATS Product Line (**APL**)*:

```
APL = { core,    // kernel of ATS
```

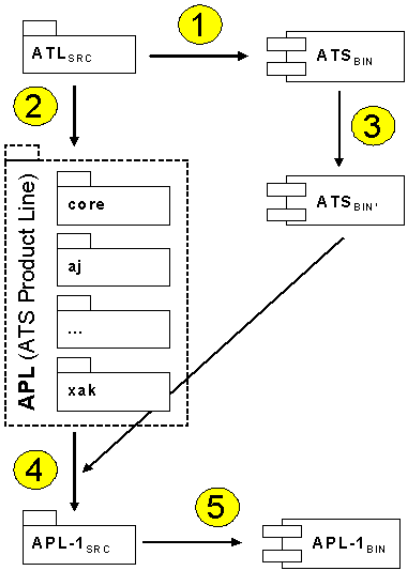



**Figure 3.8**: ATS Refactoring Process

```
            aj,       // aspectj translator
            cpp,      // c++ tools
            guidsl,   // feature modeling tool
            xc,       // XML composer
            drc,      // design rule tool
            jedi,     // javadoc-like tool
            me,       // modelexplorer
... }
```

Given **APL**, we can synthesize different variants of **ATS**:

$$ATS_1 = ...cpp \bullet drc \bullet xc \bullet core;\ \text{// full set of tools}$$
$$ATS_2 = ...cpp \bullet core;\ \text{// a subset of tools}$$
$$ATS_3 = ...jedi \bullet guidsl \bullet m \bullet core;$$
$$ATS_4 = ...drc \bullet aj \bullet core;$$

An essential tool in synthesizing variants of ATS is XAK, whose capabilities we described before in section 3.4.2.

### 3.5.4 The Process of Feature Refactoring ATS

Let $ATS_{src}$ denote the AHEAD feature that contains the source artifacts for the AHEAD Tool Suite. The tool binaries (i.e., JAR files) are produced by an *ant*

XML build which we model by the function *antBuild*:

$$ATS_{bin} = antBuild(ATS_{src})$$

That is, $ATS_{bin}$ differs from $ATS_{src}$ in that tool binaries have been created and added to the **ATS** directory. The build process itself creates directories to contain tool JARs, newly created batch and bash executables[11], and runs the regression tests to evaluate the correctness of tool executables. An **ATS** build time is about 1/2 hour.

Figure 3.8 illustrates the five-step process that we used to feature refactor $ATS_{src}$ into an SPL of **ATS** variants.

**Step 1**. $ATS_{bin}$ was created by an *ant* build of $ATS_{src}$:

$$ATS_{bin} = antBuild(ATS_{src})$$

**Step 2**. $ATS_{src}$ was feature refactored (more on this in Section 3.5.5) into the APL:

$$APL = \{core,\ jedi,\ cpp,\ aj,\ ...\}$$

Each **APL** feature has **XAK** artifacts, that either define or refine HTML and XML documents in $ATS_{src}$.

**Step 3**. Although **APL** is itself an AHEAD model, we could not compose its features using the binaries of $ATS_{bin}$ created in Step 1. The reason is that **APL** features encapsulate **XAK** artifacts and their refinements[12], and the **XAK** tool is not part of $ATS_{bin}$. However, **XAK** can be added to $ATS_{bin}$ by refinement. Let $ATS_{xak}$ be a feature that adds the **XAK** tool to $ATS_{bin}$ to produce $ATS_{bin'}$, which is a set of tools that can compose **APL** features:

$$ATS_{bin'} = ATS_{xak} \bullet ATS_{bin} \qquad (2)$$

Note that the tools of $ATS_{bin}$ are used to evaluate *(2)*. As we explain in Section 3.5.6, this is a form of bootstrapping where a tool suite is used to build a new version of itself.

**Step 4**. We use the tools of $ATS_{bin}$ to synthesize different variants of $ATS_{src}$ by composing **APL** features, such as:

[11] Cygwin. http://www.cygwin.com/

[12] A generic XAK example is shown in section 3.4.2 where XAK tool is also described. APL contains many XAK artifacts that are refined likewise (e.g., *ant* makefiles or *html* documentation). In addition, chapter 7 introduces specific examples for *ant* makefiles, which are very similar.

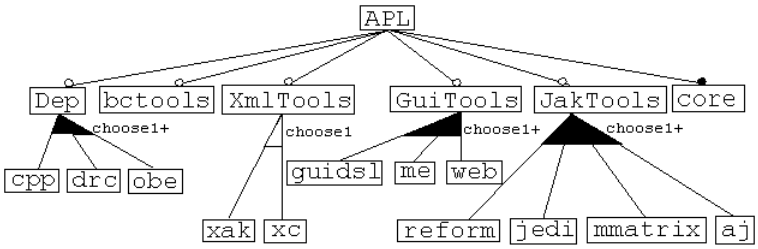



**Figure 3.9**: APL Feature Model

$$APL1_{src} = aj \bullet core$$
$$APL2_{src} = jedi \bullet core$$

**Step 5**. Tool binaries of an **ATS** variant are obtained by an *ant* build:

$$APL1_{bin} = antBuild(APL1_{src})$$
$$APL2_{bin} = antBuild(APL2_{src})$$

In the following sections, we elaborate the details of steps 2, 3, and 4 in this process.

### 3.5.5 Step 2: Refactoring ATS

One of the core contributions of SPL research is the use of features and feature models to define members of an SPL. Although the process of identifying features, refactoring them by extraction from the original program, and creating a feature model is iterative, we proceeded in a largely straightforward fashion (discussed below) with minimal backtracking.

**The APL Feature Model** We identified the following features in **ATS**:

- **core**. The kernel of **ATS** includes *Jak* file tools *jampack*, *mixin*, and *jak2java*, and *Bali* file tools such as *balicomposer* and *bali2jak*.
- **aj, mmatrix, jedi, reform**. Tools that process *Jak* files. *aj* translates *Jak* files to *AspectJ* files, *mmatrix* collects statistics on *Jak* files, *jedi* is a *javadoc-like* tool, and *reform* is a *Jak* file pretty-printer.
- **guidsl, web, me**. GUI-based tools for declaratively specifying programs and exploring AHEAD models.

```
// APL Grammar

APL : Deprecated* [cpp] [bctools] [XmlTools]
      GuiTools* JakTools* core ;

JakTools : // optional Jakarta tools
           reform | jedi | mmatrix | aj ;

GuiTools : guidsl | web | me ; // GUI-based tools

XmlTools : xc | xak ; // XML composition tools

Deprecated: obe | drc | cpp ; // miscellaneous

%% // constraints

bctools ∨ me ⇒ mmatrix;
```

**Figure 3.10**: APL Grammar

- **xak, xc**. Tools for composing XML files.
- **bctools**. Tools that produce, compose, and analyze byte codes.
- **obe, drc, cpp**. Miscellaneous tools.

Figure 3.9 depicts a feature model for **APL** using a notation similar to [CA05a]. Alternatively, a feature model is a context free grammar with constraints (a.k.a., cross-tree constraints) [Bat05]. Figure 3.10 shows a grammar representation that is used in AHEAD to specify feature models and cross-tree constraints (e.g., the selection of **bctools** or the **me** tool requires the **mmatrix** tool). Note that the cross-tree constraints were discovered during the refactoring of **ATS**, discussed in the next section.

**The Refactoring Process** We refactored $ATS_{src}$ in a progressive manner. First, we identified the kernel of $ATS_{src}$ which we called the *core* feature. The remainder of $ATS_{src}$ was called $extra_0$. So our first step in refactoring **ATS** was described by:

$$ATS_{src} = extra_0 \bullet core \qquad (3)$$

We knew that we found a correct definition of *core* when we were able to build *core* tools and run their regression tests without errors:

$$ATS_{core} = antBuild(core) \qquad (4)$$

Unlike prior feature refactoring work, the presence of regression tests in a feature helped us confirm that our refactoring was correct. If a build failed, we tracked

down the missing pieces as files that had to be moved from $extra_0$ into $core$. That is, we failed to include all parts of *core* in our refactoring of (3).

Next, we factored out feature $F_n$ from $extra_n$ to produce a smaller $extra_{n+1}$ feature:

$$extra_n = extra_{n+1} \bullet F_n \qquad (5)$$

To verify that we had a correct definition of $F_n$, we composed it with *core*, built the binaries, and ran the regression tests correctly, which takes over 20 minutes:

$$ATS_n = antBuild(F_n \bullet core) \qquad (6)$$

Again, the regression tests validated both the core tools and the additional tool(s) encapsulated in $F_n$. Failed builds helped us identify pieces that were not moved from $extra_{n+1}$ to $F_n$ .

We repeated *(5)* and *(6)* to remove incrementally all features from *extra* that we identified in the last section. The final version of *extra* was empty (i.e., it contained no files). This refactoring took around 10 person/days.

This process required adjustment when we discovered dependencies among **APL** features. For example, the *bctools* feature required the *mmatrix* feature because the *bctools* invoked the *mmatrix* tool. We had to create a version *v* of ATS with both tools before we could build and test:

$$ATS_v = antBuild(bctools \bullet mmatrix \bullet core)$$

**Refactoring Order** The order in which features were refactored from $ATS_{src}$ into features was progressive. The actual **order** is as follows: *core, bc, drc, aj, cpp, jedi, me, reform, xc, guidsl, mmatrix*, and *obe*. However, this order was not predefined beforehand. The decision on which feature to extract was taken based on intuition (i.e., which feature was more feasible at each point). Further work should study strategies to decide the extraction order and whether this has influence on the refactoring process. This order implies that the creation of the feature model shown in Figure 3.9 was also progressive. Doing so, feature model was also refactored [AGM+06].

**Feature Contents** Each feature encapsulated a tool or group of tools that roughly contains the same content. There were new artifacts such as source files, makefiles, HTML documentation, and regression tests specific to that feature. Refinements

| | SIZE | # FILES | #Java | #LOC | #Jak | #LOC | #XML | #LOC | #XAK | #LOC |
|---|---|---|---|---|---|---|---|---|---|---|
| **core** | 33M | 3904 | 1072 | 162878 | 1326 | 65243 | 78 | 24830 | 0 | 0 |
| **aj** | 106K | 28 | 0 | 0 | 25 | 1293 | 2 | 41 | 2 | 41 |
| **bc** | 903K | 88 | 13 | 3406 | 44 | 1285 | 13 | 2140 | 5 | 81 |
| **cpp** | 128K | 48 | 4 | 501 | 0 | 0 | 8 | 1241 | 5 | 103 |
| **drc** | 348K | 149 | 2 | 93 | 24 | 1708 | 7 | 1128 | 5 | 80 |
| **guidsl** | 901K | 255 | 0 | 0 | 225 | 11631 | 5 | 1186 | 3 | 50 |
| **jedi** | 884K | 200 | 0 | 0 | 133 | 12250 | 5 | 740 | 3 | 53 |
| **jrename** | 55K | 28 | 9 | 406 | 5 | 88 | 2 | 29 | 2 | 29 |
| **me** | 7.7M | 393 | 83 | 17862 | 96 | 2583 | 7 | 3237 | 3 | 53 |
| **mmatrix** | 34K | 8 | 0 | 0 | 0 | 0 | 6 | 412 | 5 | 84 |
| **obe** | 495K | 41 | 23 | 2369 | 0 | 0 | 10 | 1072 | 7 | 128 |
| **reform** | 712K | 114 | 0 | 0 | 105 | 2303 | 6 | 787 | 4 | 60 |
| **xak** | 3.9M | 56 | 17 | 1736 | 0 | 0 | 20 | 2757 | 9 | 171 |
| **xc** | 1.4M | 42 | 5 | 1002 | 0 | 0 | 21 | 4896 | 4 | 114 |

**Figure 3.11**: Feature Size and Content Distribution

were generally limited to *ant* makefiles and HTML document files, both of which were encoded as **XAK** files and **XAK** refinements.

The refactorings of *(3)* and *(5)* were straightforward. We first identified the parts of *ant* makefiles that triggered the building of a target tool **T**, its parts, and its regression tests. These statements were factored from *extra* into a **XAK** file that refined the *ant* build script of *core*. (Initially, we simply commented them out, and once we knew we had a correct set of statements, we moved them into a ***XAK*** refinement file). We followed a similar procedure for refactoring HTML documents. The remaining work was to move files (source, HTML, regression) that were specific to the tools of **T** into the feature **T**, being careful to retain the directory hierarchy in which these files were to appear. There were other files in addition to *ant* makefiles and HTML documents that had to be refined, but these were few and simple (e.g., largely text file concatenation); AHEAD had tools to perform these refinements.

Figure 3.11 shows the detail of disk volume, the number of files and their number of lines of code in each **ATS** feature. The **FILES** column indicates the total number of files in each feature. The *Java*, *Jak*, and *XML* files were introductions (constants), while **XAK** files indicates the number of files that an **ATS** feature refines.

### 3.5.6 Step 3: Bootstrapping ATS/lib

One of the unique parts about refactoring **ATS** is its reliance on bootstrapping: to build **ATS** tools requires **ATS** tools. Many **ATS** tools are written in the *Jak*

language and are themselves composed from features. Seven **ATS** tools are needed for bootstrapping:

- *jampack*, *mixin* tools to compose *Jak* files,
- *jak2java* a *Jak* to *Java* file translator,
- *balicomposer* a tool to compose *Bali* files.
- *bali2jak* a *Bali* file to *Jak* file translator,
- *bali2javacc* a *Bali* file to *JavaCC* translator, and
- *composer* a tool that composes features.

These seven tools are stored as JAR files, which we call *bootstrapping JARs*, in directory *ATS/lib* (see Figure 3.7). As soon as one of the above tools is synthesized during an **ATS** build, its JAR file replaces its bootstrapping JAR from that point on in an **ATS** build. At the end of an **ATS** build, no bootstrapping *JARs* are used.

To synthesize customized versions of **ATS** from features required three distinct changes to be made to **ATS** itself. What makes this intellectually challenging is whether the changes could be expressed using AHEAD concepts. We wanted to stress AHEAD to understand better its generality. In the following, we outline the three changes that we made, and explain how we realized them using refinements.

First, most **APL** features encapsulated regression tests that were specific to the tool(s) the feature encapsulated. Among the tests for Jak tools are Jak files that are syntactically incorrect. (These files are used to test language parsers). The Jak file composition tools (*mixin* and *jampack*) parse all Jak input files before performing any actions. We refined both tools so that if only one Jak file was listed on their command line, the file itself would not be parsed, but merely copied to the target feature directory. In this way, *composer* could invoke *mixin* or *jampack* on syntactically incorrect Jak files without discovering their syntactic errors.

Second, we needed to add *xak.jar* to the bootstrapping JARs to compose **XAK** files representing HTML documents and *ant* build scripts.

Third, we needed to refine JAR files. In particular, the bootstrapping JAR file for *composer*. Remember that this JAR contains a version of *composer* that understands how to compose a predefined set of artifacts (Jak files, Bali files, etc.). If *composer* is to compose new artifact types (such as XAK files), *composer's* bootstrapping JAR must be refined to add this capability before it can compose **APL** features.

*Composer* was designed so that new artifact-composing tools could be added easily. For each artifact type, two Java class files are placed in a subdirectory (called **Unit**) of the *composer* tool. These class files contain information that tell composer how to invoke a tool to compose artifacts of a specific type. For example, there is a pair of files that tell *composer* how to compose *Jak* files using the *mixin* tool; there is another pair of files that tells *composer* how to compose *Bali* grammar files using the *balicomposer* tool, etc. To support **XAK** file composition, two additional class files had to be added to the **Unit** directory to tell *composer* how to compose ***XAK*** files.

All three changes could be made by refining *ATS/lib* using *jarcomposer*. We chose to modify *mixin* and *jampack* directly as its changes were permanent, while adding *xak.jar* and its modification of *composer.jar* in *ATS/lib* were optional. We accomplished the last two changes (adding *xak.jar* and refining *composer.jar*) by refining *ATS/lib*. The general problem is as follows: *lib* is itself an AHEAD module that contains bootstrapping JARs:

$$lib = \{composer.jar,\ mixin.jar,\ ...\}$$

A refinement of *lib* includes new JAR files plus a refinement of the *composer.jar* to tell *composer* how to compose new file types using the new JAR files. For example, the refinement to *lib* that adds **XAK** is:

$$lib_{xak} = \{composer.jar_{xak}, xak.jar\}$$

where $composer.jar_{xak}$ contains the **Unit** file extensions of *composer* that calls **XAK**. To produce a refined *lib* (denoted $lib_{new}$ below), we compose $lib_{xak}$ with *lib*:

$$\begin{aligned} lib_{new} &= lib_{xak} \bullet lib = \\ &= \{composer.jar_{xak} \bullet composer.jar,\ xak.jar,\ mixin.jar,\ ...\} \quad (7) \end{aligned}$$

The key to this bootstrapping step is the ability to compose JAR files. By adding a *jarcomposer* to the bootstrapping JARs, we can use the unrefined lib to evaluate *(7)*, and in principle can now add any number of new artifact composition tools to AHEAD.

Our *jarcomposer* is simple: it unjars the base JAR file and refinement JARs into distinct directories. It then uses the *composer* bootstrapping JAR to compose these directories, forming their union. The contents of the resulting directory are then

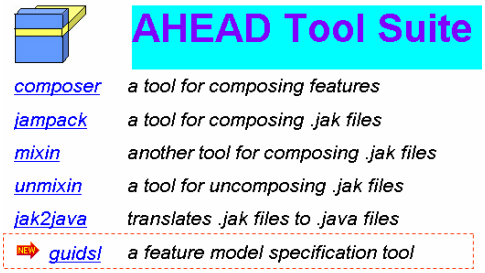



**Figure 3.12**: Customized HTML Documentation

placed into a new JAR file. In *(7)*, this JAR file (i.e., $lib_{new}/composer.jar$) becomes the refined composer bootstrapping JAR. At this point, $lib_{new}$ can compose an enlarged set of artifact types (e.g., **XAK** artifacts), thus allowing **ATS** features to be composed.

### 3.5.7 Step 4: Synthesizing APL-Specific Programs

Once $ATS_{bin'}$ and **APL** were developed, it was a simple matter to define, synthesize, and build different programs of **ATS**. Over one hundred different versions are possible (see Figure 3.9). For example, a version *K* that contained the *core* and *guidsl* tools is specified and built by:

$$ATS_{Ksrc} = guidsl \bullet core \qquad (8)$$
$$ATS_{Kbin} = antBuild(ATS_{Ksrc}) \qquad (9)$$

In the synthesis of $ATS_{Ksrc}$, makefiles and HTML documents of *core* were refined, and the code and document files for *guidsl* were added to *core*. Figure 3.12 displays part of the synthesized *index.html* document of $ATS_K$, which shows a document index for all the tools of *core* (*composer*, *jampack*, *mixin*, *unmixin*, and *jak2java*) and the *guidsl* tool (red dotted box). That is, only documents for tools that are present in $ATS_{Ksrc}$ are listed; no other tool documents are included. This customization is the result of refactoring **ATS**. The evaluation of *(8)* and *(9)* is accomplished using the tools in $ATS_{bin}$.

### 3.5.8 Lessons Learned and Future Tool Support

We encountered many problems during the refactoring of **ATS**. Most were not related with **ATS**, but with the decomposition process itself, and these problems

apply to the refactoring of programs in general. Solutions to these problems and tool support to simplify future refactoring tasks are discussed below.

### Lessons Learned

The greatest challenge in feature refactoring is understanding the original program. When a program grows beyond a few thousand lines of code, it becomes hard to understand what artifacts are impacted by individual features and what dependencies exist among features. Refactoring adds architectural knowledge to a design that was never documented or that was lost. Minimal knowledge of the program and its structure is essential, and we had such knowledge prior to refactoring **ATS** (i.e., we knew approximately the **ATS** directory structure and organization of makefiles and documents).

**Dependencies among Features.** Dependencies among features are not always evident. For example, the *bctools* feature requires the *mmatrix* feature; we discovered this requirement when *bctools* failed to compile without *mmatrix*. Once we recognized this relationship, we remembered it by adding a cross-tree constraint to our feature model (Figure 3.10). Discovering such dependencies and updating feature models is a manual process that we feel could largely be automated.

**Error Exposure.** Feature refactoring may expose existing errors. For instance, the documentation for the *cpp* tools was not referenced by the **ATS** documentation home page and consequently was not accessible. (A link to the documentation of all tools should appear in the home page). We detected this error during the creation of the *cpp* feature, as we knew the home page document had to be refined, but was not.

**Program Extensions.** AHEAD and GenVoca are generalizations of object-oriented frameworks [BCS00]. Extending a framework involves adding new classes and extending hook methods that are defined in abstract classes of a framework. While frameworks have been developed primarily for code, AHEAD generalizes this idea to frameworks to non-code artifacts as well. That is, there are variation points in documents of all types, and document extensions are made at these points. It is well-known that manually extending frameworks is error prone, and could be substantially helped by tools that guide users in how to properly extend frameworks [FHLS97]. The same ideas apply to non-code documents as well.

The lack of documentation or tool support for adding tools to **ATS** hampered our abilities to feature refactor **ATS**. Extending **ATS** has historically been ad hoc, where variations arise due to different programmers following different procedures. We now know that a new tool would *(i)* add source files, *(ii)* add one or more document pages that are linked with existing documentation at predefined points, *(iii)* add new build documents that are linked to existing build script at predefined locations, and *(iv)* regression tests must appear in a designated directory. A standard procedure for extending **ATS** would have significantly helped us in refactoring because we would have known exactly where changes would be made (as opposed to discovering these places later when *ant* builds fail or when synthesized documentation is found to be erroneous).

**Accidental Complexities (Grandmas Teeth).** Accidental complexity lies at the heart of many problems in software engineering [Bro87]. It is unnecessary and arbitrary complexity that obscures the similarity of designs, patterns, and processes that could otherwise be unified. Years ago, we were building different tools for the same language (e.g., pretty-printers, composers, translators), and discovered the process by which each tool was designed and created was unique. Odd details whose only rationale was that was the way we did it distinguished different tools. Called Grandmas teeth, meaning a gross lack of alignment in an otherwise identical design, was an indicator of accidental complexity. By standardizing designs (which reduces complexity), we were able to significantly simplify the design and synthesis of tool suites [BLS03], and make the process of design and implementation more rigorous, structured, repeatable, and streamlined.

Our refactoring of **ATS** exposed other forms of Grandmas teeth that we were unaware of. Makefiles passed parameters to other makefiles, but the same parameter was given different names in different places. Classpaths were defined in different makefiles in different manners. The place where makefiles are altered to add build scripts for similar tools varied significantly. The removal of accidental complexity and the alignment of similar concepts would solve this, and would substantially simplify program refactoring.

**Generality of Experiences.** We believe that there is nothing special about **ATS** or our procedure to refactor it. (Although **ATS** is unusual in that it is bootstrapped, this made it harder to refactor. Few applications require bootstrapping). The incremental way in which we refactored features from **ATS** is analogous to aspect

mining and refactoring (see Section 3.5.9), and is not unusual. Further, although the dependencies among features were minimal, the basic approach in Section 3.5.4 would be our starting point for future refactorings.

Even though our work is preliminary, we believe that it is possible to feature refactor an application with minimal knowledge of its structure. That we had regression tests per feature helped substantially in validating our efforts. Tool support (discussed in the next section) would help greatly in a refactoring process.

### Future Tool Support

**A Tool for Initial Refactoring.** We learned from **ATS** that when a feature encapsulates one or more tools, most artifacts of a feature are new files. Only specific artifacts (makefile gateways, documentation home pages, etc.) are refined. A simple tool could be created to partition the files of a composite feature (i.e., directory) into a pair of directories (representing the contents of different features), thus simplifying an important step in feature refactoring. A more complex tool could provide some heuristics to automate feature refactoring. However, this tool would require further research.

**Artifact-Specific Refactoring Tools.** **ATS** currently has tools for refactoring Java/Jak source classes [LBL06]. Exactly the same kind of tool would be needed for refactoring XML (e.g., **XAK**) artifacts into base and refinement files. Other artifact-specific tools would be needed for refactoring other documents (e.g., Word files, JPEG images).

**Safe Composition Tools.** A recently added set of tools to the **ATS** arsenal are those that support safe composition [BT06]. *Safe composition* is the guarantee that programs composed from features are absent of references to undefined elements (such as classes, methods, and variables). Existing safe composition tools are targeted to Java and Jak source files. However, the same concept holds for other artifact types. XML and HTML define elements that can be subsequently referenced. We want to synthesize documents of all types that are devoid of references to undefined elements, including cross-references to elements of documents in different types. Safe composition tools could also be useful in detecting unreferenced elements or benign references. A *benign reference* is a reference to a non-existent file, but no error is reported. *ant* makefiles allow benign references in *fileset* definitions: if a listed file in a *fileset* is not present, *ant* does not complain. Benign

references and unreferenced elements suggest *(i)* they are no longer needed and should be removed, *(ii)* they are not linked to other elements (the topic of Error Exposure of the previous section), or *(iii)* they belong to other features. In general, we believe that a tremendous amount of automated support for feature refactoring could be provided by detecting unused element definitions or benign references.

There are some prototype applications for the future tool support presented in this section[13]. So far, there is a *win32* prototype application for initial refactoring of artifacts into layers. Current work is focusing on the automation of this extraction instead of manual. As well, there is a prototype eclipse plugin tool for refactoring of XML artifacts using XAK tooling.

### 3.5.9 Related Work

Three SPL adoption models have been proposed, namely, extractive, reactive, and proactive [CK02]. **ATS** refactoring exemplified the extractive approach which *"reuses one or more existing software products for the product lines initial baseline"*. Few experiences have been published on the extractive approach for SPLs. In [CD03], re-engineering techniques are used to obtain an SPL from already available programs. The Unix *diff* command is used to extract differences among artifacts. Pairs of files are compared to obtain the lines matched, lines inserted, lines deleted and lines replaced. These hints are then used to ascertain common abstractions, and to group artifacts with a shared common structure, code and functionality. Similar concerns are addressed in [BCK03] where the focus is on abstraction elicitation in SPLs; the approach looks for cut-copy-paste clones within distinct pieces of code that can be moved into an abstract superclass. *Diff*-like facilities are also used for this purpose.

These efforts aim at improving reuse, modularity and legibility of the software artifacts. By contrast, feature refactoring is not only concerned about reuse but on engineering a system for variability. Moreover, and unlike prior research, our work underlines the importance and necessity of refactoring and encapsulating multiple representations of programs in features. Our work can be seen as an instance of a general approach to feature-oriented refactoring of legacy programs [LBL06].

Features can use aspects to implement program refinements. Feature refactoring is thus related to aspect mining and refactoring [vDMM03, LHB06, LH06, Mea01, ZJ04], which strives to surface hidden concerns, much like our goal to

[13]This is largely thanks to the term-projects of two masters students at the University of Texas at Austin: Yi Li and Jasraj U. Dange

strive to surface particular features [HK01]. We see three distinctions between our work and aspect refactoring. First is the scale on which we are refactoring: **ATS** features encapsulate tools or groups of tools; very few pieces of simple advice (i.e., refinements of designated variation points) are used. Second, we are largely refactoring artifacts other than Java code. And third, while aspect-based approaches rely on labeling methods and inferring other labels via program analysis [RM02], the approach described in this work relies on regression tests and build scripts to surface features. Through failures in a build process, the missing pieces are identified and moved to the appropriate feature. This is more akin with SPLs where the production process (basically supported through *build* scripts) plays a major role.

Feature refactoring includes three main activities, namely, feature identification, feature refactoring as such (a.k.a., decomposition or extraction), and refactoring validation. In our approach, the first two are mainly manual whereas regression tests are used to validate the result and, if a build fails, to guide refactoring. A more intensive use of regression tests are presented in [MH02]. First, test cases are grouped to identify feature implementation. Clustering and textual pattern analysis is conducted to find test-case clusters. A cluster execution serves to locate source code that implements the feature through the use of code profilers. Once feature implementation is located, the code is refactored (e.g., global variables are removed and implicit communication is moved to explicit interfaces) into fine-grained components.

## 3.6 Contributions

The contribution of this chapter was first to argue the need of endogenous transformations in an SPL setting. Among existing approaches, AHEAD was selected to deal with endogenous transformations. Some specific contributions were then described: we extended the AHEAD Tool Suite (**ATS**) with further tooling and we feature refactored ATS.

*Feature refactoring* is the process of decomposing a legacy program into a set of building blocks called features. Each feature implements an increment in program functionality; it encapsulates new artifacts (code, documentation, regression tests, etc.) that it adds to a program, in addition to the changes it makes to existing artifacts that integrate the new artifacts into a coherent whole. The result of feature refactoring is a software product line, where different variations of the original program can be synthesized by composing different features.

**ATS** is the largest program, by almost two orders of magnitude, that we have feature refactored. **ATS** consists of 24 different tools expressed in 200 KLOC Java. We feature refactored **ATS** so that hundreds of **ATS** variants (with or without specific tools) could be synthesized. We expressed the process of refactoring by a simple mathematical model that relates algebraic factoring to artifact refactoring, and program synthesis to expression evaluation. Refining and composing XML documents was critical to our work, and we were able to verify a correct feature refactoring by successfully executing ATS build scripts and running regression tests for all synthesized **ATS** tools. We showed how an integral part of refactoring ATS, namely its need for bootstrapping, could be explained by refinements. And most importantly, our work revealed generic problems, solutions, and an entire suite of tools that could be created to simplify future feature refactoring tasks.

Our work is a valuable case study on the scalability of feature-based multiple-representations program refactoring and synthesis. We believe our work outlines a new generation of useful program refactoring tools that can simplify future feature refactoring efforts.

Parts of the work described in this chapter has been presented before.

1. *Feature Refactoring a Multi-Representation Program into a Product Line*. S. Trujillo, D. Batory and O. Diaz. 5th International Conference on Generative Programming and Component Engineering (GPCE 2006). Portland, Oregon, USA. October 2006 [TBD06]. Acceptance Rate: 28 % (25+5/88).

2. *Supporting Production Strategies as Refinements of the Production Process*. O. Díaz, S. Trujillo and F. I. Anfurrutia. 9th International Software Product Lines Conference (SPLC 2005). Rennes, France. September 2005 [DTA05]. Acceptance Rate: 23 % (17+3/71).

3. *On Refining XML Artifacts*. F. I. Anfurrutia, O. Diaz, and S. Trujillo. Draft under Review. November 2006 [ADT06a].

The first publication describes the ATS feature refactoring case and introduces XAK partially. The second describes a different subject (introduced in chapter 7) where ATS extensions were used. The third is a draft on XAK specifically.

# Chapter 4

# SPL need *Exogenous* Transformations

"If you are out to describe the truth, leave elegance to the tailor."
*– Albert Einstein.*

## 4.1 Abstract

Chapter 2 introduced *Model Driven Development* (MDD) where higher-level abstract models are transformed into implementation artifacts. Such transformations are *exogenous transformations* that map models expressed in different languages. Doing so, development shifts from code implementation to modeling.

The previous chapter introduced how *endogenous transformations* are used to synthesize SPL programs from features. However, it is not evident how to achieve such transformations that realize a feature. An alternative is to use exogenous transformations in order to generate endogenous transformations. Doing so, it raises abstraction level in feature realization.

Specifically, this chapter covers how to generate the realization of a base program (i.e., a constant feature for other function features). How to generate these function features will be described in the next chapter. Overall, this chapter paves the way for the combination of SPL and MDD together (introduced in the next chapter).

Shortly, we will see that our work exposes no specific MDD contributions. Our MDD contribution (if any) is an approach for modeling Portlets. This chapter begins with the rationale for exogenous transformations. *Model transformations* are then described to go into details of the approach we propose, which is illustrated with a case study.

## 4.2 Rationale for *Exogenous* Transformations

*Exogenous transformations* are transformations between models expressed using different languages [MG06b]. Particularly, we are interested in those transformations that can be used to map from a source model (e.g., language A) to a target model (e.g., language B).

This scenario is common in MDD where higher-level models (e.g., PIM in MDA) are transformed into lower-level models (e.g., PSM in MDA), and eventually into code [OMG03]. Hence, techniques to realize such transformations are central to MDD.

This need increases when shifting towards an SPL scenario because an SPL represents not only one, but a number of programs. The key idea is to reuse transformations in building multiple SPL programs. Consequently, significant leverage is achieved because the benefits obtained using MDD (e.g., productivity gain) would not only encompass one, but all the product-line programs.

In this direction, this chapter describes specifically how to generate the realization of a base program (i.e., constant feature). It proposes the use of exogenous transformations to get the base program implementation (i.e., constant feature realization). Doing so, the realization of the base program is expressed as models from which implementation is obtained. We begin with a review of model transformations.

## 4.3 Model Transformations

Model transformations are central of MDD because they turn the use of models for *drawing* into a more extensive model-driven usage where *implementations* are directly obtained [SK03]. We begin with a review of *model, metamodel* and *transformation* concepts.

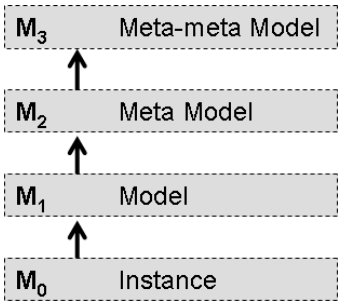



**Figure 4.1**: OMG four-layers stack

### 4.3.1 Models

"*Ceci n'est pas une pipe*" is a French sentence painted in simple script on ancient Magritte's painting [Mag29]. It states that "*this is not a pipe*", but a drawing (model) of a pipe. Likewise, "*do not take the map for the reality*" (sorry for the lack of proper Chinese characters) is a Chinese proverb by Sun Tse stating the same idea.

Bézivin uses in MDD parlance the phrase "*everything is a model*" [Béz05], which is reminiscent of the 1980s object-orientation (OO) phrase "*everything is an object*", to convey a similar idea where every real object can have a model counterpart.

Model is a term widely used in several fields with slightly different meanings. Based on the discussion [Kur05], a model *"represents a part of the reality called the object system and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system"*.

### 4.3.2 Metamodels

A model is frequently considered an instance conforming a meta-model. Based on [Kur05], a meta-model *"is a model of a modeling language"* where the language is specified.

According to [Béz01], there is a four-layers modeling stack where (see Figure 4.1):

- $M_0$ level is the instance of the model (on the bottom of Figure 4.1). $M_0$ corresponds to the execution of a program (e.g., Java instance).

- $M_1$ level corresponds to the definition of the model. $M_1$ is the program's definition (e.g., Java program). An undetermined number of $M_0$ executions may exist for each $M_1$.

- $M_2$ level is the meta-model (e.g., Java grammar). This level states how models are created.

- $M_3$ level is the meta-meta-model (e.g., the EBNF[1]). For instance, the UML meta-model is in $M_3$ level.

### 4.3.3 Transformations

Based on OMG [OMG03], a model transformation "*is the process of converting one model to another model of the same system*". In general, a model transformation is the process of automatic generation from a source model to a target model, according to a transformation definition, which is expressed in a model transformation language [Kur05].

According to Kurtev [Kur05], there are three types of model transformations:

- **Refactoring** transformations reorganize a model based on some precisely defined criteria. The output refactored model is a slightly versioned model. For instance, this type comprises renaming or other simple refactorings.

- **Model-to-model** transformations convert one model to another model (i.e., translation from a source to a target model). A typical transformation is from a level of abstraction to another level (lower or higher depends on direction). Frequently, this transformation is not only unidirectional, but it is possible for some transformations to be backwards (e.g., from PSM to PIM). Doing so, the transformation becomes bidirectional.

- **Model-to-code** transformations convert models into text. This text is usually a source code fragment not limited to OO languages (e.g., C++ or Java), but to any sort of text. This capability is broadly available in existing tooling (a.k.a., code generators [CE00]).

*Model transformations* can be specified in different languages such as XSL [W3C05], QVT [OMG05a], ATL [BDJ+03], RubyTL [CMT06], and MISTRAL [Kur05]. The level of automation in transformations ranges from fully automated to manual

[1] http://en.wikipedia.org/wiki/Extended_Backus-Naur_form

where human intervention is required [CH03, MG06b]. It depends on the domain at hand, and on the expressivity of the models.

## 4.4 *PMDD*: Model Driven Development of Portlets[2]

*Portlet MDD (PMDD)* is a model-driven approach that automates portlet implementation. This section introduces PMDD together with a case study where exogenous transformations are used in the development of an individual application.

### 4.4.1 Approaching

Existing tool support to realize model transformations was introduced in section 2.3.6. In general, these approaches provide assorted model, metamodel and transformation representations. The key is to have a language to represent models, a language to represent metamodels (i.e., models should conform to metamodels), and a language to represent transformations between models.

The approach we took does not differ from this pattern. Models are represented in *Unified Modeling Language* (UML) [OMG05b]. Doing so, modeling languages should be defined using the *Meta Object Facility* (MOF) [OMG06]. This choice allows us to serialize model content to the *XML Metadata Interchange* (XMI) [OMG05c]. Doing so, models are represented using *eXtensible Markup Language* (XML) [W3C06a].

The transformations of models are not described using a specific model transformation language (e.g., QVT: *Queries/Views/Transformations* [OMG05a]). Instead, a general transformation language was selected: *eXtensible Scripting Language* (XSL) [W3C05].

The selection of XSL as the transformation language can be criticized by others arguing for specific benefits of model transformation languages. However, the selection of XSL was not trivial. XSL is represented using XML. This convenience enables *endogenous* and *exogenous* transformations to be represented and computed equally. Doing so, an input model in XML is transformed to an output XML using an XSL exogenous transformation, and similarly a base XML is composed with a refinement XML using XAK endogenous transformations.

Alternatively, it would be possible to use other transformation languages. However, our results would not change because XSL is only the language to specify the

[2]The core of this chapter comes from our ICSE 2007 paper [TBD07].

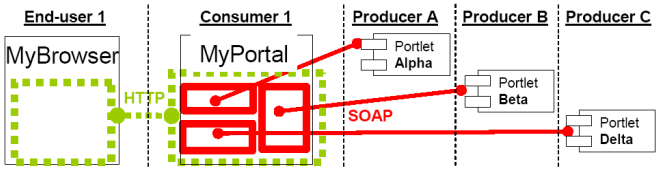



**Figure 4.2**: A 3-Tier Architecture for Portlets

logic of the exogenous transformation.

We begin revisiting Portlets and presenting a case study. The steps of PMDD approach to develop a Portlet application from models are then detailed.

### 4.4.2 Revisiting Portlets

A *portal* is a web page that provides centralized access to a variety of services [DR04]. An increasing number of these services are not offered by the portal itself, but by a third-party component called a *portlet*. Figure 4.2 depicts a 3-tier architecture for portlets, where an end-user's *MyBrowser* accesses the *MyPortal* page through HTTP. *MyPortal* is hosted by *Consumer_1* and consists of a layout aggregating (through SOAP [W3C03]) the *Alpha*, *Beta*, and *Delta* portlets that are hosted by different producers.

Unlike web services, which offer only business logic methods, portlets additionally provide a presentation-oriented web service. Hence, portlets not only return raw data but also renderable markup (e.g. XHTML) that can be displayed within a portal page.

Until recently, portlet realization was dependent on the infrastructure of the producer of portlets (service provider) and the portal consumer (service container). This changed with the release of the *WSRP* [OAS03] and the *JSR 168* [JCP03]. These standardization efforts foster a COTS market, where portlets can be deployed independently of the platform on which they were developed. Furthermore, different customers demand different portlets that overlap in functionality. Consequently, techniques for customizing portlets are increasingly sought.

Our experience with portlet implementations is that a sizable fraction of their code is common. This led us to create an OO framework (using eXo portal platform[3]) that is realized by 85 classes and 9 KLOC Java. It encapsulates and reuses logic and infrastructure common to all portlets and provides the base functionality

[3] Exo Portal Platform. http://www.exoplatform.com/

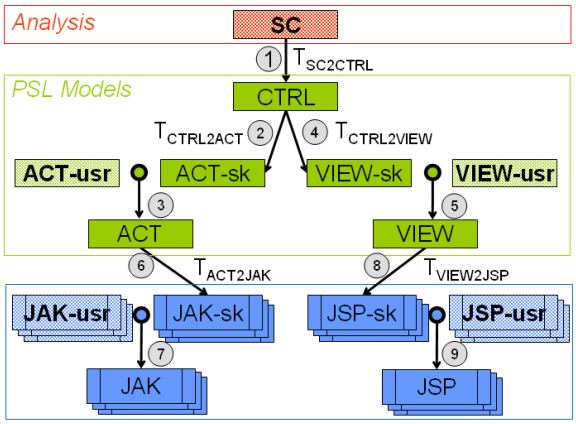



**Figure 4.3**: PMDD Process

on top of which application-specific functionality is built. We created a *Domain Specific Language for Portlets (PSL)* to define this functionality. A PSL specification is represented by several XML documents. Next sections describe how portlets can be synthesized using PMDD.

### 4.4.3 A Case Study: *PinkCreek*

*PinkCreek* is a portlet that provides *flight reservation capabilities* to different portals. Its functionality is roughly: *(i)* search for flights, *(ii)* present flight options, *(iii)* select flights, and *(iv)* purchase tickets (appendix A describes the complete case study).

### 4.4.4 Big Picture

PMDD is a model-driven approach that provides models and transformations in order to automate portlet implementation. Figure 4.3 represents the transformations from higher-level models to implementation. It sketches the overall process where *(i)* rectangles represent models (those with shady background are input models), and *(ii)* directed arrows are transformations (that generate remaining models from input). Transformations are numbered to denote the sequence to create portlets (see circles in Figure 4.3). The following sections describe the steps to create PinkCreek using PMDD.

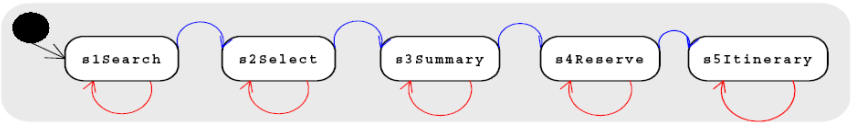



**Figure 4.4**: SC Diagram Fragment for PinkCreek

```
<scxml version="1.0" initialstate="s1Search">
    <state id="s1Search">
        <transition event="actionEvent">
            <target next="s1Search"/>
            <action>LoadAirport</action>
        </transition>
        <transition event="viewEvent">
            <target next="s2Select"/>
            <action>SearchFlightView</action>
        </transition>
    </state>
    <!-- remaining states omitted -->
</scxml>
```

**Figure 4.5**: SC.xml Fragment for PinkCreek

### 4.4.5 Step 1: Define Portlet Controller

A *State Chart* (**SC**) provides a platform independent model for representing the flow of computations in a portlet [dOTM01, Har87, OMG05b]. Each portlet consists of a sequence of *states* where each *state* represents a portlet page. *States* are connected by *transitions* whose handlers either execute some action, render some view, or both.

Figure 4.4 shows an SC diagram fragment for *PinkCreek* where each state represents a step in making a flight reservation. We removed transition details to make the figure clearer. Existing tools (e.g., IBM Rational Rose, Poseidon) can draw an SC model in UML notation, and serialize the model to an XMI specification [OMG05c]. From this XMI specification, a state chart in the W3C SCXML language can be obtained [W3C06b]. Figure 4.5 lists a fragment of the *PinkCreek* specification.

The SC of a portlet defines its controller; the details of actions and views are defined elsewhere. So the first step is to define the SC of the portlet's controller. The next step is to map an SC specification to a PSL specification.

```
<Portlet.actions id="PinkCreek">
    <Action id="LoadAirport" type="JAVA_CLASS">
        <class>pinkcreek.LoadAirport</class>
        <!-- TODO Params -->
        <!-- TODO Results -->
    </Action>
    <!-- remaining actions omitted -->
</Portlet.actions>
```

**Figure 4.6**: PSLact-sk.xml Skeleton ($PSL_{act-sk}$) for PinkCreek

```
<xak:refines id="PinkCreek">
    <xak:extends id="LoadAirport">
        <xak:super id="LoadAirport"/>
        <Param name="from" value="$data/orgn"/>
        <Param name="to" value="$data/dest"/>
        <!-- remaining Params omitted -->
        <Result name="flight"/>
        <!-- remaining Results omitted -->
    </xak:extends>
</xak:refines>
```

**Figure 4.7**: PSLact-usr.xml User Refinement ($\triangle PSL_{act-usr}$) for PinkCreek

### 4.4.6 Step 2: Map SC to PSL

A PSL specification of a portlet consists of three distinct XML documents. One document ($PSL_{ctrl}$) defines a state machine controller for a portlet. The other two documents ($PSL_{act}$, $PSL_{view}$) define the actions to be performed and the views to be rendered during the controller execution. The production of each is described in the following sections.

**Step 2.1: Transforming *SC* to $\mathbf{PSL_{ctrl}}$** We designed the PSL controller language prior to the release of the SCXML standard. As we now use the standard, we reused our PSL language and framework by writing an XSL transformation ($T_{sc2ctrl}$) that maps a state chart specification *SC* to a PSL controller document ($PSL_{ctrl}$):

$$PSL_{ctrl} = T_{sc2ctrl}(SC) \qquad (1)$$

The $PSL_{ctrl}$ document is interpreted by our Portlet framework at portlet execution time.

**Step 2.2: Transforming $\mathbf{PSL_{ctrl}}$ to $\mathbf{PSL_{act}}$** Activities in a state chart are defined as actions to perform and views to render when a transition occurs. An *action skeleton* is an interface that defines only the names of action methods. (We will see

shortly that there is a corresponding view skeleton that defines the names of view methods). An action skeleton ($PSL_{act-sk}$) is derived by a simple analysis of the $PSL_{ctrl}$ document that extracts action names. The transformation ($T_{ctrl2act}$) implements this derivation:

$$PSL_{act-sk} = T_{ctrl2act}(PSL_{ctrl}) \qquad (2)$$

Figure 4.6 shows a snippet of $PSL_{act-sk}$ skeleton, where the name of action *pinkcreek.LoadAirport* was extracted[4].

The name of an action is not sufficient: we still need to specify the input (*Params*) and output (*Results*) of each action method. This information is added by (locally) refining the generated $PSL_{act-sk}$ document. Figure 4.7 shows a snippet of such a refinement (denoted $\triangle PSL_{act-usr}$) that extends the *pinkcreek.LoadAirport* method name with its input parameters and output result. XAK is the tool we used to compose an XML file with its refinements (described in section 3.4.2).

Composing the generated action skeleton ($PSL_{act-sk}$) with its user hand-written refinement ($\triangle PSL_{act-usr}$) yields a complete PSL action document ($PSL_{act}$), which defines the name, type, and parameters of each action method:

$$PSL_{act} = \triangle PSL_{act-usr} \bullet PSL_{act-sk} \qquad (3)$$

**Step 2.3: Transforming $\mathbf{PSL_{ctrl}}$ to $\mathbf{PSL_{view}}$** An identical procedure is used to create a PSL view document ($PSL_{view}$) from the PSL controller document ($PSL_{ctrl}$). A view skeleton ($PSL_{view-sk}$) is generated from $PSL_{ctrl}$, and it is composed with a hand-written refinement ($\triangle PSL_{view-usr}$) that refines view methods with their input parameters, to yield the desired view document[5]:

$$PSL_{view-sk} = T_{ctrl2view}(PSL_{ctrl}) \qquad (4)$$
$$PSL_{view} = \triangle PSL_{view-usr} \bullet PSL_{view-sk} \qquad (5)$$

At this point, we have a PSL specification for a portlet. However, more platform-specific implementation details remain to be given[6].

[4] For sake of clarity, this code was simplified. Note that references to *xak:modules* are omitted on purpose.

[5] View methods return no results.

[6] $\triangle PSL_{act-usr}$ and $\triangle PSL_{view-usr}$ are platform-specific, as the parameters to actions and views are not platform invariant. Alternatively, some MDD approaches define code in a platform-independent language and translate code to a platform-specific language [KWB03].

```
import java.util.Hashtable;
import org.onekin.pf.action.jc.IJavaAction;
public class LoadAirport implements IJavaAction
{
 /** This is the default Constructor ... */
 public LoadAirport()
 { /* empty */ }

 /** This method executes ... */
 public void execute(Hashtable prm,Hashtable rs)
 { /* empty */ }
}
```

**Figure 4.8**: LoadAirport.jak for PinkCreek

```
refines class LoadAirport {
 public LoadAirport()
 { /* USER CODE GOES HERE */ }

 public void execute(Hashtable prm,Hashtable rs)
 { /* USER CODE GOES HERE */ }
}
```

**Figure 4.9**: LoadAirport.jak Refinement for PinkCreek

### 4.4.7 Step 3: from PSL to Implementation

A PSL specification almost completely defines what the interpreter needs to execute a portlet. What is lacking is *(i)* business logic of each action method, and *(ii)* the logic to draw the layout page of each view method.

**Step 3.1: Transforming $PSL_{act}$ to Jak Code** *Jak(arta)* is a superset of the Java language (described in section 3.4.1), where class and method refinements can be declared [BSR04]. Jak is the primary language for implementing refinements in FOP.

$PSL_{act}$ is an XML document that sketches the source code skeleton of a set of Jak classes: it specifies the signatures of all portlet-specific methods. We can generate skeletal Jak classes ($Jak_{sk}$) by a transformation ($T_{act2jak}$) of $PSL_{act}$. A unique *Jak* class is generated for each action in $PSL_{act}$. Also generated are portlet-specific methods and members that are required by our portlet framework.

$$Jak_{sk} = T_{act2jak}(PSL_{act}) \qquad (6)$$

Figure 4.8 shows the derived *Jak* code for the action *pinkcreek.LoadAirport*. Note that extra methods (e.g., *execute*) that are specific to our portlet framework are also produced, along with additional data members (not shown). Their generation simplifies the development of user code provided in the next step.

Jak code is generated instead of Java because the actions of the generated methods must be completed by a programmer. We complete the generated skeleton ($Jak_{sk}$) by composing it with a hand-written refinement ($\triangle Jak_{usr}$ ) that encapsulates the business logic for each method:

$$Jak_{code} = \triangle Jak_{usr} \bullet Jak_{sk} \qquad (7)$$

Figure 4.9 shows the corresponding refinement for Figure 4.8.

**Step 3.2: Transforming $PSL_{view}$ to JSP** In an analogous manner, JSP code skeletons ($Jsp_{sk}$) are created from $PSL_{view}$, one JSP page per view. Each skeleton is completed by composing it with a hand-written (local) refinement ($\triangle Jsp_{usr}$). The result is a compilable set of JSP files ($Jsp_{code}$), one per $PSL_{view}$ view method:

$$Jsp_{sk} = T_{view2jsp}(PSL_{view}) \qquad (8)$$
$$Jsp_{code} = \triangle Jsp_{usr} \bullet Jsp_{sk} \qquad (9)$$

### 4.4.8 Step 4: Building the Program

A PSL specification together with $Jak_{code}$ and $Jsp_{code}$ form the *raw material* of a Portlet program ($P_{raw}$). Other artifacts ($\triangle P_{additional}$) are needed, such as deployment descriptors and *JAR* libraries, to complete a portlet's source ($P_{src}$). These artifacts are introductions (i.e., new artifacts that do not refine existing ones). They are added by composing $\triangle P_{additional}$ to $P_{raw}$:

$$P_{raw} = \{PSL_{ctrl},\ PSL_{act},\ PSL_{view},\ Jak_{code},\ Jsp_{code}\} \qquad (10)$$
$$P_{src} = \triangle P_{additional} \bullet P_{raw} \qquad (11)$$

Among the artifacts added by $\triangle P_{additional}$ is an *ant* makefile [SC04], which builds the *web archive (WAR)* of the portlet. Executing the makefile translates *Jak* files to *Java* files, compiles Java files into class files, and creates the portlet web archive ($P_{war}$) which is deployed into the target portal (see section 7.4.2).

$$P_{war} = antBuild(P_{src}) \qquad (12)$$

### 4.4.9 Recap and Perspective

Our PMDD process for building *PinkCreek* (and other portlets) is a straightforward metaprogram (see Figure 4.10). Given all the inputs (i.e., MDD models) that define

```
T_mkraw(SC,ΔPSL_act-usr,ΔPSL_view-usr,ΔJak_usr,ΔJsp_usr)
{   PSL_ctrl = T_sc2ctrl (SC);                  // (1)
    PSL_act-sk = T_ctrl2act (PSL_ctrl);         // (2)
    PSL_act = ΔPSL_act-usr • PSL_act-sk;        // (3)
    PSL_view-sk = T_ctrl2view (PSL_ctrl);       // (4)
    PSL_view = ΔPSL_view-usr • PSL_view-sk;     // (5)
    Jak_sk = T_act2jak (PSL_act);               // (6)
    Jak_code = ΔJak_usr • Jak_sk;               // (7)
    Jsp_sk = T_view2jsp (PSL_view);             // (8)
    Jsp_code = ΔJsp_usr • Jsp_sk;               // (9)
    P_raw = { PSL_ctrl, PSL_act, PSL_view,
              Jak_code, Jsp_code };             // (10)
    return P_raw; }
T_raw2war(P_raw,ΔP_additional)
{   P_src = ΔP_additional • P_raw               // (11)
    P_war = antBuild (P_src)                    // (12)
    return P_war; }
T_sc2war(SC,ΔPSL_act-usr,ΔPSL_view-usr,ΔJak_usr,ΔJsp_usr)
{   P_raw = T_mkraw(SC,ΔPSL_act-usr,ΔPSL_view-usr,
              ΔJak_usr,ΔJsp_usr);
    return T_raw2war(P_raw); }
```

**Figure 4.10**: Metaprograms for PMDD

a portlet ($SC$, $\triangle PSL_{act-usr}$, $\triangle PSL_{view-usr}$, $\triangle Jak_{usr}$, $\triangle Jsp_{usr}$) the process automatically generates the portlet's WAR ($P_{war}$). Model (local) refinements are expressed as endogenous transformations, and model derivations are exogenous transformations [MG06b]. Figure 4.10 shows this metaprogram as three functions ($T_{mkraw}, T_{raw2war}, T_{sc2war}$). (The reason why we used three functions, instead of one, will become clear in section 5.4). $T_{sc2war}$ automates significant and tedious tasks in portlet development. For example, 59 files and 4.250 LOC are derived from an input of 10 files and 730 LOC.

Of the five inputs that we need to specify, only one (the statechart) is platform-independent. The remaining are platform-specific, expressing customized business logic and view logic. Ideally, these remaining inputs should be derived from one or more PIMs, which would marginally alter the metaprogram of Figure 4.10. Although we do not yet have such PIMs, this does not impact the results of this work. In general, our situation is symptomatic of a general problem in MDD on how to express customized business logic in PIMs. It is common to use model *escapes* from which code can be specified. Sometimes a generic programming language is used to express code fragments, from which Java or C# is produced [KWB03]. Creating declarative models for all PMDD inputs seems unlikely.

## 4.5 Contributions

The use of MDD to get an SPL feature exposes a different nature in which models can be operated: *refinement* and *transformation*. Transformations typically involve a PIM model being converted into a PSM, whereas refinement is the gradual change of a model to better match the desired system. In general, model derivation refers to *Wirth's* refinement [Wir71] and model refinement to *Parnas'* extensions [Par79]. Refer to [Ape07] for a complete discussion.

In this chapter, the term *refinement* is used with a slightly different meaning from the previous chapter 3. In FOP context, *refinement* was used to represent a refinement *function* that increments in (feature) functionality [BSR04]. Alternatively, in the context of MDD, it is referred to a specification that is gradually refined into a full-fledged implementation by means of successive concrete steps that *add more concrete details* [MG06b]. For sake of clarification, in this work the latter is denoted as a *local refinement*. Although both are similar, it is important to clarify that each is intended for a different purpose in different contexts. Refinement is adding feature functionality, whereas local refinement is completing code (e.g., when skeletal code is completed with user hand-written code). This difference would become clearer in chapter 6.

This chapter introduced the need for exogenous transformations that SPL demand. As mentioned at the beginning, this chapter provided *no* specific MDD contributions. The contribution (if any) in this chapter was an approach to Portlet Model Driven Development (PMDD) where a sample case Portlet application was developed, and where some real benefits were reported. PMDD paves the way for the insights of the next chapter.

Parts of the work described in this chapter has been presented before.

1. *Enhancing Decoupling in Portlet Implementation*. S. Trujillo, I. Paz and O. Díaz. 4th International Conference on Web Engineering (ICWE 2004). Munich, Germany. July 2004 [TPD04]. Acceptance Rate: 12% (25+60/204)[7].

2. *Feature Oriented Model Driven Development: A Case Study for Portlets*. S. Trujillo, D. Batory and O. Diaz. 29th International Conference on Software Engineering (ICSE 2007). Minneapolis, Minnesota, USA. May 2007 [TBD07]. Acceptance Rate: 15% (50/334).

[7] Our contribution was accepted as poster where 60 research papers were included, as either short papers or posters.

The first publication describes (partially) the domain specific language we used to specify Portlets (a.k.a., PSL), whereas the second describes the model-driven approach for developing Portlets (a.k.a., PMMD) using a working example (together with further content introduced in the next chapter).

# Chapter 5

# Combining *Endogenous* and *Exogenous* Transformations[1]

"There is hope for people like us."
– *Anonymous.*

## 5.1 Abstract

*Model Driven Development* (MDD) is an emerging paradigm for software construction that uses models to specify programs, and exogenous model transformations to synthesize executables. *Feature Oriented Programming* (FOP) is a paradigm for software product lines where programs are synthesized by composing features using endogenous transformations. *Feature Oriented Model Driven Development* (FOMDD) is a blend of FOP and MDD that shows how programs in a software product line can be synthesized in an MDD way by composing models from features, and then transforming these models into executables. We present a case study of FOMDD on a product line of portlets, which are components of web portals. We reveal mathematical properties (i.e., commuting diagrams) of portlet synthesis that helped us to validate the correctness of our abstractions, tools, and specifications, as well as optimize portlet synthesis.

[1]The core of this chapter comes from our ICSE 2007 paper [TBD07].

## 5.2 Rationale for *Combination*

*Model Driven Development (MDD)* is an emerging paradigm for software development that specifies programs in *domain-specific languages (DSLs)*, encourages greater degrees of automation, and exploits standards [Béz05, BBI+04, KWB03]. MDD uses models to represent a program. A model is written in a DSL that specifies particular details of a program's design. As an individual model captures limited information, a program is often specified by several different models. A model can be derived from other models by exogenous transformations, and program synthesis is the process of transforming high-level models into executables (which are also models).

*Feature Oriented Programming (FOP)* is a paradigm for software product lines where programs are synthesized by composing features [BSR04]. A *feature* is an increment of program functionality. It is implemented by refinements that extend existing artifacts (by means of endogenous transformations), and by introductions that add new artifacts (code, makefiles, documentation, etc.). When features are composed, consistent artifacts that define a program are synthesized. A tenet of FOP is the use of algebraic techniques to specify and manipulate program designs.

*Feature Oriented Model Driven Development (FOMDD)* is a blend of FOP and MDD. Models can be *refined* by composing features (a.k.a., *endogenous transformations* that map models expressed in the same DSL [MG06b]), and can be *derived* from other models (a.k.a., *exogenous transformations* that map models written in different DSLs [MG06b]).

We present a case study of FOMDD that is a product-line of portlets, which are building blocks of web portals. We explain how we specify a portlet as a set of models from which we refine and derive an implementation. Combining model *derivation* and model *refinement* in FOMDD exposes a fundamental commuting relationship; namely, the transformation of a composed model equals the composition of transformed models. Hence, an executable can be synthesized in very different ways. Commuting relationships impose stringent properties on our domain model and implementation, and have helped us to validate the correctness of our abstractions, tools, and portlet specifications, as well as optimize portlet synthesis. We begin revisiting background.

## 5.3 Revisiting MDD and FOP

**Model Driven Development** Program specification in MDD uses one or more *models* to define a target program. Ideally, these models are *platform independent (PIM)*. Model *derivations* convert platform independent models to *platform specific models (PSM)*, where assorted technology bindings are introduced. Possible results of transforming PIMs can be an executable or an input to an analysis tool, where both an executable and an analysis-input file are themselves considered models.

**Feature Oriented Programming** FOP is a paradigm for creating software product lines [BSR04]. Features (a.k.a., feature modules) are the building blocks of programs. An *FOP model* of a product line is an algebra that offers a set of operations, where each operation implements a feature. We write $M = \{f, h, i, j\}$ to mean model $M$ has operations or features $f$, $h$, $i$ and $j$. FOP distinguishes features as *constants* or *functions*. Constants represent base programs. Functions represent *program refinements* that extend a program that is received as input. The design of a program is a named expression, e.g.: $prog_2 = i \bullet j \bullet h$. Program $prog_2$ has features *h*, *j*, and $i$.

**Metaprogramming** is the concept that program development is a computation. Batory assert that MDD is a metaprogramming paradigm [Bat06]. That is, models are values and transformations are functions that map these values. Scripts that transform models into executables are *metaprograms* (i.e., programs that manipulate values that themselves are programs). For example, *ant* makefiles are metaprograms; the values of a makefile are files (i.e., programs) and the execution of a makefile can produce an executable [Fou]. An MDD process can be written as a makefile (metaprogram) whose input values are DSL specifications (i.e., models) of target programs, and whose output values are synthesis targets (examples of such metaprograms in section 4.4). The connection of FOP to metaprogramming and MDD is simple: FOP treats programs as values, and features are functions that map values. In section 5.4, we show how FOP and MDD can be integrated. We briefly recap on our case study.

**A Case Study** *PinkCreek* is presented in this chapter as a *product-line of portlets* that provides flight reservation capabilities to different portals. PinkCreek con-

sisted of up to 20 features, yielding hundreds of distinct Portlet products (section A.3 describes the complete approach).

## 5.4 Feature Oriented MDD

*Portlet MDD (PMDD)* was introduced in section 4.4 as a model-driven approach that automates portlet implementation. The input to our PMDD process is a 5-tuple $< SC, \triangle PSL_{act-usr}, \triangle PSL_{view-usr}, \triangle Jak_{usr}, \triangle Jsp_{usr} >$, which we abbreviate as <$s$, $a$, $v$, $b$, $j$ >. Given a tuple that defines a portlet, the transformation $T_{sc2war}$ synthesizes the portlet's *WAR* file.

Portlets are like other software applications: there is a family of related portlet designs and capabilities that we want to create. The designs and capabilities that differentiate one portlet from another can be explained in terms of features (i.e., increments in portlet functionality). Instead of manually creating portlet specifications (i.e., 5-tuples), we want to synthesize their 5-tuples using FOP.

An FOP model of a portlet domain includes one or more base portlets called *constants*, and one or more refinements, called *functions*, that add functionality to a portlet (section 5.3). A constant *C* is a 5-tuple $< s, a, v, b, j >$. A function is also a 5-tuple <$\triangle s$, $\triangle a$, $\triangle v$, $\triangle b$, $\triangle j$ > that defines changes to a base tuple in terms of:

- refinements to a base *s*tate chart ($\triangle s$ )
- refinements to a base *a*ction document ($\triangle a$)
- refinements to a base *v*iew document ($\triangle v$)
- refinements to a base actions *b*usiness logic ($\triangle b$)
- refinements to a base *j*sp page ($\triangle j$)

Suppose we want to synthesize the 5-tuple <*s, a, v, b, j*> of a portlet *P* by starting with a base portlet *C* (a constant) and refining it by the features (functions) *F1* and *F2*. Our portlet specification *P* is:

$$
\begin{aligned}
&P = F2 \bullet F1 \bullet C \\
&= < \triangle s_2, \triangle a_2, \triangle v_2, \triangle b_2, \triangle j_2 > \bullet < \triangle s_1, \triangle a_1, \triangle v_1, \triangle b_1, \triangle j_1 > \bullet \\
&\quad < s, a, v, b, j > \\
&= <\triangle s_2 \bullet \triangle s_1 \bullet s, \triangle a_2 \bullet \triangle a_1 \bullet a, \triangle v_2 \bullet \triangle v_1 \bullet v, \triangle b_2 \bullet \triangle b_1 \bullet b, \triangle j_2 \bullet \triangle j_1 \bullet j >
\end{aligned}
$$

$$P = F2 \bullet F1 \bullet C =< s, a, v, b, j > \textit{(13)}$$

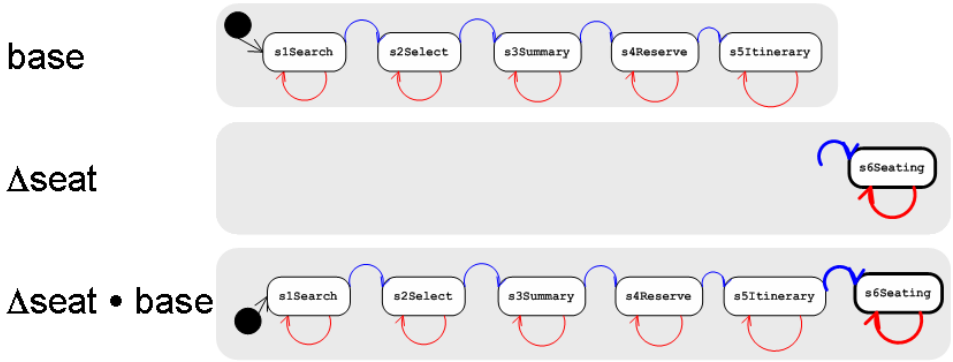



**Figure 5.1**: SC Refinement for PinkCreek seating

That is, the *5-tuple* of our desired portlet is synthesized by composing the base state chart with its refinements, the base action document with its refinements, and so on [BSR04]. In this section, we explain the interesting challenges we faced in developing portlet features in a model-driven way. It required an extension of PMDD to cope with product lines.

### 5.4.1 Developing Feature Constant

A feature constant is developed as a stand-alone portlet (described previously in section 4.4). It is defined by a 5-tuple and represents a base portlet to which more features can be added.

### 5.4.2 Developing Feature Functions

**Challenge 1: Model Refinement** A model in MDD is a specification of a program (or some part or view of a program). Model refinement elaborates a model to reflect the changes made by adding a feature.

A state chart refinement[2] adds new states, new transitions, and refines the activities associated with existing states or transitions [Bea02, LXK98, MS03]. For example, consider a feature *seat* that extends the *PinkCreek* portlet to allow passengers to select their seat after purchasing flight tickets. Figure 5.1 shows how a new state *s6Seating* is added by feature *seat* to handle the seat selection pages, and how state *s5Itinerary* is refined by linking it with *s6Seating*.

[2]Note that a refinement is based on mixin inheritance. Weber et al. extend statecharts by regular inheritance [WM98].

```
T'mkraw(ΔFsc,ΔFact-usr,ΔFview-usr,ΔFjak-usr,ΔFjsp-usr)
{
     ΔFctrl = T'sc2ctrl (ΔFsc);                    // (1)
     ΔFact-sk = T'ctrl2act (ΔFctrl);               // (2)
     ΔFview-sk = T'ctrl2view (ΔFctrl);             // (3)
     ΔFact = ΔFact-usr • ΔFact-sk;                 // (4)
     ΔFview = ΔFview-usr • ΔFview-sk;              // (5)
     ΔFjak-sk = T'act2jak (ΔFact);                 // (6)
     ΔFjsp-sk = T'view2jsp (ΔFview);               // (7)
     ΔFjakcode = ΔFjak-usr • ΔFjak-sk;             // (8)
     ΔFjspcode = ΔFjsp-usr • ΔFjsp-sk;             // (9)
     ΔFraw = {ΔFctrl,ΔFact,ΔFview,
              ΔFjakcode,ΔFjspcode}                 // (10)
     return ΔFraw;
}
```

**Figure 5.2**: The Metaprogram $T'_{mkraw}$

A state chart is defined by an XML document. A refinement of a state chart can also be defined in an XML document. Doing so, XAK can compose such documents (similarly to section 3.4.2).

**Challenge 2: Transforming Refinements** Recall that a function feature is a 5-tuple of refinements <$\triangle s$, $\triangle a$, $\triangle v$, $\triangle b$, $\triangle j$ >. Defining the changes to a state chart is easy. However, defining the remaining artifact refinements is a bit harder. This section presents the approach that we took to develop feature function 5-tuples.

Recall from section 4.4 that $P_{raw}$ is the raw material from which we could build a portlet WAR file. We want to generate a raw material refinement $\triangle F_{raw}$ for each feature *F* that can be added to a base portlet. If we could do so, we could synthesize the raw material for a target portlet. For example, suppose we want to synthesize the raw material for portlet $P = F2 \bullet F1 \bullet C$ by composing the raw material ($C_{raw}$) for base feature *C* and the raw material ($\triangle F1_{raw}$ and $\triangle F2_{raw}$) of its feature functions *F1* and *F2*. The raw material for portlet *P* would be:

$$P_{raw} = \triangle F2_{raw} \bullet \triangle F1_{raw} \bullet C_{raw}$$

To accomplish this, we had to derive the raw material refinements for each feature function. More precisely, let feature function *F* be defined by the 5-tuple $<$ $\triangle s$, $\triangle a$, $\triangle v$, $\triangle b$, $\triangle j$ $>$. We want a transformation $T'_{mkraw}$ that maps the 5-tuple of any feature function *F* to its raw material $\triangle F_{raw}$. The details of the $T'_{mkraw}$ process are given next and are virtually identical to the metaprogram of

Figure 5.2 that maps a tuple $< s, a, v, b, j >$ to raw materials. Let feature function *F* be defined by the tuple:

$$< \triangle F_{sc}, \triangle F_{act-usr}, \triangle F_{view-usr}, \triangle F_{jak-usr}, \triangle F_{jsp-usr} >$$

We can map F to a $\triangle F_{raw}$ in the following way. First, we define a new transformation ($T'_{sc2ctrl}$) that maps a refinement of a state chart ($\triangle F_{sc}$) to a refinement or delta of a PSL controller ($\triangle F_{ctrl}$):

$$\triangle F_{ctrl} = T'_{sc2ctrl}(\triangle F_{sc}) \quad (1)$$

Second, we need other transformations to map $\triangle F_{ctrl}$ to an action skeleton delta ($\triangle F_{act-sk}$) and a view skeleton delta ($\triangle F_{view-sk}$):

$$\triangle F_{act-sk} = T'_{ctrl2act}(\triangle F_{ctrl}) \quad (2)$$
$$\triangle F_{view-sk} = T'_{ctrl2view}(\triangle F_{ctrl}) \quad (3)$$

Third, we composed the action skeleton delta ($\triangle F_{act-sk}$) computed above with a hand-written refinement ($\triangle F_{act-usr}$) to yield a complete PSL action delta ($\triangle F_{act}$). The same applies to producing a complete PSL view delta ($\triangle F_{view}$) by composing its skeleton and hand-written refinement:

$$\triangle F_{act} = \triangle F_{act-usr} \bullet \triangle F_{act-sk} \quad (4)$$
$$\triangle F_{view} = \triangle F_{view-usr} \bullet \triangle F_{view-sk} \quad (5)$$

Given these deltas ($\triangle F_{act}$, $\triangle F_{view}$), we wrote additional transformations to map them to their delta *Jak* and *Jsp* code skeleton counterparts:

$$\triangle F_{jak-sk} = T'_{act2jak}(\triangle F_{act}) \quad (6)$$
$$\triangle F_{jsp-sk} = T'_{view2jsp}(\triangle F_{view}) \quad (7)$$

and composed them with their code refinements:

$$\triangle F_{jakcode} = \triangle F_{jak-usr} \bullet \triangle F_{jak-sk} \quad (8)$$
$$\triangle F_{jspcode} = \triangle F_{jsp-usr} \bullet \triangle F_{jsp-sk} \quad (9)$$

The delta raw material that a feature *F* adds to its base is:

$$\triangle F_{raw} = \{\triangle F_{ctrl}, \triangle F_{act}, \triangle F_{view}, \triangle F_{jakcode}, \triangle F_{jspcode}\} \quad (10)$$

where the components of $\triangle F_{raw}$ are derived by *(1)-(10)*.

Figure 5.2 shows this process as the metaprogram $T'_{mkraw}$. For a typical feature, the output size is 3-5 times the size of the input. For the *seat* feature in the PinkCreek product line, 27 files (755 LOC) are derived from an input of 9 files and 163 LOC. As in the case of $T_{mkraw}$, $T'_{mkraw}$ automates significant and tedious tasks in portlet development.

### 5.4.3 Program Synthesis

Our *PinkCreek* product line has 26 features (constants and functions), yielding hundreds of interesting and distinct portlets. A particular portlet program is specified by an FOP expression that composes a *base* portlet with zero or more extending features (*seat*, *checkin*, etc.):

$$PinkCreek_1 = seat \bullet base$$
$$PinkCreek_2 = checkin \bullet seat \bullet base$$
... // other products

We synthesized portlets by deriving the raw materials of the base and refining features, and composing them. Let the 5-tuples for *base*, *seat*, and *checkin* be $<>_{base}$, $<>_{seat}$, $<>_{checkin}$. The raw material for $PinkCreek_2$ is computed by:

$$PinkCreek_{raw} = T'_{mkraw}(<>_{checkin}) \bullet T'_{mkraw}(<>_{seat}) \bullet T_{mkraw}(<>_{base}) \quad (11)$$

Given the raw material of a portlet, we invoke the $T_{raw2war}$ transformation of Figure 4.10 to produce the portlet's *WAR*:

$$PinkCreek_{war} = T_{raw2war}(PinkCreek_{raw})$$

## 5.5 Commuting Diagrams

Synthesizing portlets by composing raw materials is not the way we originally planned. Our intent in section 5.4 was to synthesize the 5-tuple of a portlet by composing the *5-tuples* of its base and refining features, such as:

$$<>PinkCreek_2 = <>_{checkin} \bullet <>_{seat} \bullet <>_{base}$$

And use $T_{mkraw}$ of Figure 4.10 to derive portlet raw material:

$$PinkCreek_{2raw} = T_{mkraw}(<> PinkCreek_2) \quad (12)$$

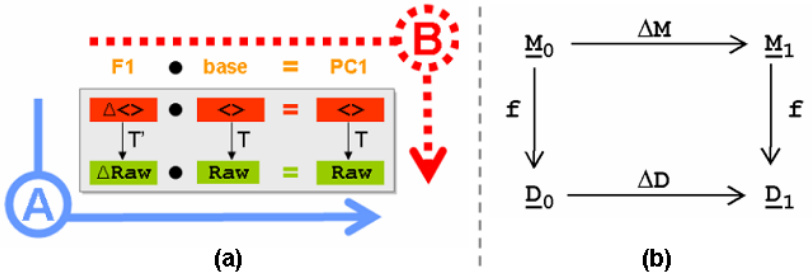



**Figure 5.3**: Alternative Synthesis Path (a) and Commuting Diagram (b)

We now had two different ways to produce the raw materials of a portlet, namely *(11)* and *(12)*. That is, a transformation of a composition of 5-tuples *(12)* equals the composition of the transformation of each 5-tuple *(11)*:

$$T_{mkraw}(<>_{F1} \bullet <>_{base}) = T'_{mkraw}(<>_{F1}) \bullet T_{mkraw}(<>_{base}) \qquad (13)$$

Figure 5.3a illustrates *(13)*: we synthesized portlets via the path labeled *A (11)*, but had an alternate path *B (12)*.

FOMDD explicitly combines model refinement and model derivation. Our research exposed a fundamental relationship between the two, which is expressed in Figure 5.3b as a commuting diagram [Pie91], where $\underline{M}_0$, $\underline{M}_1$, $\underline{D}_0$, $\underline{D}_1$ are domains and $\triangle M : \underline{M}_0 \rightarrow \underline{M}_1$, $\triangle D : \underline{D}_0 \rightarrow \underline{D}_1$ and $f : (\underline{M}_0 \cup \underline{M}_1)\ (\underline{D}_0 \cup \underline{D}_1)$ are functions satisfying:

$$f \bullet \triangle M = \triangle D \bullet f \qquad (14)$$

In PMDD, we encountered instances of these domains: $M_0 \in \underline{M}_0$, $M_1 \in \underline{M}_1$, $D_0 \in \underline{D}_0$ and $D_1 \in \underline{D}_1$. We refined model $M_0$ by $\triangle M$ to produce model $M_1$. Function or transformation $f$ derived model $D_1$ from $M_1$. Alternatively, we could derive model $D_0$ from $M_0$ using function *f*, and then refine $D_0$ by $\triangle D$ (that corresponds to $\triangle M$) to produce $D_1$. An operator *f'* maps function $\triangle M$ to function $\triangle D$. The general relationship is:

$$f(\triangle M \bullet M_0) = f'(\triangle M) \bullet f(M_0) \qquad (15)$$

where *(13)* is a PMDD instance of *(15)* which in our case states that the transformation of a composed model equals the composition of transformed models. Note that no special restrictions are placed on models and features by commuting diagrams, except that *(15)* must hold.

The reason why *(13)* holds is because functions $T_{mkraw}$ and $T'_{mkraw}$ are *morphisms* (i.e., structure preserving mappings [Pie91]). Formally proving structure preservation is difficult, as it requires a formalization of the input and output domains, a formalization of the properties to be preserved, and a faithful implementation of this formalization, each step of which is a non-trivial undertaking. An alternative approach is to validate each *instance* of a transformation. For example, Narayanan and Karsai [NK06] presented an algorithm to validate that the translation between two different state chart representations preserves each state, transition, and activity (and no additional states, transitions, and activities are added). This is accomplished by maintaining an internal mapping between input and output representations and validating that there is a 1-1 correspondence between input/output states, transitions, and activities.

We took a different approach by computing the results in both *directions* (paths *A* and *B*) and used a source equivalence *diff* to test for equality. (*Source equivalence* is syntactic equivalence with two relaxations: it allows permutations of members when member ordering is not significant and it allows white space to differ when white space is unimportant). We added this extra computation as an option to our metaprograms to validate *(13)*.

We soon discovered that there are *many* other commuting diagrams/relationships in PinkCreek. For example, state charts can be refined and then mapped to PSL controllers, or a PSL controller can be derived from a state chart and then refined:

$$T_{sc2ctrl}(\triangle F_{sc} \bullet B_{sc}) = T'_{sc2ctrl}(\triangle F_{sc}) \bullet T_{sc2ctrl}(B_{sc}) \quad (16)$$

where $B_{SC}$ is a base state chart and $\triangle F_{SC}$ is a state chart refinement of feature *F*. These relationships helped us validate individual transformations of Figure 4.10 and Figure 5.2.

### 5.5.1 Experience

Initially, our tools did not satisfy *(13)*. That is, synthesizing portlet raw material via paths *A* and *B* yielded different results. Upon closer inspection, we discovered errors in both our tools and portlet specifications. Such errors were not exposed until we synthesized raw materials via path *B*.

We soon realized the significance of commuting diagrams/relationships. While checking validity increases build times (more in section 5.5.2), *we obtain assurances on the correctness of our PMDD abstractions, our portlet specifications,*

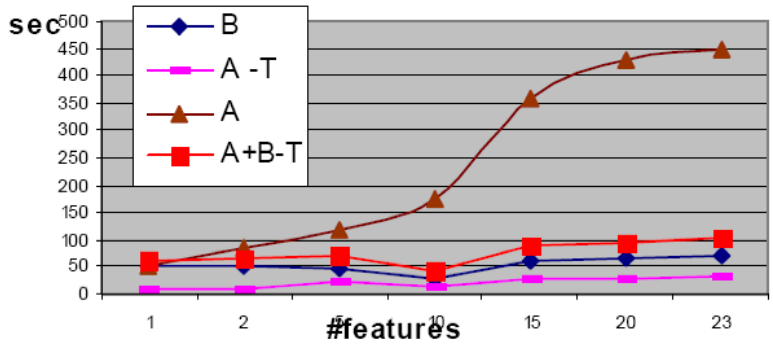



**Figure 5.4**: PinkCreek Build Time Alternatives

*and our tools. (13)* defines stringent properties that our models, tools, and specifications must satisfy, *and without these diagrams, we were unaware that these constraints existed. Our results are general: their benefits will hold in the development of tools, models, and specifications for other domains using FOMDD, as they too will have commuting diagrams like Figure 5.3b.* More on this in section 5.6.

### 5.5.2 Optimization

Figure 5.3a offers two ways in which portlet raw materials can be synthesized: either build raw materials via path *A* or via path *B*. Figure 5.4 shows experimental results of synthesizing portlets via each path. The *A* line indicates the cost of traversing path *A*, which includes the cost of transforming 5-tuples to raw materials. Tuple transformations only have to be computed once, as raw materials of model refinements are portlet invariant. This offers a very useful optimization: raw materials are computed once and can be composed. The *A*-*T* line shows the reduction in cost by this optimization. Note that the *A*-*T* path is 2-3 times faster than path *B* (indicated by line *B*). The *A* line confirms our intuition: without raw material optimization, composing models and transforming (path *B*) is substantially faster than composing transformed models (path *A*). If we validate compositions by building both ways (*A*+*B*-*T*), build times increase, but this is a one-time cost.

## 5.6 Related Work

Model derivation and model refinement are common in FOMDD. We expressed derivations by exogenous transformations (mappings of models written in differ-

ent DSLs) and refinements by endogenous transformations (mappings of models written in the same DSL). We explicitly represent MDD processes as functional metaprograms (where programs are values and transformations are functions that map programs); this idea is latent in the MDD community. We took an additional step forward by merging MDD ideas with those of FOP, which itself has a long history of development [BCRW00, Bea02, BSR04, Bat05, Bat06]. A complimentary view which describes MDD and FOP as an object-oriented metaprogramming paradigm is given in [Bat06].

Horizontal and vertical transformations are also common in MDD [MG06b]. Horizontal transformations map source and target models at the same level of abstraction (e.g., refactoring), while vertical transformations map models that reside at different levels of abstraction (PIM to PSM mappings). We have clearly used both kinds of transformations in PinkCreek, but we found no advantage in making horizontal and vertical distinctions in our work.

Much of the tooling effort in MDD today is focused on UML models. What is generally lacking are tools to express *refinements* of UML models, on which FOMDD relies. Building such tools is the subject of future work (see section 8.3.2).

Kurtev uses XML transformations to develop XML applications [KvdB05]. The design of web applications includes not only functionality but also content, navigation and presentation issues. This calls for a model-based approach (e.g. W2000 [BGP00], WebML [CFM02], UWE [NA02] or OO-HMethod [PGIP01]) from which web applications are derived [PM00]. However, we are unaware of MDD approaches for building portlets.

Merging MDD and product lines is not new [AFM05, BdOB06, CA05b, CA05a, DSvGB03, GBLC05, Gea04, SNW05], we know of few examples that explicitly use features in MDD. One is *BoldStroke*: a product-line written in several millions lines of C++ for supporting a family of mission computing avionics for military aircraft. Gray used MDD to express maintenance tasks on BoldStroke [Gea04]. Adding a feature required both updating BoldStroke's model and code base. Although build optimizations were used (e.g., delaying the updates of the code base), no commuting relationships were found (although we believe that they exist). Additionally, MDD can be used with other paradigms. Kulkarni combined MDD with AOP (Aspect Oriented Programming) [KR03].

Proving properties of large programs remains a difficult challenge. The programs used in PMDD (*javac*, *XSLT*, *ATS*) may be on a scale that is appropriate for

the Verified Software Grand Challenge of Hoare, Misra, and Shankar [Lea], which seeks scalable technologies for program verification.

We believe commuting diagrams are common in FOMDD. In the construction of the AHEAD Tool Suite, customized parsers were built by first composing a base grammar with its refinements, and then using *JavaCC* to derive a parser [BSR04]. This is comparable to path B in Figure 5.3a. A counterpart to path A would be to compose a base parser with its refinements. Unfortunately, *JavaCC* translates only complete grammars into complete parsers (not grammar refinements into parser refinements), so path A could not be evaluated [Bat07a].

Commuting diagrams are fundamental to *category theory (CT)* [Pie91], which is a general mathematical theory of structures and of systems of structures (see section 6.4.5). A benefit of FOMDD is that it is mathematically based, and this makes connections with category theory easier to recognize. PinkCreek has provided us with an invaluable example that has enabled us to unify the ideas of FOMDD program synthesis and CT. An exposition of these ideas is the subject of forthcoming work [Bat07a].

## 5.7 Contributions

MDD and FOP are complementary paradigms. MDD derives models and FOP refines models. Metaprogramming unifies models with values; transformations and compositions are functions. This unification of FOP and MDD, here called FOMDD, offers a powerful paradigm for creating product lines using MDD technology.

We presented a case study of FOMDD that created a product line of portlets. We showed how the MDD production of a portlet is a synthesis metaprogram that transforms a multi-model specification of a portlet into a web archive file. We expressed variations in portlet functionality as features, and synthesized portlet specifications by composing features. Our work exposed a fundamental relationship between model derivation and model refinement, which we expressed as a commuting diagram/relationship. We exploited this relationship (the transformation of a composition of models equals the composition of transformed models) to validate the correctness of our domain abstractions, tools, and portlet specifications at a cost of longer synthesis times. The relationship could also be used to reduce synthesis times if validation is not an issue.

While the portlet domain admittedly has specific and unusual requirements,

there is nothing domain-specific about the need for MDD and FOP and their benefits. In this regard, PMDD is not unusual; it is an example of many domains where both technologies naturally complement each other to produce a result that is better than either could deliver in isolation. FOMDD offers a fresh perspective in program and product-line synthesis where mathematical properties (in addition to engineering feats) guide a principled design of complex systems. Research on MDD and FOP should focus on infrastructures that support their integration (next chapter), and researchers should be cognizant that their synergy is not only possible, but desirable.

The work of this chapter is accepted for publication in the paper:

1. *Feature Oriented Model Driven Development: A Case Study for Portlets*. S. Trujillo, D. Batory and O. Diaz. 29th International Conference on Software Engineering (ICSE 2007). Minneapolis, Minnesota, USA. May 2007 [TBD07]. Acceptance Rate: 15% (50/334).

# Chapter 6

# Generative Metaprogramming for Synthesis Process

"Freedom is not worth having if it does not include the freedom to make mistakes."
– *Mahatma Gandhi*.

## 6.1 Abstract

*Software product-line synthesis* defines a process to synthesize individual programs. The previous chapter combines the use of composition and derivation to realize synthesis. Our work exposed a fundamental relationship between model composition and derivation: *commuting diagrams*. This was symptomatic of an important structure behind our metaprograms that drive the synthesis process. Our work explores these ideas in relationship to product-line synthesis. This work describes a way to synthesize metaprograms, which when executed, will synthesize a target program of a product-line. Specifically, we elaborate on the generation of metaprograms from abstract specifications. We use commuting diagrams to generate a metaprogram from which our target program can be ultimately synthesized. A case study is used to illustrate the *GeneRative metaprOgramming for Variable structurE* (GROVE) approach.

## 6.2 Rationale for *Generation*

*Software product-line synthesis* drives the process to synthesize individual programs[1]. These processes were initially developed using hand-crafted scripting. A typical script to realize a synthesis path consists of around 500 LOC of batch processes that use 300 LOC of ant makefiles and 2 KLOC of Java code, taking around 4 person/day to complete. A small change in the synthesis process (e.g., in order to modify the synthesis path) involves significant effort to modify the script. Overall, this implies time-consuming, repetitive and cumbersome tasks. Our intention is to accelerate the development of synthesis metaprograms by generating them from abstract specifications (i.e., MDD is used to generate FOMDD metaprograms). Doing so, the changes in the synthesis process are modified in the abstract specification level instead of in the implementation.

### 6.2.1 From Scripting to Generation

Our metaprograms have been developed so far as scripts (i.e., a specific script was created for each individual program of the product-line). Commuting relationships expose the nature of synthesis. We realize that our metaprograms are not monolithic code blocks, but consist of well-known primitive operations (e.g., composition or derivation). We believe this structured nature not only holds in our case, but in general.

*Metaprogramming* is the concept that program synthesis is a computation. Specifically, we study the primitives that form such metaprograms. These primitives form the *architecture of metaprograms* [Bat07b]. Our aim is a generative approach to metaprogramming.

To attain this, we explore *(i)* the primitive operations that form synthesis, *(ii)* the abstract specification that can be used to represent synthesis, and *(iii)* the generation of metaprograms from specifications. These are the ideas behind the *GeneRative metaprOgramming for Variable structurE* (GROVE) approach. GROVE demands a new generation of models and tools. This support could eventually foster to widespread metaprogramming use in synthesis.

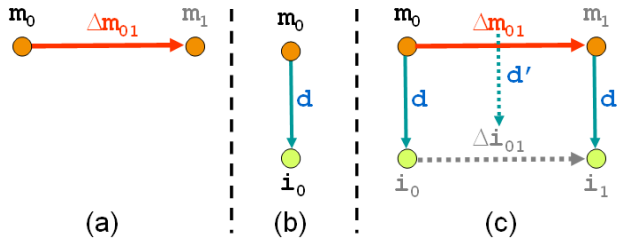



**Figure 6.1**: Simple Primitives

### 6.2.2 Synthesis Primitives

A metaprogram consists of basic primitives. The intention is not only to analyze the primitives used for synthesis, but to study them in order to generate a metaprogram. We begin introducing an initial subset of primitives.

**A Composition** primitive is used to synthesize programs in *Feature Oriented Programming* (see section 3.3). This primitive composes a base artifact with refinements to increment its functionality [BSR04]. Figure 6.1a sketches the composition where nodes represent models $m_0$ (initial) and $m_1$ (initial extended). The edge represents a composition operation $\mathbf{c}ompose$ with model refinement $\triangle m_{01}$ (extending initial). Note that $m_1$ is synthesized as a result of composition. The expression for this primitive operation is[2]: $m_1 = \triangle m_{01}.compose(m_0)$.

**A Derivation** primitive is used to obtain an implementation from abstract models in *Model Driven Development*. This primitive derives models from other models (see section 4.3). Figure 6.1b shows an example of a derivation primitive where nodes represent models $m_0$ (source) and $i_0$ (target). The edge represents a derivation operation $\mathbf{d}erive$. Note that $i_0$ is synthesized as a result of the derivation. The expression for this primitive operation is: $i_0 = m_0.derive()$.

**Composition and Derivation.** Our previous chapter combined the use of composition and derivation in synthesis. In general, models can be refined by composing features (a.k.a., *endogenous* transformations that map models expressed in the

[1]Synthesis is commonly understood to be an integration of two or more pre-existing elements which results in a new creation (from http://en.wikipedia.org/wiki/Synthesis).

[2]Although we write the composition of A and B as $A.compose(B)$, it can be represented alternatively as expression $A \bullet B$.

same language [MG06b]), and can be derived from other models (a.k.a., *exogenous* transformations that map models written in different languages [MG06b]).

Figure 6.1c shows an example of this combination. Nodes represent models where $m_0$ is an input and its remaining ($m_1$, $i_0$ , $i_1$) are synthesized. Edges represent composition operations (e.g., red edge $compose$ is a composition with refinement $\triangle m_{01}$) or derivation operations (e.g., blue edge $derive$ is a model transformation)[3]. Figure 6.1c is actually a *pushout* of Figure 6.1a and Figure 6.1b [Fia05, Pie91].

This exposed a fundamental relationship: the composition of transformed models (1) equals the transformation of a composed model (2). This implies that there are two alternative paths to synthesize $i_1$.

$$i_1 = (\triangle m_{01}.derive'()\,).compose\,(m_0.derive()\,) \quad (1)$$

$$i_1 = (\triangle m_{01}.compose(m_0)\,).derive() \quad (2)$$

When $i_1$ is the same through (1) and (2), commuting holds. This means that composition and derivation operations preserve the structure (see section 5.5).

**Other Primitives.** The primitives introduced so far are aimed to product-line synthesis where composition and derivation are used. However, this set is clearly open to new primitives such as weaving of aspects [ALS06], refactoring of features [LBL06, TBD06], interactions of features [LBN05], harvesting of models [RGvD06], and others. These are different implementation edges. Refer to [Bat07b] for an exploratory study.

### 6.2.3 Synthesis Geometries

The introduced primitives are not used in isolation, but are combined in metaprogram synthesis. Doing so, synthesis is not a monolithic code block, but comprises simple primitives to yield a complex metaprogram, which has structure[4]. By structure, we mean what are the primitives that form synthesis and how these primitives are connected. The combination of primitives forms a complex structure of

[3]Note that a new edge $d'$ appears. This edge represents a different *derive'* from edge $\triangle m_{01}$ to edge $\triangle i_{01}$. Remember *d* was from node to node previously. This $d$' is a model refinement-to-model refinement transformation that enables commuting diagrams.

[4]The structure of something is how the parts of it relate to each other, how it is put together, and how they are connected (from http://en.wikipedia.org/wiki/Structure).

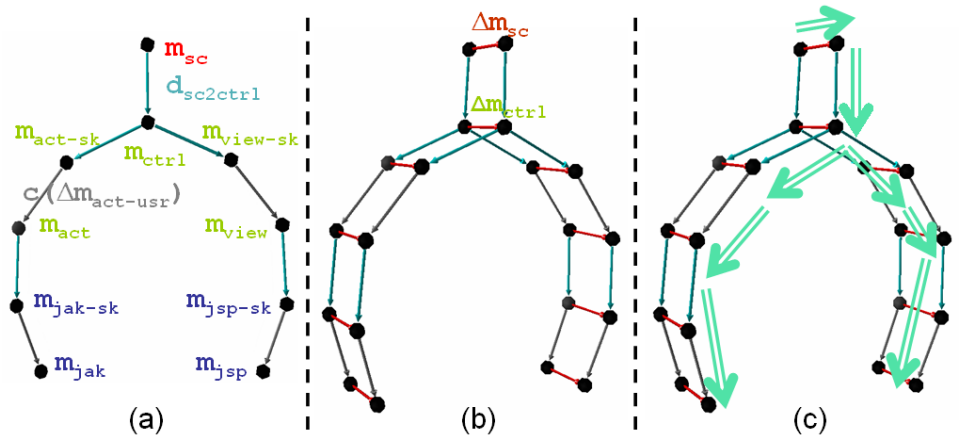



**Figure 6.2**: Geometries

metaprograms (a.k.a., *architectural metaprogramming* [Bat07b]). Specifically, we analyze the *geometry* that depicts a synthesis metaprogram[5].

This work uses PinkCreek case study, which is a product-line of portlets (see appendix A), to generalize on the synthesis geometries. PinkCreek was developed using a model-driven approach for Portlets (a.k.a., PMDD introduced in section 4.4), which specifies a portlet as a set of models from which its implementation can be derived[6]. Figure 6.2a shows PMDD synthesis geometry where the upper-most node represents a statechart model $m_{sc}$. The nodes on the bottom are implementation artifacts: $m_{jak}$ represents a *Jak* class and $m_{jsp}$ a JSP page. The nodes in between are *Portlet Specific Language* (see section 4.4) models: $m_{ctrl}$ defines the portlet controller, $m_{act}$ contains the portlet actions, and $m_{view}$ specifies the portlet views. Edges represent model transformations (e.g., $derive_{sc2ctrl}$ is a derivation from $m_{sc}$ to $m_{ctrl}$) and compositions (e.g., $\triangle m_{act-usr}$).

However, Figure 6.2a shows only the derivation geometry, without the composition (i.e., the model-driven synthesis is sketched, but not the product-line synthesis). Figure 6.2b introduces the composition dimension for PinkCreek synthesis where the geometry of Figure 6.2a is enlarged with a set of edges. Each edge

[5]Geometry arose as the field of knowledge dealing with spatial relationships (from http://en.wikipedia.org/wiki/Geometry).

[6]Figure 4.10 shows the complete synthesis metaprograms to derive a PinkCreek base feature. The notation of this figure adjusted to the notation introduced in this chapter is as follows: $m_{ctrl} = m_{sc}.derive_{sc2ctrl}()$, $m_{act-sk} = m_{ctrl}.derive_{ctrl2act}()$, $m_{act} = \triangle m_{act-usr}.compose(m_{act-sk})$, and so on (where each *m* represents a different model, *derive* denotes derivation and *compose* is composition).

represents a refinement of a node (e.g., $\triangle m_{sc}$). Doing so, Figure 6.2b shows the geometry of the synthesis metaprogram where derivation and composition are combined together. This example shows only two features: one constant with the base nodes and one function extending base with edges (i.e., a constant feature consists of initial base nodes, a function feature consists of edges extending initial nodes to target nodes, and result consists of target nodes).

Figure 6.2c shows how to traverse a geometry to generate a synthesis metaprogram. The study of these geometries paves the way to the generation of synthesis metaprograms.

## 6.3 Generative Metaprogramming

*Generative Programming* is the process to generate programs [CE00][7]. Our work goes one step farther. Starting from a synthesis geometry specification, GROVE generates a synthesis metaprogram, which is used to generate a program. We name this process *Generative **Meta**programming*[8].

The product-line synthesis is not aimed to build one individual application, but a number of them. This demands a change in the engineering processes where a distinction between domain and application engineering process is introduced. *Domain engineering* (a.k.a., core asset development) determines the commonality and the variability of the SPL, whereas *application engineering* (a.k.a., product development) synthesizes individual applications from the SPL.

This work does not focus on the development of the product-line[9]. Conversely, our work concentrates on how to synthesize individual programs out of a product-line. Therefore, domain engineering described next is restricted to the process of defining the geometry of metaprograms, and application engineering to the synthesis of individual metaprograms. GROVE thus consists of two major activities.

First, the geometry specification is defined for the domain of the application. Figure 6.2b shows the synthesis geometry for PinkCreek (details explained shortly in section 6.3.1).

Second, starting from this geometry, a specific metaprogram can be specified by its synthesis path (i.e., how to traverse the geometry space). Figure 6.2c shows

[7] Generative programming uses automated source code creation through code generators to improve programmer productivity (from http://en.wikipedia.org/wiki/Generative_programming).

[8] This is related to multi-stage programming [TS00].

[9] We consider that the SPL has been already built using existing approaches [BSR04, CN01] (see appendix A for a complete approach).

the synthesis path for a PinkCreek program of two features (more details in section 6.3.2).

### 6.3.1 Geometry Specification

**Substructures** A synthesis geometry is basically formed by two substructures: base and refinement synthesis substructures.

*Base synthesis substructure* contains the set of primitives from which the implementation of a base program is derived. It represents derivation synthesis without composition. Figure 6.3a shows this where nodes represent different models and edges represent derivations and local compositions.

*Refinement synthesis substructure* represents the refinement of a base synthesis substructure in order to realize a feature (i.e., the synthesis a feature realizes). Figure 6.4a shows this situation where a node represents a model that can be composed with an edge representing a model refinement. The result of composition yields another node representing a model. Hence, the edges represent model (or code) refinements and *edge-to-edge* transformations represent model (or code) refinement transformations. To simplify this representation, edges are represented as nodes, and edge-to-edge transformations as edges. Figure 6.4b shows this simplification of Figure 6.4a.

Base synthesis substructure corresponds to constant features, whereas refinement synthesis substructure corresponds to features that extend base (i.e., refinement substructures are repeatable). The point to stress is that synthesis geometry of Figure 6.2b can be generated combining base and refinement synthesis substructures together. To specify the geometry, a graphical language is needed.

**Specification** There are currently several graphical specification languages (e.g., SVG[10]). *Graph eXchange Language* (GXL) was selected to depict graphical structures because it provides a meta-model, language, and tool support [Sou, HWS00]. GXL enables to specify both substructures from which the synthesis geometry is formed.

GXL metamodel is designed to specify directed graphs [HWS00]. Figure 6.3b shows a fragment of the GXL code that specifies the base synthesis substructure of Figure 6.3a. This code is actually the representation of a model conforming the GXL metamodel. This figure was sketched using existing tool support [Sou,

[10]SVG (Scalable Vector Graphics). http://www.w3.org/Graphics/SVG/

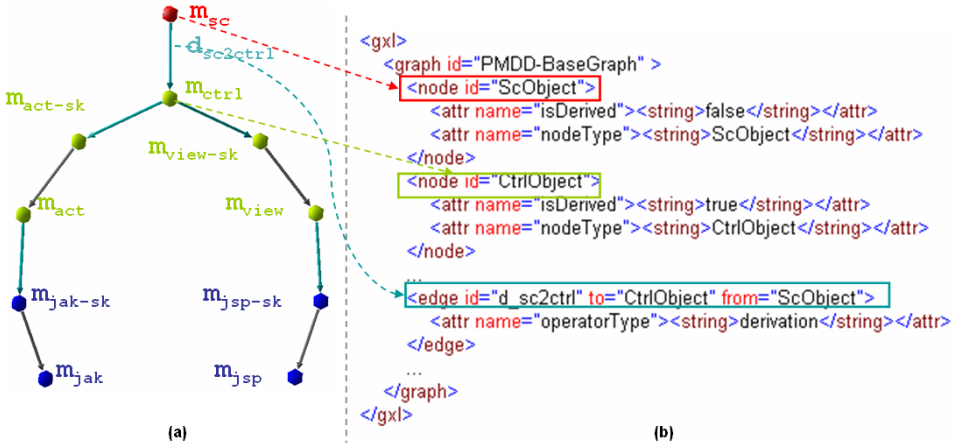



**Figure 6.3**: Base Synthesis Substructure Specification

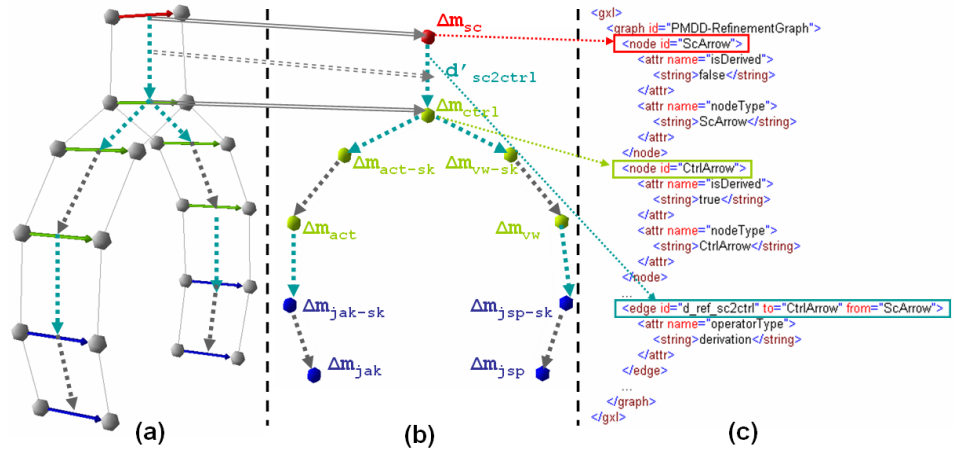



**Figure 6.4**: Refinement Synthesis Substructure Specification

HWS00]. This tool produces the snippet of Figure 6.3b, which lists a fragment of code with two nodes and one edge connecting both nodes. Nodes specify models $m_{sc}$ and $m_{ctrl}$, and edge specifies derivation primitive $d_{sc2ctrl}$. In general, this code fragment represents the expression $m_{ctrl} = m_{sc}.d_{sc2ctrl}()$.

The elements of Figure 6.3b require additionally the specification of some attributes (e.g., name, type, primitive, whether is derived from other, etc). Nodes and edges have slightly different attributes. The attribute *nodeType* determines the type of a node. The attribute *isDerived* states whether a node is derived from other node. The attribute *operatorType* determines the primitive of an edge. This allows

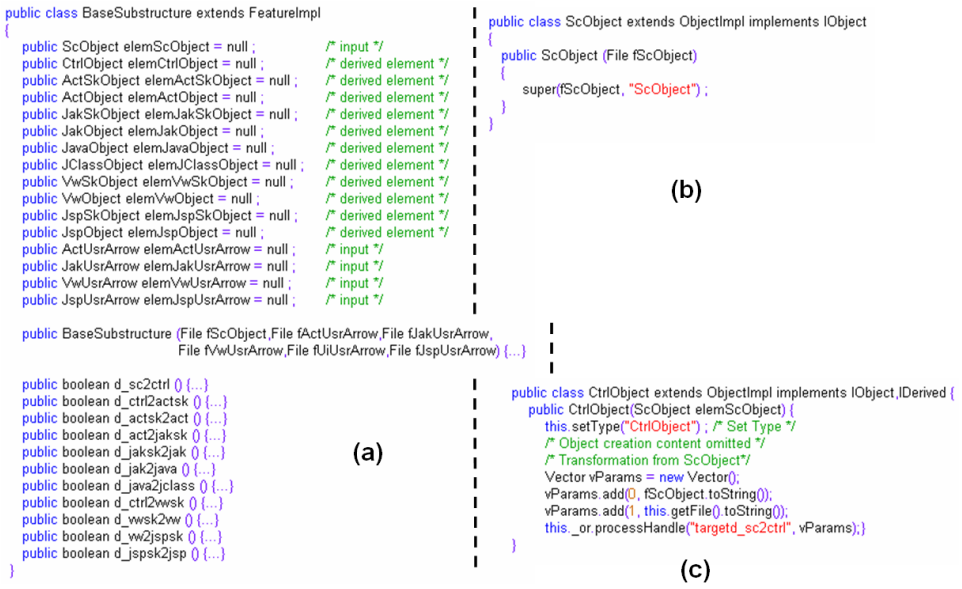



**Figure 6.5**: Generated Code Fragments

later to specify how to handle (edge) operations over (node) elements. This extra information of the specification would be required later to generate geometry implementation.

Likewise, Figure 6.4c shows partially the code snippet of a refinement synthesis substructure. This code fragment represents the expression $\triangle m_{ctrl} = \triangle m_{sc} .d'_{sc2ctrl}()$. The specification is similar to Figure 6.3b, but note that types of nodes are not models, but model refinements, and the edge is a transformation between such model refinements. This specification differs from the base synthesis substructure because the actual way to operate it also differs.

**Transforming from Specification to Implementation** The next goal is to create an implementation to handle each element of the geometry (i.e., node or edge). This implementation is the result of a model transformation from the specification of the two substructures.

The result of the transformation is a set of *Java* classes that realize the geometry and give semantics to the primitive operations. The generated classes represent *(i)* the objects contained in the substructures (for each node a class is generated), and *(ii)* the substructures that form the geometry (for each substructure a class is generated).

First, a *Java* class is created for each node in the geometry. These classes represent nodes that are used later on by metaprograms. Some nodes are inputs and others are derived (those that depend on the input from another model). Figure 6.5b shows a snippet where an example for an input is shown. *ScObject* class encapsulates the synthesis functionality for models of type $m_{sc}$. As this is an input model, the functionality is reduced to create a handle for this file. Note that our framework provides a default implementation for the constructor (*ObjectImpl* in Figure 6.5b). The way to construct derived nodes is by creating constructors in their respective classes. Figure 6.5c shows an example for a derived class. *CtrlObject* class encapsulates the synthesis functionality for $m_{ctrl}$. Note that a constructor is used to derive $m_{ctrl}$ from $m_{sc}$. Currently, this constructor invokes an *ant* target that performs the actual transformation and should be defined elsewhere (described shortly). As composition is always similar, all node classes are composable (e.g., $m_{ctrl}$ can be composed with $\triangle m_{ctrl}$). We provide a default implementation for composition (*ObjectImpl* in Figure 6.5b). Similarly, each node element has a corresponding class.

Second, a *Java* class is created for each substructure that form the geometry (see Figure 6.5a). Actually, two classes are created for base and refinement substructures. These classes contain the objects that form each substructure. Figure 6.5a shows a snippet where an example for *BaseSubstructure* is shown. This substructure contains references to its constituent parts (*ScObject*, *CtrlObject*, and so on). The substructure classes contain also one method for each edge of the substructure (*d_sc2ctrl*, *d_ctrl2actsk*, and so on) that enables to construct derived nodes. Additionally, as the composition of substructures is always similar, all substructure classes are composable to yield complex synthesis geometries. We provide a default implementation for this (*FeatureImpl* in Figure 6.5a). In general, base substructure contains model objects and model transformation methods, whereas refinement substructure contains model refinement objects and their transformation methods.

As a result of these transformations, we obtain a class per node (with object functionality) where edge functionality is included and a class per each substructure (containing object references and edge methods). All these generated classes are later used by the actual synthesis metaprograms. This is the reason why we choose this way to represent metaprograms.

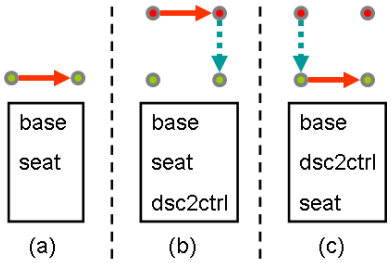



**Figure 6.6**: Path Specification Example

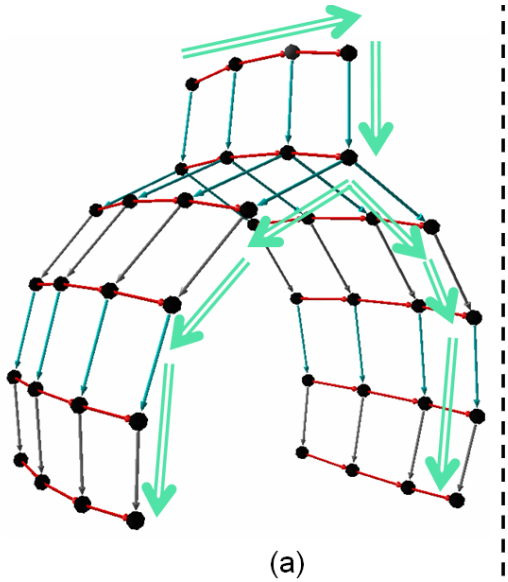
**Tool Support** is required for the domain engineer to specify the synthesis geometry. *GROVE Tool Suite* (GTS) is a set of tools we created to support GROVE. First, the synthesis geometry is specified using *Graph eXchange Language* (GXL). Existing tools can be used by the domain engineer to draw the required geometry [Sou]. Second, the synthesis geometry specification needs to be transformed into the implementation code, which is used later on by the metaprogram. Thus, a code generator has been created (a.k.a., GTS geometry generator). Third, the semantics of some primitives needs to be specified completely (e.g., we use *ant* to specify how to handle the transformation). This is introduced by GTS primitive handler.

### 6.3.2 Path Generation

**Specification** A synthesis path represents how to synthesize a program traversing a geometry by means of composition and derivation. The fundamental idea is that the synthesis geometry should be specified before drawing the synthesis path. Figure 6.2c shows the path for a PinkCreek program where the geometry sketches two features. To specify this synthesis path, a specification is necessary to deal not only with feature composition, but as well with model derivation.

Existing specifications for program synthesis (e.g., AHEAD equation) do not consider other primitives but composition. An equation only specifies the set of features that distinguishes the program [BSR04]. Figure 6.6a shows an example of this where features are listed in composition order. This would produce a synthesis geometry of one base substructure for *base* feature and one refinement substructure for *seat* feature. The expression is as follows $seat.compose\,(base)$. However, it lacks the derivation direction.

Hence, equation should be enhanced to introduce such extra information. Doing so, it introduces other primitives besides feature composition where compo-




```
public class SynthesisMetaprogram
{
  public static void main(String[] args)
  {
    // Build objects for equation features
    /* Create feature "1Base" */
    File f1BaseScObject = new File ("../1_fbp.scxml") ;
    /* Remaining files omitted */
    BaseSubstructure f1Base = new BaseSubstructure (...) ;
    /* Create feature "2Asst" */
    RefSubstructure f2Asst = new RefSubstructure (...) ;
    /* Remaining features creation omitted */
    // Synthesis path for features
    BaseSubstructure f21 = f2Asst.compose (f1Base) ;
    BaseSubstructure f321 = f3Bull.compose (f21) ;
    BaseSubstructure f4321 = f4Chec.compose (f321) ;
    f4321.d_sc2ctrl () ;
    f4321.d_ctrl2actsk () ;
    f4321.d_actsk2act () ;
    f4321.d_act2jaksk () ;
    f4321.d_jaksk2jak () ;
    f4321.d_wsk2ww () ;
    f4321.d_ctrl2wsk () ;
    f4321.d_ww2jspsk () ;
    f4321.d_jspsk2jsp () ; /* This is the result */
    }
}
```

(b)

**Figure 6.7**: Synthesis Path Metaprogram Example

sition and derivation can be used together. Figure 6.6b shows an example where *dsc2ctrl* derivation primitive is introduced after composition. Although this example is simple, it enables the equation to deal with derivation as well. The expression is as follows $(seat.compose\,(base)).derive_{sc2ctrl}()$.

Figure 6.6c is a similar example where the equation differs. The expression is as follows $(seat.derive'_{sc2ctrl}()).compose\,(base.derive_{sc2ctrl}())$[11]. The product synthesized via Figure 6.6b and 6.6c is the same (remind commuting diagrams). However, the synthesis time is different (remind section 5.5.2).

**Transforming from Specification to Implementation** The final goal is to synthesize a metaprogram (i.e., specifying how the geometry is traversed to get a metaprogram). This is obtained from the synthesis path specification as the result of applying a model transformation where the path specification is the input and the metaprogram implementation is the output. This transformation creates a specific metaprogram for each synthesis path in order to synthesize an end program.

As we define previously the objects to handle the geometry of synthesis, the transformation is straightforward. For each feature in the synthesis path, an object is created in the metaprogram. Then, the synthesis path is traversed as follows. For

[11] Figure 6.6c is simplified and $d'_{sc2ctrl}$ is omitted. Refer to the previous sections for further details.

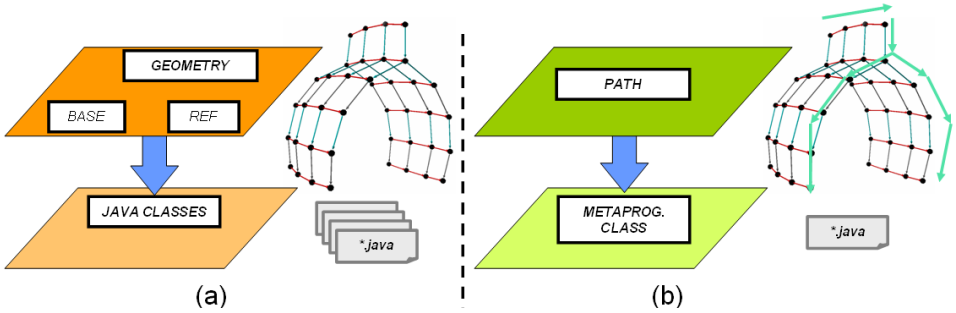



**Figure 6.8**: Big Picture

each feature composition edge, a feature composition is called. For each derivation edge, a derivation is invoked.

Consider a geometry of 4 features: *base*, *seat*, *checkin* and *assistance*. Figure 6.7a shows this geometry and one synthesis path that traverses such geometry to synthesize a metaprogram. From such path specification, a Java class implementing the metaprogram is derived as follows. First, one substructure object per feature is created. Second, the horizontal path is traversed (i.e., feature substructures are composed). Then, the vertical path is traversed (i.e., derivations are applied to the resulting substructure to derive implementation artifacts). Figure 6.7b lists a fragment of this code. This resulting metaprogram is directly executable to synthesize an end-program.

**Tool Support** is required by the application engineer to specify program synthesis. We create specifically tooling to support the generation of synthesis path metaprograms from a synthesis geometry. First, the features of the program are selected using existing GUI tools to select features (see section 3.4.3). Second, from the feature selection, we can plot the whole synthesis geometry from which the user could select a synthesis path (see Figure 6.2b). This path is then translated to a Java executable metaprogram. Another tool is needed to generate such executable metaprogram from which the program is synthesized finally. Our tooling also invokes the suitable metaprogram.

### 6.3.3 Recap and Perspective

So far, we describe how to generate synthesis implementation from a specification where GROVE is illustrated with our case study: PinkCreek.

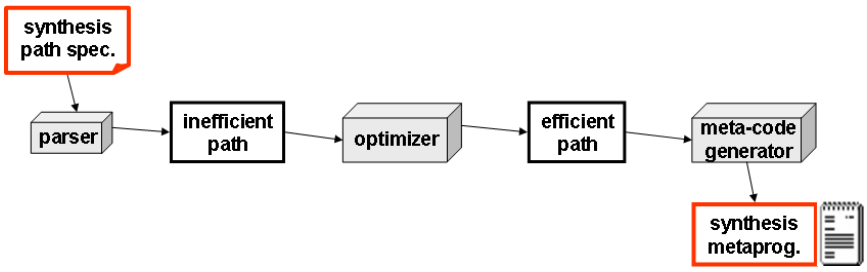



**Figure 6.9**: Generative Metaprogramming Process

First, PinkCreek's synthesis geometry is specified (see Figure 6.8a). Actually, base and refinement synthesis substructures are specified (base corresponds to 6.3 and refinement corresponds to 6.4). Model-to-code transformations are used to transform from this synthesis specification to the Java classes implementing that geometry (classes are similar to Figure 6.5). Doing so, synthesis geometry classes are available to be used later on by each metaprogram class (traversing a synthesis path).

Second, the aim is to generate PinkCreek's programs (see Figure 6.8b). Specifically, we generate those synthesis metaprograms from which programs are synthesized. We start by selecting the program features (creating an equation). Doing so, the geometry of Figure 6.7a is rendered. A synthesis path specifies then how to traverse such geometry (e.g., Figure 6.7a is an example of such traversal). From the synthesis path, a synthesis metaprogram code is derived using the model-to-code transformation we created (see 6.7b). This metaprogram code uses the geometry classes created previously. Hence, the execution of this metaprogram leads to the synthesis of the program.

**Process** Figure 6.9 depicts the process to generate metaprograms that eventually synthesize programs. A number of challenges are addressed in this work to turn this figure from an envision into a reality. This process is analogous to the SQL process where *(i)* a synthesis path statement is specified, from this input *(ii)* the parser generates *(iii)* some inefficient path, then *(iv)* the efficiency of this path is evaluated by the optimizer, eventually *(v)* obtaining an efficient path, which is *(vi)* the input to the meta-code generator. The output of this process is *(vii)* the synthesis metaprogram code that is directly executable to synthesize a program.

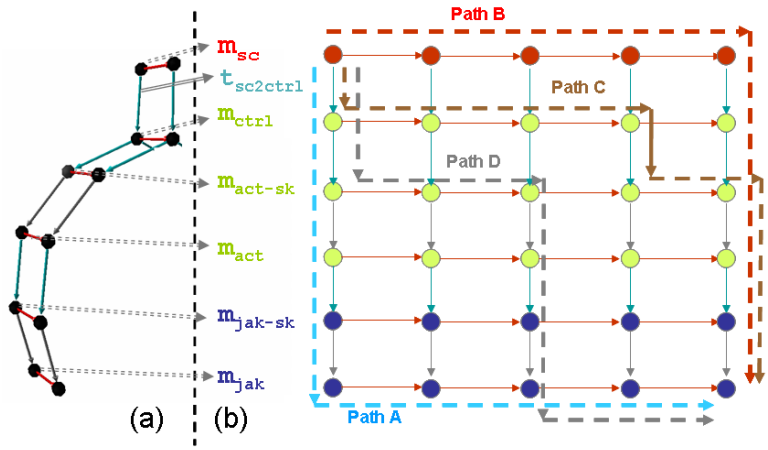



**Figure 6.10**: PinkCreek Optimization

**Benefits** GROVE automates significant and tedious tasks in synthesis metaprogramming. The benefit of this approach is that it is no longer necessary to create/modify the implementation of the synthesis metaprogram, but its specification (i.e., the way to specify synthesis is not coding, but drawing a graphical specification using existing tools). Our approach reduces the development time of synthesis metaprograms and facilitates their evolution. For our PinkCreek example, 33 Java classes and 1.188 LOC are derived from an input of two substructure files and 498 LOC. Likewise, synthesis path metaprograms are not coded, but derived from path specifications. For a typical metaprogram of 5 features, specified by an equation of 18 LOC, our tools produce an output metaprogram of 210 LOC. This metaprogram uses the 33 geometry classes generated before. The number of LOC increases proportionally with the number of features: 380 LOC for 10 features (equation specification of 23 LOC); 550 LOC for 15; 720 LOC for 20; and 822 LOC for 23 (equation specification of 36 LOC). These figures pose a remarkable improvement from our previous scripts (see section 6.2.1).

## 6.4 Future Work

### 6.4.1 Multiple Paths

Multiple paths are possible while traversing a geometry design space to generate a synthesis metaprogram. Section 5.5.2 describes only two possible paths. However, additional paths exist. Figure 6.10 illustrates this situation where further paths are

shown for PinkCreek synthesis. Figure 6.10a shows partially the synthesis. This geometry can be traversed by multiple synthesis paths to yield the same product (see Figure 6.10b).

Each synthesis path implies a different cost. A number of rules are required to optimize cost. These rules would be based on different optimization criteria (e.g., build-time, program size or quality). Under each criteria, one or more optimized path appears. This shortest path between two points is known as *geodesic* [Bat07a][12].

In our process of Figure 6.9, the optimizer would take the synthesis path selected by the user as input and would return a geodesic. To turn this vision into a reality we created tools to measure the cost of traversing each edge of the geometry. We have measured the cost of operation time and the program size. However, we do not yet solve the optimization problem for synthesis. An alternative is the use of rules to find the geodesic. Synthesis optimization is a matter for future work.

### 6.4.2 Multiple Dimensions

The set of primitives presented so far configure a space of synthesis design (similar to composition design [Bat07a]). In general, this space consists of multiple dimensions. Thus, the synthesis geometry becomes more complex. So far, we come up with two types of dimensions: space and time.

**Space and Origami** Multidimensions in space arise when there are orthogonal feature models. Origami deals with these complex relationships among orthogonal features by using matrices [BLS03]. An Origami matrix is a n-dimensional matrix where each dimension is formed by a set of features and its cells are the feature modules that implement the functionality at the intersection of their coordinates in the n-dimensions. The benefit of Origami is that represents largely linear specifications of systems that have potentially exponential complexity. For the purpose of this work, note that Origami poses multiple *space* dimensions to be considered.

Figure 6.11a shows an example with a simple 2x2 matrix. Figure 6.11b sketches the synthesis geometry for this matrix (i.e., translates Origami matrix into synthesis geometry representation). The geometry becomes more complicated as the number of dimensions increase. Figure 6.11c shows a 2x2x2 matrix (i.e., a cube) and

[12] A geodesic can be regarded also as the shortest spanning graph between an input set of points and an output set of points. Its resolution is not straightforward. According to Batory, it involves solving the Directed Steiner-Tree problem, which is NP-hard [Bat07a, CCC+99].

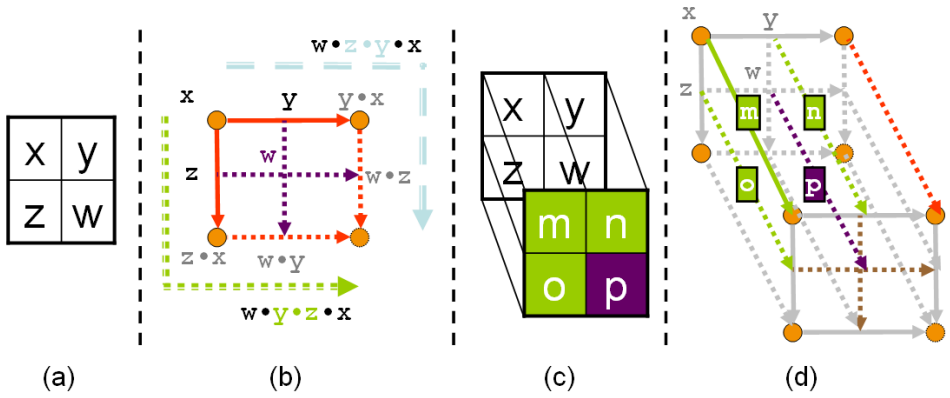



**Figure 6.11**: Origami Multi-dimensions

Figure 6.11d sketches its geometry. Note that these examples only use composition, but not derivation. These figures illustrate how the geometry explodes with the number of dimensions.

The representation of multiple dimensions exposes some limitations of our work. It is not evident how to draw these synthesis structures even for small dimensions, and it is almost impossible to depict out the commuting diagrams for n-matrices. Future work might address a generalization of an Origami matrix, that allows complex, multi-dimensional commuting diagrams to be expressed. Building multi-dimensional matrices is, actually, rather easy [Tak06]. In this future scenario, derivation should be represented using Origami as well.

**Time and Evolution** The geometrical spaces presented so far represent a static view of the synthesis metaprogram geometry. However, as any software, a synthesis metaprogram also evolves over time towards a dynamic geometry. Remarkably, this evolution forces to consider also multiple *time dimensions*. This is closely related to the variability of synthesis introduced in the next chapter.

### 6.4.3 Multiple Artifacts

The primitives presented in section 6.2.2 are applied to individual artifacts. However, operations frequently involve more than one artifact. This is the case of composition where multiple artifacts are simultaneously composed *á la* AHEAD (see section 3.3). A feature in AHEAD is implemented by multiple artifacts: introductions add new artifacts and refinements extend existing artifacts. Such artifacts can

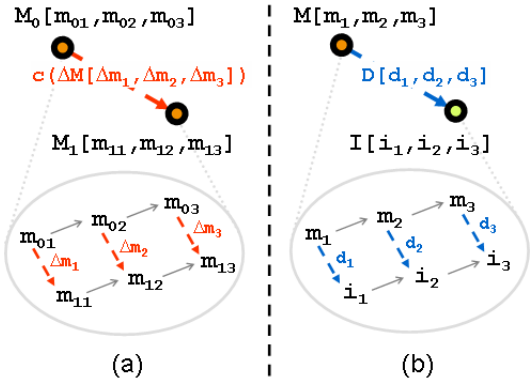



**Figure 6.12**: Feature Composition

have multiple representations (e.g., code, makefiles, documentation, etc.).

Figure 6.12a shows an example of a simultaneous composition of multiple artifacts. Coarse-grained nodes $M_0$ and $M_1$ represent a set of artifacts ($M_0$ contains $m_{01}$, $m_{02}$, and $m_{03}$; $M_1$ contains $m_{11}$, $m_{12}$, and $m_{13}$). Likewise, edge $\triangle M$ consists of a set of artifacts ($\triangle m_1$,$\triangle m_2$ ,$\triangle m_3$) to refine the set of artifacts a node comprises. Note that $M_1$ is synthesized as a result of a composition: $M_1 = \triangle M.compose\,(M_0)$.

In general, coarse-grained nodes and edges contain the realization of a feature. Some features are base or constants (e.g., node $M_0$), some features are increments in functionality or functions (e.g., edge $\triangle M$), and a composition of features is a program (e.g., $M_1$).

Similarly to composition, derivation comprises frequently multiple artifacts. Figure 6.12b shows an example of a derivation of multiple artifacts.

### 6.4.4 Multiple Product Lines

Synthesis is typically scoped to an individual product-line. However, in certain contexts synthesis of multiple product-lines may be desirable. Consider an example where different Portlets are combined together to synthesize a Portal. Although this setting is unlikely nowadays in SPL, it is common in service-oriented architectures.

Future work should address how to automate synthesis involving external product-lines. More to the point, this would impact on the geometries we presented before. Consequently, the architecture metaprogramming should also consider the service-

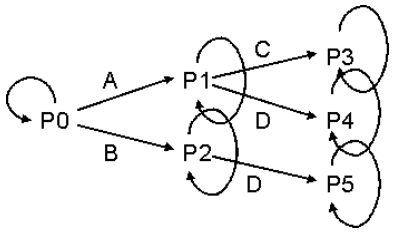



**Figure 6.13**: Simple Category

oriented architectures [Bat07b, Tea07].

### 6.4.5 Category Theory

Structure is a recurring issue in computer sciences [Gog91], and also fundamental in software product-lines. This work raises the issue of a structural theory for product-line metaprogram synthesis. The final aim is to have a strong theory behind, an algebra describing this theory, and eventually some programming language with metaprogramming primitives. We discuss the need for a structural theory.

*Category Theory* (CT) is a general mathematical theory of structures and of systems of structures. In mathematics, CT deals in an abstract way with mathematical structures and relationships between them. CT offers a way to *(i)* reason about structure, and mappings that preserve structure; *(ii)* abstract away from details; and *(iii)* automation (constructive methods exist for many useful categorical structures) [Eas98].

A category is a directed graph with special properties. Nodes are called objects and edges are arrows. An arrow drawn from object *X* to object *Y* is a function with *X* as its domain and *Y* as its codomain. Arrows compose like functions, and arrow (function) composition is associative. In addition, there are identity arrows (identity functions) for each object, indicated by loops. Figure 6.13 shows a base category where object *P0* has two arrows: *A* linking to *P1* and *B* to *P2*. An identity function links the objects itself (*P0*).

According to [Bat07a], a product line is a category: Figure 6.13 shows a category of programs ($P_i$). Each object $P_i$ is a domain with one element (the i-th program). Arrows (e.g., A, B, ...) are total functions that compose like functions. Commuting diagrams are fundamental to CT [Fia05, Pie91], which was proposed recently as a theory to support program synthesis [Bat07a, TBD07], where

CT is represented in an applied way that links with our work. Our case study PinkCreek provided an invaluable case study that enabled to unify the ideas of synthesis metaprograms and CT [TBD07, Tru07]. However, further work is needed to study how relevant CT is to product synthesis.

Present work is not far from this vision. Our geometries resemble to CT representations. *Base synthesis substructure* specification resembles actually a category in CT. Furthermore, *refinement synthesis substructure* resembles to the concept of functor[13]. We also encounter pushouts in our geometries.

CT imposes some constraints on our work that enable us to validate the correctness of our abstractions, tools, and specification. We found errors because of the constraints imposed by CT. Specifically, the commuting diagrams force us to fix transformations and compositions to commute (e.g., derivations should consider order of elements to preserve the order of compositions).

There are many results on CT that may not be applicable to Pinkcreek (specifically) and FOMDD (in general). A clear example is adjoints [Pie91]. We believe that there could be lots of results in CT that are useless to us. Nonetheless, the study of these issues is the subject of future work.

## 6.5 Related Work

Feature Oriented Model Driven Development (FOMDD) is a blend of FOP and MDD where the structure of features imposed by AHEAD is translated to MDD. GROVE is a model to support the generation of synthesis in FOMDD.

FOMDA stands for Feature Oriented Model Driven Architecture. Although the name is close to FOMDD, the issue addressed by FOMDA is different. FOMDA aims to specify the flow of transformations in MDA using a feature model [BdOB06]. Doing so, features are not used to denote increments in program functionality, but also model transformations.

AHEAD is an algebraic model for feature composition that structures SPL artifacts for composition (see section 3.3). Inspired by AHEAD, GROVE is focused on the geometry of synthesis metaprograms where the aim is the generative metaprogramming using geometry specification. GROVE aims to introduce a structural model for SPL synthesis. Actually, GROVE introduces an algebraic representa-

[13] In category theory, a functor is a special type of mapping between categories. Functors can be thought of as morphisms in the category of small categories (from http://en.wikipedia.org/wiki/Functor).

tion, even though further work is needed to assess the mathematical implications of this representation.

The basic ideas behind model driven development pushed us to abstract the synthesis process. Actually, we first implemented those classes that later we attempted to model in order to generate them. Essentially, GROVE applies model-driven ideas to the generation of synthesis metaprograms.

*Intentional programming* is a collection of concepts which enable software source code to reflect the precise information, called an intention, which programmers have in mind when conceiving their work. Those intentions are actually abstractions [Sim96, SCC06]. Similarly, GROVE introduces geometry abstraction that enables the generation of code.

Increasingly complex systems require novel approaches where analogies with biological phenomena could be useful to inspire them (e.g., the study of structural biology to inspire GROVE geometries) [MB01].

The geometries presented in this work are far from the *"perception-representation-action loops"* (a.k.a., *Design Animism*) presented by Laurel in her OOPSLA Keynote [Lau06]. Nonetheless, we believe the exploration of this aesthetic conception is worth with regard to GROVE geometries.

Synthesis is also known as product production [McG04]. Production planning defines how programs are built. It provides the strategical vision of the production process [CM02]. GROVE specifically concentrates on the synthesis process of this managerial plan.

## 6.6 Contributions

This chapter described ideas to synthesize metaprograms, which when executed, will synthesize a target program of a product-line. Specifically, we elaborated on the generation of metaprograms from abstract specifications. To attain this, we presented the *GeneRative metaprOgramming for Variable structurE* (GROVE) as an approach *(i)* to specify a synthesis geometry, *(ii)* from which a synthesis path can be specified *(iii)* in order to generate the code of a synthesis metaprogram. The execution of such metaprogram code synthesizes a target program of a product-line. So far, the geometries allow a limited number of primitives mainly for composition and derivation.

GROVE called for a companion set of tools. *GROVE Tool Suite* (GTS) supports the generative approach for synthesis metaprogramming. GTS is realized

by 26 classes and 4 KLOC Java. We implemented also 3 model transformations (two for synthesis substructures and one for synthesis path) realized by 600 LOC of XSL code. GTS encapsulates and reuses logic and infrastructure common to generative metaprogramming and provides the base functionality on top of which metaprogram-specific functionality is built.

Evolution introduces the time dimension into the synthesis. Models are frequently updated after their generation (e.g., to fix possible errors). This affects the whole synthesis (i.e., a new operation is performed afterwards). The issue is how to keep track of these updates at model level, and how to propagate those changes back to the model. Another issue is whether model validation and debugging would affect synthesis. To study the relationships of these ideas to CT is also worth.

GROVE introduces an initial set of primitives for synthesis. Future work should address a specific GROVE algebraic model. Considering Origami multidimensional models could be a starting point [BLS03, Tak06]. Synthesis is also subject to variability to accommodate different strategies during program synthesis (see next chapter). The study of this variability in synthesis should be considered similarly to [DTA05]. These issues are on our agenda for future work.

# Chapter 7

# Variability on the Production Process[1]

"It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change."

– *Charles Darwin.*

## 7.1 Abstract

*Software product-line synthesis* defines the process to produce individual products. The promotion of a clear separation between artifact construction and artifact synthesis for product production is one of the hallmarks of software product lines. This work rests on the assumption that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the artifacts but on the production process itself. This leads to promoting production processes as first-class artifacts, and as such are liable to vary when accommodating distinct features. Production process variability and its role to support either product features or production strategies are analyzed. As a proof of concept, the *AHEAD Tool*

[1]The core of this chapter comes from our SPLC 2005 paper [DTA05]. We adjust some terms to better fit the terminology used throughout this thesis. The text mostly sticks to the original published version. However, as this work was produced before previous chapters, some footnotes are added for clarification.

*Suite* is used to support a sample application where features require variations on the production process.

## 7.2 Rationale for *Production Variability*

**Software Product Lines** (SPL) are defined as "*a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [CN01]. In this chapter, we focus on "the prescribed manner" in which products are synthesized: **the production plan.**

A production plan is "*a description of how core assets are to be used to develop a product in a product line*" [CM02]. Among the distinct concerns involved in a production plan, this chapter focuses on the **production process** which specifies how to use the production plan to synthesize the end-product [BF93, CM02]. As stated in [McG04] "*product production has not received the attention that software architecture or programming languages have*". It is often so tightly coupled to the techniques used to create the product pieces that both are indistinguishable. For example, integrated development environments (e.g., *JDeveloper*) make it seamless by automatically creating a build script for the project or system under development so that the programmer can be unaware of the synthesis process that leads to the end-product.

Indeed, production processes have been traditionally considered as mere scripts. They are created by the same programmers that also developed other reusable artifacts. In a traditional setting, such scripts are often kludged together. They are by people who would rather be writing source code than developing a production process. Such scripts are notorious for their poor or misleading documentation [Cre03], which was thought to be consumed by other core asset developers.

An SPL changes this situation by explicitly distinguishing between core-asset developers and product developers where the latter are involved in intertwining the core assets to obtain the end-product. This distinction not only reinforces a separation of concerns between programming and production, but explains the preponderant and strategic role that production has in SPL. That is, there is a growing evidence that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the components but on the production process itself. Despite this observation, most approaches just support a textual description of the production process [CDM02], where variability or requirements specific to

the production process are almost overlooked.

Based on these observations, we strive to turn production processes into first-class artifacts. Specifically, the main contribution of this work rests on observing how the explicit and separate specification of the production process accounts for variations at both the product and process level. To this end, this work distinguishes between product features and process features. By product features we mean those that characterize the product as such, whereas process features refer to variations on the associated synthesis process. Hence, two end-products can share the same product features but being produced along distinct production process standards.

We show some evidence of how this process variability impacts both the modifiability (i.e., variability along time) and the configurability (i.e., variability in the product space) of SPL. To this end, these ideas are supported for *AHEAD* [BSR04], a methodology for SPL based on step-wise refinement. So far, the companion tool suite, *AHEAD Tool Suite* [Bata], *(i)* hides the process into the integrated development environment, and *(ii)* excludes build scripts from refinement. Hence, the upgrades include, *(i)* an explicit representation of the synthesis process that AHEAD implicitly conducts, and *(ii)* a refinement operator for production processes. These processes are specified using *ant* [Fou], a popular scripting language in the Java world.

## 7.3 Revisiting AHEAD

*Step-wise refinement* (SWR) is a paradigm for developing a complex program from a simple program by incrementally adding details [Dij76]. AHEAD is a design methodology for creating application families and architecturally extensible software (i.e., software that is customizable via module additions [BO92]). It follows traditional SWR with one major difference: instead of composing thousands of microscopic program refinements, AHEAD scales refinements so that each adds a whole **feature** to a program, being a feature a "*product characteristic that is used in distinguishing programs within a family of related programs*" [BSR04]. Hence, a final program (i.e., a product) is characterized as a sequence of refinements (i.e., features) applied to the core artifacts.

This approach is supported by the *AHEAD Tool Suite (ATS)* [BSR04] where refinements to realize a feature are packaged into a **layer**. Broadly speaking, the base layer comprises the core artifacts, where other layers provide the refinements that permit enhancing the core artifacts with a specific feature (see section 3.3).

### 7.3.1 Production Process in AHEAD

From the perspective of the production process, it is important to distinguish between:

- the **build process**[2], which specifies the construction process for the set of artifacts included within a layer. This is specified as *ant* files in *ATS*. This would correspond to the "*product-build process*" in Chastek's terminology [CM02].

- the **synthesis process**[3], which specifies how layers are composed to synthesize the end-product. This is hard-coded in *ATS*. This is referred to as "*product-specific plan"* in Chastek's parlance [CM02].

Unfortunately, *ATS* does not consider yet *XML* artifacts[4]. Since the processes are *XML* documents[5], production processes are not refined as such. A layer always overrides the *build.xml* file of a previous layer so that the *build.xml* of the last layer is the only one that endures.

This implies that layers should be aware of how to compose the whole set of artifacts down in the refinement hierarchy. This could be a main stumbling block to achieve a loose coupling among layers, and leads to increasingly complex *build.xml* files as the layer hierarchy grows.

Turning production processes into first-class artifacts makes production processes liable to be refined as any other artifact. This permits to account for both ***product features*** and ***process features***. By product features we mean those that characterize the product as such (i.e., impact on the build process), whereas process features refer to variations on the associated synthesis process.

It is worth noting that product features commonly impact the build process (i.e., the process adds a new artifact to build the end-product). By contrast, process features influence the synthesis process (i.e., the process that indicates how feature layers are composed).

---

[2] In the published version [DTA05], the build process was named ***intra**-layer production process* to denote that this process builds the artifacts within a layer.

[3] In the published version [DTA05], the synthesis process was named ***inter**-layer production process* to denote that this process synthesizes (composes) layers.

[4] At the time of writing our SPLC'05 paper [DTA05], ATS did not support the refinement of XML artifacts. Indeed, this was the reason behind the creation of XAK (see section 3.4.2).

[5] *ATS* names it *ModelExplorer.xml*, but it plays the same role than *build.xml* in traditional Java projects.

### 7.3.2 ATS Upgrades

Different upgrades were conducted into *ATS* to accommodate variability into the production process, namely

- build processes are currently specified as *ant* files. A layer currently overrides the *build.xml* file of the previous layer so that the *build.xml* of the last layer is the only one that prevails. This problem is solved by using XAK (see section 3.4.2). Doing so, the *refinement* operator has been extended to handle extensions of *ant* files. This enables to apply refinements also to *ant* files.

- synthesis process is hard-coded into *ATS*. This synthesis process is made explicit likewise, and hence, subject to refinement.

The next sections illustrate the advantage of bringing refinement to the build and synthesis processes realm through a running example.

## 7.4 A Case Study for Production

*PinkCreek* is the case study introduced in previous chapters. It is roughly a product-line of portlets that provide flight reservation capabilities to different portals. Broadly speaking, a layer comprises the set of artifacts that realize a given feature (see appendix A). This might include a build process. Being in a Java setting, *ant* is used to specify this process; the so-called, *build.xml* file [SC04].

### 7.4.1 Ant Makefile Process

*Ant* is a Java-based tool for scripting build processes. Scripts are specified using XML syntax: <*project*> is the root element whose main child is <*target*>. A target describes a unit of enactment in the build process. This unit can be an aggregate of atomic tasks such as *compile, copy, mkdir* and the like. The process itself (i.e., the control flow between targets) is described through a target's attribute: *"depends"*. A target is enacted as long as the target it depends on, has already been enacted. This provides a backward-style description of the process flow. Data sharing between targets is achieved through the external file directory.

```
<project name="PKProductProcess" default="usage" basedir="."
         xak:artifact="aPKProductProcess" xak:module="mPKProductProcess">
  <property file="build.properties"/>
  <!-- content omitted -->
  <target name="prepare" description="Prepare compilation">
    <mkdir dir="${pk.classes}"/>
  </target>
  <target name="compile" depends="prepare" description="Compile classes under classes">
    <javac srcdir="${pk.src}" destdir="${pk.classes}" debug="on"
           deprecation="on" optimize="off" source="1.4">
      <classpath refid="classpath"/>
    </javac>
  </target>
  <target name="recompile" depends="clean, compile" description="Recompile All for pk"/>
  <target name="all" depends="compile" description="Make All" xak:module="mAll"/>
</project>
```

**Figure 7.1**: Base Build Process

### 7.4.2 The Build Process

Figure 7.1 shows partially a snippet of the specification of a specific build process for the *base* layer[6]. The synthesis process includes the following steps:

1. compile the *Java* classes into byte codes,
2. package the artifacts (classes, libraries, pages, resources, etc) into a *WAR* file,
3. deploy the web application into a container.

The use of *ant* for specifying build processes is not new. After all, *Java* programmers have been using *ant* as a scripting language for years. However, instead of burying it into the integrated development environment, we make it explicit as any other artifact. This enables ant scripts to be refined.

## 7.5 Variability on the Build Process

Previous section describes the base build process of *PinkCreek*. This process might then be refined to account for distinct product-features. The example introduces two features which imply a refinement in the build process, namely

[6]Space limitations prevent us from giving the complete *build.xml* files. Some targets are collapsed or omitted and variables are defined in external properties files (<*property file="build.properties"*/>).

```
<xak:refines xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xak="http://www.atarix.org/xak"
             xak:artifact="mPKProductProcess">
  <xak:extends xak:module="mPKProductProcess">
    <xak:super xak:module="mPKProductProcess"/>
    <target name="prebuild" depends="compile" description="Copy files to prepare WAR file generation">
      <mkdir dir="${pk.deploy}/${pk.name}"/>
      <!-- content omitted -->
    </target>
    <target name="build" depends="prebuild" description="Generate packaged WAR file"
          xak:module="mTargetPrebuild">
      <zip destfile="${pk.deploy}/${pk.name}.war" basedir="${pk.deploy}/${pk.name}"/>
    </target>
    <target name="deploy" depends="build" description="Generate packaged WAR file">
      <delete dir="${tomcat.home}/pks/${pk.name}"/>
      <delete file="${tomcat.home}/pks/${pk.name}.war"/>
      <copy todir="${tomcat.home}/pks" file="${pk.deploy}/${pk.name}.war"/>
    </target>
  </xak:extends>
  <xak:extends xak:module="mTargetAll">
    <antcall target="deploy"/>
  </xak:extends>
</xak:refines>
```

**Figure 7.2**: Refinement for Product Feature *Tomcat*.

- the container feature. The variants include *Tomcat*[7] and *JBoss*[8]. By default, there is no *base* web container[9],

- the locales feature. The alternatives are *EN* (English), *es_ES* (Spanish in Spain), and *eu_ES* (Basque in Spain). The base locale is *EN*.

One possible layer composition is expression: *es_ES(Tomcat(base))*.

### 7.5.1 Base Build Process

The base build process is contained in the base layer together with other artifacts, whereas the other layers contain either refinements on existing artifacts or new artifacts. The important point to note is that both *Tomcat* and *es_ES* features imply the refinement of the synthesis process as well. That is, deploying *PinkCreek* in *Tomcat* requires to refine the *build.xml* accordingly.

```
<xak:refines xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xak="http://www.atarix.org/xak"
             xak:artifact="mPKProductProcess">
  <xak:extends xak:module="mTargetPrebuild">
    <xak:super xak:module="mTargetPrebuild"/>
    <delete file="${pk.deploy}/${pk.name}/WEB-INF/classes/view/ApplicationResources.properties"/>
    <copy file="public_html/WEB-INF/classes/view/ApplicationResources_es_ES.properties"
          tofile="${pk.deploy}/${pk.name}/WEB-INF/classes/view/ApplicationResources.properties"/>
  </xak:extends>
</xak:refines>
```

**Figure 7.3**: Refinement for Product Feature *es_ES*.

### 7.5.2 Refinement Build Process

*AHEAD* does not provide a way to refine *XML* artifacts[10]. However, Batory et al. state the Principle of Uniformity whereby *"when introducing a new artifact* (type), *the tasks to be done are (1) to support inheritance relationships between instances and (2) to implement a refinement operation that realizes mixin inheritance*" [BSR04]. This principle is realized by XAK for the refinement of XML artifacts. Likewise, this is realized for *build.xml* artifacts as follows. Inheritance is supported by building on the uniqueness of the name (i.e., *name* attribute within <*xak:module*>) within a given project (i.e., a given document). Basically, the project maps to the notion of class, and the target corresponds to a method (though XAK allows other modularization). This permits to re-interpret inheritance for *ant* artifacts by introducing the following tags:

1. <*xak:refines*> which denotes a refinement (a kind of "is_a"),
2. <*xak:extends*> which denotes to extend a given element (e.g., <*target*>) within a document (e.g., <*project*>),
3. <*xak:super/*> which is the counterpart of the *"super"* constructor found in object-oriented programming languages.

Hence a <*xak:refines*> can refine a <*project*> by introducing a new <*target*>, extending a previously existing <*target*> (calling <*xak:extends*> and <*xak:super*>)

[7] http://jakarta.apache.org/tomcat/

[8] http://www.jboss.org/

[9] A design rule can be used here to ensure that the final product will have a container.

[10] This was at the time of this writing. Nowadays, this is not accurate since XML refinement is supported by XAK into AHEAD (see section 3.4.2). The following text is partially a rewording from the published version. The syntax used is slightly different to the published version where a previous version of XAK was described [DTA05]. The original syntax is replaced by the use of XAK to better fit with section 3.4.2.

or overriding a <*target*> (by using <*xak:extends*> to introduce this target with new content).

An example is given for the *Tomcat* feature (see figure 7.2). Feature *Tomcat* permits to deploy *PinkCreek* in the namesake container. This requires the refinement of the *build.xml* artifact found in the base layer, as follows:

- a new <*target*> is added to prepare the *WAR* building (prebuild),
- a new <*target*> is added to build the *WAR* (build) specific for *Tomcat*,
- a new <*target*> is added to deploy it into the *Tomcat* container,
- target <*target name= "all"*> is overridden (i.e., specific syntax is <*xak:extends name= "mTargetAll"*>).

Likewise, feature *es_ES* overrides the English locale of the base build process to the Spanish locale. The counterpart refinement is shown in figure 7.3. It extends <*target name= "prebuild"*> to copy the appropriate resource files. Here the <*xak:super/*> constructor is used[11].

Both examples illustrate how refinements have been realized for *ant* artifacts. Implementation wise, the composition operator for *ant* is implemented using *XAK* (see Section 3.4.2). Hence, when *build.xml* artifacts are found, the composition process is governed by the XAK composer to compose *ant* artifacts.

## 7.6 Variability on the Synthesis Process

### 7.6.1 Base Synthesis Process

Previous section focuses on *ant* artifacts found within a layer. These artifacts describe the build process to produce a product within a layer. By contrast, this section focuses on synthesis processes that state how layers themselves should be composed. This comprises the steps of the synthesis methodology being used. For AHEAD, these steps include:

1. feature selection. Output: a feature equation (e.g. *"es_ES(Tomcat(base))").*

[11] It is worth noticing that the *es_ES* refinement requires the container been already selected. This implies a design rule to regulate how layers are composed.

```
<project name="PKSynthesisProcess" default="usage" basedir="." xmlns=""
         xmlns:xak="http://www.onekin.org/xak" xak:artifact="aPKSynthesisProcess"
         xak:module="mPKSynthesisProcess" xak:type="xml">
  <target name="compose" description="Compose layers" xak:module="mTargetCompose">
    <echo message="Composing layer equation for product: '${equation.name}.equation'"/>
    <ant antfile="${ahead.home}/build/lib/ModelExplorer.xml" target="composer">
      <property name="layer" value="${equation.name}"/>
      <property name="jts.home" value="${ahead.home}"/>
    </ant>
  </target>
  <target name="compose-build-xml" depends="compose" description="Run XRefine for build.xml">
    <echo message="Composing 'build.xml' for product: '${equation.name}.equation' "/>
    <ant antfile="${xrefine.home}/build-run.xml" target="xmlcomposer">
      <property name="equation.name" value="${equation.name}"/>
      <property name="artifact.name" value="build.xml"/>
      <property name="layers.base.dir" value="${layers.home}"/>
      <property name="xrefine.home" value="${xrefine.home}"/>
    </ant>
  </target>
  <target name="execute-build-xml" depends="compose-build-xml" description="Execute build.xml">
    <echo message="Executing 'build.xml' for product: '${equation.name}.equation'"/>
    <ant antfile="build.xml" dir="${equation.name}" target="all"/>
  </target>
  <target name="produce" description="Produce a Product" xak:module="mTargetProduce">
    <antcall name="execute-build-xml"/>
  </target>
</project>
```

**Figure 7.4**: Base Synthesis Process.

2. feature synthesis (i.e., layer composition in Batory's parlance). Output: collective of artifacts that support an end-product[12].

3. enactment of the *build.xml* associated with the end-product. Output: end-product ready to be used.

Figure 7.4 illustrates the targets that realize previous steps *(*the *equation.name* property holds the feature equation):

- *compose*, which calls the *ATS* composer,
- *compose-build-xml*, which supports the composition operator for the *build.xml* artifact that *ATS* lacks[13],
- *execute-build-xml,* which runs the *ant* script supporting the build process of the end-product,
- *produce*, which performs the whole production (using previous targets).

[12] At the time of this writing, we did not even consider the use of model derivation together with composition. Nonetheless, currently model derivation is an important part of the synthesis process. So, it might be subject to variability as well.

[13] Nowadays, this process is not necessary since XAK is integrated into ATS. However, the code of this figure sticks to the original published version [DTA05].

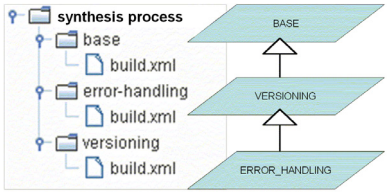



**Figure 7.5**: Refinement Layers: each accounts for a "process feature".

```
<xak:refines xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:xak="http://www.atarix.org/xak" xak:artifact="aPKSynthesisProcess">
  <xak:extends xak:module="mPKSynthesisProcess">
    <xak:super xak:module="mPKSynthesisProcess"/>
    <target name="versioning" depends="execute-build-xml" description="Saves curren
      <zip destfile="${equation.name}.zip">
        <fileset dir="${equation.name}"/>
        <fileset file="${equation.name}.equation"/>
      </zip>
      <copy file="${equation.name}.zip" todir="${repo.home}"/>
      <delete file="${equation.name}.zip"/>
      <property name="svn.exe" location="${svn.home}/bin/svn.exe"/>
      <echo message="Adding to SVN"/>
      <exec executable="${svn.exe}" spawn="false" dir="${repo.home}">
        <arg value="add"/>
        <arg value="${equation.name}.zip"/>
      </exec>
      <echo message="Commiting to SVN"/>
      <exec executable="${svn.exe}" spawn="false" dir="${repo.home}">
        <arg value="commit"/>
        <arg value="-m 'Equation: ${equation.name}' "/>
      </exec>
    </target>
  </xak:extends>
  <xak:extends xak:module="mTargetProduce">
    <antcall target="versioning"/>
  </xak:extends>
</xak:refines>
```

**Figure 7.6**: Refinement for Process Feature *versioning*.

The enactment of this synthesis process leads to an end-product that exhibits the process features of the input equation. *ATS* hard-codes this script.

However, this work rests on the assumption that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the artifacts but on the synthesis process itself. From this viewpoint, the synthesis process can accommodate important production strategies that affect the synthesis process rather than the characteristics of the final product. These strategies can affect the product costs, increase product quality, or improve the synthesis process.

### 7.6.2 Refinement Synthesis Process

Based on this observation, the previous *base synthesis process* might be refined to account for distinct "process-features" (see Figure 7.5). The example introduces

```
<xak:refines xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:xak="http://www.atarix.org/xak" xak:artifact="aPKSynthesisProcess">
  <xak:extends xak:module="mPKSynthesisProcess">
    <xak:super xak:module="mPKSynthesisProcess"/>
    <taskdef resource="net/sf/antcontrib/antlib.xml"/>
  </xak:extends>
  <xak:extends xak:module="mTargetCompose">
    <!-- @Override this target -->
    <target name="compose" description="Compose layers Manage possible errors">
      <trycatch property="exception.message" reference="exception.refObject">
        <try>
          <xak:super xak:module="mTargetCompose"/>
        </try>
        <catch>
          <echo>CATCH EXCEPTION: Get last product from version system</echo>
          <delete dir="${layers.home}/${equation.name}"/>
          <mkdir dir="${layers.home}/${equation.name}"/>
          <unzip src="${repo.home}/${equation.name}.zip" dest="${layers.home}/${equ
        </catch>
      </trycatch>
      <echo>Exception Property: ${exception.message}</echo>
      <property name="exception.object" refid="exception.refObject"/>
      <echo>Exception reference: ${exception.object}</echo>
    </target>
  </xak:extends>
</xak:refines>
```

**Figure 7.7**: Refinement for Process Feature *errorHandling*.

two features which imply a refinement in this process, namely

- the *versioning* feature. Consider that security reasons recommend to version each new delivery of an end-product. This implies that artifacts that conform the end-product, should have appropriate backups.
- the *errorHandling* feature. Errors can rise during the production processes. How these errors are handled is not a characteristic of the product but depends on managerial strategies. Hence, the base synthesis process can be customized to support distinct strategies depending on the availability of resources or the quality requirements of the customer.

Figure 7.6 shows how the *base* synthesis process can now be refined to account for the *version* feature, namely:

- a new <*target name=”versioning”*> is added to back up artifacts into the versioning system. For this purpose, *Subversion* is used[14],
- target <*target name= “produce”*> is overridden (using <*xak:extends*>).

The equation *versioning(base)* leads to a synthesis process that supports the naive security policy of the organization. As further experience is gained, and stringent demands are placed, more sophisticated plans can be defined.

[14] http://subversion.tigris.org/

Likewise, figure 7.7 shows the *"substitution_eh"* policy for error handling:

- a new <*taskdef/*> is added in order to extend *ant* targets with try&catch routines[15].
- target <*target name= "compose"*> is overridden in order to handle possible errors. The base task *"compose"* is monitored so that when an error occurs, the last error-free version of the artifacts outputted by *"compose"* are taken. This policy might be applicable under stringent time demands or if debugging programmers are on shortage.

The process feature expression (*"substitution_eh(version(base)))"*) then strives to reflect the managerial and strategic decisions that govern the production process. Making theses strategies explicit facilitates knowledge sharing among the organization, facilitates customization, eases evolution, and permits to manage resources for product synthesis in the same way as the product itself.

The latter is shown for the *version* and *errorHandling* features: a design rule is needed to state that the *substitution_eh* policy requires the *version* feature to be in place. The *version* feature in turn requires a new artifact, namely, *subversion*. It is a well-known fact among programmers of complex systems, that setting the appropriate environment is a key factor for efficient and effective throughput. SPL are complex systems, and SPL techniques should be used not only to manage the artifacts of the product itself, but also those artifacts that comprise the environment/framework where these products are built. These include a large number of artifacts such a compilers, debugger, monitors or backup systems. Making explicit the synthesis process facilitates this endeavour.

## 7.7 Contributions

The clear separation between artifact construction and artifact synthesis is one of the hallmarks of software product lines. However, little attention has been devoted to the production process itself, and how this process might realize important process strategies.

This work strives to illustrate the benefits of handling production processes as first-class artifacts, namely *(i)* it permits to focus on how the product is synthesized rather than on what the product does. Programmers and assemblers can wide their

[15] This is achieved using Ant-Contrib (from http://ant-contrib.sourceforge.net/).

minds to ascertain how features might affect the production process itself so that scripting is no longer seen as a byproduct of source code writing, and *(ii)* it extends variability to the production process.

Using *ant* for process specification, and *AHEAD* as the SPL methodology, this work illustrates this approach for a sample application. Our future work is to increase the evidence of the benefit of this approach by addressing more complex problems, and to investigate the impact that distinct SPL quality measures have into the production process.

The previous chapter introduces GROVE (*GeneRative metaprOgramming for Variable structurE*). So far, the variability is not address into the generative metaprogramming. Future work might study the implications of variability into the synthesis geometry.

The work of this chapter was presented in the paper:

1. *Supporting Production Strategies as Refinements of the Production Process*. O. Díaz, S. Trujillo and F. I. Anfurrutia. 9th International Software Product Lines Conference (SPLC 2005). Rennes, France. September 2005 [DTA05]. Acceptance Rate: 23 % (17+3/71).

# Chapter 8

# Conclusions

"An expert is a person who has made all the
mistakes that can be made in a very narrow field"

*– Niels Bohr.*

## 8.1 Abstract

This dissertation analyzed a combination of *Feature Oriented Programming* (FOP) and *Model Driven Development* (MDD) into *Feature Oriented Model Driven Development* (FOMDD) where a non-trivial example for a product line of *Portlets* was used throughout to assess the applicability of presented ideas.

This chapter reviews our central results and primary contributions, evaluates the limitations of this work, and proposes new areas for future research.

## 8.2 Results and Contributions

There are three strategies to improve software *productivity*: working faster, working smarter and avoiding unnecessary work [Boe99]. The latter promises the highest payoff and can be achieved by software reuse approaches where FOP and MDD can be regarded as ways to capitalize on given reuse efforts [Mut02].

We review the concrete *contributions* of our research in more detail below:

- In chapter **3,** a valuable case study on the scalability of feature-based multiple-representations program refactoring and synthesis was presented. ATS is the largest program, by almost two orders of magnitude, that we have feature refactored from a program onto a product line. Refining and composing XML documents was critical to our work, and we were able to verify a correct feature refactoring by using regression tests. And most importantly, our work revealed generic problems, solutions, and an entire suite of tools that could be created to simplify future feature refactoring tasks.

- Chapter **5** combined *Feature Oriented Programming* and *Model Driven Development*. *FOMDD* (Feature Oriented Model Driven Development) is a blend of both. The new challenges, such as model refinement to build MDD-increments, were presented. Additionally, this work exposed properties of FOMDD synthesis (e.g., commuting diagrams) that ultimately enabled us to check correctness properties of our models and tools. This experience exposed the nature of synthesis where commuting was simply symptomatic of fundamental structures that were "behind the scenes".

- Chapter **6** described ideas to synthesize metaprograms, which when executed, will synthesize a target program of a product-line. Specifically, we elaborated on the generation of metaprograms from abstract specifications. A case study was used to illustrate the *GeneRative metaprOgramming for Variable structurE*. GROVE automates significant and tedious tasks in synthesis metaprogramming. The benefit of this approach is that it is no longer necessary to create/modify the metaprogram implementation, but its specification. This reduces considerably the development time of synthesis metaprograms and facilitates their evolution.

- Chapter **7** described how variability was considered into the production processes. Specifically, it enables the refinement of production processes (build and synthesis process).

- Appendix **A** detailed a complete case study of Portlets where a software product line was developed. This chapter explored the *key challenges* while facing Portlets from a product-line perspective.

### 8.2.1 Publications

Parts of the results presented in this thesis have been presented and discussed before on distinct peer-review forums. The distinct publications in which the author of this thesis was involved are listed below.

#### Selected Publications

- *Feature Oriented Model Driven Development: A Case Study for Portlets*. S. Trujillo, D. Batory and O. Diaz. 29th International Conference on Software Engineering (ICSE 2007). Minneapolis, Minnesota, USA. May 2007 [TBD07]. Acceptance Rate: 15% (50/334).

- *Turning Portlets into Services: Introducing the Organization Profile*. O. Diaz, S. Trujillo, and S. Perez. 16th International World Wide Web Conference (WWW2007). Banff , Canada. May 2007 [DTP07]. Acceptance Rate: 14% (110/750).

- *Feature Refactoring a Multi-Representation Program into a Product Line*. S. Trujillo, D. Batory and O. Diaz. 5th International Conference on Generative Programming and Component Engineering (GPCE 2006). Portland, Oregon, USA. October 2006 [TBD06]. Acceptance Rate: 28 % (25+5/88).

- *Supporting Production Strategies as Refinements of the Production Process*. O. Díaz, S. Trujillo and F. I. Anfurrutia. 9th International Software Product Lines Conference (SPLC 2005). Rennes, France. September 2005 [DTA05]. Acceptance Rate: 23 % (17+3/71).

#### International Conferences/Workshops

- *On the Modularization of Feature Models*. D. Benavides, S. Trujillo and P. Trinidad. 1st EWMT Workshop (jointly with SPLC 2005). Rennes, France. September 2005 [BTT05].

- *Enhancing Decoupling in Portlet Implementation*. S. Trujillo, I. Paz and O. Díaz. 4th International Conference on Web Engineering (ICWE 2004). Munich, Germany. July 2004 [TPD04]. Acceptance Rate: 12% (25+60/204)[1].

[1] Our contribution was accepted as poster where 60 research papers were included, as either short papers or posters. This makes a global acceptance rate of 41%.

- *User-Facing Web Service Development: a Case for a Product-Line Approach*. O. Díaz, S. Trujillo and I. Azpeitia. VLDB-TES (Workshop). Berlin, Germany. September 2003 [DTA03].

- *Moving Co-Branding to the Web: Service-Level Agreement Implications*. O. Díaz and S. Trujillo. PROVE 2003 (Conference). Lugano, Switzerland. October 2003 [DT03].

**Spanish Conferences**

- *Experience Measuring Maintainability in Software Product Lines*. G. Aldekoa, S. Trujillo, G. Sagardui, O. Díaz. JISBD 2006. Sitges, Spain. October 2006 [ATSD06]. Acceptance Rate: 33 % (40/120).

- *A Product-Line Approach to Database Reporting*. F. I. Anfurrutia, O. Diaz and S. Trujillo. JISBD 2005. Granada, Spain. September 2005 [ADT05]. Acceptance Rate: 31 % (29+10/92). This paper was among best 10 conference papers (it was later published in a journal version [ADT06b]).

**Journals**

- *Una Aproximación de Línea de Producto para la Generación de Informes de Bases de Datos* (spanish version of [ADT05]). F. I. Anfurrutia, O. Diaz and S. Trujillo. IEEE América Latina Journal 4 (2). April 2006 [ADT06b].

**Tutorials**

- *An Introduction to Software Product Lines*. O. Diaz and S. Trujillo. JISBD 2006. Sitges, Spain. October 2006 [DT06].

**Drafts under Review**

- *On Refining XML Artifacts*. F. I. Anfurrutia, O. Diaz and S. Trujillo. Draft. November 2006 [ADT06a].

### 8.2.2 Research Visits

The aim of this work was to be open, influenced and enriched by distinct research streams, works, visions and schools. Thus, along this work two *research visits* were accomplished. The author was first visiting the informally called Product

Line Research Group headed by Prof. Dr. Don Batory at the University of Texas at Austin (USA) from January to April 2006. Later, a short visit for a week was done by the end of October 2006. The author was also visiting the department of Product Line Architectures headed by Dr. Dirk Muthig at the Fraunhofer IESE at Kaiserslautern (Germany) from July to September 2006. Both visits fostered discussion and eventually imposed new perspectives on this work that otherwise would not be reached.

## 8.3 Assessment

The work presented so far reveals insights combining Software Product Lines and Model Driven Development in the synthesis of Portlet applications. Although our work exposed some challenges, a close assessment is necessary to reveal some limitations of this work and propose some future work.

### 8.3.1 Limitations

- **Portlet Modeling approach**: follows an approach to MDD where code generation is not complete. Although this approach automates cumbersome tasks, it requires some human intervention (e.g., to complete skeletal generated code). This is not exceptional, but is common in other approaches. Nevertheless, future work should address the generation of further code. This would likely embrace the definition of further models.

- **Techniques Generalization**: FOMDD is a general paradigm combining FOP and MDD. As pointed out in chapter 2, there are many techniques for each of them. So far, we used AHEAD for FOP and XSL for MDD. We also make some proofs with other MDD engines (e.g., RubyTL [CMT06]). More work is needed to assert that our approach supports the general use of other techniques.

- **Scope**: the ideas presented in this work were used with a product-line of portlets that we developed. This worked fine so far. However, *further case studies* (industrial if possible) are needed to validate our findings. We know that an ongoing external research work is currently looking for commuting diagrams with other cases.

To overcome these limitations is subject of future work.

### 8.3.2 Future Research

*Feature Oriented Model Driven Development* and Portlets (in general, service-oriented applications) are still in their infancy. They are promising for the future of software construction and offer several interesting directions for future research. Particularly, FOMDD requires further tool support to generalize its adoption to other domains and case studies. To contribute towards those goals, our work presents an agenda for future research.

- **FOMDD Engineering**. A centerpiece for *practitioners* guidance in industrial application are engineering processes. *Model Driven Engineering* (MDE) emerged to support the general MDD paradigm for software development [Ken02]. *Software Product Line Engineering* provides similar support in terms of processes. Further work is needed to combine both together into a common engineering processes similarly to [AFM05].

- **FOP-UML.** *Feature Oriented Programming* (FOP) is a product-line paradigm where artifact composition mechanisms are provided. This work applies FOP mechanisms to the composition of models (i.e., a base model is extended with a refinement model). Doing so, composition of artifacts comprises also models. However, there is not yet a UML profile to support this. This is the subject of ongoing work [LH07].

- **Theoretical Support**. Commuting diagrams revealed a mathematical insight of FOMDD. This suggested that some structural theory was behind. *Category theory* was recently proposed as such theory [Bat07a]. Further work is needed to formally elaborate all the implications of this theory.

- **GROVE Algebra**. AHEAD is an algebra to represent structure and composition of structures in a product-line setting. Likewise, GROVE offers a structural model to represent synthesis in a product-line setting. GROVE structure would require an algebra to formalize it. GROVE is closely related to the ideas behind *Architectural Metaprogramming* that are promising for the future [Bat07b].

- **FOMDD Refactoring**. FOMDD combines two paradigms for the creation of SPL starting from scratch. This is not the situation in many cases where a legacy program already exists. *Feature Oriented Refactoring* is a paradigm

to refactor such existing program into a product-line [TBD06], whereas *Reverse Engineering to Model Driven* (a.k.a., harvesting or refactoring) is a paradigm to refactor an existing program into a modeling approach [RGvD06]. The challenge is how to combine both together (in the same way as FOMDD did with FOP and MDD) to yield *FOMDD refactoring*. We are seeking for case studies on this subject. We are aware that Freeman is currently working on a promising case [FB07].

- **Portlet Issues**. Although Portlets are merely used as a case study in this work, alone they deserve specific research attention. On the one hand, some interests are inherent of Portlets (e.g., orchestration [DII05]). On the other hand, they impose new scenarios for a product-line setting and for a model-driven setting (e.g., service-orientation). The exploration of the impact of service-orientation distributed production is the subject of forthcoming work [Tea07].

- **Tool Support**. Some tools were implemented to support our ideas (e.g., XAK, GROVE Tool Suite, etc). Although we spent a great amount of effort on them to work, still they require further work to be used by a regular customer.

# Appendix A

# Portlet Product Lines: A Case Study

"Any fool can make history, but it takes a genius to write it."
*– Oscar Wilde.*

## A.1 Abstract

*Families of Portlet* applications are steadily emerging to meet the requirements of *different Portal customers*. Conference management, ticket reservation, etc, are services gathered by distinct Portals from assorted Portlets. This scenario offers a large potential to customize a Portlet for a target Portal where variability issues should be resolved beforehand. *Software Product Lines* offer a paradigm to face this challenge in a cost-effective way. Portlets impose some idiosyncrasies (exposed in this chapter) that make them different from traditional software. Hence, a number of challenges appear while facing a *Product line of Portlets*. This work describes an approach where those issues were exposed. A working example where a family of Portlets is developed illustrates the product-line approach.

## A.2 Rationale for *Portlet Lines*

Most web applications (also Portlets) are conceived in a one-to-one basis nowadays. This crafted approach is also found on the early stages of the production of other items such as engines, cars, planes, etc. However, as the technology matures and a better comprehension of the domain is obtained, the production process evolves from single-item production to product lines which are capable of delivering a whole family of items. This transition has to do with the production benefits of product lines. Software is not an exception.

*Software Product Lines* (SPL) present a paradigm to build a family of applications. The adoption of this paradigm is reported in several case studies for traditional software [CN01]. Some cases are even reported for web applications (see section 2.5.2). In general, these experiences reported a number of general and potential benefits, namely, *(i)* productivity gains, *(ii)* improved product quality, *(iii)* faster time-to-market, and *(iv)* decreased labour efforts [CN01, Coh01]. However, as in the real world, it depends on the specific case at hand. The same is applicable for Portlets.

The web community is maturing to reach the SPL momentum. Indeed, we believe that the web is more inclined to the SPL paradigm than traditional software. First, its ubiquitousness makes web applications reach a broader spectrum of customers, markets and cultures. While traditional software has a more limited scope, web applications tend to cope with a more diverse set of stakeholders. The emphasis that the web community places on personalization/customization/privacy is a case in point. SPL techniques enable to face these issues in a cost-effective way.

A second argument is the heterogeneous and quick pace at which web technology is evolving. SPL facilitate the coexistence of the same product being delivered in distinct platforms or developed using distinct technologies (*.NET* vs *J2EE*, *JSP* vs. *ASP*, etc). The variability of the SPL paradigm enables to face this heterogeneity.

A third argument is the service-orientation which changes not only the way in which applications are operated, but even the way in which they are constructed. Typically, services use other services (i.e., several Portlets are involved to satisfy a Portal). Those issues are to be handled in this setting.

Finally, web applications have been reported to have shorter life cycles than no-web software [Ove00]. An essential reason for introducing SPL is the reduction of costs and time-to-market. Once the upfront investment of developing the SPL is

done, custom products are developed quicker, cheaper and at higher quality levels.

Specifically, this chapter elaborates on those differences that cause a number of challenges while facing a *Product line of Portlets*. To illustrate them, a working example on the development of a family of Portlets (where those issues are faced) is used to show the product-line approach.

## A.3 Product Lines of Portlets

**SPL** are defined as "*a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [CN01]. Previous statement is refactored onto five major issues:

- **Products**: "*a set of software-intensive systems...*". SPL shift the focus from single software development to SPL development. The development processes are not intended to build one application, but a number of them.
- **Features**: "*...sharing a common, managed set of features...*". Features are units (i.e. incrementing application functionality) by which different products can be distinguished and defined within an SPL [BSR04].
- **Domain**: "*...that satisfy the specific needs of a particular market segment or mission...*". An SPL is created within the scope of a domain. A domain is "*a specialized body of knowledge, an area of expertise, or a collection of related functionality*" [Nor02].
- **Core Assets**: "*...are developed from a common set of core assets...*". A core asset is "*an artifact or resource that is used in the production of more than one product in a software product line*" [CN01].
- **Production Plan**: "*...in a prescribed way*". It states how each product is produced. The production plan is "*a description of how core assets are to be used to develop a product in a product line and specifies how to use the production plan to build the end product*" [CM02]. It ties together all the reusable assets to allow end-product production.

### A.3.1 Approaching

SPL development process is not intended to build one application, but a number of them. This forces a change in the engineering processes where a distinction

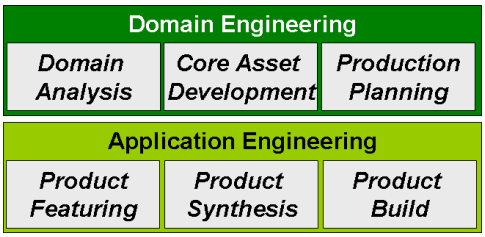



**Figure A.1**: SPL Engineering Process

between *domain* and *application* engineering is introduced. In general, *domain engineering* (a.k.a., core asset development) determines the commonality and the variability of the SPL, whereas *application engineering* (a.k.a., product development) produces individual products from the SPL. Doing so, the construction of the reusable assets and their variability is separated from production of the product-line applications. Distinct approaches exist to face SPL [CN01]. Figure A.1 sketches the overall process[1]:

- **Domain Analysis** studies the variability of the domain at hand. Frequently, this study is done in terms of *features* and represented using a feature model.

- **Core Asset Development** conceives, designs and implements the core assets. This not only involves the development of domain functionality, but also defines how core assets should be extended.

- **Production Planning** defines how individual products are created. In general, it involves to set-up a factory capability.

- **Product Featuring** chooses desired features to differentiate a target product. Typically, customers start this process with feature selection.

- **Product Synthesis** assembles core assets to get the raw material (a product is made up of). Typically, variability realization techniques are used.

- **Product Build** processes the raw material following the build process (e.g., compile, deploy, etc) to yield an end-product.

[1] Note that this process is a simplified version with the major activities. For a detailed account on all practices and processes involved, please refer to [CN01].

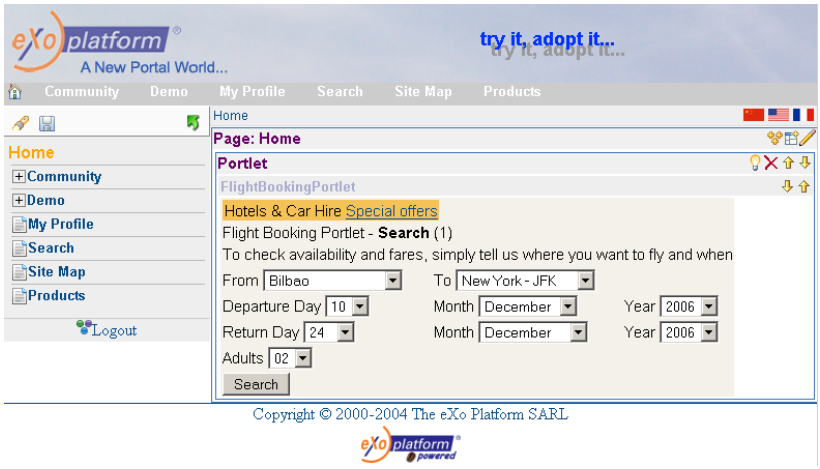



**Figure A.2**: PinkCreek Screenshot

**Impact of Portlet Issues** Although the overall process is similar, some particularities appear while developing Portlets. In general, the differences are in *(i)* how the domain analysis is performed, *(ii)* core asset development depends certainly on the Portlet technology and on the domain at hand, *(iii)* the variability realization is similar than in other approaches, except the way to cope with heterogeneous artifacts (e.g., XML documents), and *(iv)* synthesis process is clearly affected by the distributed nature of Portlets.

### A.3.2 A Case Study: PinkCreek

***PinkCreek*** is a product-line to build a family of portlets that provides flight reservation capabilities to different portals[2]. Its functionality is roughly: *(i)* search for flights, *(ii)* present flight options, *(iii)* select flights, and *(iv)* purchase tickets. Figure A.2 displays a screenshot of PinkCreek Portlet where the initial page showing a flight search form is rendered inside a Portal.

### A.3.3 Domain Analysis

*Feature Oriented Domain Analysis* (FODA) is a domain engineering methodology developed by the CMU Software Engineering Institute (SEI) [Kea90]. It focuses on the SPL development through a structured domain engineering process based

[2]PinkCreek's name is well worth a footnote. Many ideas of this thesis benefit from the jogging on the beautiful shores of Barton creek at Austin, Texas.

on the notion of feature models.

**Feature Modeling** is a main output of domain engineering. A feature is a "*product characteristic that is used in distinguishing products within a family of related products*" [BSR04]. *PinkCreek* consisted of 26 features:

```
PinkCreek = {
  core // portlet base
  assistance // this feature allows to get airport assistance (people with accessibility needs)
  bulletin // this feature allows to subscribe to a newsletter with offers
  checkin // this feature allows check-in capability
  commercialbanner // a commercial banner is added (using a picture)
  commercialstext // a commercial banner is added (using text)
  dbmysql // a mysql dbms is used to access flight information
  exojboss // exo platform is on jboss J2EE container
  exotomcat // exo platform is on tomcat J2EE container
  flightchildren // it enables to pick the number of children
  flightclass // it enables to select the flight class
  includejavadoc // it includes javadoc of the generated code in the delivered product
  includesourcecode // it includes source code in the delivered product
  login // it forces users to login before search for flights
  meal // it allows to select desired meal (after purchase)
  mytrip // it allows a personal place where you can see your trips, ...
  onlydirectflights // it allows to restrict search only to direct flights
  paymentclick // payment depends on the number of clicks
  paymenttransaction // payment depends on the number of complete purchases
  quickpurchase // it accelerates the purchase (only 3 steps to purchase)
  refundable // it permits the flight to be refundable
  reservationcheck // a quick access to reservation info
  reservationmodify // modification of reservation
  seat // it allows to select flight seat (after purchase)
  testmode // it activates a test-mode (to be used only during production time)
  weather // check the weather at destination
}
```

The feature model characterizes the SPL in terms of the supported variability. Among the distinct variations available in the domain, the SPL domain analyst should decide which one to provide to the different stakeholders, and the variants or alternatives to be supported/excluded.

Figure A.3 depicts partially[3] a feature model for our sample problem using a notation similar to [CA05a]. The model organizes features into a composition hierarchy where optionality is annotated. Portlet products are then characterized in

[3]For simplification, this feature model does not show all features. Dependencies between features are omitted as well [Ben07] (see section 3.3.1).

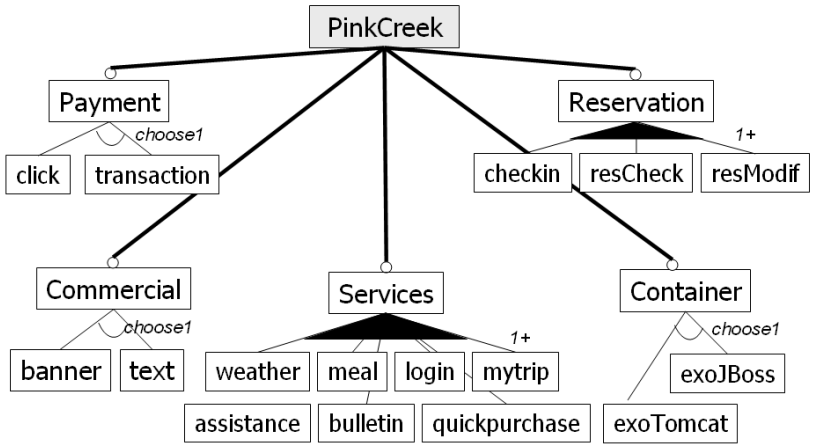



**Figure A.3**: *PinkCreek* Feature Model

terms of features, and the scope of the SPL is given by its feature model. Consequently, an SPL must support variability for those features that tend to differ from product to product.

### A.3.4 Core Asset Development

**Platform Creation** The domain engineer should be experienced enough to determine the common ground shared by all family products (i.e., the platform). A **platform** of an SPL is *"the set of software subsystems and interfaces that form a common structure from which a set of derived products can be efficiently developed and produced"* [ML97]. Its constituent parts are referred to as **core assets** [CN01]. It may include the architecture, software components, design models, etc. In general, any artifact that is liable to be reused [PBvdL06]. The platform is the base on top of which products are created adding (feature) variability. Shortly, we will see distinct techniques to realize variability. First, a platform for Portlets is described.

**Portlet Platform** *PinkCreek* follows a *J2EE* architecture [SSJ02], and specifically a *JSR168* [JCP03]. Portlet architecture is based on the model-view-controller paradigm [KP88]. Besides the architecture, the commonality of the SPL is supported by the following types of artifacts:

- ***Model:*** the business logic of the application (i.e., flight search, flight checkin, and so on) is implemented by *Java* class files.

- ***View***: the application is rendered through server pages to interact with the end-user. *Java Server Pages* (JSP) are used for this purpose.
- ***Controller***: drives the flow of the application in executing actions and rendering views (see section 4.4 for specific models).
- ***Deployment descriptors***: which configure how the application is to be deployed into a container (*web.xml* in a *J2EE* setting, and *portlet.xml* in *JSR168*).
- ***Build Script***: which compiles code, packages application artifacts, and deploys them into a WAR[4]. The *ant*[5] scripting language is used for this purpose.
- ***Miscellaneous***: the application contains other artifacts such as images, CSS resource files, configuration files, tag libraries, DTD/XML Schemas and JAR libraries.

These core artifacts realize the base ground on top of which any family product is built upon. But this is feasible only if these artifacts have been engineered for variability. Built-in flexibility is the means to ensure reuse. Next sections look at realizing variability of these assets.

**Variability Realization** Handling variability in a cost-effective way, implies engineering core artifacts for variability in a planned way. Distinct techniques have been proposed to achieve variability (see section 3.2.1). We select Feature Oriented Programming and its realizing model: AHEAD.

*Feature Oriented Programming* (FOP) is a paradigm of SPL synthesis where features are the building blocks of products (see section 3.3). Features are units (i.e., incrementing application functionality) by which different products can be distinguished and defined within an SPL [BSR04]. In general, an SPL is characterized by the set of features it supports, (e.g., *PinkCreek={checkin, seat, core}*) whereas a product is obtained as the synthesis of some of those features in the base or platform of the SPL (e.g., *P_A* from *customerA* selection is *P_A = seat • core* where • stands for synthesis or composition).

*Algebraic Hierarchical Equations for Application Design* (AHEAD) is a model of FOP where each feature implementation (a.k.a., layer) encapsulates the set of

[4]Portlets applications are packaged into a *Web ARchive* (WAR) which follows a directory structure defined in *Java Servlet Specification* [CY03].

[5]http://ant.apache.org/

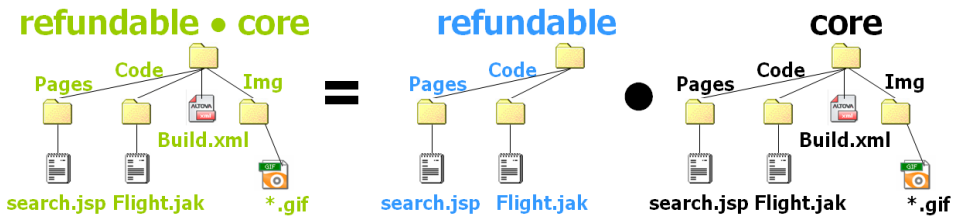



**Figure A.4**: Feature Composition for the Expression *refundable•core.*

files (a.k.a., artifacts) realizing its functionality [BSR04]. Feature realization is described as increments to functionality (a.k.a., refinements) that provides the required feature. The refinement depends on the artifact at hand. It is not the same the notion of refinement in Java that in XML (see section 3.4.2).

AHEAD provides *(i)* a general model to represent variability based on incremental development (*Step-Wise Refinement* [Dij76]) where *(ii)* a feature is bound to a set of artifacts; and *(iii)* a particular variability realization technique representation for refinements. Additionally, it describes, *(iv)* a particular composition mechanism for such refinements; *(v)* this composition is polymorphic enabling multiple and heterogeneous artifact representations; *(vi)* there are also free tools available; and all together *(vii)* allow to build a product automatically starting from customer selected features.

The set of artifacts that realize a given feature are composed using AHEAD. In general, there is a base layer (commonality) and some optional features (variability). For our sample problem, the ***core* feature layer** includes different types of artifacts that set the grounds for the model, view, controller, deployment descriptor, build script, CSS default style guidelines and so on. Subsequent feature layers provide the extensions (a.k.a., refinements) that permit enhancing these core artifacts with a specific feature. Figure A.4 shows this situation where the *refundable* feature layer is composed with the *core* feature layer. However, the *core* feature layer should be engineered for variability beforehand. This implies to define extensions points (i.e., design for variability[6]).

**Feature Realization** A layer is the implementation of a feature (in the AHEAD parlance). *Layer composition* (denoted by the • operator) implies the composition

[6]The platform should be engineered for variability (i.e., a class offers methods that may be extended). Likewise, web artifacts offer modules. The design of the platform states also how the platform is extended (see section 3.4.2).

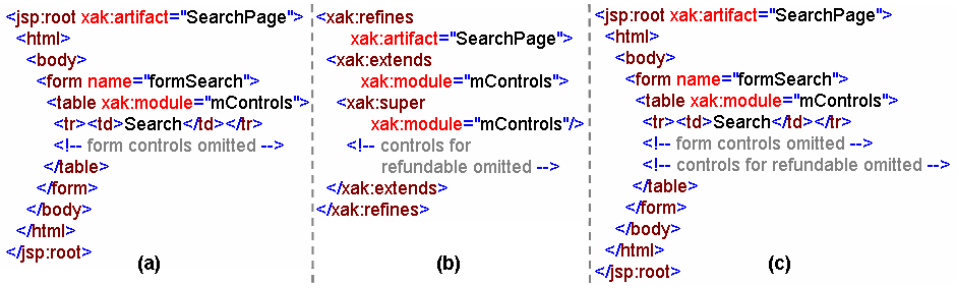



**Figure A.5**: JSP Refinement

of the namesake artifacts found in each layer. Implementation wise, a layer is a directory. Hence, feature composition is *directory composition* which, in turn, implies the composition of the contained artifacts. Figure A.4 shows this situation where the *search.jsp* file found in the *core* layer is later refined by the namesake file at the *refundable* layer. Note that feature realization impacts commonly on more than one artifact (i.e., features are cross-cutting concerns). Hence, a layer frequently refines/introduces more than one artifact. *PinkCreek*'s features are no exception, as shown shortly in table A.1.

Figure A.5 shows the refinement of the *search.jsp* artifact. This refinement is done using XAK (see section 3.4.2). Figure A.5a represents the base artifact (that belongs to the core layer) where some *xak:modules* for extensions are defined (real file content is extremely simplified). Figure A.5b represents a refinement (of the refundable layer) that extends *xak:module="mControls"* with further form controls to cope with the *refundable* functionality. Figure A.5c shows the result of composing the base artifacts and the refinements to yield a product.

The realization of one feature involves frequently more than one artifact. If a feature can be accounted for by introducing bright new artifacts, then adding this feature is just a question of assembling these artifacts to the platform. This is a regular case, since a feature commonly implies the refinement of existing artifacts. *Seat* is a case in point. The realization of this feature includes *(i)* adding a new view to cope with seating reservation (e.g., the *seating.jsp* artifact); *(ii)* introducing code to deal with seating functionality at the back-end (e.g., the *Seating.jak* artifact); and *(iii)* propagating the GUI event to its model counterpart (i.e., enhancing the controller artifact).

### A.3.5 Production Planning

Artifacts are engineered for variability in terms of features. The next step is to define how the product line synthesizes Portlets. To this end, we need to create a factory.

**Setting Up the Factory** As industrialization of the automobile manufacturing process led to increased productivity and higher quality at lower costs, industrialization of the software development process is leading to the same advantages [GS04]. A *software factory* is defined as "*a facility that assembles (not codes) software applications to conform to a specification following a strict methodology*"[7]. In our case, the specification is in term of features, and the methodology is AHEAD.

We created *PinkCreek's* factory to receive as input the selection of features of the customer, and manufacture as output the Portlet product (a.k.a., production process). Basically, our factory *(i)* composes product feature layers using AHEAD composer, *(ii)* compiles the resulting composition, *(iii)* creates a Portlet WAR, and *(iv)* deploys it to a given location. This factory is specified using scripts [DTA05] (see chapter 7).

This Portlet factory has two major particularities: *(i)* some variability is required in the very same production process (see chapter 7), and *(ii)* being the process distributed makes it dependent from other processes.

### A.3.6 Application Engineering

Portlet production is the process to synthesize a customized Portlet. A *PinkCreek* product is the output of our factory in the same way that a car is the output of an assembly line. A given car is characterized by its platform (e.g., *Ford Focus* chassis) plus its features (*red* as the color, *gasoline* as the engine, etc). Likewise, our products are characterized by their features (e.g., *exoTomcat* as the *container*) together with the *core* layer.

This process is conceptualized as an expression [BSR04] where selected features (*checkin, seat* and *exoTomcat*) are applied to a *core*: *checkin( seat (exoTomcat (core)))*. This expression denotes a product which is synthesized by composing feature *exoTomcat* with *core*, then applying the changes by layer *seat*, and finally those by *checkin* (i.e., the product is the accumulation of all these features).

[7] http://en.wikipedia.org/wiki/Software_factory

| | SIZE | | FILES | | Java | | Jak | | XML | | XAK | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Feature | # | % | # | % | # | LOC | # | LOC | # | LOC | # | LOC |
| core | 8.900 | 83,65% | 112 | 21,41% | 8 | 688 | 28 | 1032 | 20 | 3437 | 8 | 1840 |
| assistance | 114 | 1,07% | 39 | 7,46% | | | 8 | 196 | 12 | 473 | 13 | 473 |
| bulletin | 111 | 1,04% | 36 | 6,88% | | | 10 | 230 | 12 | 473 | 13 | 473 |
| checkin | 106 | 1,00% | 37 | 7,07% | | | 10 | 230 | 12 | 473 | 13 | 473 |
| commercialbanner | 14 | 0,13% | 8 | 1,53% | | | | | 2 | 39 | 2 | 39 |
| commercialstext | 14 | 0,13% | 8 | 1,53% | | | | | 2 | 39 | 2 | 39 |
| dbmysql | 252 | 2,37% | 5 | 0,96% | 3 | 489 | | | 1 | 14 | 1 | 14 |
| exojboss | 388 | 3,65% | 8 | 1,53% | | | | | 5 | 94 | 2 | 87 |
| exotomcat | 11 | 0,10% | 5 | 0,96% | | | | | 2 | 87 | 2 | 87 |
| flightchildren | 15 | 0,14% | 6 | 1,15% | | | | | 3 | 130 | 3 | 130 |
| flightclass | 8 | 0,08% | 5 | 0,96% | | | | | 2 | 38 | 2 | 38 |
| includejavadoc | 4 | 0,04% | 1 | 0,19% | | | | | 1 | 31 | 1 | 31 |
| includesourcecode | 4 | 0,04% | 1 | 0,19% | | | | | 1 | 14 | 1 | 14 |
| loginrequired | 22 | 0,21% | 7 | 1,34% | | | 1 | 21 | 4 | 93 | 4 | 93 |
| meal | 104 | 0,98% | 35 | 6,69% | | | 8 | 196 | 12 | 473 | 13 | 473 |
| mytrip | 19 | 0,18% | 7 | 1,34% | | | 2 | 34 | 3 | 78 | 3 | 78 |
| onlydirectflights | 9 | 0,08% | 6 | 1,15% | | | | | 3 | 38 | 3 | 38 |
| paymentclick | 13 | 0,12% | 7 | 1,34% | | | 1 | 20 | 3 | 33 | 3 | 33 |
| paymenttransaction | 58 | 0,55% | 26 | 4,97% | 4 | 800 | 2 | 40 | 12 | 112 | 12 | 112 |
| quickpurchase | 20 | 0,19% | 11 | 2,10% | | | | | 6 | 51 | 6 | 51 |
| refundable | 9 | 0,08% | 6 | 1,15% | | | | | 3 | 38 | 3 | 38 |
| reservationcheck | 106 | 1,00% | 34 | 6,50% | | | 8 | 234 | 12 | 471 | 13 | 471 |
| reservationmodify | 107 | 1,01% | 35 | 6,69% | | | 8 | 234 | 12 | 471 | 13 | 471 |
| seat | 122 | 1,15% | 41 | 7,84% | | | 10 | 230 | 15 | 566 | 16 | 566 |
| testmode | 9 | 0,08% | 3 | 0,57% | | | | | 1 | 9 | 1 | 9 |
| weather | 100 | 0,94% | 34 | 6,50% | | | 8 | 234 | 12 | 473 | 13 | 473 |
| TOTAL | 10.639 | | 523 | | 15 | 1.977 | 104 | 2931 | 173 | 8.248 | 166 | 6644 |

**Table A.1**: *PinkCreek* Feature Implementation

Hence, the feature oriented programming approach supported by AHEAD is able to obtain a customized application by simply specifying the features to be exhibited by the product (i.e., the product's equation expression). AHEAD checks this expression for compliance against the feature model (i.e., obeying dependencies and cardinalities), and produces a set of artifacts that provide the required features.

This example illustrates one of the hallmarks of SPL, namely, keeping specifications of how features affect a product (i.e., supported as refinements in our approach) separated from the core assets (i.e., the platform). In this way, the programmer does not have to know how variants affect the product parts, how to produce customized parts and how to assemble customized parts into a custom product, as this knowledge is embedded within the SPL. The outcome is effective reuse as well as enhanced maintainability which stems from the separation of concerns between commonality and variability preached by SPL proponents. This separation is orthogonal to the distinction among content, navigation and presentation commonly found in web methodologies.

### A.3.7 Experience

**Features Data.** Table A.1 shows the details of each feature layer (note that empty cells correspond to zero values). It shows the total size in kilobytes (KB), and the number of files (#). Percentages (%) are also shown for each value. Besides, it details information for 4 types of artifacts (namely, *Java* source classes, *Jak* refinements, XML documents, and XAK refinements). A common pattern is usually followed where first some Jak extend some model functionality, then some XAK extend view and controller functionality. Variability implementation *cross-cuts* different concerns and artifacts. However, there are exceptions where only one artifact (within view, model or controller) is extended. The total number of Jak (104) is higher than the number of Java (15). The difference in terms of Lines Of Code (LOC) is smaller. The total number of XAK files (166) is similar to XML's ones (173). However, note that the LOC of XML (8.248) is considerably higher than the LOC of XAK (6.644). Note that the LOC rate between *Jak* and Java is not aligned with previous studies where refinements represent around 10% [ALS06]. This is presumably altered by the extensive presence of non-code artifacts.

**Production Data.** Empirical data shows that product synthesis time is around 30 seconds. Note that build time increases with the number of features the product has. Our factory needs more time to manufacture a product with more features (confirming our intuition). A *PinkCreek* product consists approximately of 200 files. In average, products range between 9-11 MB in size.

## A.4 Discussion

Portlets idiosyncrasies impose some requirements when building an SPL: *(i)* variability of Portlet variants should be handled, *(ii)* flexibility to both the consumer and the end-user is to be considered, *(iii)* availability matters because the built system is to be available online, *(iv)* usually several (and distributed) organizations are involved during the construction (and operation) process, and *(v)* reconfiguration due to frequent updates should be catered for. From a Portlet perspective, these requirements involve a number of research problems:

- **Distinct Stakeholders** are typically involved in a service-oriented architecture. Each requires a specific study of organizational variability requirements. From a Portlet perspective, it is not the same the variability require-

ments that the Portal *demands* or what the end-user *expects*. Hence, a study of variability in these scenarios is necessary [DTA03].

- **Ubiquity**. A Portlet can be deployed into distinct locations (a.k.a., Portlet producers). Doing so, it is necessary to cope with the variability of different producers (from distinct vendors). Likewise, the online availability is an issue together with the frequent updates.

- **Customization/Personalization/Privacy**. SPL offer an innovative way to resolve these issues in a web setting (also Portlets). Recently, Wang et al. introduced a case to deal with privacy [WKvdHW06].

- **Artifact Heterogeneity**. Implementation contains heterogeneous artifacts (not just traditional code classes). These artifacts can range from code classes (e.g., *Java*) to other artifacts such as HTML or JSP pages. Most of the techniques to manage variability are geared towards code artifacts [BSR04], and few experiences exist to bring these techniques to non-code artifacts [LS04, TBD06].

- **Standards and Platforms**. So far, there was a lack of proper platforms and approaches due to the novelty of Portlet [TPD04].

Previous issues have a counterpart where some research problems appear in the SPL universe:

- **Distributed Variability**. Being distinct stakeholders involved, variability study is different. *Stage configuration* is a technique to schedule variability decisions in time [CHE04]. Moreover, customers distribution should be catered for when they are distributed (in space) across different organizations.

- **Production Variability**. Different locations (ubiquity) demands usually some variation in the synthesis (i.e., synthesis process demands some variability that requires to be addressed). This is achieved by separating the product from the process itself (see chapter 7).

- **Distributed Production**. The production of the Portlet takes part of a larger process for creating a Portal. It is distributed. Hence, issues like consistency and time in production matter imposing a new challenge in the way the production is done. Further work is necessary to deal with them. Likewise,

product is delivered and deployed, and reconfiguration is often a requirement that needs to be resolved beforehand [LK06].

- **Implementation Variability**. Variability implementation techniques are not only restricted to SPL, but to other unplanned situations (e.g., ubiquity, customization/personalization, privacy [WKvdHW06]). This spreads the use of variability realization techniques towards unforeseen domains.

- **Heterogeneous Artifact Variability**. Specifically, the variability of XML documents could be resolved by refining them [TBD06]. The idea is to engineer for variability not only code classes, but documents. Doing so, it enables documents to cope with variability (see section 3.4.2).

## A.5 Future Work

**Distributed Feature Modeling.** In a Portlet setting, customers are distributed in different portals. Each portal typically requires different variability. The issue on how to delay (in time) variability decisions is resolved by stage-configurations [CHE04]. However, in a Portlet scenario decisions should be also distributed in space (i.e., different decisions should be done by distributed stakeholders that usually may belong to distinct organizations and portals). The challenge is how to assess the consistency of the feature selection when this process is distributed across those separated stakeholders. This chapter points this issue for future work.

**Distributed Factory.** The factory to produce a Portlet takes usually part of a larger process for creating a Portal. Thus, the factory to create a Portal uses *distributed* Portlet factories. In this setting, issues like consistency and production time need to be considered. This subject imposes new challenges in the way the production is done. First, the orchestration of production of several product factories where a clear invocation interface is necessary and some rules to manage the whole system to get a consistent end-product. As well, production time matters (i.e., production planning is necessary). Production automation could improve this issue. These issues are the subject of ongoing research [Tea07]. Likewise, product is delivered and deployed, and reconfiguration is often a requirement that needs to be considered [LK06].

## A.6 Contributions

The need for customization, the distributed service-oriented architecture and the existence of mature domains, vindicate a turning point in Portlet development from individual applications to SPL. The differences mainly stem from variability being the pivotal notion. This chapter illustrated an approach to an SPL of Portlets (together with a sample case). In particular, we outlined the major research problems, and provided some solutions to them in the context of Portlets.

This chapter introduced a new generation of issues either in Portlets and in SPL that should be catered for by future work. As well, we illustrate the approach using an academic case study. This limitation should be strengthened using some industrial case in the future.

SPL introduce the variability dimension to application development, orthogonal to the subject-based dimension that is commonly found in web methodologies (e.g., content, presentation, navigation). This in turn, rises the issue of cross cuts. In our experience, the notion of "refinement" is a convenient approach for handling cross cuts in a web setting. The reason is twofold. First, modularization. A refinement comprises a set of artifacts with the very same aim: the realization of a given feature variant. Feature evolution is then facilitated. Second, the principle of uniformity permits a uniform treatment of artifact composition. This is paramount to face the diversity of artifact types found in web applications.

An extended version of the work presented in this chapter has been accepted for publication:

1. *Turning Portlets into Services: Introducing the Organization Profile*. O. Diaz, S. Trujillo, and S. Perez. 16th International World Wide Web Conference (WWW2007). Banff , Canada. May 2007 [DTP07]. Acceptance Rate: 14% (110/750).

The knowledge we acquired during this work allowed us to prepare the following tutorial:

1. *An Introduction to Software Product Lines*. O. Diaz and S. Trujillo. JISBD 2006 (Spanish Software Engineering Conference). Sitges, Spain. October 2006 [DT06].

# Bibliography


[ABM00] C. Atkinson, J. Bayer, and D. Muthig. Component-based Product Line Development: the KobrA Approach. In *1st International Software Product Lines Conference (SPLC 2000), Denver, Colorado, USA, August 28-31*, pages 289–310, 2000. 12

[ADT05] F. I. Anfurrutia, O. Díaz, and S. Trujillo. A Product-Line Approach to Database Reporting. In *Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2005), Granada, Spain, September 14-16*, 2005. 126

[ADT06a] F. I. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *Submitted for Review*, November 2006. 37, 38, 56, 126

[ADT06b] F. I. Anfurrutia, O. Díaz, and S. Trujillo. Una Aproximación de Línea de Producto para la Generación de Informes de Bases de Datos (A Product-Line Approach to Database Reporting). *IEEE América Latina*, 4, April 2006. ISSN: 1548-0992. 126

[AFM05] M. Anastaspoulos, T. Forster, and D. Muthig. Optimizing Model-Driven Development by Deriving Code Generation Patterns from Product Line Architectures. In *NetObject Days, Erfurt, Germany, September 19-22*, 2005. 22, 23, 84, 128

[AG01] M. Anastasopoulos and C. Gacek. Implementing Product Line Variabilities. In *Symposium on Software Reusability (SSR 2001), Toronto, Canada, May*, pages 109–117, 2001. 13, 28, 31

[AGESR06] O. Avila-Garcia, A. Estévez, E.V. Sanchez, and J.L. Roda. Integrando Modelos de Procesos y Activos Reutilizables en una Herramienta MDA. In *Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2006), Sitges, Spain, October 2-6*, 2006. 22

[AGM+06] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *5th International Conference on Generative Programming and Component Engineering (GPCE 2006), Portland, Oregon, USA, October 24-27*, 2006. 46

[ALS06] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28*, 2006. 90, 143

[Ape07] S. Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, School of Computer Sciences, University of Magdeburg, Germany, March 2007. 32, 70

[ATSD06] G. Aldekoa, S. Trujillo, G. Sagardui, and O. Díaz. Experience Measuring Maintainability in Software Product Lines. In *Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2006), Sitges, Spain, October 2-6*, 2006. 126

[Bas97] P. Bassett. *Framing Software Reuse - Lessons from Real World*. Yourdon Press, Prentice Hall, 1997. 29

[Bata] D. Batory. AHEAD Tool Suite. http://www.cs.utexas.edu/users/schwartz/ATS.html. 13, 31, 34, 35, 40, 111

[Batb] D. Batory. Product-Line Architectures, Invited presentation, Smalltalk und Java in Industrie and Ausbildung (STJA), Erfurt, Germany, October 1998. 13

[Bat05] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *9th International Software Product Lines Conference (SPLC 2005), Rennes, France, September 26-29*, 2005. 13, 32, 45, 84

[Bat06] D. Batory. Multi-Level Models in Model Driven Development, Product-Lines, and Metaprogramming. *IBM Systems Journal*, 45(3), 2006. 22, 75, 84

[Bat07a] D. Batory. From Implementation to Theory in Program Synthesis. In *Keynote at Principles of Programming Languages (POPL), Nice, France, January 17-19*, 2007. 85, 102, 105, 128

[Bat07b] D. Batory. Program Refactoring, Program Synthesis, and Model Driven Development. In *Keynote at European Joint Conferences on Theory and Practice of Software (ETAPS) Compiler Construction Conference, Braga, Portugal, March 24-April 1*, 2007. 88, 90, 91, 105, 128

[BB04] J. Blair and D. Batory. A Comparison of Generative Approaches: XVCL and GenVoca. Technical report, The University of Texas at Austin, Department of Computer Sciences, December 2004. 29

[BBdF+06] J. Bézivin, S. Bouzitouna, M. Didonet del Fabro, M.P. Gervais, F. Jouault, D.S. Kolovos, I. Kurtev, and R.F. Paige. A Canonical Scheme for Model Composition. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13*, pages 346–360, 2006. 17

[BBG05] S. Beydeda, M. Book, and V. Gruhn, editors. *Model-Driven Software Development*. Springer, 2005. 15

[BBI+04] G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, and B. Selic. The IBM MDA Manifesto. *The MDA Journal*, May 2004. 74

[BC90] G. Bracha and W. R. Cook. Mixin-based Inheritance. In *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP), Ottawa, Canada, October 21-25*, pages 303–311, 1990. 31, 34

[BCK03] R. Buhrdorf, D. Churchett, and C.W. Krueger. Salions Experience with a Reactive Software Product Line Approach. In *5th International Workshop on Software Product-Family Engineering (PFE 2003), Siena, Italy, November 4-6*, 2003. 54

[BCM+04] G. Bockle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid. Calculating ROI for Software Product Lines. *IEEE Software*, 21(3):23–31, May/June 2004. 10

[BCR05] A. Boronat, J. A. Carsí, and I. Ramos. Automatic Support for Traceability in a Generic Model Management Framework. In *1st European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2005), Nuremberg, Germany, November 7-10*, pages 316–330, 2005. 16, 17

[BCR06] A. Boronat, J.A. Carsí, and I. Ramos. Algebraic Specification of a Model Transformation Engine. In *9th International Conference on Fundamental Approaches to Software Engineering (FASE 2006), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2006), Vienna, Austria, March 27-28*, pages 262–277, 2006. 16

[BCRW00] D. Batory, G. Chen, E. Robertson, and T. Wang. Design Wizards and Visual Programming Environments for GenVoca Generators. *IEEE Transactions on Software Engineering (TSE)*, 26(5):441–452, May 2000. 32, 84

[BCS00] D. Batory, R. Cardone, and Y. Smaragdakis. Object-Oriented Frameworks and Product-Lines. In *1st International Software Product Lines Conference (SPLC 2000), Denver, Colorado, USA, August 28-31*, 2000. 51

[BDJ+03] J. Bézivin, G. Dupe, F. Jouault, G. Pitette, and J. E. Rougui. First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the Context of MDA, Anaheim, California, USA, October 27*, 2003. 16, 60

[BdOB06] F.P. Basso, T. Cavalcante de Oliveira, and L. B. Becker. Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development. In *9th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2006), Gyeongju, Korea*, pages 374–381, April 2006. 84, 106

[BdRPdA05] L. Balzerani, D. di Ruscio, A. Pierantonio, and G. de Angelis. A Product Line Architecture for Web Applications. In *ACM Symposium on Applied Computing (SAC 2005), Santa Fe, New Mexico, USA, March 13-17*, pages 1689–1693, 2005. 24

[Bea99] J. Bayer and et al. PuLSE: A Methodology to Develop Software Product Lines. In *Symposium on Software Reusability (SSR 1999), Los Angeles, California, USA, May*, pages 122–131, 1999. 12

[Bea02] D. Batory and et al. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, April 2002. 77, 84

[Ben07] D. Benavides. *On the Automated Analysis of Software Product Lines using Feature Models*. PhD thesis, University of Seville, Spain, March 2007. 32, 136

[Beu03] D. Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, School of Computer Sciences, University of Magdeburg, Germany, 2003. 10

[Béz01] J. Bézivin. From Object Composition to Model Transformation with the MDA. In *Technology of Object-Oriented Languages and Systems (TOOLS USA), Santa Barbara, Californmia, USA, August*, 2001. 59

[Béz04] J. Bézivin. In Search of a Basic Principle for Model Driven Engineering. *Novatica*, (1), June 2004. 15, 24

[Béz05] J. Bézivin. Model Driven Engineering: Principles, Scope, Deployment, and Applicability. In *International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005), Braga, Portugal, July 4-8*, 2005. 59, 74

[BF93] S. Bandinelli and A. Fuggetta. Computational Reflection in Software Process Modeling: The SLANG Approach. In *15th International Conference on Software Engineering (ICSE 1993), Baltimore, Maryland, USA, May 17-21*, 1993. 110

[BFJ+03] J. Bézivin, N. Farcet, J.M. Jezequel, B. Langlois, and D. Pollet. Reflective Model-Driven Engineering. In *6th International Conference on The Unified Modeling Language, Modeling Languages and Applications (UML 2003), San Francisco, California, USA, October 20-24*, 2003. 15

[BGP00] L. Baresi, F. Garzotto, and P. Paolini. From Web Sites to Web Applications: New Issues for Conceptual Modeling. In *Conceptual Modeling for E-Business and the Web, ER 2000 Workshops on Conceptual Modeling Approaches for E-Business and The World Wide Web, Salt Lake City, Utah, USA, October 9-12*, pages 89–100, 2000. 20, 24, 84

[BJT05] J. Bézivin, F. Jouault, and D. Touzet. An Introduction to the ATLAS Model Management Architecture. Technical report, University of Nantes, 2005. Research Report LINA (05-01). 17

[BLS03] D. Batory, J. Liu, and J.N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *11th ACM Symposium on Foundations of Software Engineering (SIGSOFT) held jointly with 9th European Software Engineering Conference (ESEC/FSE), Helsinki, Finland, September 1-5*, 2003. 52, 102, 108

[BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, October 1992. 111

[Boe99] B.W. Boehm. Managing Software Productivity and Reuse. *IEEE Computer*, 32(9):111–113, 1999. 123

[Bos00] J. Bosch. *Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000. 8, 13, 30

[BPSP04] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, 2004. 13

[BRCT05] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. In *17th International Conference on Advanced Information Systems Engineering (CAiSE 2005), Porto, Portugal, June 13-17*, volume 3520, pages 491–503, 2005. 32

[Bro87] F.P. Brookes. No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987. 52

[BSR04] D. Batory, J.Neal Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, June 2004. 1, 8, 13, 29, 30, 31, 32, 33, 34, 35, 37, 39, 40, 67, 70, 74, 75, 77, 84, 85, 89, 92, 97, 111, 116, 133, 136, 138, 139, 141, 144

[BSTRC07] D. Benavides, S. Segura, P. Trinidad, and A. Ruíz-Cortés. FAMA: a Framework for the Automated Analysis of Feature Models. In *1st International Workshop on Variability Modelling of Software-intensive Systems (VAMOS). Limerick, Ireland, January 16-18*, 2007. 32

[BT06] D. Batory and S. Thaker. Towards Safe Composition of Product Lines. Technical Report TR-06-33, The University of Texas at Austin, Department of Computer Sciences, 2006. http://www.cs.utexas.edu/ftp/pub/techreports/index/html/Abstracts.2006.html. 53

[BTT05] D. Benavides, S. Trujillo, and P. Trinidad. On the Modularization of Feature Models. In *1st European Workshop on Model Transformation (SPLC-EWMT 2005), Rennes, France, September 25*, 2005. 39, 125

[CA05a] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *4th International Conference on Generative Programming and Component Engineering (GPCE 2005), Tallinn, Estonia, September 29 - October 1*, 2005. 13, 23, 45, 84, 136

[CA05b] K. Czarnecki and M. Antkiewicz. Model-Driven Software Product-Lines. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), San Diego, CA, USA, October 16-20*, 2005. 22, 84

[CCC+99] M. Charikar, C. Chekuri, T.Y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation Algorithms for Directed Steiner Problems. *J. Algorithms*, 33(1):73–91, 1999. 102

[CD03] R. Capilla and J. C. Dueñas. Light-weight Product-lines for Evolution and Maintenance of Web Sites. In *7th European Conference on Software Maintenance and Reengineering (CSMR 2003), Benevento, Italy, March 26-28*, pages 53–62, 2003. 23, 54

[CDM02] G. Chastek, P. Donohoe, and J.D. McGregor. Product Line Production Planning for the Home Integration System Example. Technical report, CMU/SEI, September 2002. CMU/SEI-2002-TN-029. 110

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 2000. 8, 11, 12, 13, 60, 92

[CFM02] S. Ceri, P. Fraternali, and M. Matera. Conceptual Modeling of Data-Intensive Web Applications. *IEEE Internet Computing*, 6(4):20–30, July/August 2002. 20, 24, 84

[CH03] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *OOPSLA Workshop on Generative Programming on the Context of MDA, Anaheim, California, USA, October 27*, 2003. 61

[CHE04] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *3rd International Software Product Lines Conference (SPLC 2004), Boston, Massachusetts, USA, August 30-September 2*, 2004. 144, 145

[CK02] P. Clements and C. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, July/August 2002. 11, 54

[CM02] G. Chastek and J.D. McGregor. Guidelines for Developing a Product Line Production Plan. Technical report, CMU/SEI, June 2002. CMU/SEI-2002-TR-06. 9, 107, 110, 112, 133

[CMC05] P. Clements, J.D. McGregor, and S.G. Cohen. The Structured Intuitive Model for Product Line Economics (SIMPLE). Technical report, CMU/SEI, 2005. CMU/SEI-2005-TR-003. 10

[CMT06] J. Sánchez Cuadrado, J. García Molina, and M. Menárguez Tortosa. RubyTL: A Practical, Extensible Transformation Language.

In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13*, pages 158–172, 2006. 16, 17, 60, 127

[CN01] P. Clements and L.M. Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2001. 1, 8, 9, 10, 11, 12, 13, 92, 110, 132, 133, 134, 137

[Coh01] S. Cohen. Predicting when Product Line Investment Pays. In *2nd Workshop on Software Product Lines: Economics, Architectures, and Implications held in conjunction with the 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Ontario, Canada, May 12-19*, 2001. 9, 132

[CPRS04] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. XML-Based Feature Modelling. In *8th International Conference on Software Reuse (ICSR 2004), Madrid, Spain, July 5-9*, pages 101–114, 2004. 13

[Cre03] J. Creasman. Enhance Ant with XSL Transformations, 2003. http://www-128.ibm.com/developerworks/xml/library/x-antxsl/. 110

[CY03] D. Coward and Y. Yoshida. JSR 154, Java Servlet 2.4 Specification, 2003. http://www.jcp.org/en/jsr/detail?id=154. 22, 138

[dFBRG06] G. de Fombelle, X. Blanc, L. Rioux, and M.P. Gervais. Finding a Path to Model Consistency. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13*, pages 101–112, 2006. 17

[DII05] O. Díaz, J. Iturrioz, and A. Irastorza. Improving Portlet Interoperability through Deep Annotation. In *14th International Conference on World Wide Web (WWW 2005), Chiba, Japan, May 10-14*, pages 372–381, 2005. 22, 129

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976. 32, 111, 139

[Don00] P. Donohoe, editor. *Proceedings of the 1st International Software Product Lines Conference (SPLC 2000), Denver, Colorado, USA, August 28-31*. Kluwer, 2000. ISBN 0-7923-7940-3. 8

[dOTM01] M. C. Ferreira de Oliveira, M. A. Santos Turine, and P. C. Masiero. A Statechart-Based Model for Hypermedia Applications. *ACM Transactions on Information Systems (TOIS)*, 19(1):28–52, January 2001. 64

[DR04] O. Díaz and J.J. Rodríguez. Portlets as Web Components: an Introduction. *Journal of Universal Computer Sciences (JUCS)*, 10(4):454–472, 2004. 18, 20, 21, 62

[DR05] O. Díaz and J.J. Rodriguez. Portlet Syndication: Raising Variability Concerns. *ACM Transactions on Internet Technology (TOIT)*, 5(4), November 2005. 24

[DSvGB03] S. Deelstra, M. Sinnema, J. van Gurp, and J. Bosch. Model Driven Architecture as Approach to Manage Variability in Software Product Families. In *Workshop on Model Driven Architecture: Foundations and Applications (MDAFA), Enschede, The Netherlands, June 26-27*, 2003. 22, 84

[DT03] O. Díaz and S. Trujillo. Moving Co-Branding to the Web: Service-Level Agreement Implications. In *Processes and Foundations for Virtual Organizations, IFIP TC5/WG5.5 Fourth Working Conference on Virtual Enterprises (PRO-VE'03), Lugano, Switzerland, October 29-31*, volume 262 of *IFIP Conference Proceedings*. Kluwer Academic Publishers, 2003. 126

[DT06] O. Díaz and S. Trujillo. An Introduction to Software Product Lines (Tutorial). In *Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2006), Sitges, Spain, October 2-6*, 2006. 126, 146

[DTA03] O. Díaz, S. Trujillo, and I. Azpeitia. User-Facing Web Service Development: a Case for a Product-line Approach. In *4th Workshop on Technologies for E-Services (TES) held jointly with the 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany, September 9-12*, 2003. 126, 144

[DTA05] O. Díaz, S. Trujillo, and F. I. Anfurrutia. Supporting Production Strategies as Refinements of the Production Process. In *9th International Software Product Lines Conference (SPLC 2005), Rennes, France, September 26-29*, pages 210–221, 2005. 13, 24, 37, 56, 108, 109, 112, 116, 118, 122, 125, 141

[DTP07] O. Díaz, S. Trujillo, and S. Perez. Turning Portlets into Services: Introducing the Organization Profile. In *16th International World Wide Web Conference (WWW2007), Banff, Canada, May 8-12*, November 2007. 125, 146

[DvdHT05] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):199–245, 2005. 13

[Eas98] S. Easterbrook. Category Theory for Beginners (An Introduction to Category Theory for Software Engineers). In *Tutorial at 13th IEEE Conference on Automated Software Engineering (ASE 1998), Honolulu, Hawaii, USA*, 1998. http://www.cs.toronto.edu/ sme/presentations/cat101.pdf. 105

[EBB05] M. Eriksson, J. Börstler, and K. Borg. The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realization. In *9th International Software Product Lines Conference (SPLC 2005), Rennes, France, September 26-29*, 2005. 12

[Ecl] Eclipse. Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf/. 16

[FB07] G. Freeman and D. Batory. MDD with XML Domain Specific Languages: SVG Case Study. In *Draft under Preparation*, 2007. 129

[FHLS97] G. Froehlich, H.J. Hoover, L. Liu, and P.G. Sorenson. Hooking into Object-Oriented Application Frameworks. In *19th International Conference on Software Engineering (ICSE 1997), Boston, Massachusetts, USA, May 17-23*, 1997. 51

[Fia05] J.L. Fiadeiro. *Categories for Software Engineering*. Springer, 2005. 90, 105

[Fou] Apache Software Foundation. Apache Ant. http://www.ant.apache.org/. 75, 111

[Fra03] D.S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003. 15

[GBLC05] B. Gonzalez-Baixauli, M.A. Laguna, and Y. Crespo. Product Lines, Features, and MDD. In *1st European Workshop on Model Transformation (SPLC-EWMT'05), Rennes, France, September 25*, 2005. 22, 84

[Gea04] J. Gray and et al. Model Driven Program Transformation of a Large Avionics Framework. In *3rd International Conference on Generative Programming and Component Engineering (GPCE 2004), Vancouver, Canada, October 24-28*, 2004. 23, 84

[GFd98] M.L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *5th International Conference on Software Reuse (ICSR 1998), Victoria, BC, Canada*, Vancouver, BC, Canada, 1998. 12

[GMY06] D. Ganesan, D. Muthig, and K. Yoshimura. Predicting Return-on-Investment for Product Line Generations. In *10th International Software Product Lines Conference (SPLC 2006), Baltimore, Maryland, USA, August 21-24*, 2006. 10, 13

[Gog91] J.A. Goguen. A Categorical Manifesto. *Mathematical Structures in Computer Sciences*, 1(1):49–67, 1991. 105

[Gom04] H. Gomaa. *Designing Software Product Lines with UML*. Addison-Wesley, 2004. 11, 12, 30

[Gro] Onekin Research Group. XAK Tool for XML Refinement in AHEAD. http://www.onekin.org/xak. 37

[GS04] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. ACM Press New York, 2004. 16, 141

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3), 1987. 64

[HK01] J. Hannemann and G. Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns held jointly with the 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Ontario, Canada, May 12-19*, 2001. 55

[HR03] D. Herst and E. Roman. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) - Approach: A Productivity Analysis. Technical report, TMC Research Report, 2003. 15

[HWS00] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *7th Working Conference on Reverse Engineering (WCRE 2000), Brisbane, Australia*, November 2000. 93, 94

[IBM] IBM. Rational Software Design. http://www.ibm.com/software/rational. 16

[JBR98] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1998. 16

[JBZZ03] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based Variant Configuration Language. In *25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, USA, 3-10 May*, pages 810–811, 2003. 13, 23, 29

[JCP03] JCP. JSR 168 Portlet Specification Version 1.0, September 2003. at http://www.jcp.org/en/jsr/detail?id=168. 19, 21, 22, 62, 137

[JCP06] JCP. JSR 286 Portlet Specification Version 2.0 Early Draft 1, August 2006. at http://www.jcp.org/en/jsr/detail?id=286. 22

[Kea90] K. C. Kang and et al. Feature Oriented Domain Analysis Feasability Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990. 8, 12, 30, 135

[Ken02] S. Kent. Model Driven Engineering. In *3rd International Conference on Integrated Formal Methods (IFM 2002), Turku, Finland, May 15-18*, 2002. 14, 15, 128

[KKL+98] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998. 12

[KLD02] K. C. Kang, J. Lee, and P. Donohoe. Feature-Oriented Project Line Engineering. *IEEE Software*, 19(4):58–65, 2002. 12

[Kle06] A. Kleppe. MCC: A Model Transformation Environment. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13*, pages 173–187, 2006. 17

[KMSW00] P. Knauber, D. Muthig, K. Schmid, and T. Widen. Applying Product Line Concepts in Small and Medium-Sized Companies. *IEEE Software*, 17(5), 2000. 10

[Kon] Mikko Kontio. Architectural Manifesto: The MDA Adoption Manual. http://www-128.ibm.com/developerworks/wireless/library/wi-arch17.html. 15

[KP88] G. Krasner and S. Pope. A Cookbook for Using the MVC User Interface Paradigm in Smalltalk. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988. 137

[KR03] V. Kulkarni and S. Reddy. Separation of Concerns in Model-Driven Development. *IEEE Software*, 20(5):64–69, 2003. 84

[Kru02] C. W. Krueger. Variation Management for Software Production Lines. In *2nd International Software Product Lines (SPLC 2002), San Diego, California, USA, August 19-22*, pages 37–48, 2002. 13

[Kur05] I. Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, The Netherlands, 2005. 15, 59, 60

[KvdB05] I. Kurtev and K. van den Berg. Building Adaptable and Reusable XML Applications with Model Transformations. In *14th International Conference on World Wide Web (WWW 2005), Chiba, Japan, May 10-14*, 2005. 24, 84

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003. 15, 66, 69, 74

[Lau06] B. Laurel. Designed Animism: Poetics of a New World. In *Keynote at 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), Portland, Oregon, USA, October 22-26*, 2006. 107

[LB04] J. Liu and D. Batory. Automatic Remodularization and Optimized Synthesis of Product-Families. In *3rd International Conference on Generative Programming and Component Engineering (GPCE 2004), Vancouver, Canada, October 24-28*, 2004. 32

[LBL06] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28*, 2006. 13, 40, 53, 54, 90

[LBN05] J. Liu, D. Batory, and S. Nedunuri. Modeling Interactions in Feature Oriented Designs. In *International Conference on Feature Interactions (ICFI 2005), Leicester, United Kingdom, June 28-30*, June 2005. 40, 90

[Lea] G. Leavens and et al. Roadmap for Enhanced Languages and Methods to Aid Verification. Technical report. ftp://ftp.cs.iastate.edu/pub/techreports/TR06-21/TR.pdf. 85

[Len06] C. Lengauer. Discussion on AHEAD's Mathematical Structure. Personal communication, 2006. 30

[LH06] R.E. Lopez-Herrejon. *Understanding Feature Modularity*. PhD thesis, The University of Texas at Austin,

Department of Computer Sciences, USA, September 2006. http://www.cs.utexas.edu/ftp/pub/techreports/tr06-45.pdf. 30, 54

[LH07] R.E. Lopez-Herrejon. Language and UML Support for Features: Two Research Challenges. In *1st International Workshop on Variability Modelling of Software-intensive Systems (VAMOS). Limerick, Ireland, January 16-18*, 2007. 128

[LHB06] R.E. Lopez-Herrejon and D. Batory. From Crosscutting Concerns to Product Lines: A Functional Composition Approach. Technical Report TR-06-24, The University of Texas at Austin, Department of Computer Sciences, May 2006. 54

[LK06] J. Lee and K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *10th International Software Product Lines Conference (SPLC 2006), Baltimore, Maryland, USA, August 21-24*, 2006. 13, 145

[LS04] S. C. Lee and A. I. Shirani. A Component based Methodology for Web Application Development. *Journal of Systems and Software*, 71:177–187, 2004. 144

[LXK98] J. Lee, N.L. Xue, and T.L. Kuei. A Note on State Modeling through Inheritance. *SIGSOFT Soft. Eng. Notes*, 23(1):104 – 110, January 1998. 77

[Mag29] R. Magritte. Painting: The Treachery of Images (from the original: La Trahison des Images), 1929. 59

[Mat04] M. Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In *26th International Conference on Software Engineering (ICSE 2004), Edinburgh, United Kingdom, 23-28 May*, pages 127–136, 2004. 12

[MB01] D.C. Marinescu and L. Bölöni. Biological Metaphors in the Design of Complex Software Systems. *Future Generation Computer Sciences*, 17(4):345–360, January 2001. 107

[McG04] J.D. McGregor. Product Production. *Journal Object Technology*, 3(10):89–98, November/December 2004. 107, 110

[McI68] M.D. McIlroy. Mass Produced Software Components. In *NATO Science Committee, Garmisch, Germany, October 7-11*, 1968. 8

[MCL06] B. Mesing, C. Constantinides, and W. Lohmann. Limes: An Aspect-Oriented Constraint Checking Language. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13*, pages 299–315, 2006. 17

[Mea01] G. C. Murphy and et al. Separating Features in Source Code: An Exploratory Study. In *23rd International Conference on Software Engineering (ICSE 2001), Toronto, Ontario, Canada, May 12-19*, 2001. 54

[Mel07] S. Melia. *WebSA: Un Metodo de Desarrollo Dirigido por Modelos de Arquitectura para Aplicaciones Web (in Spanish)*. PhD thesis, University of Alicante, Spain, March 2007. 24

[MG06a] S. Melia and J. Gomez. The WebSA Approach: Applying Model Driven Engineering to Web Applications. *Journal of Web Engineering*, 5:2, 2006. 24

[MG06b] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006. 28, 32, 58, 61, 69, 70, 74, 84, 90

[MH02] A. Mehta and G.T. Heineman. Evolving Legacy System Features into Fine-grained Components. In *22nd International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA, May 19-25*, 2002. 55

[ML97] M. Meyer and A. Lehnerd. *The Power of Product Platforms*. Free Press, 1997. 137

[MND02] M. Matinlassi, E. Niemelä, and L. Dobrica. Quality-Driven Architecture Design and Quality Analysis. Technical Report 456, VTT Publications (Technical Research Center of Finland), 2002. 12

[Mod] Modelbased.net. MDA Tools. http://www.modelbased.net/mda_tools.html. 17

[MS03] A.T. McNeile and N. Simons. State Machines as Mixins. *Journal of Object Technology*, 2(6):85–101, November/December 2003. 77

[Mut02] D. Muthig. *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. PhD thesis, Fraunhofer IESE, Germany, 2002. 10, 123

[NA02] N.Koch and A.Kraus. The Expressive Power of UML-based Web Engineering. In *2nd International Workshop on Web-Oriented Software Technology (IWWOST02), Malaga, Spain, June 10*, 2002. 20, 24, 84

[NK06] A. Narayanan and G. Karsai. Towards Verifying Model Transformations. In *GT-VMT Workshop held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2006), Vienna, Austria, March 25-April 2*, 2006. 82

[Nor02] L. M. Northrop. SEI's Software Product Line Tenets. *IEEE Software*, 19(4):32–40, 2002. 8, 133

[OAS03] OASIS. Web Service for Remote Portlets (WSRP) Version 1.0, 2003. http://www.oasis-open.org/commitees/wsrp/. 18, 19, 21, 62

[OAS06a] OASIS. Reference Model for Service Oriented Architecture 1.0. OASIS Standard, October 2006. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html. 21

[OAS06b] OASIS. Web Service for Remote Portlets Specification Version 2.0, June 2006. http://www.oasis-open.org/commitees/tc_home.php?wg_abbrev=wsrp. 22

[OMG] OMG. MDA Success Stories. http://www.omg.org/mda/products_success.htm. 15

[OMG03] OMG. MDA Guide version 1.0.1. OMG document 2003-06-01, 2003. 14, 16, 24, 58, 60

[OMG05a] OMG. MOF Query/Views/Transformations (QVT) Draft Adopted specification: OMG document ptc/05-11-01, 2005. 16, 24, 60, 61

[OMG05b] OMG. Unified Modeling Language (UML), version 2.0, 2005. http://www.uml.org/#UML2.0. 16, 61, 64

[OMG05c] OMG. XML Metadata Interchange (XMI) Mapping Specification, version 2.1, 2005. http://www.omg.org/technology/documents/formal/xmi.htm. 16, 61, 64

[OMG06] OMG. MetaObject Facility (MOF) Core Specification version 2.0. OMG document 2006-01-01, 2006. 16, 61

[Ove00] S. P. Overmyer. What's Different about Requirements Engineering for Web Sites? *Requirements Engineering*, 5(1):62–65, 2000. 132

[Par76] D.L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering (TSE)*, 2(1):1–9, March 1976. 8

[Par79] D.L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)*, 5(2):128–138, 1979. 70

[PBvdL06] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer, 2006. 11, 12, 137

[PGIP01] O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano. The OO-method Approach for Information Systems Modeling: from Object-Oriented Conceptual Modeling to Automated Programming. *Information Systems*, 26(7):507–534, 2001. 20, 24, 84

[Pie91] B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991. 81, 82, 85, 90, 105, 106

[PM00] F. Paterno and C. Mancini. Model-Based Design of Interactive Applications. *ACM Intelligence*, 11(4):26–38, December 2000. 20, 84

[Pre97] C. Prehofer. Feature Oriented Programming: A Fresh Look at Objects. In *11th European Conference on Object-Oriented Programming (ECOOP 1997), Jyväskylä, Finland, June 9-13*, pages 419–443, 1997. 13

[RGJ04] J. Rumbaugh, G.Booch, and I. Jacobsen. *The UML Reference Manual*. Addison-Wesley, 2004. 16

[RGvD06] T. Reus, H. Geers, and A. van Deursen. Harvesting Software Systems for MDA-Based Reengineering. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13*, pages 213–225, 2006. 17, 90, 129

[RJ05] D. C. Rajapackse and S. Jarzabek. An Investigation of Cloning in Web Applications. In *5th International Conference on Web Engineering (ICWE 2005), Sydney, Australia, July 27-29*, pages 252–262, 2005. 23, 37

[RM02] P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *22nd International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA, May 19-25*, 2002. 55

[RW06] A. Rensink and J. Warmer, editors. *Model Driven Architecture - Foundations and Applications, 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13*, volume 4066 of *Lecture Notes in Computer Science*. Springer, 2006. 17

[SAC+79] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access Path Selection in a Relational Database System. In *ACM SIGMOD, Boston, Massachusetts, USA, June*, pages 22–34, 1979. 32

[SB02] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, Abril 2002. 34

[SC04] N. Serrano and I. Ciordia. Ant: Automating the Process of Building Applications. *IEEE Software*, 21(6):89–91, November/December 2004. 68, 113

[SCC06] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. *SIGPLAN Not.*, 41(10):451–464, 2006. 107

[Sch06] D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006. 15

[SEI] SEI. Product Line Practice Initiative (PLPI). http://www.sei.cmu.edu/productlines/. 12

[Sim96] C. Simonyi. Intentional Programming: Innovation in the Legacy Age, 1996. Presented at IFIP Working group 2.1. Available from URL http://www.research.microsoft.com/research/ip/. 107

[SK97] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–111, April 1997. 16

[SK03] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003. 58

[Sma99] Y. Smaragdakis. *Implementing Large-Scale Object-Oriented Components*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, USA, September 1999. ftp://ftp.cs.utexas.edu/pub/predator/yannis-thesis.pdf. 30, 31

[SNW05] D. Schmidt, A. Nechypurenko, and E. Wuchner. MDD for Software Product-Lines: Fact or Fiction. In *Workshop at 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005), Montego Bay, Jamaica, October 2-7, 2005*, 2005. 22, 23, 84

[Sou] Sourceforge.net. GXL (Graph eXchange Language). http://gxl.sourceforge.net/. 93, 94, 97

[SSJ02] I. Singh, B. Stearns, and M. Johnson. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley, 2002. 19, 137

[str] Apache Struts. http://struts.apache.org/. 37

[Sun] Sun. JavaServer Faces Technology (JSF). http://java.sun.com/javaee/javaserverfaces/. 22

[Sys] Pure Systems. Pure::variants tool. http://www.pure-systems.com/. 13, 29

[Tak06] S. Taker. Design and Analysis of Multidimensional Program Structures. Master's thesis, The University of Texas at Austin, Department of Computer Sciences, USA, 2006. 103, 108

[TBD06] S. Trujillo, D. Batory, and O. Díaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *5th International Conference on Generative Programming and Component Engineering (GPCE 2006), Portland, Oregon, USA, October 24-27*, 2006. 13, 37, 39, 56, 90, 125, 129, 144, 145

[TBD07] S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA, May 20-26*, 2007. 61, 70, 73, 86, 105, 106, 125

[Tea07] S. Trujillo and et al. Exploring a Distributed Production Setting for Product Lines. In *Draft under Preparation*, 2007. 13, 105, 129, 145

[TPD04] S. Trujillo, I. Paz, and O. Díaz. Enhancing Decoupling in Portlet Implementation. In *4th International Conference on Web Engineering (ICWE 2004 Posters), Munich, Germany, July 26-30*, 2004. 70, 125, 144

[Tru07] S. Trujillo. *Feature Oriented Model Driven Product Lines*. PhD thesis, School of Computer Sciences, University of the Basque Country, Spain, March 2007. 106

[TS00] W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000. 92

[vdL02] F. van der Linden, editor. *4th International Workshop on Software Product-Family Engineering (PFE 2001), Bilbao, Spain, October 3-5, 2001, Revised Papers*, volume 2290 of *Lecture Notes in Computer Science*. Springer, 2002. 8

[vDMM03] A. van Deursen, M. Marin, and L. Moonen. Aspect Mining and Refactoring. In *1st International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE). University of Waterloo*, 2003. 54

[vOvdLKM00] R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000. 13

[W3C01] W3C. Web Services Description Language (WSDL) 1.1, 2001. http://www.w3.org/TR/wsdl. 21

[W3C03] W3C. Simple Object Access Protocol (SOAP) 1.2, 2003. http://www.w3.org/TR/soap12/. 19, 21, 62

[W3C05] W3C. XSL Transformations (XSLT) Version 2.0, 2005. http://www.w3.org/TR/xslt20/. 60, 61

[W3C06a] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition), August 2006. http://www.w3.org/TR/2006/REC-xml-20060816/. 61

[W3C06b] W3C. State Chart XML (SCXML): State Machine Notation for Control Abstraction, W3C Working Draft 24 January 2006, 2006. http://www.w3.org/TR/scxml/. 64

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, April 1971. 70

[WKvdHW06] Y. Wang, A. Kobsa, A. van der Hoek, and J. White. PLA-based Runtime Dynamism in Support of Privacy-Enhanced Web Personalization. In *10th International Software Product Lines Conference (SPLC 2006), Baltimore, Maryland, USA, August 21-24*, 2006. 13, 144, 145

[WL99] D.M. Weiss and C.T.R. Lai. *Software Product-Line Engineering. A Family-Based Software Development Process*. Addison-Wesley, 1999. 12

[WM98] W. Weber and P. Metz. Reuse of Models and Diagrams of the UML and Implementation Concepts Regarding Dynamic Modeling. In *The Unified Modeling Language: Technical Aspects and Applications*, pages 190–203. Physica-Verlag, 1998. 77

[WvdS06] D. Wagelaar and R. van der Straeten. A Comparison of Configuration Techniques for Model Transformations. In *2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, July 10-13*, pages 331–345, 2006. 17

[ZJ04] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, BC, Canada, October 24-28*, 2004. 40, 54

# Epilogue

The road to doctorate is not an easy one. Being a doctor does not imply to be an expert, but to sip some initial lessons as training for research in the future. Actually, this document is not just intended to turn into the end, but is conceived to be the amazing beginning of the forthcoming future.

To apprehend knowledge was the intention when joining the doctoral program. However, this experience demonstrated that there are many other personal skills that a researcher should acquire such as communication, discussion, and abstraction of ideas. Being this important, the work on this thesis posed some intellectual challenges that allowed us to have a lot of fun! making this experience unforgettable.

# Acknowledgments

Seldom is any worthy undertaking tackled alone, and this is no exception. Throughout my work many people have influenced my thoughts, provided assistance, guided my work, afforded opportunities, and comforted me. Listing them here immortalizes their contribution towards my thesis.

First and foremost, my supervisor *Oscar* must be thanked for investing a great deal of time in this work from the early stages to the very end. I learned the highest values of research from him. He also pushed me to attend relevant meetings across the world, and to visit vanguard research institutions. I would like to express my deep gratitude to Txema Perez from Mondragon University who put me in contact with Oscar Diaz.

*Onekin* is the research group Oscar leads at the University of the Basque Country. Our group fosters collaborative work among its members as important part of our daily activity. Hence, parts of this work would not be possible without the effort of others. Specifically, Iñaki Paz struggled in interminably discussions during the early stages of Portlet modeling, Sergio F. Anzuola was always ready to give a helping hand, Felipe I. Anfurrutia pushed XAK always forward, and Maider Azanza give many clever pushes towards GROVE and provides invaluable assistance during the review process. I would like to express my gratitude to remaining members of Onekin: Luis M. Alonso, Iker Azpeitia, Oscar Barrera, Arantza Irastorza, Jon Iturrioz, Arturo Jaime, Jon Kortabitarte, Felipe Martin, Sandy Perez and Juanjo Rodriguez. We shared together many hours of intense work. Oscar was working hard to keep up with us all the way.

The project was funded by the *Spanish Ministry of Education and Science* allowing me to concentrate full-time on this work. I benefited immensely from its financial support to visit the University of Texas at Austin (USA) and the Fraunhofer IESE (Germany).

The day I landed at Austin's Bergstrom Airport to visit *Don Batory* changed

the course of this work. The accomplishment of this visit was remarkable. The hard work we did so. I was fortunate for attending his course on Feature Oriented Programming. I also thank him for reviewing this thesis.

I am very grateful for having the opportunity to get in contact with eminent people in the field. Part of the results presented in this work are the outcome of collaborations together with other individuals and research groups, alphabetically, Gentzane Aldekoa (University of Mondragon), Don Batory (University of Texas at Austin) and David Benavides (University of Seville). All were fusion from different visions and fruits of hard collaborative work where distances were shortened to accomplish new ideas. Many other pals helped me somehow during this work: Sven Apel from the University of Magdeburg; Jia Liu, Sahil Taker, and Greg Freeman from the University of Texas at Austin; Thomas Forster from the Fraunhofer IESE; and Roberto L. Herrejon from Oxford University.

Despite the demand of this work, there was fortunately life outside of the tower (Arbide is where we work). This is thanks to many relatives and *lacuadri* friends that helped me to distract from the thesis matters.

*My family* (Almu, Bego and Truji) fostered the grand *value* of education from my earliest days, and always encouraged and supported me to improve my education. Gracias.

Special affection to *Gen*. She was there during all the hard times, always providing encouragement. Now that I have to do the same for her, I wish I can be as supportive and loving as she was. Eskerrik asko.

As last note for the reviewers and members of the dissertation committee, let me appreciate your *invaluable* work beforehand.

# Vita

*Salvador Trujillo Gonzalez* was born in *Durango*, Spain on May 6th, 1978, the son of Salvador Trujillo Ruiz (father) and Begoña Gonzalez Alonso (mother). After a wonderful childhood, he entered the Department of Computer Sciences at the University of Mondragon in 1996, where he received the degree of Technical Engineer in Computer Sciences (Bachelor of Science) in 1999. He followed his studies in the same university receiving later the degree of Engineer in Computer Sciences (Master of Science) in 2002. This year, he entered the Doctoral School at the University of the Basque Country where he conducted his doctoral research until 2007.

Salva finds referring to himself in the third person to be *weird* and hopes nobody will ever get to read this.

Permanent Address:
San Ignacio, 9C - 2G
ES-48200 Durango, Biscay, Spain
struji@gmail.com
http://www.struji.com

This dissertation was typed by the author.