Model Driven Product Line Engineering: Core Asset and Process Implications

Dissertation
presented to
the Department of Computer Languages and Systems of
the University of the Basque Country
in Partial Fulfillment of
the Requirements
for the Degree of

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

Maider Azanza Sesé

Supervisor: *Prof. Dr. Oscar Díaz García*
San Sebastián, Spain, 2011

Esta obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-SinObraDerivada 3.0 Unported

This work was hosted by the *University of the Basque Country* (Faculty of Computer Sciences). The author enjoyed a doctoral grant from the Basque Goverment under the *"Researchers Training Program"* during the years 2005 to 2009. The work was was co-supported by the Spanish Ministry of Education, and the European Social Fund under contracts WAPO (TIN2005-05610) and MODELINE (TIN2008-06507-C02-01).

The important thing is not to stop questioning; curiosity has its own reason for existing. One cannot help but be in awe when contemplating the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of the mystery every day.

– Albert Einstein

Summary

Reuse is at the heart of major improvements in productivity and quality in Software Engineering. Both *Model Driven Engineering (MDE)* and *Software Product Line Engineering (SPLE)* are software development paradigms that promote reuse. Specifically, they promote systematic reuse and a departure from craftsmanship towards an industrialization of the software development process. *MDE* and *SPLE* have established their benefits separately. Their combination, here called *Model Driven Product Line Engineering (MDPLE),* gathers together the advantages of both.

Nevertheless, this blending requires *MDE* to be recasted in *SPLE* terms. This has implications on both the core assets and the software development process. The challenges are twofold: *(i)* models become central core assets from which products are obtained and *(ii)* the software development process needs to cater for the changes that *SPLE* and *MDE* introduce. This dissertation proposes a solution to the first challenge following a feature oriented approach, with an emphasis on reuse and early detection of inconsistencies. The second part is dedicated to assembly processes, a clear example of the complexity *MDPLE* introduces in software development processes. This work advocates for a new discipline inside the general software development process, i.e., the *Assembly Plan Management*, which raises the abstraction level and increases reuse in such processes. Different case studies illustrate the presented ideas.

Contents

1 Introduction **1**

1.1 Overview 1
1.2 Context 2
1.3 General Problem 3
1.4 This Dissertation 4
1.5 Contributions 6
1.6 Outline 8
1.7 Conclusions 10

2 Background **11**

2.1 Overview 11
2.2 Model Driven Engineering 11
2.2.1 Definition 12
2.2.2 Motivation 13
2.2.3 Models 14
2.2.4 Metamodels 15
2.2.5 General Purpose Languages vs. Domain Specific Languages 16
2.2.6 Transformations 18
2.2.7 The Four Layer Architecture 20
2.2.8 Technical Spaces 21
2.2.9 Successful Case Studies 24
2.2.10 Current Research Issues 25

2.3 Software Product Line Engineering 25
2.3.1 Definition 26
2.3.2 Motivation 27
2.3.3 Engineering a Software Product Line 29
2.3.4 Successful Case Studies 31
2.3.5 Current Research Issues 32
2.4 Software Processes 33
2.4.1 Definition 33
2.4.2 Motivation 34
2.4.3 Successful Case Studies 35
2.4.4 Current Research Issues 36
2.5 Model Driven Product Line Engineering 37
2.5.1 Definition 38
2.5.2 Motivation 38
2.5.3 Successful Case Studies 38
2.5.4 Current Research Issues 39
2.6 Conclusions 40

3 Problem Statement 41
3.1 Overview 41
3.2 Core Asset Implications 42
3.3 Process Implications 45
3.3.1 General Overview 46
3.3.2 The Assembly Process in *MDPLE* 48
3.4 Conclusions 52

4 Domain Specific Composition of Model Deltas 53
4.1 Overview 53
4.2 The Crime and Safety Survey Questionnaire SPL 54
4.3 Delta Metamodels 57
4.4 Delta Composition 61
4.5 Defining Deltas for UML Interaction Diagrams 64

4.6 Incremental Consistency Management in Delta Composition . . . 68
4.7 Discussion . . . 70
4.8 Related Work . . . 71
4.9 Conclusions . . . 72

5 Increasing Reuse in Model Delta Composition **75**
5.1 Overview . . . 75
5.2 Realizing Model Composition . . . 76
5.3 Realizing Generic Delta Composition . . . 78
5.4 Realizing Domain-Specific Delta Composition . . . 81
5.5 A Model for Composition Annotations . . . 84
5.6 Relating to Other Technical Spaces: Jak . . . 89
5.7 Discussion . . . 92
5.8 Related Work . . . 93
5.9 Conclusions . . . 94

6 The Assembly Process in *MDPLE* **95**
6.1 Overview . . . 95
6.2 Assembly Plan Management Overview . . . 96
6.3 Megamodel Engineering Phase . . . 98
6.3.1 Process . . . 99
6.3.2 Models . . . 100
6.3.3 Transformations . . . 102
6.3.4 Case Study . . . 104
6.4 Family Assembly Engineering Phase . . . 108
6.4.1 Process . . . 108
6.4.2 Models . . . 109
6.4.3 Case Study . . . 110
6.5 Assembly Program Engineering Phase . . . 111
6.5.1 Process . . . 112
6.5.2 Models . . . 112

6.5.3 Transformations 113
6.5.4 Case Study 113
6.6 Product Assembling Phase 115
6.7 Discussion 116
6.8 Related Work 117
6.9 Conclusions 118

7 Conclusions 121
7.1 Overview 121
7.2 Results 122
7.3 Publications 123
7.4 Research Stages 125
7.5 Assessment and Future Research 125
7.6 Conclusions 128

Bibliography 131

List of Figures

1.1 Chapter Map 9

2.1 Basic Concepts of Model Transformations 19
2.2 The 4 /3+1 Layer Architecture 21
2.3 Modeling Architecture Examples 22
2.4 Essential Activities in SPL Development 30

3.1 Cone of Instances of a Metamodel: Model Instances are the SPL Products 43
3.2 Models, Arrows and Product Lines 44
3.3 The Assembly Space 51

4.1 Feature Model for the CSSPL 54
4.2 Feature Implementation Options 56
4.3 Questionnaire Metamodel 58
4.4 Constraint Violation by the *Minor* Feature 59
4.5 Delta Questionnaire Metamodel 60
4.6 Delta Metamodel Cone 61
4.7 Composition of Minor and Base Features 62
4.8 Component Metamodel and Component Composition Example 64
4.9 UML Interactions as Sequence Diagrams 65
4.10 *ThrowDice* • *Move* Interaction Composition 67

5.1 Example of Match Rule in ECL 76

5.2 Example of Merge Rule in EML 77
5.3 Extended Questionnaire Metamodel 79
5.4 Generic Composition Implementation 80
5.5 Domain-Specific Composition Implementation 82
5.6 Annotated Questionnaire Metamodel 83
5.7 Obtaining Composition Implementation from the Annotation Model 85
5.8 Annotation Metamodel 85
5.9 Annotation Model Example 86
5.10 Match Customization Example 87
5.11 Copy Customization Example 88
5.12 Annotated Jak Delta Metamodel 90

6.1 SPEM Diagram of the Assembly Plan Management Discipline 97
6.2 Megamodel Engineering Phase 99
6.3 Assembly Megamodel Metamodel 100
6.4 Assembly Machine Tool Metamodel 101
6.5 Portlet MDD Assembly Megamodel 104
6.6 Portlet MDD Assembly Machine Tool Model 106
6.7 Portlet Assembly Machine Tool (Class Examples) 107
6.8 Family Assembly Engineering Phase 108
6.9 Family Assembly Metamodel 109
6.10 PinkCreek Family Assembly Model (simplified) 110
6.11 Assembly Program Engineering Phase 111
6.12 Assembly Equation Metamodel 112
6.13 Assembly Equation Example for PinkCreek 114
6.14 Assembly Program Example for PinkCreek 114
6.15 Product Assembling Phase 116

Chapter 1

Introduction

"Wayfarer, there is no path; you create the path as you keep on walking."
– *Antonio Machado.*

1.1 Overview

The software industry remains reliant on the craftsmanship of skilled individuals engaged in labor intensive manual tasks. However, growing pressure to reduce cost and time to market and to improve software quality may catalyze a transition to more automated methods. In this context, advanced software development paradigms such as *Model Driven Engineering (MDE)* and *Software Product Line Engineering (SPLE)* represent forward steps on the path to the industrialization of software development. *Model Driven Product Line Engineering (MDPLE)* gathers together the advantages of both. This dissertation contributes to realize this vision by proposing a solution to two fundamental challenges that appear when combining these paradigms.

1.2 Context

Over the years, the software industry has met the demands of their clients by mostly relying on the skills of individual developers, just as artisans met the demands of an increasingly industrialized society by relying on the skills of individual craftsmen. Nevertheless, this is effective only up to a point. Beyond such point, the means of production are overwhelmed. If we consider the evolution of other industries when faced with similar challenges many years ago, they moved from craftsmanship to industrialization by learning to customize and assemble standard components to produce similar but distinct products, by standardizing, integrating and automating their production processes, by developing product lines and so on. These changes enabled the cost effective production of a wider variety of products to satisfy a broader range of customer demands [GS03]. This experience has provided insights on possible paths for software development industrialization.

MDE and *SPLE* are two increasingly popular paradigms that promote the industrialization of software development and increase software reuse. The former achieves reuse through abstractions (i.e., models) and model transformations. Models capture the essence of the software at hand while transformations are the means for reuse, as they encode reusable mappings of models to lower abstraction levels. As for *SPLE*, the goal is to build a set of related software products (i.e., a product family) out of a common and previously built set of core assets. Unlike *MDE*, now the stress is not so much on the abstraction level at which software is specified, but on conceiving software as preplanned variations obtained from core assets. The set of software products that can be derived from them forms the product family and the preplanned usage of core assets is the means to achieve reuse.

These differences also impact the way software is developed. In *MDE*, coding is substituted by modeling and transforming. Hence, the software development process becomes a pipeline of model transformations that

eventually leads to an end product. By contrast, in *SPLE* programs are built, step-by-step, by incrementally adding or removing features (i.e., increments in program functionality that customers use to distinguish one application from another [KCH+90]). Not only does this alleviate software complexity and improve program understandability, it also permits the reuse of features, as multiple products of a product line can share the same feature.

Therefore, *MDE* and *SPLE* differ in both the reuse strategy and their respective development processes. The benefits of these paradigms have been reported in academia and industry [BLW05, Béz05, CN01, OMG, PBvdL06]. More to the point, being both paradigms orthogonal, they can be synergistically combined leading to *Model Driven Product Line Engineering*.

1.3 General Problem

In order to achieve the blending of *MDE* and *SPLE*, *MDE* needs to incorporate *SPLE* principles. The implications are twofold: *(i)* models are developed from core assets instead of from scratch and *(ii)* the development process needs to accomodate both *MDE* and *SPLE* principles.

Regarding the first point, in *SPLE* reuse is planned, enabled and enforced, the opposite of opportunistic [CN01]. This implies that two complete development cycles exist. The first one builds artifacts *for reuse*, i.e., the core assets that will be reused at a later stage. In the second one, products are built *with reuse*, leveraging on the core assets built previously [CE00]. Being models the primary focus in *MDE*, the same two cycles should be applied to them. Although there is work on developing models *with reuse* [MKBJ08, MBJ08, VG07], less effort has been dedicated to developing models *for reuse*. If models are core assets that will later be reused to form products, there are certain issues that need to be explored. The metamodel such core assets conform to and its relation with the metamodel of the resulting products is only one example, along with the way

in which products are obtained from core assets and how core assets are transformed. Benefits in a product line are derived from the reuse of core assets in a strategic and prescribed way [CN01]. Consequently, a clear definition of model core assets is required if the benefits of *MDPLE* are to be obtained.

The second point refers to the changes *MDPLE* brings to the software development process. An explicit process prevents errors from being introduced in the product and provides means for controlling the quality of what is being produced [CG98]. Hence, an explicit process for *MDPLE* is desirable. Nevertheless, *MDE* does not make any assumptions on the software development process or the design methodology [MD08]. This may lead to adoption problems, particularly in industry [BLW05]. Regarding *SPLE*, several methodologies to build product lines have been established during the past few years [BFK+99, CN01, KKL+98, PBvdL06, WL99]. Nonetheless, these methodologies do not explicitly cater for the changes *MDE* introduces. Both *SPLE* and *MDE* introduce changes in the traditional software development process, which *MDPLE* needs to accomodate. Compared to traditional development processes, new activities and artifacts emerge. Among other reasons, defining an explicit process for *MDPLE* permits to reuse such process in different settings and improves communication among managers, workers and customers [CG98, Ost87].

Assembly processes, i.e., processes that *assemble* a product from the product line core assets, illustrate the process implications of *MDPLE*. Being assembly processes in both *SPLE* and *MDE* complex in their own right [DSB04, RRGLR+09, VAB+07], their combined use in *MDPLE* puts even more stringent demands on the assembly process [TBD07].

1.4 This Dissertation

This dissertation proposes a solution to the problems posed above. First, it describes the development of models *for reuse* (i.e., as core assets) following a feature oriented paradigm. Second, it advocates for a new discipline

inside the general software development process, i.e., the *Assembly Plan Management*, that permits to face the assembly process complexity that exists in *MDPLE*. The following paragraphs delve into the details.

Feature Oriented Software Development (FOSD) is a general paradigm for product synthesis in *SPLE* [BSR04]. It advocates for the incremental development of the products, a technique to handle design complexity and improve design understandability [Wir83]. Here, features are not only increments in program functionality that customers use to distinguish one application from another, but are the actual building blocks of the artifact at hand (i.e., features are the actual core assets). We can define features as *arrows* that map one artifact to another with enhanced functionality [BAS08]. Hence, the final artifact with the desired features is obtained by successively applying the corresponding arrows. The challenge rests on applying this paradigm to model development. Traditionally, *MDE* works with models that stand for a complete representation of the software to be built. This dissertation describes incremental development of models following the feature oriented paradigm, where arrows are realized as model deltas and domain specific composition of such deltas is presented. A metamodel definition for model deltas and the application of incremental consistency checking to delta composition results in earlier error detection. Moreover, automatic generation of delta composition implementation increases reuse and facilitates the usage for domain experts.

In the second part, this dissertation presents the *Assembly Plan Management* as a new discipline inside *MDPLE*. Three are the main benefits. First, it reduces the complexity of assembly processes by separating them in different phases and by applying *MDE* to assembly processes themselves. Second, the specification of such processes at a higher level of abstraction, as a result of using *MDE,* permits developers to concentrate on the essential information and to ponder the advantages and trade-offs of each design decision. Last, it increases reuse by applying *MDE* and by dividing the assembly process in different phases, were the output of each phase can be reused in different instances of the successive ones.

The following section provides a summary of the main contributions of this dissertation.

1.5 Contributions

From Complete Models to Model Deltas

- *Problem Statement.* In *FOSD* software artifacts are developed by incrementally adding features. In *MDE*, software development revolves around models. When models are the artifact to be developed using a feature oriented approach, we need to define how features are realized.

- *Contribution.* Features are realized as model deltas (i.e., models that encompass the additions a feature makes). The main advantage rests on the fact that such model deltas conform to a metamodel, which is derived from the metamodel complete models conform to. In this way, complete models and model deltas are described using the same constructs, which permits to check certain constraints at the model delta building time and makes deltas easier to understand and analyze.

Domain Specific Composition of Model Deltas

- *Problem Statement.* Defining model features as models implies that such models need to be composed together to yield the final product, which contains all the features requested by the user. Consequently, an algorithm that composes deltas together is required. This algorithm is domain independent in most cases but it also needs to cater for some specificities of each domain at hand.

- *Contribution.* Although a generic and metamodel agnostic composition algorithm is sufficient in the majority of cases, some domains have composition specific semantics. Our approach permits domain

experts to specify such semantics so that they are taken care of while composing.

Annotations to Automate Model Delta Compositon

- *Problem Statement.* As stated above, delta composition can be domain specific. Such specificity can be captured through code written in a model composition language (e.g., *Epsilon Merging Language* [KPP06a]). However, even though such code is generally not as verbose as general purpose programming languages, it still imposes an important entry barrier for domain experts. Moreover, certain domain specific compositions are liable to be reused in different domains and manual implementation often leads to work repetition.

- *Contribution.* We introduce an annotation-based mechanism for metamodels that hides the composition implementation and eases the specification of domain-specific composition semantics, hence lowering the entry barrier for domain experts. Taken an annotated metamodel as input, the composition algorithm implementation is automatically generated. This approach leads to the automation of repetitive work, better understandability of the composition semantics and better maintenance of the generated implementation.

Assembly Plan Management

- *Problem Statement.* Greenfield and Short venture that one of the consequences of the industrialization of software development, of which *MDPLE* is an example, will be that development becomes mainly component assembly [GS03]. At the same time, both *SPLE*-based and *MDE*-based assembly processes are complex in their own right [RRGLR+09, VAB+07] and *MDPLE* puts even more stringent demands on the assembly process.

- *Contribution.* Based on the previous insights, this work advocates

for a new discipline inside the general software development process, i.e., the *Assembly Plan Management*, that permits to face complexity in assembly processes. For this discipline, new phases, roles, tasks and workproducts are introduced. Such plan is divided into phases, which increases reuse as the result of each phase is the input to different instances of the successive ones.

Model Driven Engineering for Assembly Processes

- *Problem Statement.* Previous experience showed us that assembly process implementation in *MDPLE* is complex, requires design and becomes repetitive, as it lacks reuse mechanisms.

- *Contribution.* By applying *MDE*, assembly processes are specified at a higher abstraction level. This reduces the complexity of assembly processes by increasing reuse and allows developers to make design decisions more accurately since they can concentrate on the essentials of the problem. The repetitive parts of code become part of the transformation that yields code from the abstract specification, which increases reuse.

1.6 Outline

This section summarizes briefly the content of each chapter of this dissertation. Figure 1.1 presents a chapter map to help to put each of them in context.

Chapter 2

This chapter provides background on *Model Driven Engineering*, *Software Product Line Engineering* and their combination in *Model Driven Product Line Engineering* together with a brief introduction to software development processes. These are the main concepts on top of which this work is

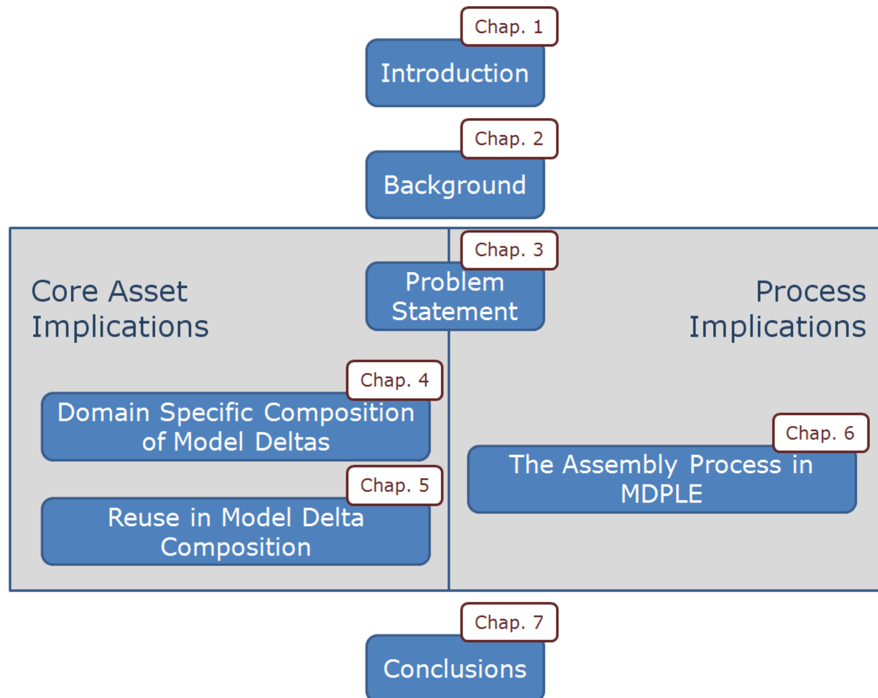


Figure 1.1: Chapter Map

built.

Chapter 3

A general overview is given and the two fundamental challenges which are the topic of this dissertation are motivated, namely the development of model core assets following a feature oriented approach and process implications in *Model Driven Product Line Engineering*.

Chapter 4

This chapter presents incremental development of models following a feature oriented development approach where features are realized as deltas. How these deltas can be composed to yield the end product is also described. Moreover, domain specific composition of deltas is motivated.

Two case studies are presented: a questionnaire family, and the incremental specification of UML interaction diagrams.

Chapter 5

In this chapter, metamodel annotations are presented as the means to specify domain-specific composition. In this way, the composition implementation is automatically generated from the metamodel. This not only permits the composition implementation to be reused in different domains, but also shields the implementation details from domain experts.

Chapter 6

This chapter describes the *Assembly Plan Management,* a new discipline inside the general software development process that permits to face the complexity in assembly processes introduced by *MDPLE*. A case study of flight booking portlets illustrates the presented ideas.

Chapter 7

This chapter concludes the dissertation. It summarizes the obtained results, makes an assessment and also identifies future research topics that this work raised.

1.7 Conclusions

The intention of this chapter was to give an overview of the contents of this dissertation. The topic was introduced and what, in our opinion, are its contributions were listed. The next chapter starts with a review of the background.

Chapter 2

Background

"We are like dwarfs standing on the shoulders of giants."
– Bernard of Chartres

2.1 Overview

Model Driven Engineering and *Software Product Line Engineering* are two increasingly popular paradigms that represent an advance towards the industrialization of software development. *Model Driven Product Line Engineering* is their combination, which brings together the advantages of both.

This chapter covers the background on *MDE*, *SPLE* and *MDPLE,* together with a brief introduction to software development processes. These are the main topics on top of which this work is built.

2.2 Model Driven Engineering

MDE is a consolidating trend in software engineering in which abstract models are created and systematically transformed to concrete implemen-

tations [FR07]. Reuse is here achieved in the form of transformations, which are built once but can be enacted for a variety of input models that yield different results.

MDE is the result of a paradigm shift from object-orientation, where the main principle was *everything is an object,* into a model driven paradigm based on the principle that *everything is a model* [Béz05]. In object orientation objects and classes are the main concepts. These concepts are related using *instantiation* (i.e., an object is an instance of a class) and *inheritsFrom* (i.e., a class can inherit from another class). In the same way, the main concepts in *MDE* are models, the system they represent, metamodels and model transformations. The main relations in this case are *representation* of a system (i.e., a model is the representation of a particular view of a system) and *conformance* to metamodel (i.e., each model is written in the language defined by its metamodel) [Béz05].

The following subsections define and motivate *MDE* and describe its main concepts.

2.2.1 Definition

Model Driven Engineering describes software development approaches that are concerned with reducing the abstraction gap between the problem domain and the software implementation domain. This is achieved through the use of technologies that support systematic transformation of problem-level abstractions to software implementations. The complexity of bridging the abstraction gap is tackled through the use of models that describe complex systems at multiple levels of abstraction and from a variety of perspectives, combined with automated support for transforming and analyzing those models. In the *MDE* vision of software development, models are the primary artifacts of development and developers rely on computer-based technologies to transform models into running systems [FR07].

2.2.2 Motivation

Considering models as the primary artifacts in software development has many benefits, aside from increasing reuse by means of automatic transformations. Some of these benefits are still being discovered and researched. We can cite the following as examples of the motivations to apply or evaluate *MDE* in industry [MD08]:

- *Increase productivity and shorten development time.*

- *Improve quality.* Improve the quality of the generated code, improve the quality (assurance) of system requirements and manage requirement volatility, improve the quality of intermediate models, and earlier detection of bugs.

- *Automation.* Generate code and other artifacts and introduce automation into the development process. Model-based simulation and testing.

- *Standardization and formalism.* Provide a common framework for software development across the company and phases of the lifecycle that formalizes and organizes software engineering knowledge at a higher level of abstraction and a common data exchange format.

- *Maintenance and evolution concerns.* Maintain the architecture intact from analysis to implementation, evolution of legacy systems, concerns over software method and tool obsolescence, verification of the system by producing models from traces and that platform independent models have a considerable lifespan.

- *Improved communication and information sharing.* Between stakeholders and within the development team. Ease of learning.

The main concepts of *MDE* are described next.

2.2.3 Models

The *MDE* hallmark is that software development's primary focus are models rather than traditional programs. Consequently, it is important to understand what a model is. Models play an important role in different sciences (e.g., mathematics or biology) and each one may have a specialized view of the concept of a model. Even inside the software engineeering community different definitions of a model have been given [MFB09]. In this work, we will adopt the following definition [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.

This definition highlights three characteristics of models: *(i)* models are abstractions of part of the reality, *(ii)* which are expressed in a language and *(iii)* which provide knowledge about the reality for a certain purpose.

Models have been an essential part of engineering from antiquity. Engineering models aim to reduce risk by helping us better understand both a complex problem and its potential solutions before undertaking the expense and effort of a full implementation. To be useful and effective, an engineering model must possess, to a sufficient degree, the following five key characteristics [Sel03]:

- *Abstraction.* A model is always a reduced rendering of the system that it represents. By removing or hiding detail that is irrelevant for a given viewpoint, it permits us understand the essence more easily.

- *Understandability.* It is not sufficient just to abstract away detail; we must also present what remains in a form (e.g., a notation) that most directly appeals to our intuition. A good model provides a shortcut by reducing the amount of intellectual effort required for understanding.

- *Accuracy.* A model must provide a true-to-life representation of the modeled system's features of interest.
- *Predictiveness.* We should be able to use a model to correctly predict the modeled system's interesting but non obvious properties, either through experimentation or through some type of formal analysis.
- *Inexpensiveness.* A model must be significantly cheaper to construct and analyze than the modeled system.

2.2.4 Metamodels

The previous model definition emphasized that each model is expressed in a modeling language. A metamodel is the model that defines such language. It is important to note that a metamodel defines a language in an abstract way, regardless of its specific syntax. The following definition highlights such abstraction:

> A metamodel is an "abstract language" for describing different kinds of data; that is, a language without a concrete syntax or notation [OMG08a].

We have already stated that in *MDE everything is a model*. Hence, metamodels are also models:

> A metamodel is a model of a language of models [Fav04].

A metamodel also defines constraints that every model written in that language must satisfy. Consequently, it delimits the way in which the system can be modeled (i.e., it constraints the set of models that can be written in the language defined by the metamodel). The FRISCO report definition emphasizes this point:

> A metamodel gives the conception of the structure of concepts on which a modelling language is based, and all constraints thereupon. Thus, a metamodel determines the way one may view, conceive or model the "world" [FHL+98].

Metamodels facilitate separation of concerns. When dealing with a given system, one may work with different views of the same system, each characterized by a given metamodel [Béz04], which can later be composed into an integrated application [KR03].

In summary, metamodels define the languages used to describe models. There are two main trends when choosing the metamodel to model a system. On one hand, a general purpose modeling language, a prominent example being the *Unified Modeling Language (UML)* [OMG09], can be used. On the other hand, a *Domain Specific Language (DSL)* might suit some systems better. Both approaches are briefly described next.

2.2.5 General Purpose Languages vs. Domain Specific Languages

This section provides a concise summary on the main trends that exist when choosing a language (i.e., a metamodel) for a particular domain. Both are trends described and a comparison between them is made.

General Purpose Languages

The main advantage of these languages is their generality, i.e., they can be used in a myriad of domains. The *Unified Modeling Language* is a prominent example of such languages. UML is a modeling language whose objective is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software based systems as well as for modeling business and similar processes [OMG09]. It is defined by the *Object Management Group (OMG)*, an international, open membership, not-for-profit computer industry consortium that develops enterprise integration standards. UML is a general purpose modeling language that can be used with all major object and component methods and can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms (e.g., Common

Object Request Broker Architecture (CORBA), Java 2 Enterprise Edition (J2EE), .NET).

There are situations, however, in which a language that is so general and of such a broad scope may not be suitable for modeling applications of some specific domains. In this case, *Domain Specific Languages* tackle this problem.

Domain Specific Languages

DSLs are focused languages for specifying systems at a high-level of abstraction, using a notation very close to the problem domain. Their goal is to allow domain experts to specify and reason about their systems using intuitive notations, closer to the language of the problem domain, and at the right level of abstraction [Val10].

Two alternatives exist to develop such a language. The first one is to specialize UML. In this case, some elements of the language are specialized, imposing new restrictions on them, while respecting the UML metamodel and leaving the original semantics of the UML elements unchanged. UML provides a set of extension mechanisms (stereotypes, tagged values, and constraints) for specializing its elements, allowing customized extensions of UML for particular application domains. These customisations are grouped into *UML Profiles* [FV04]. The second alternative is to define a completely new language. In this way, the syntax and semantics of the elements of the new language are defined to fit the specific characteristics of the domain.

Comparison

Each alternative has its advantages and disadvantages. Defining a tailor-made language will produce a notation that will perfectly match the concepts and nature of the specific application domain. However, as the new language does not respect UML semantics, it will not allow the use of commercial UML tools for drawing diagrams, generating code, reverse

engineering, and so forth. Conversely, UML Profiles (which are amenable to be handled by most commercial UML tools) may not provide such an elegant and perfectly fitting notation as may be required for those systems. It is not therefore always easy to decide when to create a new language and when to define a set of extensions to the standard UML metamodel by grouping them into a UML Profile [FV04]. The decision depends on the specificities of the domain at hand.

Besides models and metamodels, the other main concept in *MDE* are model transformations which will be described next.

2.2.6 Transformations

Model transformations define relationships between sets of models [FR07]. As a result of the *everything is a model* principle, model transformations tend to operate on a more diverse set of artifacts than program transformations. Model transformation literature considers a broad range of software development artifacts as potential transformation subjects. These include, but are not limited to, UML models, interface specifications, data schemas, component descriptors, and program code. Model transformations play a key role in *MDE*. Their applications include the following [CH06]:

- Generating lower-level models, and eventually code, from higher-level models.
- Mapping and synchronizing among models at the same level or different levels of abstraction.
- Creating query-based views of a system.
- Model evolution tasks such as model refactoring.
- Reverse engineering of higher-level models from lower-level models or code.

Figure 2.1 presents a simplification of the model transformation pattern [JABK08] that gives an overview of the main transformation concepts. A

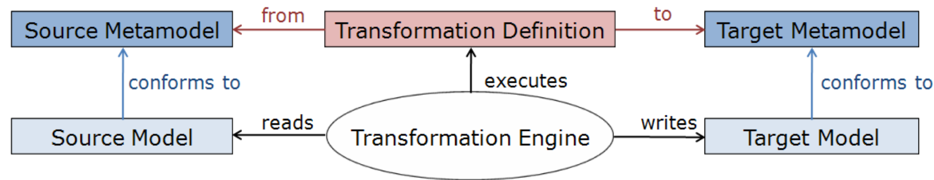


Figure 2.1: Basic Concepts of Model Transformations

transformation definition is a relationship defined between a source and a target metamodel. Once this transformation is defined, a transformation engine will execute the definition thus creating a model that conforms to the target metamodel, from a model that conforms to the source metamodel. In general, transformations may have multiple source and target metamodels. Furthermore, the source and target metamodel can be the same in some settings. These are known as endogenous transformations.

A classification of the different transformation approaches is given in [CH06]. The major characteristics used to organize the vast range of existing transformation approaches are:

- *Specification.* A particular transformation specification may represent a function between source and target models and be executable; however, in general, specifications describe relations and are not executable.

- *Transformation rules.* Transformation rules can be understood in a broad sense as the smallest units of transformation. Rewrite rules with a lefthand side (LHS) and a righthand side (RHS) are obvious examples of transformation rules; however, functions or procedures implementing some transformation step can be understood as a transformation rule. In fact, the boundary between rules and functions is not so clear-cut; for example, function definitions in modern functional languages such as Haskell resemble rules with patterns on the left and expressions on the right.

- *Rule application control.* It has two aspects: location determination

and scheduling. Location determination is the strategy for determining the model locations to which transformation rules are applied. Scheduling determines the order in which transformation rules are executed.

- *Rule organization*. This comprises general structuring issues, such as modularization and reuse mechanisms.

- *Source-target relationship*. The source-target relationship is concerned with issues such as whether source and target are one and the same model or two different models.

- *Incrementality.* This refers to the ability to update existing target models based on changes in the source models.

- *Directionality.* Directionality describes whether a transformation can be executed in only one direction (unidirectional transformation) or multiple directions (multidirectional transformation).

- *Tracing.* Tracing is concerned with the mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements.

Transformations play a pivotal role in *MDE* and are one of the enablers of reuse. Consequently, they have raised considerable interest among the research community [ICM], several model transformation languages have been developed [JABK08, OMG08a, CMT06] and different approaches to ease their realization have been proposed [LWK10, WKK$^+$10].

2.2.7 The Four Layer Architecture

The OMG defines a four layer architecture for metamodeling [OMG02]. This architecture, which has also been named as the 3+1 architecture [Béz04], is presented in Figure 2.2. At the bottom level, the M0 layer is the real system. Two models represent different views of such system at level M1.

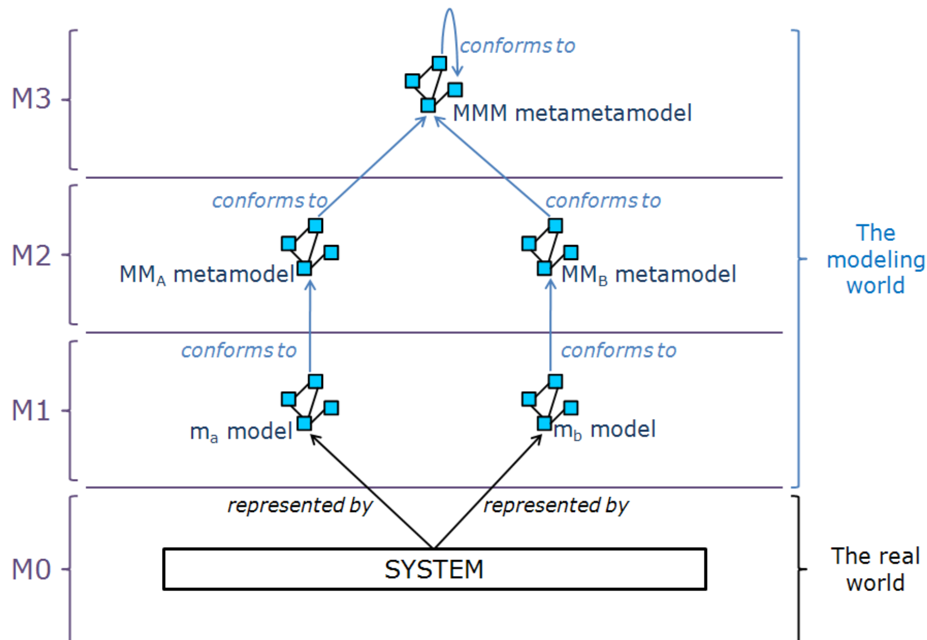


Figure 2.2: The 4 /3+1 Layer Architecture

These models conform to their respective metamodels, which are defined at level M2. Metamodels conform to a common metametamodel defined at level M3. An important aspect of this architecture is the fact that the M3 level conforms to itself. This means that the metametamodel at M3 provides a set of language constructs to express the whole metametamodel. This brings an important engineering benefit: the metametamodel may be treated in the same way as the metamodels that conform to it. This allows to build tools that handle the models at different levels in the model management system in a uniform way [BK06].

2.2.8 Technical Spaces

The definition of models given in Section 2.2.3 is general. An XML document, a running program, a database, etc. are representations of systems found in the real world and they are all expressed using a language. That is, following our definition, they are models. These examples expose an

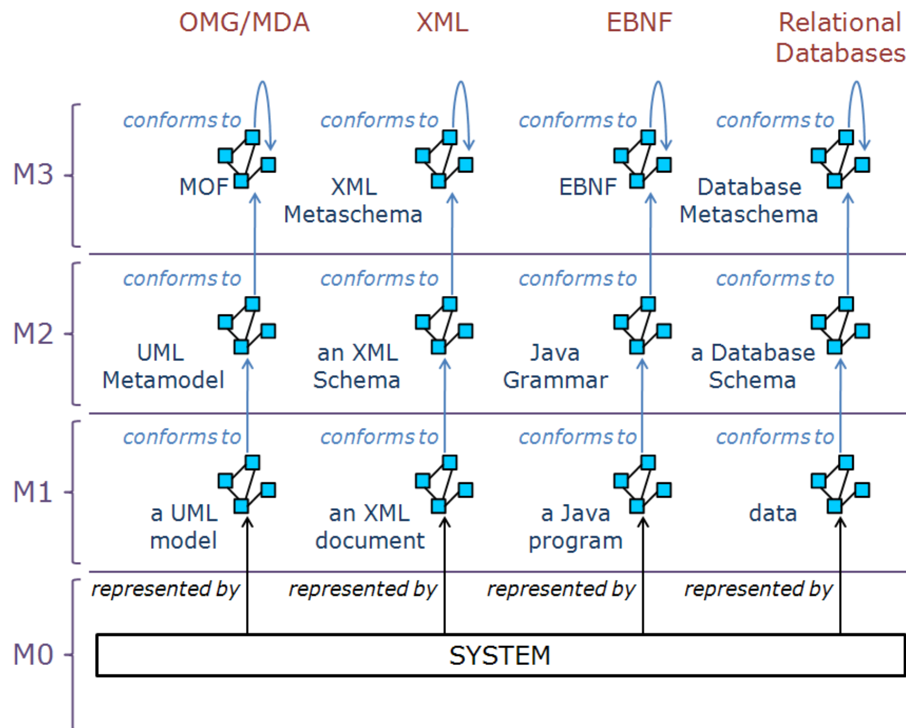


Figure 2.3: Modeling Architecture Examples

important commonality. They provide an organizational structure usually based on a single model that is used to create other models in the space (e.g., programming languages, XML schemas, database schemas, etc.). These models are related via the *conforms to* relationship and are organized in the 3+1 level architecture, akin to the one presented above [BK06].

Figure 2.3 presents the modeling architecture observed in four examples: OMG/MDA, XML, EBNF (also known also as grammarware) and Relational Databases. In every example the metametamodel at level M3 provides the foundation for defining metamodels and models that conform to them.

Model Driven Architecture (MDA) is the *MDE* framework launched by the OMG [OMG]. Its metametamodel is known as *Meta Object Facility (MOF)* [OMG08a], which provides a language for defining the abstract

syntax of modeling languages. One of such languages is UML. In *XML*, XML documents conform to XML schemas, which in turn conform to the XML Metaschema. In the grammarware example *EBNF* is a framework for defining the syntax of programming languages. These definitions take the form of context-free grammars. In *Relational Databases* the data conform to the database schema which conforms to the database metaschema.

This architecture is the cornerstone for building a model management framework. It is mainly based on the fixed metametamodel at M3 and the meaning of the *conforms to* relation between levels. It should be noted that the *conforms to* relationship is defined differently in each example (e.g., in XML it is based on the notion of validity of XML documents and in grammarware *conforms to* means that a sentence is syntactically correct according to the grammar rules).

These examples also share another common characteristic: they have a set of tools associated with them (e.g., tools to check the *conforms to* relationship, navigation languages or transformation languages). Based on the previous commonalities, we can define a common ground that will enable us to describe technologies at a more abstract level in order to allow reasoning about their similarities and differences and possibilites for integration. Each of the four examples above can be denoted as a *Technical Space (TS)* [BK06]:

> A technical space is a model management framework accompanied by a set of tools that operate on the models definable within the framework.

Other examples of technical spaces are the *Eclipse Modeling Framework (EMF) TS* [SBPM08], the *Microsoft DSL Tools TS* [GS04] and the *Generic Modeling Enviroment (GME) TS* [LMB+01].

Technical spaces allow reasoning about multiple technologies at a higher level of abstraction and about possible relations among them. An important benefit of this reasoning is the recognition of the various capabilities offered by different technical spaces and the possibility of combining them

together to solve a problem. This requires the ability to transfer an artifact from one space to another space and vice versa. This is accomplished using technical projectors [BK06].

After describing the main concepts related to *MDE*, the following section describes the experiences of its application.

2.2.9 Successful Case Studies

Several companies have reported the benefits of applying *MDE* to software development. With over 13.000 software engineers and an experience of over 15 years with modeling languages, Motorola is a case in point [BLW05]. This company has gained consistent benefits from *MDE* and code generation. Typical results collected over the past few years have shown the following benefits when compared to hand written code:

- *Quality.* A 1.2x - 4x overall reduction in defects and a 3x improvement in phase containment of defects.
- *Productivity.* A 2x - 8x productivity improvement when measured in terms of equivalent source lines of code.

Moreover, OMG describes several MDA Success Stories [OMG]. The U.S. Goverment Intelligence Agency, Siemens Transformation Systems, ABB Research Centre, Austrian Railways and Daimler Chrysler are some of the companies that have chosen MDA as their developing architecture. Among the benefits they reported there are:

- Facilitates communication among different stakeholders (U.S. Goverment Intelligence Agency).
- High level of reuse (Siemens Transformation Systems).
- Efficient tailoring of existing functionality (ABB Research Centre).
- Uniform and easy-to-maintain component infrastructure (Austrian Railways).

- ROI in less than 12 months (Daimler Chrysler).

2.2.10 Current Research Issues

France and Rumpe grouped the major challenges that researchers face when attempting to realize the *MDE* vision into the following categories [FR07]:

- *Modeling language challenges.* These challenges arise from concerns associated with providing support for creating and using problem-level abstractions in modeling languages, and for rigorously analyzing models.

- *Separation of concerns challenges.* These challenges arise from problems associated with modeling systems using multiple, overlapping viewpoints that utilize possibly heterogeneous languages.

- *Model manipulation and management challenges.* These challenges arise from problems associated with *(i)* defining, analyzing, and using model transformations, *(ii)* maintaining traceability links among model elements to support model evolution and roundtrip engineering, *(iii)* maintaining consistency among viewpoints, *(iv)* tracking versions, and *(v)* using models during runtime.

This thesis is linked to the first and third challenges. First, Chapter 4 defines model deltas as feature realizations in an *FOSD* setting. Consequently, it focuses on the definition of the language to define such deltas (i.e., the delta metamodel). Second, model management is a complex issue, which is exacerbated when using models in *SPLE*. This thesis addresses this problem in Chapter 6 in the specific case of product assembling.

2.3 Software Product Line Engineering

Software Product Line Engineering is a paradigm to develop software where a family of related products is built out of a common set of core assets,

thus reducing development costs for each individual product. It is important to indicate that reuse in *SPLE* is managed, meaning that core assets are developed with their reuse opportunities in mind. This is in contrast to opportunistic reuse, where components are developed on the hope that they will be reused at some point in the future.

2.3.1 Definition

With the aim to help the reader understand the concept, two definitions will be analyzed next.

> A *Software Product Line (SPL)* is 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].

This definition can be divided into five main concepts [Tru07]:

- *Products (i.e., "a set of software-intensive systems"). SPLE* shifts the focus from single product development to the development of a family of related products. Software product families are defined (analogously to hardware families) as sets of products whose common properties are so extensive that it is advantageous to study the common properties of the products before analyzing individual members [Par76].

- *Features (i.e., "sharing a common, managed set of features").* Features are increments in program functionality that stakeholders use to distinguish one product from another [KCH$^+$90].

- *Domain (i.e., "that satisfy the needs of a particular market segment or mission").* An SPL is created within the scope of a well defined domain.

- *Core Assets (i.e., "are developed from a common set of core assets").* Core assets are artifacts or resources that are used in the production of more than one product in a software product line [CN01].
- *Production Plan (i.e., "in a prescribed way").* The production plan describes how products are produced from the core assets [CN01]. It is the guide for reuse within the product line scope.

An alternative definition is presented in [PBvdL06]:

> Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization.

Where:

- *Platform.* A software platform is a set of software subsystems and subsystems that form a common structure from which a set of derivative products can be efficiently developed and produced [ML97].
- *Mass customization.* Mass customization is the large-scale production of goods tailored to individual customers' needs [Dav87]. In the case of *SPLE*, software is the product to be obtained.

2.3.2 Motivation

The aim of *SPLE* is to provide customized products at reasonable costs. The key motivations for developing software using this paradigm are the following [PBvdL06]:

- *Reduction of Development Costs.* Core assets are reused in several products, which implies a cost reduction for each product. However, it is important to note that, before core assets can be reused, an investment has been made in creating them. Moreover, the way in which they will be reused has to be planned beforehand to provide managed reuse. This means that an up-front investment needs to be

made to create the product line before it can reduce the costs per product by reusing the core assets.

- *Enhancement of Quality.* The core assets are reused and tested in many products. This implies a significantly higher chance of detecting faults and correcting them, thereby increasing the quality of all products.

- *Reduction of Time-to-Market.* The initial time-to-market is higher, as the core assets need to be built first. However, after this phase has finished, the time-to-market is considerably shortened as core assets can be reused in each product.

- *Reduction of Maintenance Effort.* Whenever a core asset is changed (e.g., for error correction), the changes are propagated to all products in which the core asset is used. This can be exploited to reduce maintenance effort. As changes have been made, product testing is unavoidable but the reuse of test procedures within the product line also reduces maintenance effort.

- *Coping with Evolution.* The introduction of a new core asset (or a change of an existing one) triggers the evolution of all products that are built using such core asset.

- *Coping with Complexity.* As more and more functionality is put into software, the complexity of the products increases. Embedded systems are an example, where functionality is being moved from hardware to software. The fact that core assets are reused throughout the product line helps to cope with the complexity of each product.

- *Improving Cost Estimation.* The development organization can focus its marketing efforts on those products that can be easily produced within the product line. It can also allow extensions not covered by the product line, but products that need such extensions can be sold for higher prices than those built within the scope of the product line.

After defining and motivating *SPLE*, how the product line approach is realized is described next.

2.3.3 Engineering a Software Product Line

Engineering a software product line traditionally involves two separate processes, namely domain engineering and application engineering. The former is dedicated to core asset development while the latter is aimed at yielding products. Both are briefly described next:

- *Domain Engineering.* Domain Engineering is the process of *SPLE* in which the commonality and the variability of the product line are defined and realized [PBvdL06]. This means that the common parts and the ones that are different (i.e., variable) among products are defined and realized in the form of core assets.

- *Application Engineering.* Application Engineering is the process of *SPLE* in which the products of the product line are built by reusing domain artifacts and exploiting the product line variability [PBvdL06]. In this case, products are obtained by reusing the previously built core assets.

We have already mentioned that managed reuse is the main driving force behind *SPLE*. How each core asset is to be reused is pre-planned. Indeed, management in general plays a critical role in *SPLE*. The activities performed in the two processes above must be given resources, coordinated and supervised. This perspective is present but intertwined in the above definitions. The *Software Engineering Institute (SEI)* [Pes03] brings management to the forefront when defining the three essential activities in SPL development [CN01] (see Figure 2.4):

- *Core Asset Development (i.e., Domain Engineering).* Its goal is to establish a production capability for products. It has three outputs: *(i)* the SPL scope, *(ii)* core assets and *(iii)* the production plan.

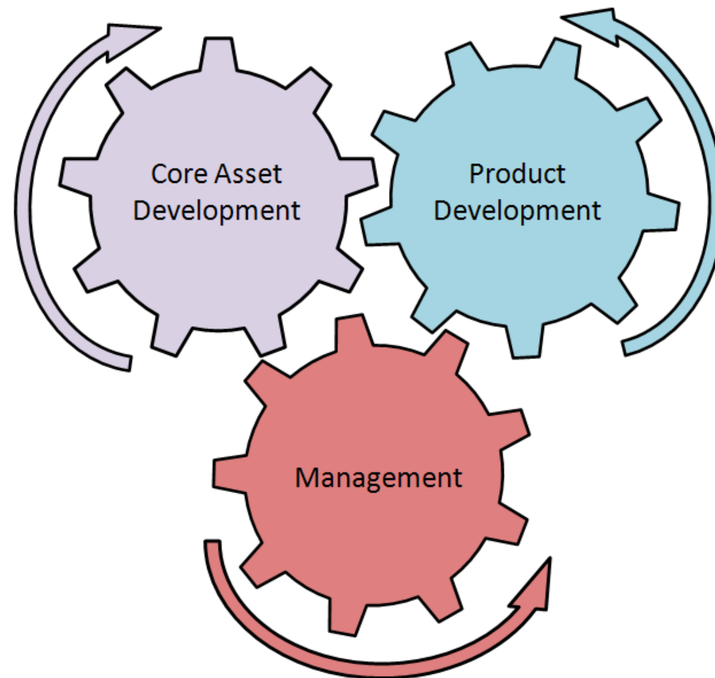


Figure 2.4: Essential Activities in SPL Development

- *Product Development (i.e., Application Engineering).* Its goal is to turn out products. It depends on the three outputs described above.

- *Management.* Management at both technical and organizational levels must be strongly commited to the SPL effort. Technical management oversees the core asset development and product development activities by ensuring that the groups that build core assets and the groups that build products are engaged in the required activities, follow the processes and collect data to track progress. Organizational management must set in place the proper organizational structure that makes sense for the enterprise and must make sure that the organizational units receive the right resources (e.g., well-trained personnel) in sufficient amounts.

Figure 2.4 illustrates how these three activities are linked together and perpetually in motion (indicated with arrows). Not only are core assets used to develop products, but also product development brings revisions of existing core assets or even new ones. There is also a constant need for strong, visionary management to invest resources in the development and sustainment of the core assets. Management must also precipitate the cultural change to view new products in the context of available assets. Either new products must align with the existing assets or assets must be updated to reflect new products.

2.3.4 Successful Case Studies

Cummings Engine Inc. is the world largest manufacturer of commercial diesel engines above 50 horsepower. Among the benefits that a product line approach has brought them since 1994 we can cite a slashed product cycle time, high software quality, high customer satisfaction and more successful projects [CN01].

The United States National Reconnaissance Office (NRO) decided to take advantage of the commonality of the software associated with satellites and build a product line for their ground based spacecraft command and control software. They report, among other benefits, 50% reduction in overall cost and schedule and nearly tenfold reductions in development personell [CN01].

Asea Brown Bovery (ABB) is a leading global technology company which has experience with different product lines and its associated benefits. As an example, the ABB Gas Turbine Family led to shorter development time, higher code quality and eased the exchange of modules [PBvdL06].

Celsius Tech. built a product line for warfare control systems. The benefits included reduced system cost by around 50%, delivery time slashed from years to months, reuse ranging from 70% to over 90% and higher product quality and customer satisfaction [CN01].

The International Software Product Line Conferences (SPLC) are the premier conference in the area where most recent ideas, innovations, trends, experiences, and concerns in software product lines and software product family engineering are presented and discussed [SPLa]. Each SPLC culminates with a session in which members of the audience nominate systems for induction into the Software Product Line Hall of Fame, the members of which constitute examples of successful case studies of *SPLE* [SPLb].

2.3.5 Current Research Issues

SPLs have generated considerable interest in the research community. The fact that, apart from the specific conference [SPLa], more general conferences [ASE, ECO] include product lines in their scope, along with the different workshops that exist [VAM], are a sign of such interest. Regarding the current research issues, as an example we can cite that the focus of the 2010 edition of SPLC is on novel approaches to effective sharing of software assets inside and across organizations. Some other topics of interest in the conference are the following:

- Techniques and tools for product line engineering.
- Evolution of product line assets.
- Business issues for product lines.
- Organizational and process issues for product lines.
- Product line life-cycle issues.
- Effective software product lines across organizational boundaries.
- Software product line in new emerging application domains (e.g., Web services and online applications, Service oriented systems, etc.).

This thesis is linked to the first and third challenges. First, *MDE* is a new paradigm that aids in the development of product lines, where models *for*

reuse need to be defined. Second, the complexity of assembly processes (i.e., process issues for product lines) are addressed.

Both *MDE* and *SPLE* represent a perspective shift in the way software is developed. Hence, the next section provides a brief description of software development processes, as both *MDE* and *SPLE* have an impact on them.

2.4 Software Processes

Processes have a profound influence on the quality of products; i.e., by controlling processes we can achieve a better control of the required qualities of products. This is especially true in software due to its intrinsic complexity [Bro87]. If an explicit process is in place, software development proceeds in a more systematic and orderly fashion. This prevents errors from being introduced in the product and provides the means for controlling the quality of what is being produced [CG98].

2.4.1 Definition

A software process can be defined as the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain a software product [Fug00].

Hence, the following factors impact on the software development process [Fug00]:

- *Software Development Technology.* Technological support used during the process. Certainly, to accomplish software development activities we need tools, infrastructures, and environments. We need the proper technology that makes it possible and economically feasible to create the complex software products our society needs.

- *Software Development Methods and Techniques.* Guidelines on how to use technology and accomplish software development activities.

The methodological support is essential to exploit technology effectively.

- *Organizational Behavior.* The science of organizations and people. In general, software development is carried out by teams of people that have to be coordinated and managed within an effective organizational structure.

- *Marketing and Economy.* Software development is not a self-contained endeavor. As any other product, software must address real customers' needs in specific market settings. Thus different stages of software development (e.g., requirements specification and development/deployment) must be shaped in such a way to properly take into account the context where software is supposed to be sold and used.

2.4.2 Motivation

The key difference between a process and a process description needs to be highlighted at this point. While a process is a vehicle for doing a job, a process description is a specification of how the job is to be done. Thus cookbook recipes are process descriptions while the carrying out of the recipes are processes. It is startling to realize that too often we develop large software systems without the aid of visible and detailed descriptions of how to proceed [Ost87]. Among the motivations to define explicit and detailed process descriptions we can cite [CG98, Ost87]:

- The manager of a project can communicate to workers, customers and other managers just what steps are to be taken in order to achieve product development or evolution goals.

- Workers can benefit in that reading them should indicate the way in which work is to be coordinated and the way in which each individual's contribution is to fit with others' contributions.

- In materializing software process descriptions it becomes possible to reuse them. At present key software process information is locked in the heads of software managers. It can be reused only when these individuals instantiate it and apply it to the execution of a specific software process. When these individuals are promoted, resign or die their software process knowledge disappears.

2.4.3 Successful Case Studies

As an example, we can mention the *Capability Maturity Model for Software (CMM)*, a process maturity framework that provides software organizations with guidance on how to gain control of their processes for developing and maintaining software and how to evolve towards a culture of software engineering and management excellence, as advocated by the *Software Engineering Institute (SEI)* [PWCC93].

In many companies CMM plays a major role in defining the software process improvement. Data in industry shows that CMM-based process improvement can make a difference. As successful case studies we can cite [DS97]:

- Raytheon yielded a twofold increase in its productivity and a return ratio of 7.7 to 1 on its improvement expenditures, for a 1990 savings of $4.48 million from a $0.58 million investment.

- Hughes Aircraft has computed a 5-to-1 return ratio for its process improvement initiatives, based on changes in its cost–performance index.

- Tinker Air Force Base recently computed a 5-to-1 return on investment for its process improvement initiatives, which generated a savings of $3.8 million from a $0.64 million investment.

2.4.4 Current Research Issues

The following were identified as software process research needs, based on the results of USC-CSE workshops with its industry and government affiliates [Boe05]:

- *Lean, Hybrid Processes for Balancing Dependability and Agility.* The trends in simultaneous need for high dependability and high agility dominate here. Additional concerns for special cases are the increased need for scalability and incrementality for large, software-intensive systems of systems involving COTS and legacy systems; and the needs to address multi-location and multi-cultural development.

- *Integrated Technical and Acquisition Processes.* Improvements in administrative and contracting processes tend to lag behind improvements in technical processes, causing the technical process to become over-constrained and unstable. Again, balancing dependability and agility is important, and the ability to administer and incentivize collaborative efforts that are performing concurrent plan-driven increments and agile next-increment preparation across multiple supplier chains are important.

- *Empirically-Evolved Process Languages, Methods, Metrics, Models, and Tools.* The need for such capabilities to be incremental and ambiguity tolerant, and to allow for incomplete, informal, and partial specifications are important, as are techniques for bridging gaps between less and more formal specifications and gaps or inconsistencies across life cycle phases or suppliers. The use of empirical evaluation testbeds to accelerate maturity and transition of research results, and to support collection of baseline process data for evaluating improvement priorities is important as well. A further attractive avenue is the development of capabilities to capitalize on computational plenty. The empirical framework could also be extended to

monitor and evaluate progress and risk areas in the wild card autonomy and bio-computing areas.

- *Virtual Process Collaboration Support*. Shifting the GUI focus from individual performance to distributed team, multi-stakeholder, and multi-cultural collaboration is a significant need.

- *Game Technology for Process Education and Training.* Game engines complemented by virtual reality modeling and simulation have become tremendously powerful, and provide an excellent support base for developing "acquire and develop the way you train; train the way you acquire and develop" capabilities.

The contribution of this thesis relates to the first topic, exploring the challenges that arise when defining processes that cater for the particularities of *MDPLE* development. This work adresses the specific case of asembly processes.

Once *MDE* and *SPLE* have been introduced and the importance of explicit software processes has been emphasized, the following section is dedicated to *Model Driven Product Line Engineering,* which is the main topic of this thesis.

2.5 Model Driven Product Line Engineering

MDE and *SPLE* differ in both what they stress as reuse strategy (i.e., broadly, abstraction vs. variability) and the development processes they follow (i.e., broadly, model transformation vs. incremental construction). *MDE* and *SPLE* complement each other. *MDE* brings abstraction to the *SPLE* realm which has historically focused on code artifacts. On the other hand, *SPLE* moves variability, through managed reuse, to the forefront. As both paradigms are orthogonal, they can be synergistically combined, leading to *Model Driven Product Line Engineering*.

The following sections briefly describe *MDPLE*.

2.5.1 Definition

Model Driven Product Line Engineering is a paradigm for software development that combines *Model Driven Engineering* and *Software Product Line Engineering* to build families of related models that are then transformed into the desired products. Consequently, metamodels and transformations become main core assets. Moreover, models become a central in the development process and have to cope with variability.

2.5.2 Motivation

SPLE reduces the development cost by leveraging on the commonalities of a family of related products, thus increasing reuse. In this setting, it is essential to maintain a relationship between requirements, architecture, design and implementation core assets, as they have to be reusable and evolvable over time. *MDE* can be used to tacke this problem, being transformations the means to trace those relationships from one model to another. The combination of both results in *MDPLE*.

MDPLE combines the benefits of *Model Driven Engineering* (see Section 2.2.2) and the benefits of *Software Product Line Engineering* (see Section 2.3.2). Moreover, it brings additional benefits such as verification of the product building platform and optimization of product production times [BAS08, TBD07].

2.5.3 Successful Case Studies

MDPLE has raised a considerable interest among researchers in the last few years. Some examples can be found in [CA05, MKBJ08, TBD07, VG07]. Another indication of this interest are the specific workshops on *MDPLE* held at relevant conferences [MAP, MDP]. However, to the best of our knowledge, industrial size case studies of *MDPLE* are still few. An example of such can be found in [TGLH$^+$10].

2.5.4 Current Research Issues

The earlier sections highlighted the benefits of *MDPLE*. Nevertheless, for this vision to be realized, some issues need to be resolved. *SPLE* poses interesting challenges to *MDE*. The first is related to core assets. All members of a product line may have models that describe them. Since the product line members have usually significant overlaps in their functionality, their models also have significant overlaps and it is desirable to factor out such overlaps in the form of model deltas. So, model languages have to take this into account [AJTK09]. This raises questions that need to be answered. Among others:

- What is the best modeling language to describe such overlaps (i.e., model deltas) and to what extent is such language similar to the languages used to describe complete models?
- Do model management operations change when considered in an *SPLE* setting?
- Is there any way to guarantee that the family of models developed using *MDPLE* is correct (i.e., that they all conform to their meta-model)?

Another fundamental issue is related to the software development process. *MDE* and *SPLE* and moreover *MDPLE* require considerable changes in an organization and in the processes that yield each product. Compared to traditional development processes, new activities and artifacts emerge, which need to be explicitly defined beforehand. In these sense, some of the questions that arise are:

- We briefly described the processes to develop a product line in Section 2.3.3. How do these processes change when applying *MDE*?
- What are the new roles, activities and artifacts?
- How can the *MDE* and *SPLE* processes be combined, with reuse as the main goal?

This dissertation will try to give answers to what the language to describe model deltas is and how the model composition operation changes when considered in an *SPLE* setting. Moreover, incremental consistency checking will be used to pave the way to guaranteeing that the products conform to their metamodel. As for the development process, the case of assembly processes will be described and a solution will be proposed.

2.6 Conclusions

This chapter was to provided a concise introduction to the existing background on the topics of this thesis, namely:

- Model Driven Engineering
- Software Product Line Engineering
- Software Development Processes
- Model Driven Software Product Line Engineering

The goal was to prepare the ground for the contributions introduced in the following chapters. The interested reader can find further details of the covered topics in the references.

Chapter 3

Problem Statement

"The worthwhile problems are the ones you can really solve or help solve, the ones you can really contribute something to."

– Richard P. Feynman

3.1 Overview

Software has evolved from relatively small and simple products to ones with a considerable size and a high degree of complexity. At the same time, the product's desired time-to-market is constantly decreasing. On top of that, extensibility and customization to each customer are nowadays more a requirement than a wish. Therefore, a paradigm shift is needed to change the way software is produced. To this end, advanced software paradigms have emerged, being *Model Driven Product Line Engineering* an example. The benefits of such approach have already been described in the previous chapter. Nevertheless, to fully exploit them, certain research challenges need to be resolved. This chapter describes how, in order to achieve the combination in *MPLE*, *MDE* needs to be adapted to *SPLE* terms. The impact is twofold: *(i)* on the core assets (i.e., models

for reuse need to be developed) and *(ii)* on the development process. Feature oriented development of models and the product assembly process are presented as illustrative examples of such impact.

3.2 Core Asset Implications

SPLs aim at building a family of related products from a common set of core assets. If models are the products to be obtained, how the corresponding core assets are developed and how products are obtained out of them needs to be defined. This raises different issues, which are described below. *Feature Oriented Software Development* [BSR04] is used as the SPL realization paradigm.

FOSD is a paradigm for product synthesis in software product lines. Programs are built, step-by-step, by incrementally adding features. Not only does this help control program complexity and improve program understandability, but it also allows for the reuse of features (i.e., multiple programs in a product line can share the same feature) [BSR04]. Hence, features in *FOSD* are not only increments in program functionality, they become the actual building blocks (i.e., the core assets) that, when composed, yield the different products of the SPL.

FOSD can also handle heterogeneous artifacts in a uniform way [BSR04]. Apart from code, it has been successfully applied to build grammars [BSR04], XML documents [ADT07] and test specifications [UGKB08] among others. Moreover, these ideas can be abstracted to support the composition of software artifacts written in different languages [AKL09].

FOSD ideas scale: they have been used to build customizable databases (80K LOC each), extensible Java preprocessors (40K LOC each), and AHEAD Tool Suite (250K LOC), a set of tools that realizes FOSD ideas [BAS08].

In the context of *MDPLE*, the question that arises is how these ideas can be combined with *MDE* to support feature oriented development of models, i.e., to recast *MDE* to follow *SPLE* principles, as mentioned be-

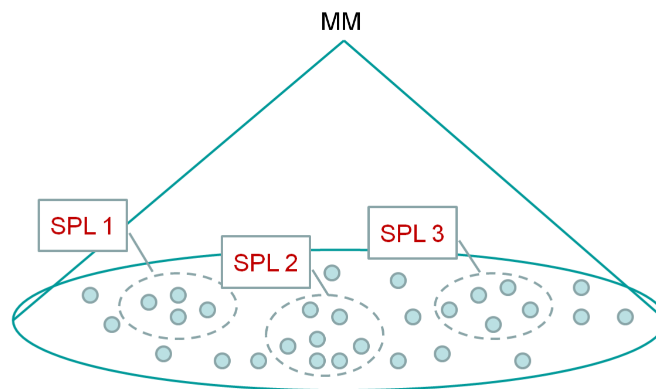


Figure 3.1: Cone of Instances of a Metamodel: Model Instances are the SPL Products

fore. Figure 3.1 shows a metamodel `MM` and its cone of instances, i.e., the set of models that conform to `MM`. For each metamodel there is typically an infinite number of instances. An SPL, in contrast, is a finite family of *n* similar products. Hence, the scope of an SPL is typically more constrained than the domain a metamodel defines, i.e., an SPL is a subset of a metamodel's domain (e.g., `SPL 1`, `SPL 2` and `SPL 3` in Figure 3.1). As a case in point, if `MM` is a metamodel of statecharts, one could find SPLs for flight booking web applications, a another of embedded systems for washing machines and many others.

As stated above, a product line in an *MDE* setting is a set of models. The *baseModel* expresses the greatest common denominator of all SPL members. In many product lines, the *baseModel* is simply the empty model $\emptyset$ [BAS08].

Figure 3.2a shows a metamodel `MM` and its cone of instances. A subset of the cone is a set of models that belongs to the `SPL` product line (e.g., $m_2 ... m_4$ belong to `SPL`, m_1 is not a member, and all $m_1 ... m_4$ conform to `MM`). Note that the empty model $\emptyset$ and model m_A are outside the cone of `MM`, meaning that $\emptyset$ and m_A do not conform to `MM`.

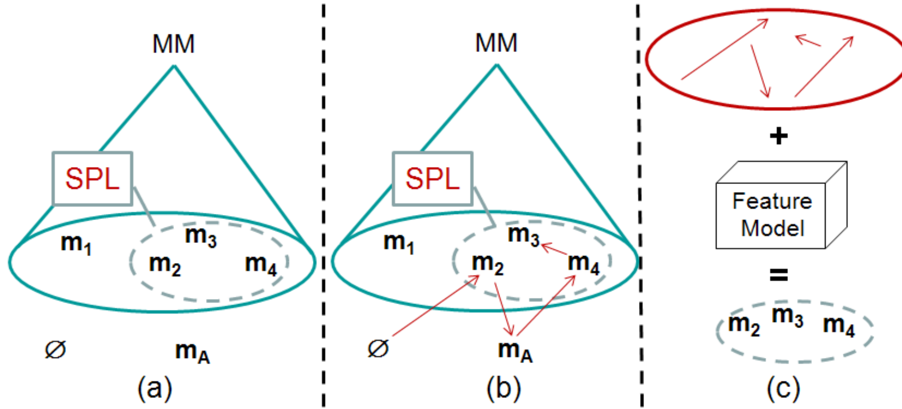


Figure 3.2: Models, Arrows and Product Lines

Figure 3.2b shows features as *arrows*[1] that map one model to another with enhanced functionality. In this sense, we can define the products of *SPL* as the composition of the arrows that represent their features (e.g., m_3 is the result of composing three arrows). An important property about arrows is that a composition of two or more arrows is yet another arrow [BAS08]. Further, observe that a composition of arrows may produce an intermediate model (e.g., m_A) that does not conform to `MM`. Only the members of `SPL` (e.g., m_2, m_3 and m_4) are required to conform to `MM`, regardless of the steps needed to obtain them. The members of `SPL` are produced by composing arrows. The collection of arrows defines the features of `SPL`.

A specification of all legal compositions of features in a product line is defined by the *Feature Model* [KCH$^+$90]. Every model in the SPL has a derivation (i.e., a composition of arrows starting from $\emptyset$) and the feature model defines it. Figure 3.2c illustrates the product line perspective: a collection of arrows (i.e., features) plus a feature model that constrains how these arrows can be composed yields the set of models of a product

[1]The term comes from Category Theory. Arrows are maps between objects. In this work we are interested in arrows that build SPLs. See [BAS08] for a description of the links between *MDE*, *SPLE* and Category Theory.

line.

The relationships among arrows described above are very general; they hold for *all* artifact types. The key questions when mapping arrows to models are:

1. How is an arrow specified?
2. If arrows are themselves models then, what is their metamodel?
3. How are arrows composed?

These issues are addressed in Chapter 4 and Chapter 5.

This section described how model development can be tackled in *MDPLE*. This impacts on the core assets, arrows (i.e., models *for reuse*) need to be specified and then composed together to yield the final product. Nevertheless, *MDPLE* also has repercussions on the way the final product is obtained, i.e., on the development process. Domain and application engineering have to cater for the specificities of models (e.g., metamodels and transformations have to be developed). The following section presents an overview of the impact of *MDPLE* on the software development process.

3.3 Process Implications

Both *MDE* and *SPLE* depart from one-off development to provide an infrastructure where different (though related) products can be obtained. The benefits of applying them separately in an industrial setting have been reported in the literature [BLW05, CN01]. Their complementary nature permits their combination in *MDPLE*, which brings benefits that were enumerated in the previous chapter. Different case studies support this claim [CA05, FBL08, TBD07, TGLH+10, UGKB08, VG07, WJE+09, ZSS+09].

Nevertheless, *MDE* and *SPLE,* and also *MDPLE*, being the combination of both, require considerable changes in an organization and in the processes that yield each product. Compared to traditional development

processes, new activities and artifacts emerge, which need to be explicitly defined. The following section introduces such changes.

3.3.1 General Overview

This section introduces the necessity to define an explicit process to develop software using *MDE*, *SPLE* and *MDPLE* and the changes these paradigms introduce when compared to traditional software development.

Most *tried and tested* processes are not tailored for *MDE*, which does not make any assumptions on the software development process or the design methodology [MD08]. However, the lack of a well defined process may hinder *MDE* adoption. As a case in point, Baker et al. report that many teams in Motorola encountered major obstacles in adopting *MDE* due to the lack of a well-defined process, lack of necessary skills and inflexibility in changing the existing culture [BLW05]. Consequently, an explicit process that provides new artifacts such as metamodels, transformations, etc. is desirable. Such process should address the following questions among others [FS04]:

1. How many levels of abstraction there are, and what platforms have to be integrated.

2. What the modeling notations are and the abstract syntax to be used at each level of abstraction.

3. How transformations are performed, and what platform and additional information they integrate into the lower level of abstraction.

4. How code is generated for the modeling language used at the lowest level of abstraction, and perhaps even how to deploy that code.

5. How a model can be verified against the upper level model, how it can be validated, and how it can generate test cases for the system under development.

As for *SPLE*, the introduction of the paradigm in an organization is not trivial and should fulfill certain steps. The product line adoption roadmap should include the following seven major activities according to [Nor04]:

1. Deciding and justifying what products to include in the product line (SPL scoping).
2. Defining, documenting, and following processes for software development and management.
3. Preparing the organization for a software product line approach.
4. Designing and providing the common assets that will be used to construct the products in the product line.
5. Building and using the production infrastructure (necessary plans, processes, and tools).
6. Building products from the core assets in a prescribed way.
7. Monitoring the product line effort, keeping a pulse on the adoption activities and the product line operations, and applying course corrections as necessary to keep the organization on course.

Note how the second and fifth points emphasize the need to define explicit processes for the SPL. Moreover, the first adoption phase, namely establishing context, involves paving the way for the product line adoption by determining the scope and associated business case, ensuring the necessary process capability, and performing the necessary organizational management tasks [Nor04]. Once again, the importance of having explicit processes is stated.

Several methodologies to build SPLs have been established during the past few years [BFK+99, CN01, KKL+98, PBvdL06, WL99]. In their development process, generally the idea is to first define the context (i.e., the scope of the SPL). Then, *Domain Engineering* and *Application Engineering*, which were already described in the previous chapter, are conducted

as the two main phases of the SPL development process [Mat04]. Developing an SPL also involves new artifacts (e.g., core assets, production plans, etc) and new activities (e.g., SPL domain scoping). We refer the reader to [CN01] for a comprehensive account.

Both *SPLE* and *MDE* introduce changes in the traditional software development process. *MDPLE* needs to accomodate all of them if its benefits are to be fully exploited. The *Assembly Process* (i.e., the process to assemble a product) is a case in point, as it needs to cater for new tasks such as model compositions and model transformations in order to yield a the final product.

3.3.2 The Assembly Process in *MDPLE*

Greenfield and Short [GS03] venture that one of the consequences of the industrialization of software development, of which *MDPLE* is an example, will be that product developers will build about 30% of each product. The remaining 70% will be supplied by ready-built vertical and horizontal components. Most development will be component assembly, involving customization, adaptation, and extension. This is the case for *MDPLE*, where the reuse of core assets (i.e., metamodels, models and transformations) reduces the cost of building products. However, the counterpart is that the cost of their assembly increases significantly.

Assembly processes in both *SPLE* and *MDE* are complex in their own right [CN01, RRGLR+09, VAB+07]. The combined use of *SPLE* and *MDE* puts even more stringent demands on such process. An example of this difficulty was described in [TAD07]. Completing the assembling process for just an individual product of the family took itself four people/day. This case study is used in the following paragraphs to illustrate the problem.

PinkCreek is a product line of portlets (i.e., building blocks of web portals) for the provision of flight reservation capabilities to travel-agency portals [DTP07]. The challenges during its construction were twofold. First,

both the diversity and instability of Web platforms advise to abstract away from concrete implementations. This grounds the use of *MDE*. Second, flight reservation is similar among companies but not exactly the same. Companies exhibit variations in how flight reservation is conducted. Such diversity is captured through *features* that are supported using *SPLE* techniques. Hence, features stand for variations on flight reservation whereas *transformations* account for mappings between the distinct levels of abstraction at which flight booking is captured. Next, the distinct artifacts that arise from both the *MDE* and the *SPLE* perspectives are described (a more comprehensive account can be found at [TBD07]).

- *MDE Perspective.* An *MDE* practitioner first strives to abstract from the different web platforms that can realize portlets. To this end, *State Charts* (*SC*) are introduced to model the flight-reservation control flow. The product is described as sequence of states where each state represents an HTML fragment (i.e., a unit of delivery during user interaction). States are connected by transitions whose handlers either execute some action, render some view, or both. A *State Chart* is then mapped into a model for portlet controllers: the *Ctrl* metamodel. *Ctrl* models are in turn, mapped into *Act* and *View* models, that describe the actions to be performed and the views to be rendered during the controller execution. Finally, technical platforms include *Java/Jak* (i.e., a Java language for supporting *features* [BSR04]) and *Java Server Pages (JSPs)*.

  Summing up, *PinkCreek*'s metamodels include *SC, Ctrl, Act, View,* and *Java/Jak* and *JSP* as technical platforms. This requires the existence of transformations between them, namely: *sc2ctrl*, *ctrl2act, ctrl2view, act2jak* and *view2jsp*. For instance, the equation $app = act2Jak \bullet ctrl2act \bullet sc2ctrl \bullet baseModelSC$ specifies a transformation chain from a *baseModel* statechart down to its realization to *Jak* code.

- *SPLE Perspective.* An SPL practitioner first identifies the common-

ality and variability of the product family. The variability on flight reservation includes, among others, the possibility of on-line checking or the alternatives to compensate travel agencies for their cooperation when inlaying this portlet into the agency's portal (e.g., click-through fees, where the carrier will pay the agency based on the number of users who access the portlet; bounties, where the carrier will pay the agency based on the number of users who actually sign up for the carrier services through the agency portal; and transaction fees, where the incomes of the ticket sales are split between the carrier and the agency). Hence, a base core can be leveraged by consecutively applying the required features as described in Section 3.2. PinkCreek's features include, among others, *Reservation, ClickThroughFees, BountyFees* and *TransactionFees*.

As described above, feature composition can be regarded as function composition. For instance, the equation $app = bountyFees \bullet reservation \bullet baseModelSC$ increments the *baseModel* described as a statechart with the *reservation* functionality, and this in turn, with the *bountyFees* functionality. Notice that both features should be realized at the same level of abstraction as the *baseModel* (e.g., statecharts).

Transformations and features define a two-dimensional space for product assembly (see Figure 3.3). Moving down implies adding more details about the technical platform: from abstract (e.g., statecharts) to implementation (e.g., Java/Jak). On the other hand, moving along the horizontal axis adds features to the final product.

Contrasting features with traditional vertical transformations introduces the question of whether such operations are commutative. As Figure 3.3 suggests, the order (i.e., first add a feature and then transform or first transform and then add the feature) may not matter in *some* domains. From this perspective, ensuring commutativity can be regarded as an additional proof of the validity of transformations and features [BAS08, FBL08, TBD07,

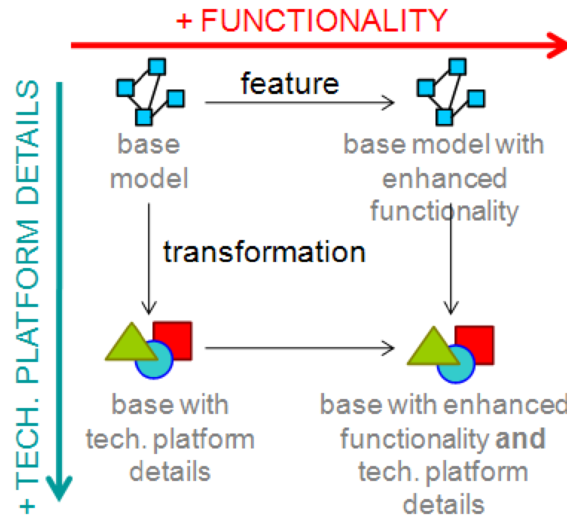


Figure 3.3: The Assembly Space

UGKB08].

An *Assembly Process* can then be described as a path along this assembly space. For instance, the equation $app = act2jak \bullet ctrl2act \bullet sc2ctrl \bullet bountyFees \bullet baseModelSC$ takes the *baseModelSC* as a start, enriches its functionality by leveraging the *baseModelSC* with *bountyFees*, and finally, moves down to code by applying transformation *sc2ctrl*, *sc2act* and *act2Jak*. The outcome is a *Java/Jak* program that provides flight reservation with bounty fees as the payment mechanism.

At the time of *PinkCreek* development, there was not a clear guideline available that allowed to *declaratively* describe the assembly process as a high-level equation. As a result, intricate and compound scripts were required to achieve product assembly. The gained insights when developing such scripts include:

1. *The Assembly Process is complex. PinkCreek* scripts, which realize the assembly process, accounted on average for 500 LOC of batch processes using 300 LOC of ANT makefiles and 2 KLOC of Java code.

2. *The Assembly Process needs to be designed.* Design facilitates to ponder options and tradeoffs and is a way to handle complexity by abstracting away from a large number of accidental details and hence, focusing on the essentials [Bro87]. There is not a single way to product production. Distinct assembly alternatives may need to be contrasted and assembly counterpoints can arise.

3. *The Assembly Process becomes repetitive.* The paradox is that the assembly process in *MDPLE* is geared towards the reuse of software, but its realization often lacks such reuse. This occured in PinkCreek where no reuse mechanism was initially in place. Specifically, defining alternative assembly processes involved a great deal of potential reuse but, paradoxically in an *MDPLE* setting, the only technique available for reuse was rudimentary *"clone&own"*. It did not take long to realize that this technique did not scale.

Summing up, there was a need for a plan that manages how the assembly process is defined, allowing its design and capitalizing on reuse. Chapter 6 describes such plan.

3.4 Conclusions

The benefits of *MDPLE* are substantial. Nevertheless, the combination of *MDE* and *SPLE* impacts on both the core assets that are developed and on the process followed to develop products. This chapter described this impact, using feature oriented development of models and product assembly process as illustrative examples. The questions that were raised will be the topic of the following chapters.

Chapter 4

Domain Specific Composition of Model Deltas

"The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny...'.".

– *Isaac Asimov.*

4.1 Overview

This chapter addresses the first issue presented in the previous chapter, namely how core assets (i.e., models for reuse) are created in *MDPLE*. Arrows are here realized as model deltas that, when composed, deliver a complete model. A relationship between a metamodel and its corresponding delta metamodel will be described. We explain how model deltas can be composed and that, while generic composition algorithms suffice for the majority of cases, there is a need for domain-specific composition algorithms. These ideas are illustrated with two running examples: the *Crime and Safety Survey SPL* and the *Game SPL*. While the first defines a family of *Questionnaires*, the second defines a family of *UML Interaction Diagrams*, as an example of a general purpose language. The chapter ends

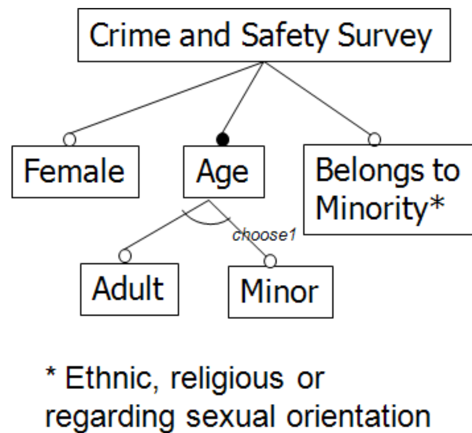


Figure 4.1: Feature Model for the CSSPL

with a brief introduction to incremental consistency management for the conformance checking of model delta composition.

4.2 The Crime and Safety Survey Questionnaire SPL

Questionnaires are a common research instrument in Social Sciences, as the means to gather information. A set of questions is presented to respondents and a research hypothesis is validated against their answers. The web is becoming an invaluable resource in this area, as online questionnaires provide access to groups and individuals who would be difficult, if not impossible, to reach through other channels. It also permits researchers to save time by reaching thousands of people with common characteristics in a short amount of time and helps to reduce costs, when compared to other questionnaire methodologies [Wri05]. However, social scientists generally lack the technical knowledge to make their questionnaires accessible online[1]. To overcome this problem, a domain specific language for question-

[1]Examples of applications that target this need include LimeService (http://www.limeservice.org), SurveyMonkey (http://www.surveymonkey.com) and ESurveyPro (http://www.esurveyspro.com).

naires was defined that abstracts domain experts from the technicalities of the Web platform.

Similar questionnaires that are targeted to different population profiles are needed on a regular basis. A one-size-fits-all questionnaire is inappropriate. Our example focusses on *Crime and Safety Surveys*, a questionnaire family that assess citizens' feelings on the safety of their enviroment. Figure 4.1 shows the feature model for the *Crime and Safety Survey Questionnaire SPL (CSSPL)*[2]. Its features define how questionnaires vary in this domain. Specifically, *(i)* if the respondent is a `female`, she is given a *Sexual Victimization Block* apart from the *Base Questionnaire*, *(ii)* if he or she `belongs to a minority`, a *Hate Crime Block* is added and *(iii)* regarding `age`, `adults` are given the *Consumer Fraud Block* while `minors` receive the *School Violence Block*. Now questionnaire creation is not just a "single shot". Rather, a questionnaire is characterized in terms of features that customize it for a target audience.

There are two basic ways to realize an arrow (i.e. a feature) in a model driven product line. One is to use a general-purpose transformation language. Figure 4.2a shows an example, presenting how the *Minor* feature is expressed in RubyTL, a model transformation language embedded in Ruby [CMT06]. That is, it presents the code that implements how the *Department of Youth* should be added to the `acknowledgments` of the *Base Questionnaire*, how the estimated completion `time` is increased in 10 minutes, and the *School Violence Block* that should be included when the questionnaire is directed to minors. Another way of realizing arrows is simply to create a *model delta* that defines the additions the *Minor* feature makes to a model (see Figure 4.2b). Deltas are models that encompass the additions a feature makes and that, when composed with a base model, deliver a complete model. The latter brings the following advantages:

- Permits the checking of questionnaire constraints (e.g., `option codes` have to follow a certain format).

[2]This *Questionnaire* family is inspired in the European Crime and Safety Survey (`http://www.europeansafetyobservatory.eu/euics_fiq.htm`).

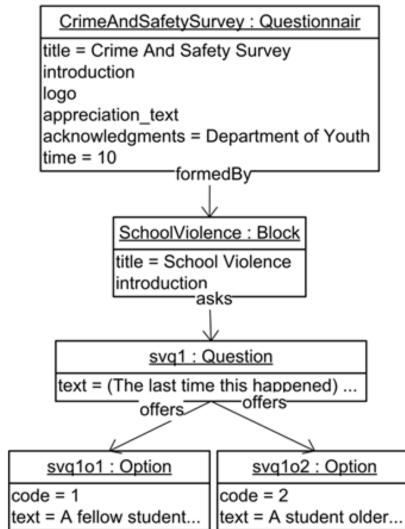
```
transformation 'minors'

rule 'TransformQuestionnaire' do
    from Q1::Questionnaire
    to   Q2::Questionnaire

    mapping do |q, qt|
    svq1o1 = Q2::Option.new
    svq1o1.code = 1
    svq1o1.text = 'A fellow student...'
    svq1o2 = Q2::Option.new
    svq1o2.code = 2
    svq1o2.text = 'A student older...'
    #Code omitted
    svq1 = Q2::Question.new
    svq1.text =
    '(The last time this happened)...'
    svq1.offers << svq1o1
    svq1.offers << svq1o2
    #Code omitted
      svBlock = Q2::Block.new
      svBlock.title = 'School Violence'
      svBlock.asks << svq1

      if (q.title == 'Crime and Safety Survey') then
         qt.title = q.title
         qt.introduction = q.introduction
         qt.logo = q.logo
         qt.appreciation_text = q.appreciation_text
         qt.acknowledgments = q.acknowledgments +
         ' Department of Youth'
         qt.time = q.time + 10
         qt.uses = q.uses
         qt.formedBy = q.formedBy
         qt.formedBy << svBlock
      end
      end
end
#Code omitted
```

(a)



(b)

Figure 4.2: Feature Implementation Options

- Makes deltas that represent features easier to understand and analyze [ZSS+09].
- Separates *what* the feature adds (i.e., new questions, options, etc.) from *how* it is added (i.e., how deltas are composed together).

These advantages are revised in Section 4.7. Note that features are not allowed to delete previously existing model elements.

Figure 4.2b presents arrows as model deltas and the advantages of such decision have been enumerated. If we define deltas as models, as opposed to defining them as transformations, the next natural questions are: *(i)* what metamodel do model deltas conform to? and *(ii)* how is such delta metamodel related to the domain metamodel (e.g., the questionnaire metamodel)? The following section delves into these questions.

4.3 Delta Metamodels

Let M and MM be a model and its metamodel, respectively. Further, let DM and DMM stand for a model delta and its delta metamodel. This section presents a definition of DMM and its relationship with MM.

Figure 4.3 is a simplified *Questionnaire Metamodel* MM. Each questionnaire begins with its `title` and an `introduction` explaining its purpose. It may also contain the `logo` of the organization doing the research and the estimated `time` required to complete it. Questionnaires are structured in `blocks`. A `Scale` is a previously validated instrument to measure a variable. A block can be defined using a scale or it can be specifically created for the questionnaire at hand. Each block can have subblocks and zero or more `questions`, where each question has a `text` that formulates it with two or more `options` as its answers. Each option has a `text` that gives a possible answer for the question and a `code` that is used to encode the answer. Questionnaires end with an `appreciation_text`, thanking respondents and there may be `acknowledgments` for the organizations that funded the research.

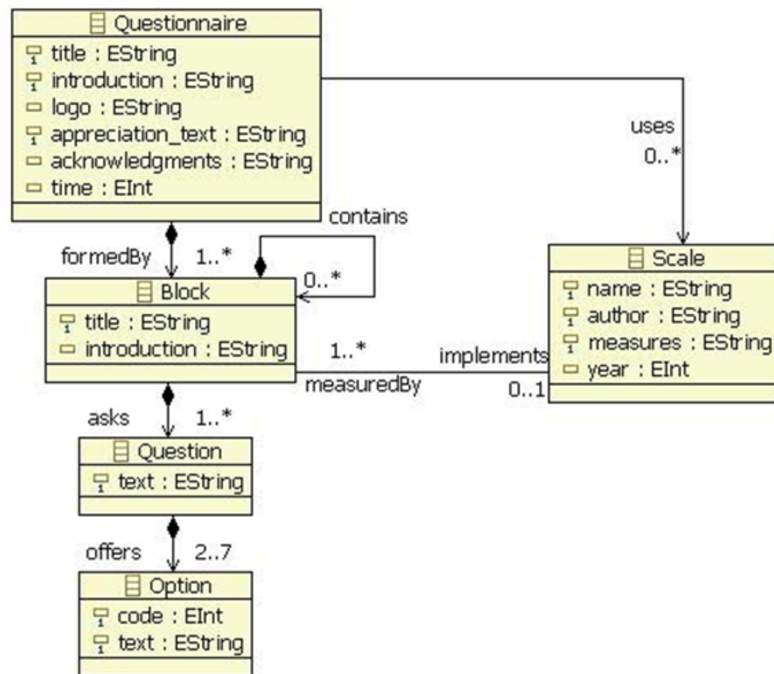


Figure 4.3: Questionnaire Metamodel

Deltas do not necessarily satisfy all constraints of MM. In the `Question` metaclass, for example, each question must have at least two options (there would not be a real question otherwise). A model delta can contribute with just one or no option at all. This is the case for the *Minor* feature (see Figure 4.4), where question `atq2` has but one option. Once this feature is composed with the base, the result has four options, which is conformant with the questionnaire metamodel. Hence, a DMM is similar to its corresponding MM but without some of its constraints.

However, not every constraint in the metamodel should be removed. Consider the `Question` metaclass which requires every question to have at most seven options. If any model delta adds more than seven options, every questionnaire created using that delta will be non-conformant. Thus, this upper-bound constraint should be fulfilled by every delta.

Three types of constraints are encountered when defining delta metamodels:

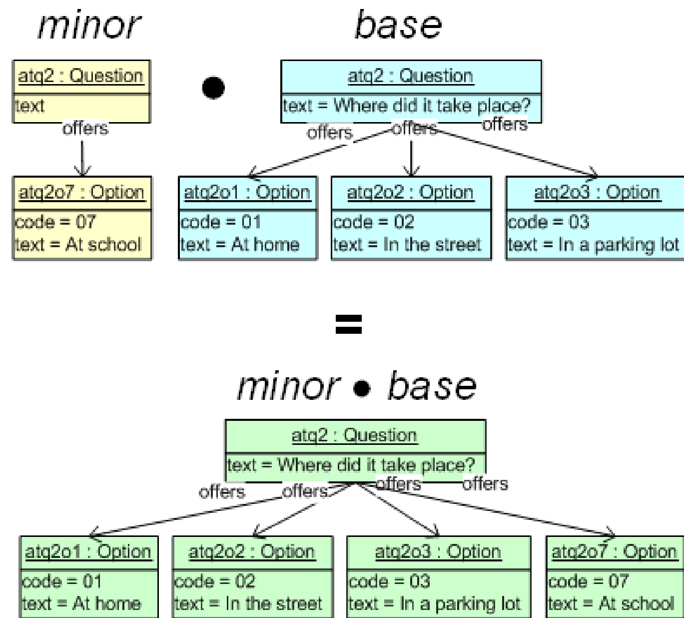


Figure 4.4: Constraint Violation by the *Minor* Feature

- *Delta Constraints.* Constraints that can be validated for every delta (e.g., upper-bounds, `option codes` have to follow a certain format, etc.). These constraints are kept in the DMM.

- *Composition Constraints.* Constraints that can be evaluated when two deltas are composed (e.g., upper-bounds, two deltas with four questions each would fullfil the constraint separately but their composition would be non-conformant).

- *Deferred Constraints:* Constraints that can only be evaluated after *all* deltas are composed, i.e., when the entire product model has been assembled (e.g., lower-bounds)[3].

[3]This has a database counterpart. The database administrator defines a database schema with constraints. Additionally, the administrator can declare a constraint to be validated as soon as a database update happens (immediate mode) or wait till the end of the transaction, once all updates were conducted (deferred mode). The deferred mode

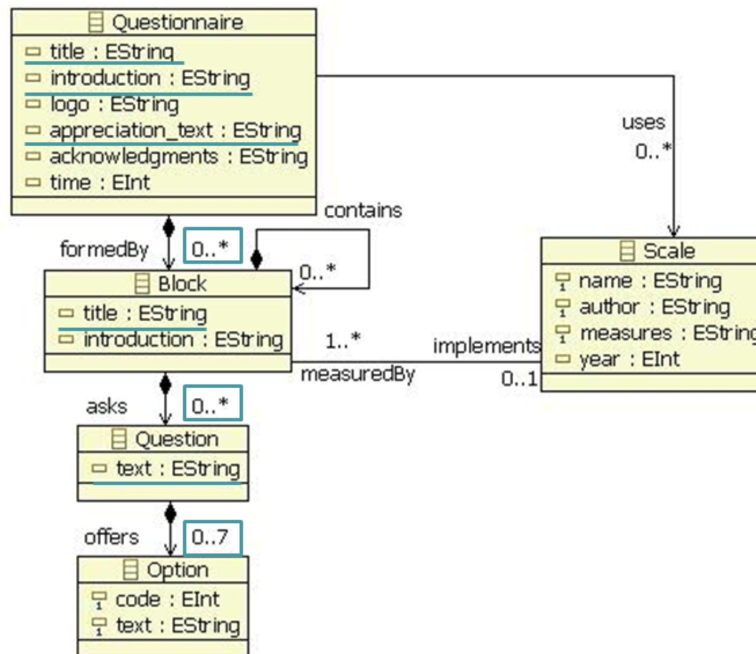


Figure 4.5: Delta Questionnaire Metamodel

Domain engineers should decide which group each constraint belongs to. A DMM can also be automatically generated from the product metamodel simply by removing all the constraints ot the metamodel except upper-bounds, leaving all constraints to be evaluated at the end. However, the first option is preferrable as it detects more inconsistencies earlier in the process (i.e., at delta creation time or at delta composition time). We derived the DMM for the Questionnaire MM by removing eight constraints (see Figure 4.5). An example of such deleted constraints can be found in the `title` attribute of the `Questionnaire` metaclass, which was previously required. Although once all deltas have been composed the result should have title, this constraint was removed because it suffices if just one of the deltas contributes with it, not all of them need to provide it. The same holds for the rest of attributes that are marked in Figure 4.5, they are

allows for some constraints to be violated during a transaction execution, as long as the constraints hold at the end of the transaction.

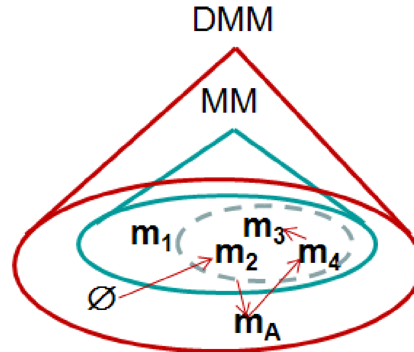


Figure 4.6: Delta Metamodel Cone

no longer required. Regarding references, the reason why lower-bounds are deleted is described above.

Figure 4.6 shows the relationship between the cones of MM and DMM. Each model m_i in MM is equivalent to an arrow $\emptyset \rightarrow m_i$ (i.e., a model delta that gathers the changes from the empty model to m_i and that conforms to DMM). This means that the cone of DMM not only includes every model in the cone of MM, but other models and deltas as well (e.g., models $\emptyset$ and m_A, and deltas that realize $m_A \rightarrow m_4$ and composite $\emptyset \rightarrow m_A$). The close relationship between DMM and MM means that arrows are defined using the same constructs as models. However, to define arrows as model deltas entails that an algorithm that composes them together to yield a product is required. This is the topic of the next section.

4.4 Delta Composition

Model composition has been defined as the operation M_{AB} = Compose (M_A, M_B, C_{AB}) that takes two models M_A, M_B and a correspondence model C_{AB} between them as input, and combines their elements into a new output model M_{AB} [BBF+06]. Delta composition is a special case, where both M_A and M_B conform to the *same* metamodel (i.e., the delta

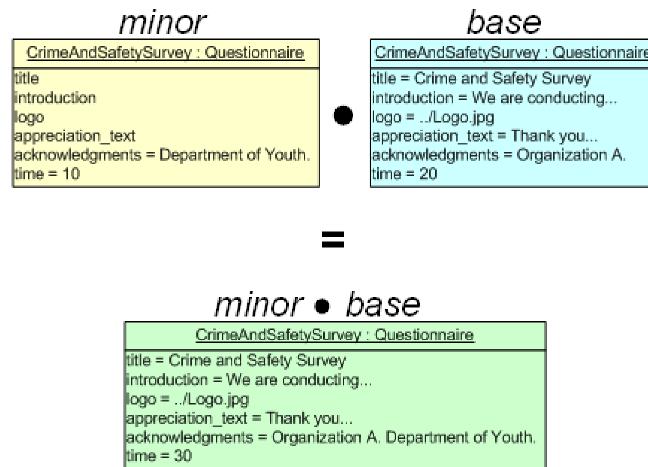


Figure 4.7: Composition of Minor and Base Features

metamodel). Further, the correspondence model C_{AB} is implicit as objects with the same name (or rather, identifier) in models M_A and M_B are, in fact, the same object (note that deltas belong to a product line and thus, have been designed with composition in mind). Delta composition is performed by pairing objects of different fragments with the same name and composing them.

There are two distinct cases in object composition:

- *Generic:* This is the default composition [AKL09, BSR04]. The composition of a pair of objects equals the composition of their corresponding contents. The composition of a pair of attributes is as follows: if both have different non-null values, composition fails and an error is raised. Otherwise, the non-null value is the result. In the case of references, their union is the result. Note that this composition is performed in the same way independent of the domain at hand; it is metamodel agnostic. Figure 4.7 presents different examples of this composition: `title`, `introduction`, `logo` and `appreciation_text` attributes, where the base gathers the content for all of them which is passed to the result, as the minor feature

has null values for those attributes.

- *Domain-Specific:* While in a majority of cases, generic composition is sufficient, the remaining cases require a domain-specific composition method. Consider the `acknowledgments` and `time` attributes in the `Questionnaire` metaclass. An organization can fund all the study or only part of it. Figure 4.7 shows how the *base* study is funded by *Organization A* and the part regarding *minors* is funded by the *Department of Youth*. If objects were composed generically, an error would be raised as both objects have a different value in the `acknowledgments` attribute. However, the convention for questionnaires is to concatenate both acknowledgments as the result.

  The `time` attribute is another example. It indicates the estimated time needed to complete the questionnaire. The *base* feature takes *20* minutes and the *minor* feature needs *10* more. The expected behavior would be to add both values to the result, not to raise an error. Therefore, the domain expert must customize the generic composition algorithm to account for domain-specific composition semantics.

  Domain-specific composition algorithms need not be limited to individual attributes as in the above examples; they apply to objects as well. A revealing example on object composition comes from the domain of software components. A software `component` implements or `provides` a set of `methods`, and references or `requires` another set of `methods`. Figure 4.8a shows a simplified metamodel. When two components are composed, the composite component `provides` the *union* of the methods of its constituent components, and `requires` the *union* of the references *minus* the references provided by the merged components (see Figure 4.8b). A special (i.e., non-generic) composition algorithm for `Component` objects is required: composed `requires` values are computed knowing `provides` values. There is another example, albeit less intu-

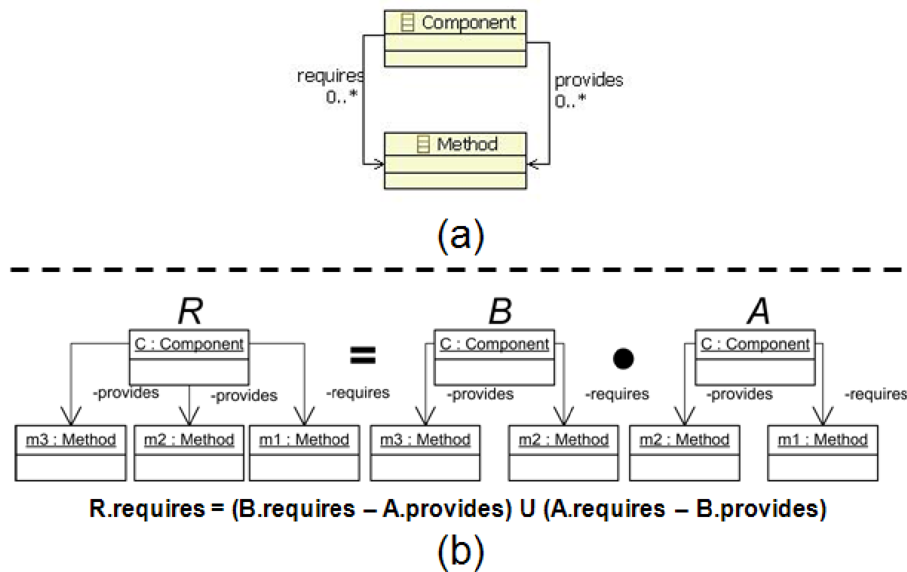


Figure 4.8: Component Metamodel and Component Composition Example

itive, in the *Questionnaire* domain. `Blocks` can be defined using a scale. When composing blocks with different scales, as results or each scale are measured separately, *Questionnaire* semantics dictate not to merge blocks but to keep them as separate sub-blocks.

We have described how arrows are defined and composed as model deltas using the *Crime and Safety Survey SPL* and *Questionnaires* as the running example. The next section describes how model deltas are defined in a general purpose modeling language.

4.5 Defining Deltas for UML Interaction Diagrams

Chapter 2 described that there are two main trends when choosing a metamodel to model a system, namely general purpose modeling languages and domain specific languages. So far, the ideas presented in this chapter have

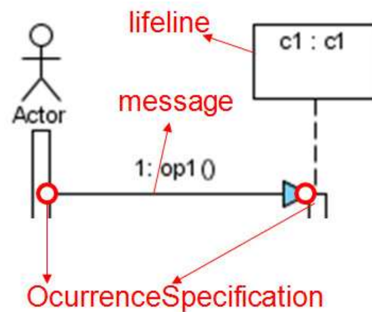


Figure 4.9: UML Interactions as Sequence Diagrams

been illustrated with an example of the latter. This section presents how the same ideas can also be applied to general purpose modeling languages. UML Interactions are used as the running example.

Interactions are a common mechanism for describing systems that can be understood and produced, at varying levels of detail, by both professionals of computer systems design, as well as potential end users and stakeholders [OMG09].

Along the OMG's metamodel for UML Interactions [OMG09], an interaction is a composition of *messages*. A *message* defines a particular communication between lifelines. A *lifeline* represents an individual participant in the interaction. A *message* then relates two happenings in, normally distinct, lifelines. These happenings are known as *OccurrenceSpecifications*. Note that *OccurrenceSpecifications* are ordered along a *lifeline*. A common graphical notation to depict models of this metamodel are *Sequence Diagrams*. Figure 4.9 shows a UML sequence diagram with its main constructs: a `message`, two `OccurrenceSpecifications` and a `lifeline`.

A main requirement when defining a product line of interaction diagrams is that current UML 2.0 compliant tools should be able to support our definition of interaction deltas. This also entails that MM (i.e., the UML Metamodel) and DMM (i.e., the UML Delta Metamodel) need to be the same, no constraints can be removed. The ability to use existing tools is a main issue to ensure practitioners will embrace the approach. The

question is twofold: *(i)* how interaction deltas differ from interactions and *(ii)* how they are composed together.

To illustrate these issues, we will consider a playing-board game product line, named the *Game Product Line (GamePL)*. These kind of games share a broad set of characteristics, such as the existence of a board, one or more players, the possible use of dice, the presence or absence of cards, policies related to the assignment of turns to the next player, etc.

Note that, similar to the *Questionnaire* example, interaction deltas use the same constructs as full interactions. Model deltas realize arrows that represent the features of an SPL. Hence, delta interactions are designed to be composed together and do not exist in isolation. Delta interactions define increments to base interactions and will lead to the complete product once they have been composed together. Message order needs to be mantained in interactions. Consequently, interaction deltas refer to an point on the base interaction that indicates the place in which the feature functionality is added. The point is indicated with a *message*, a construct present in the UML metamodel. In this way, the requirement of not changing the UML metamodel is fulfilled.

Figure 4.10 shows two UML interactions: *Move*, and *ThrowDice* which partially realizes the *Dice* optional feature, that gathers the necessary functionality for games that require dice (e.g., ludo or trivial). *ThrowDice* leverages *Move* with the ability of throwing dices. This enhancement is constrained to occur *(i)* after the player consults his turn and *(ii)* before moving. To denote which specific message is to be used as the extension point, we resort to the notion of *gate*. It is a representative of an *OccurrenceSpecification* that is not in the same scope as the *gate* [OMG09]. A *gate* is a connection point for locating a message outside an interaction with respect to a *message* inside the interaction. Figure 4.10 shows such a gate for the *ThrowDice* delta. The gate indicates that the delta is to occur just above the namesake message in the *Move* interaction.

The current UML 2.0 compliant tools support the use of *gates*. Specif-

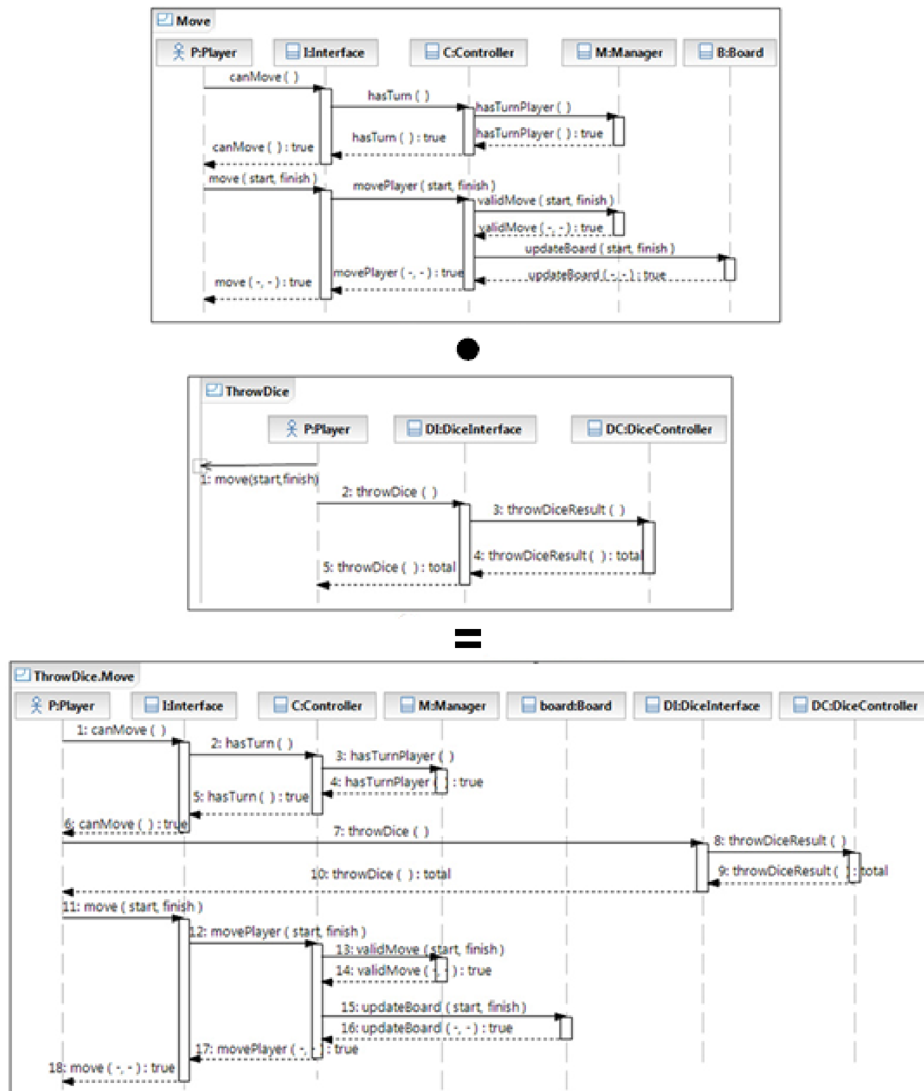


Figure 4.10: *ThrowDice* • *Move* Interaction Composition

ically, IBM Rational Software Modeler[4] was used to obtain the diagrams in Figure 4.10. As a consequence of using *gates*, model deltas can be realized using the same tools as complete interactions, thus meeting our requirement.

Regarding how interaction deltas are composed, generic composition is sufficient for most objects of interaction deltas. Deltas are composed together by identifying the message that is linked to a gate as the extension point, finding the message with the same name, and appending the feature just above the namesake message.

However, a domain-specific composition is necessary in some cases. *Gates* are a case in point. The generic behavior states that every object should be copied to the result. Message occurrence specifications linked to gates, however, should not be passed to the result (as gates are only used as the means to indicate where should the feature functionality be added). Hence, a domain specific behavior for gates has to be defined (more details of the exact composition semantics can be found in Chapter 5).

Previous sections showed how arrows can be realized as model deltas, the metamodel these deltas conform to, and how they are composed to yield a product of the SPL. An important aspect that was already mentioned in Chapter 3 is that one of our goals is to guarantee safe composition of model deltas (i.e., that the resulting product of composing deltas conforms to MM). Up to now, we only considered the constraints that are kept in the delta metamodel. How incremental consistency management can be used to check composition constraints (see Section 4.3) is described next.

4.6 Incremental Consistency Management in Delta Composition

This section details how incremental consistency management can be used for checking composition constraints in delta composition. We begin with

[4] *http://www-01.ibm.com/software/awdtools/modeler/swmodeler/*

a brief description of incremental consistency management.

Consistency checking is based on the work on *Multi-View Modeling (MVM)* [FKN$^+$92]. An extensive body of research in consistency checking exists [LMÁ09, UNKC08]. Different works typically have in common that consistency is expressed via rules. A recent trend in consistency checking is the work on incremental approaches which react to changes and evaluate only those rules on those model elements that are affected and can potentially cause an inconsistency. An advantage of these approaches is a reduced verification time over systems that follow a batch strategy. A leading tool among the incremental approaches is UML/Analyzer [Egy06, Egy07]. In this tool, when a model change occurs, it automatically, correctly and efficiently identifies what consistency rules to evaluate and on what model elements. If inconsistencies are detected, they are highlighted for the user to take an appropriate corrective action.

Incremental consistency in UML/Analyzer works as follows. First the tool loads the model to analyze. Then it identifies the places where each consistency rule can be applied. A consistency rule instance is an application of a consistency rule, and its scope is the set of model elements that are part of the instance. The work of UML/Analyzer has been mostly used in the context of UML models; however, its underlying principles are applicable to any types of models and constraints,

As an example of how incremental consistency checking with UML/Analyzer can be used to check the composition constraints defined in Section 4.3, recall the example we used when describing such constraints. In the *Questionnaire* domain, two deltas with four questions each would conform to the delta metamodel, as each contains less than seven options. However, their composition would be non-conformant.

In this setting, a consistency rule instance checks if a question contains more than seven options and returns a boolean result, true if the rule holds or false otherwise. It evaluates to true before composition, as both deltas fulfill the constraint. We have already mentioned that a delta can be seen

as a model that gathers all the changes a feature makes. As composition is performed, a rule is re-evaluated if a change in its scope elements is detected. In this case, the second delta adding four new options causes a re-evaluation of the rule, which detects a violation because the question now has eight available options. It is important to notice that this violation is signaled as soon as the eighth option is added and the place where the violation occurred is marked. This immediate notification allows the developer to take any corrective actions deemed necessary.

4.7 Discussion

We conceive model construction as a gradual composition of model deltas that are expressed in the same language as the final model. Each delta realizes an increment in application functionality (i.e., a feature), and the functionality of the final application is the added functionality of its set of deltas. The benefits include:

- Certain domain constraints can be checked at delta building time (see Section 4.3), allowing earlier error detection. It paves the way to safe composition, the assurance that all products of the SPL conform to the domain metamodel.

- It separates what a feature adds from how it is added, thus making the composition algorithms reusable for all features that conform to the same metamodel.

- Declarativeness. Model deltas are easier to read and write than their transformation counterpart. Figure 4.2a shows the same *Minor* feature but now realized using a general-purpose transformation language, RubyTL [CMT06]. Even to someone accustomed to reading transformation rules, it takes some time to grasp information that the transformation adds to the base model.

- The Delta Metamodel (DMM) can be derived from the domain metamodel (e.g., *Questionnaire*) by removing constraints.
- Existing work in incremental consistency management can be used to check constraints at composition time.

4.8 Related Work

Arrow Realization. Two main trends to realize arrows can be identified in *MDE*: transformation languages vs. model deltas. Examples of the former include C-SAW [BGL+06], MATA [WJE+09] and VML* [ZSS+09]. A transformation language is more versatile and it performs analysis that are presently outside this work. For example, MATA comes with support to automatically detect interactions between arrows. Since arrows in MATA are graph rules, the technique of critical pair analysis can be used to detect dependencies and conflicts between rules[5]. This versatility, however, comes at a cost: *(i)* it requires developers to be familiar with a graph transformation language and *(ii)* being arrows defined in a different language to the one that defines models, it reduces the amount of checking with respect to the domain metamodel (i.e., transformations conform to the transformation language metamodel, which in principle has no link with the domain metamodel). To ease the developers burden, MATA defines transformations between UML and the underlying graph rules. Nevertheless, these transformations must be defined for every metamodel at hand.

Traditionally SPL arrows are defined as model deltas — a set of changes — that are superimposed on existing models. Recently, several researchers have followed this trend, particularly using aspects as the implementation technique [AJTK09, MKBJ08, MBJ08, VG07]. To the best of our knowledge, none of these approaches defines a mechanism to check the conformance of deltas to their corresponding delta metamodels. Interestingly, SmartAdapters [MBJ08] defines pointcuts as model snippets that conform

[5] It may be possible to define a corresponding analysis on deltas.

to a metamodel that is obtained by eliminating all constraints from the domain metamodel, in a similar fashion to the way we define arrow metamodels [RBJ07]. However, no mention is made about an advice metamodel.

Along with aspects, collaborations are another candidate paradigm for realizing model deltas. Collaborations are monotonic extensions of models that encode role-based designs and are composed by superimposition [VN96]; collaborations are a centerpiece in recent programming languages (e.g., Scala [OAC+06]) and prior work on feature-based program development [BSR04]. In contrast, aspects may offer a more general approach to express model deltas, where pointcuts identify one or more targets for rewriting (a.k.a. advising). However, the generality of aspects is by no means free: it comes at a cost of increased complexity in specifying, maintaining, and understanding concerns [ALS08].

Model Differences. Model deltas are closely related to the idea of model differences [CRP07, RV08]. The main difference stems from their purpose, the former implement a feature in an SPL and are normally built separately while the latter are calculated as the difference between two models.

Model Composition. Model composition in general has been subject of extensive research [Ber03, BK06, BCRL07, PB03] and there are numerous tools that support this operation [AMW, Eps, Ker]. This work explores the specificities of this operation in an SPL setting.

Incremental Consistency Checking. There is a considerable amount of research in consistency checking. Recent literature surveys identified over 30 approaches which rely on different formalisms to represent and validate consistency [LMÁ09, UNKC08]. We leveraged on this work to check consistency during model delta composition.

4.9 Conclusions

MDE conceives software development as transformation chains where models are the artifacts to be transformed. We presented an approach to the

feature oriented development of models in an SPL setting. Models are created incrementally by progressively applying arrows that add increments in functionality. We implemented the arrows as model deltas. We explained how model deltas conform to delta metamodels, and how delta metamodels are derivable from domain metamodels. The focus of our work stressed the need for domain-specific composition algorithms. Implementing arrows as model deltas permitted us to use incremental consistency management to check constraints at delta composition time, thus paving the way to safe composition in model driven product lines.

Parts of the work described in this chapter have been previously presented:

- Maider Azanza, Don Batory, Oscar Díaz, and Salvador Trujillo. Domain-Specific Composition of Model Deltas. In *3rd International Conference on Model Transformations (ICMT 2010)*, Malaga, Spain, 2010.

- Roberto Lopez-Herrejon, Alexander Egyed, Salvador Trujillo, Josune de Sosa, and Maider Azanza. Using Incremental Consistency Management for Conformance Checking in Feature-Oriented Model-Driven Engineering. In *4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010)*, Linz, Austria, 2010.

Chapter 5

Increasing Reuse in Model Delta Composition

"By endurance, we overcome."
– *Ernest Shackleton*

5.1 Overview

Reuse is the main driving force behind *MDE* and *SPLE*. The previous chapter described how their combination can be achieved following a feature oriented approach. Arrows were realized using model deltas that encompass the additions a features makes to a model. This chapter addresses how model delta composition is realized. Metamodel annotations are presented as the means to specify domain-specific composition. In this way, the composition implementation is automatically generated from the metamodel. Annotations not only permit the composition implementation to be reused in different domains (i.e., the same annotation can be reused in different metamodels), but also shield the implementation details from developers, thus lowering the entry barrier for domain experts. Previous test cases (i.e.,

```
rule MatchQuestionnaire
 match l:Left!Questionnaire
 with r:Right!Questionnaire {
 compare:
  l.title=r.title
}
```

Figure 5.1: Example of Match Rule in ECL

Questionnaire and UML Interaction Diagram domains) are used as running examples.

5.2 Realizing Model Composition

We begin by providing some context on general model composition. Model composition can be decomposed into four phases: matching, conformance checking, merging, and reconciliation [KPP06a, PB03]. We use Epsilon, a family of consistent and interoperable task-specific programming languages that can be used to perform common *MDE* tasks [Eps], to realize this operation.

- *Matching*. Match is an operation $C = Match(M_1..M_n)$ that takes a set of models $M_1..M_n$ as input, searches for equivalences between their elements and produces a correspondence model *C* as output [Ber03, BBF$^+$06]. In Epsilon, matching is performed via *match-rules*. Each match-rule compares pairs of instances of two specific metaclasses and decides if they match. Match rules are implemented using the *Epsilon Comparison Language (ECL)*, a language that enables users to specify comparison algorithms in a rule-based manner. These rules identify pairs of matching elements between two models of potentially different metamodels and modeling technologies [KPP06b]. Figure 5.1 presents an example. The *MatchQuestionnaire* rule takes elements `l` and `r` as input, taken from input models `Left` and `Right` respectively, being both instances of the

```
rule MergeQuestionnaire
 merge l:Left!Questionnaire
 with r:Right!Questionnaire
 into t:Target!Questionnaire {
 t.title:=l.title;
 t.introduction:=r.introduction;
 t.logo:=l.logo;
 t.appreciation_text:=r.appreciation_text;
 t.acknowledgments:=t. acknowledgments;
 t.time:=r.time
}
```

Figure 5.2: Example of Merge Rule in EML

`Questionnaire` metaclass. It compares the titles of both questionnaires, returning true if they are the same and false otherwise.

- *Conformance Checking.* In this phase, elements that have been identified as matching in the previous phase are examined for conformance with each other. The purpose of this phase is to identify potential conflicts that would render merging infeasible [KPP06a].

- *Merging.* The goal of this phase is to merge to models based on the correspondance model [PB03]. It performs the actual composition using *Epsilon Merging Language (EML),* a rule-based language with tool support for merging models of diverse metamodels and technologies [KPP06a]. In Epsilon this phase takes two models and the correspondence model created in the match phase as input, and combines their elements into a new output model.

  There are two activities that produce elements in the target model: the model elements that have been identified as matching in the previous phase are merged into objects in the target model and the remaining model elements, for which no match has been found, are simply copied to the target model. This merging and copying are implemented using EML rules [KPP06a] and *Epsilon Transformation Language (ETL)* rules [KPP08] respectively. Figure 5.2 presents an

example of an EML rule. The *MergeQuestionnaire* rule takes questionnaires `l` and `r` as input and creates questionnaire `t` as output. The attributes of `t` are created from the corresponding ones in `l` and `r` alternatively.

- *Reconciliation.* After the merging phase, the target model may contain inconsistencies that need fixing. In the final step of the process, such inconsistencies are removed and the model is polished to acquire its final form.

However, EML, ECL and ETL are general purpose model management languages. As stated in Chapter 4, model delta composition is a special case of model composition, where both M_A and M_B conform to the *same* metamodel. Further, the correspondence model C_{AB} is implicit as objects with the same name (or rather, identifier) in models M_A and M_B are, in fact, the same object. The following section describes how delta composition is realized.

5.3 Realizing Generic Delta Composition

We can leverage on the particularities of delta composition to ease its implementation. We have already established that in most cases deltas are composed using *generic composition*, which is metamodel agnostic. Hence, instead of manually implementing such composition for every domain at hand, its implementation can be automatically generated from the domain metamodel, thus increasing reuse.

The `Object` metaclass defines the core metamodel of our implementaion of *ANDROMEDA (ANnotation-DRiven Oid-based coMposEr for moDel deltAs)* (see Figure 5.3a). It encompasses the generic behavior that is metamodel agnostic. `Object` instances are singled out by an explicit identifier that is used for object matching. `Object` holds the generic methods for `match`, `merge`, and `copy`.

The implementation of these generic methods is provided by AN-

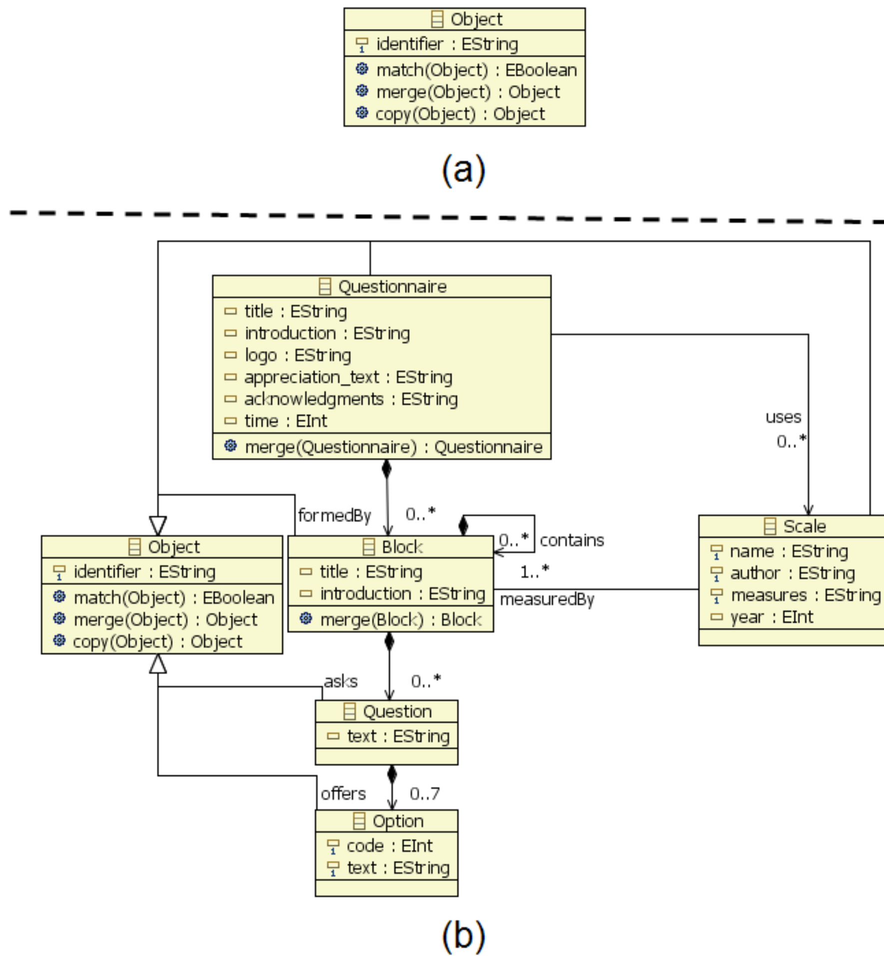


Figure 5.3: Extended Questionnaire Metamodel

```
@greedy
rule MatchObject
 match l:Left!Object
 with r:Right!Object {
 compare:
  l.class=r.class
      and
  l.identifier=r.identifier
}
                                        a
```

```
rule MergeQuestion
 merge l:Left!Question
 with r:Right!Question
 into t:Target!Question
 extends MergeObject {
 t.text:=l.text.
 defaultAttributeMerge
 ('text',r.text);
 --Code Omitted
}
                                        c
```

```
@abstract
rule MergeObject
 merge l:Left!Object
 with r:Right!Object
 into t:Target!Object {
  t.identifier:=r.identifier;
}
operation Any defaultAttributeMerge(att,obj):Any{...}
operation Any defaultBoundedReferenceMerge(ref,obj):Any{...}
operation Set defaultUnboundedReferenceMerge(set):Set{...}
                                        b
```

Figure 5.4: Generic Composition Implementation

DROMEDA. The `match` method is realized through an ECL rule (see Figure 5.4a). This rule indicates that two `Objects` match if they are instances of the same class and have the same identifier. The `merge` method is supported through an EML rule (see Figure 5.4b). The rules apply to all Object instances. Methods that define default merge behavior for attributes and references are also provided.

When implementing composition for model deltas of a given domain (e.g., questionnaires), the above rules and methods can be called, thus making it reusable. This holds whenever a metaclass can be composed using generic composition. Figure 5.4c provides an example, where the `MergeQuestion` rule extends the generic `MergeObject` rule for the *Question* metaclass. Note that the rule simply calls `defaultAttributeMerge` when composing the `text` attribute. The same would occur for any other attribute or reference, if there were any. In this way, generic rules can be automatically generated from the domain metamodel using a

transformation that outputs the composition implementation[1]. This implementation only needs to call the corresponding methods for attributes and references.

We described how generic composition can be automatically generated from the domain metamodel. However, generic composition is not sufficient in certain cases and a domain-specific composition needs to be defined. Next section delves into the details of how it is realized.

5.4 Realizing Domain-Specific Delta Composition

The previous chapter defined delta composition and motivated the need for domain-specific composition in certain cases. The `time` attribute is a case in point. This attribute indicates the estimated time needed to complete the questionnaire. As example, the *base* feature takes *20* minutes and the *minor* feature needs *10* more. The expected behavior would be to add both values to the result, not to raise an error as the generic composition would. Therefore, the domain expert must customize the generic composition algorithm to account for this domain-specific composition semantics.

To realize such domain-specific composition semantics, the generic behavior can be specialized to cater for the composition peculiarities of the domain at hand. To this end, `Object` specializations are created for each domain metaclass (see Figure 5.3). Note how `Questionnaire` and `Block` override the generic merge method to address their own specific composition semantics.

Figure 5.5 presents the domain-specific EML rules that extend the generic EML rules (rule extension is the counterpart of method inheritance). Figure 5.5a shows the rule that merges `Questionnaires`. Here, the generic merge of `acknowledgments` and `time` attributes is overridden by string

[1]Actually, rules for `copy` and `compose` are generated in separate files, e.g., *DomainMerge.eml* (which contains all the merge rules) and *DomainCopy.etl* (with all the copy rules).

```
rule MergeQuestionnaire
 merge l:Left!Questionnaire
 with r:Right!Questionnaire
 into t:Target!Questionnaire
 extends MergeObject {
 t.title:=l.title.
 defaultAttributeMerge
 ('title',r.title);
 --Code Omitted
 t.acknowledgments:=
  l.acknowledgments+
  r.acknowledgments;
 t.time:=l.time+r.time;
}                              a
```

```
rule MergeBlock
 merge l:Left!Block
 with r:Right!Block
 into t:Target!Block
 extends MergeObject {
 var lsb: new Target!Block;
 var rsb: new Target!Block;
 lsb.title:=l.title;
 lsb.implements:=l.implements;
 rsb.title:=r.title;
 rsb.implements:=r.implements;
 --Code Omitted
}                              b
```

Figure 5.5: Domain-Specific Composition Implementation

concatenation and integer addition respectively. Regarding `Blocks`, they can be defined using a scale. As results from a each scale are measured separately, when composing blocks with different scales, they are not merged but are kept as separate sub-blocks instead[2] (see Figure 5.5b).

A first approximation would be to write this domain-specific rules manually. Nevertheless, it requires domain experts to be knowledgeable with EML. A more practical approach is to allow domain engineers to annotate any class or attribute to indicate that a domain-specific composition algorithm, rather than the generic algorithm, is to be used. Additionally, since some algorithms are likely to be used in different domains (e.g., string concatenation), we provide a set of keywords to denote those recurrent algorithms as annotations on the metamodel. These annotations include: `@concat` (i.e., given two values V1 and V2, the composition delivers V1V2), `@slash_concat` (i.e., the composition delivers V1/V2), `@sum` (i.e., the composition delivers the addition of V1 and V2), `@min` (minimum value), and `@max` (maximum).

Figure 5.6 depicts the annotated *Questionnaire* metamodel. Note that the `acknowledgments` attribute in *Questionnaire* and the `introduction` attribute in *Block* are annotated with `@concat`. This indicates that

[2]Note that this domain specific composition is not limited to individual attributes as in the above examples; it applies to complete *Block* objects.

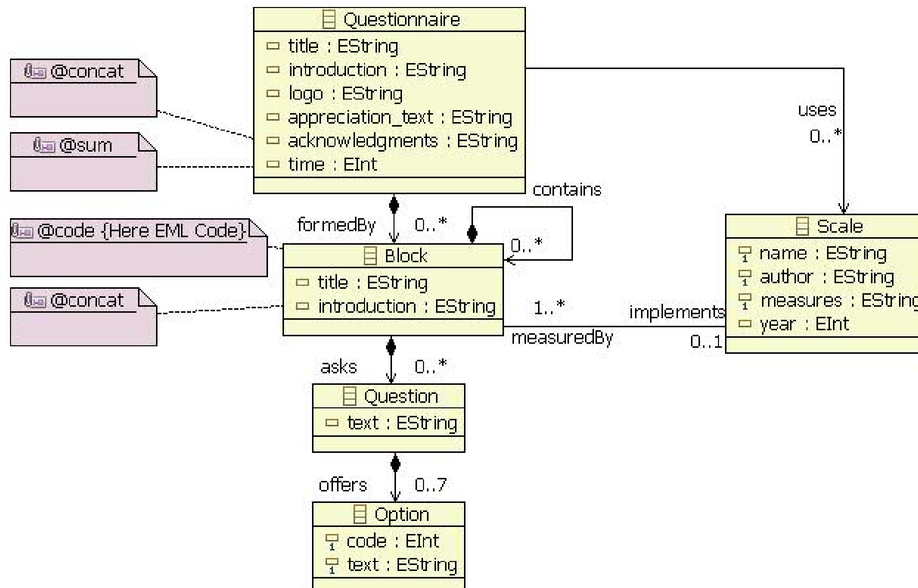


Figure 5.6: Annotated Questionnaire Metamodel

their values are to be composed by concatenation. Moreover, the `time` attribute in *Questionnaire* is annotated with `@sum`, meaning that composed contents should be added. In this case, domain-specific EML rules that override realize these semantics are automatically generated using the same transformation that also generates the generic rules (recall Figure 5.5a).

When these built-in annotations are not sufficient, engineers must resort to describing domain-specific composition algorithms procedurally using EML. Blocks are a case in point. We specify these algorithms as a piece of EML code inserted inside an annotation that is attached to the *block* class[3] (see Figure 5.6). In this case, when generating the rule, the code embedded in the annotation is directly copied into the EML rule (see Figure 5.5b).

[3]This option is debatable. Another possibility is to embed the ad-hoc algorithm directly into the generated code. We prefer to have all composition algorithms specified in a single place: the metamodel. In our opinion, this facilitates understandability and maintainability.

We have described how annotations can be used in a metamodel to indicate that a domain-specific composition algorithm is needed. This metamodel is the input for a model-to-text transformation, named *t_AMM2Comp,* and implemented using the *Epsilon Generation Language (EGL)* [RPKP08]. When this transformation is enacted, the composition implementation is generated.

Initially, the implementation that corresponds to each annotation was hard-coded in the transformation. The approach works but requires domain experts to be knowledgeable with EML when the annotations above are not sufficient. Moreover, it prevents new annotations from being created, thus limiting reuse. Next section addresses a model for annotations that improves the declarativeness (and reuse) of the solution and in so doing, facilitates domain experts themselves to define the composition semantics.

5.5 A Model for Composition Annotations

A domain-specific composition algorithm can be particular to a domain (e.g., block composition in *Questionnaires*) but it can also be reused in several domains (e.g., string concatenation). Our aim is to allow developers to define their own annotations and make them reusable in different domains. A first solution would be to hard-code the annotations in *t_AMM2Comp*. However, this implies that the set of annotations is either fixed, or the transformation implementation needs to be manually changed every time an annotation is added.

Hence, we opted for a different path. We defined an annotation model that contains the specification of each annotation (e.g., @concat, @sum) and its corresponding implementation. This model is the input of another transformation, named *t_an2t*. This transformation will then output the actual *t_AMM2Comp* that will generate the composition implementation from the annotated metamodel (see Figure 5.7).

Figure 5.8 presents the annotation metamodel. Each annotation model gathers a set of `Annotations`, where each one has a name, a specifi-

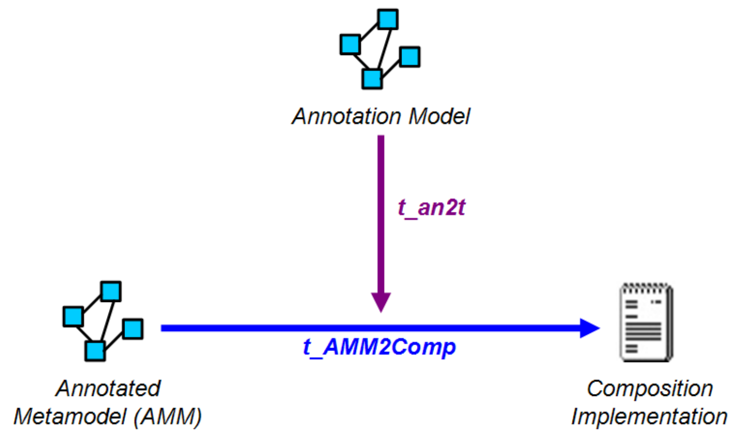


Figure 5.7: Obtaining Composition Implementation from the Annotation Model

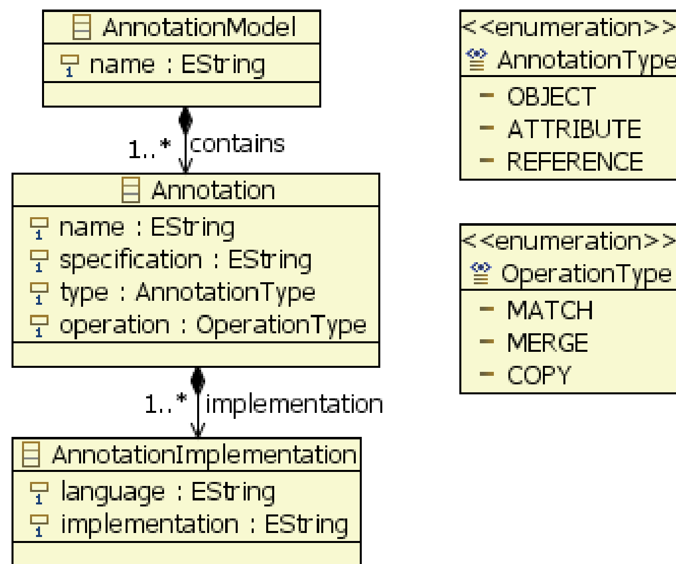


Figure 5.8: Annotation Metamodel

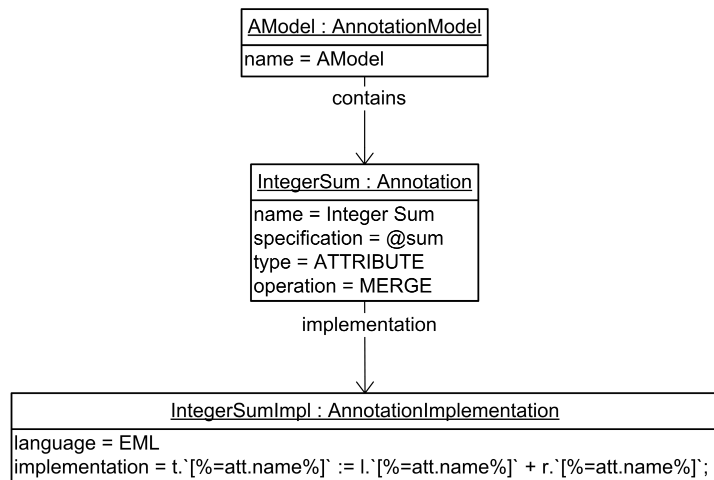


Figure 5.9: Annotation Model Example

cation, a type and an operation. The `specification` gives the name of the annotation itself (e.g., @*concat*, @*sum*...) and the `type` indicates whether the annotation is for an attribute, a reference, or a complete object (e.g., @*concat* is for attributes while the block composition in *Questionnaires* is domain-specific for the complete object). The `operation` details which method (i.e., match, merge or copy) is customized by the annotation.

Each annotation can have one or more `AnnotationImplementations`. They contain the `implementation` and the `language` such implementation is written in. Although ANDROMEDA currently only supports EML, this allows to generate distinct *t_AMM2Comp* transformations that output composition implementations written in different languages.

Figure 5.9 depicts an example of an annotation model. The integer sum annotation is an instance of the annotation metaclass that is applied to attributes and that overrides the generic merging behavior. It is specified using @*sum* and the corresponding code that will lead to an implementation in EML is indicated.

As in the above @*sum* example, all annotations in the *Questionnaire*

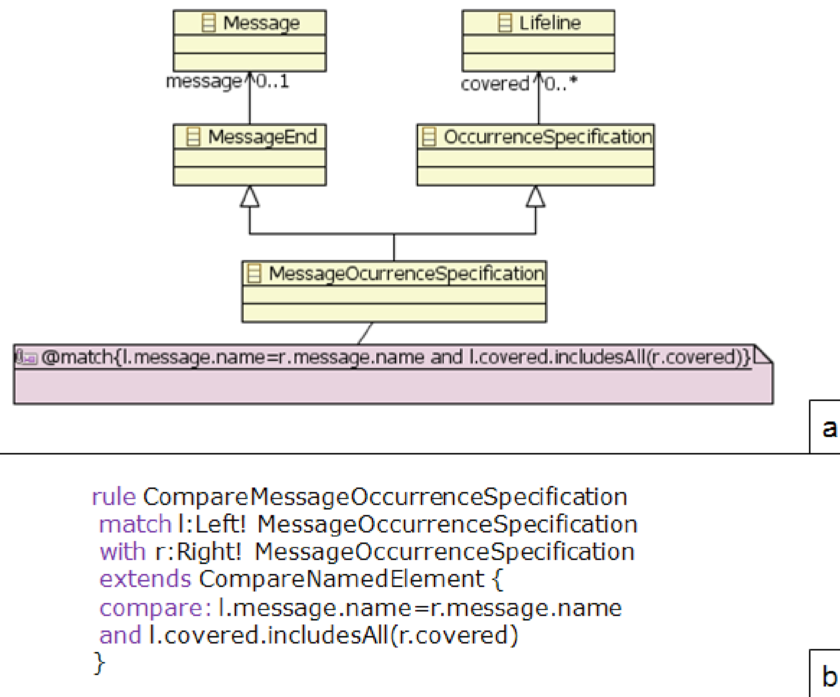


Figure 5.10: Match Customization Example

domain customize merge rules. We will now present examples of domain-specific match and copy in the context of the UML Interaction example. This example supports the need to have annotations that also those operations. As stated in Chapter 4, a main premise when composing UML Interaction deltas is to leave the UML metamodel untouched.

- *Match.* Not every object in UML is required to have a unique name. The previously mentioned *MessageOcurrenceSpecifications* are a case in point. We have defined their customized matching as follows: two message occurrence specifications are the same, if they both belong to the same message and are linked to the same lifeline (this permits to distinguish between the two occurrence specifications a message can have). Figure 5.10a presents a fragment of the UML metamodel with the corresponding annotation attached to the *Mes-*

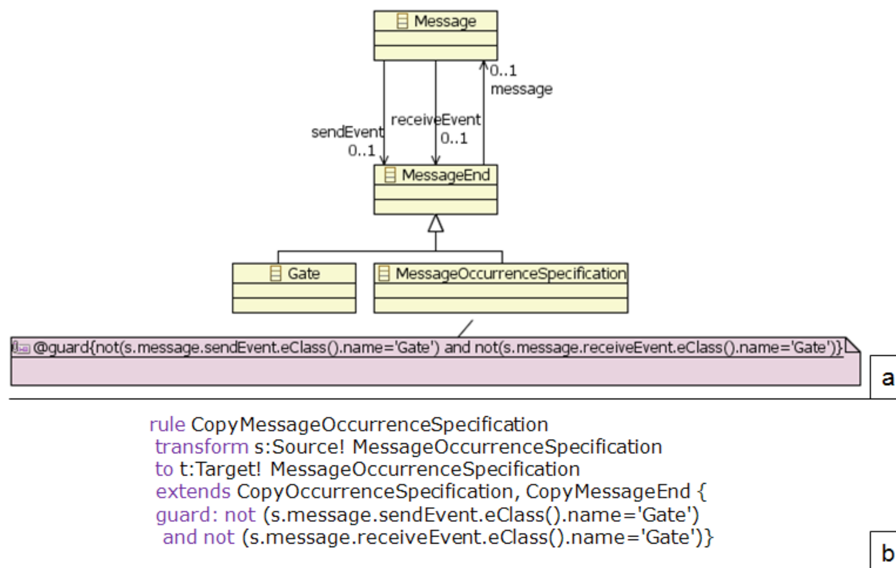


Figure 5.11: Copy Customization Example

sageOcurrenceSpecification metaclass. Figure 5.10b, in turn, shows the generated ECL rule. Note the new annotation (i.e., *@match*) that permits customization of the match method.

- *Copy.* As described in Chapter 4, the generic behavior dictates that every object for which no match is found should be copied to the result. *Gates* are case in point where this behavior should be overriden. *MessageOccurrenceSpecifications* that are linked to gates should not be passed to the result (as gates are only used as the means to indicate where the feature functionality should be added). Figure 5.11a pictures an excerpt of the UML metamodel, where *MessageOccurrenceSpecification* is annotated with a guard that indicates when instances of the metaclass should be passed to the result. Figure 5.11b presents the resulting ETL rule.

So far, we have been working in the *MDE Technical Space*. The following section describes how this work relates to SPLs defined in other technical spaces.

MM DATA	Jak	Qst
Classes	12	5
Attributes	18	15
References	25	7
Removed Constraints	0	8
Generic Comp.	37	23
D.S. Comp.	18	4

Table 5.1: Jak and Questionnaire Metamodels

5.6 Relating to Other Technical Spaces: Jak

Among the motivations for connecting arrows and product lines in Chapter 3, one was to underscore the generality the underlying ideas. This section describes how the FOSD paradigm is applied equally in different technical spaces.

We believe our previous case studies, namely the *Crime and Safety Survey Questionnaire SPL (CSSPL)* and the *Game Product Line (GPL)* are typical examples of basic *MDE* product lines, and we have explained how we handle them. But there are examples from other technical spaces [BK06] for which these same relationships can be demonstrated to hold.

The largest examples of feature-based compositions are written in Jak, a superset of Java that supports feature declarations, state machines, and metaprogramming [BSR04]. Hence, they come from the EBNF technical space [BK06]. With the aim of testing ANDROMEDA and comparing the results obtained in both technical spaces we defined a metamodel of the Jak language which exposes its main language constructs (e.g., feature, package, class, field, and method).

As Jak was specifically designed to implement product lines, no constraint had to be removed from the product metamodel to produce the delta metamodel: all constraints defined in the product metamodel were applicable to model deltas. Next, a composition strategy was chosen for each element in the Jak metamodel. The annotated Jak delta metamodel is shown in Figure 5.12 (the @code annotations are omitted to improve readibility).

Table 5.1 lists statistics about the *Jak Metamodel*, with statistics from the *Questionnaire metamodel (Qst)* for comparison purposes. The number

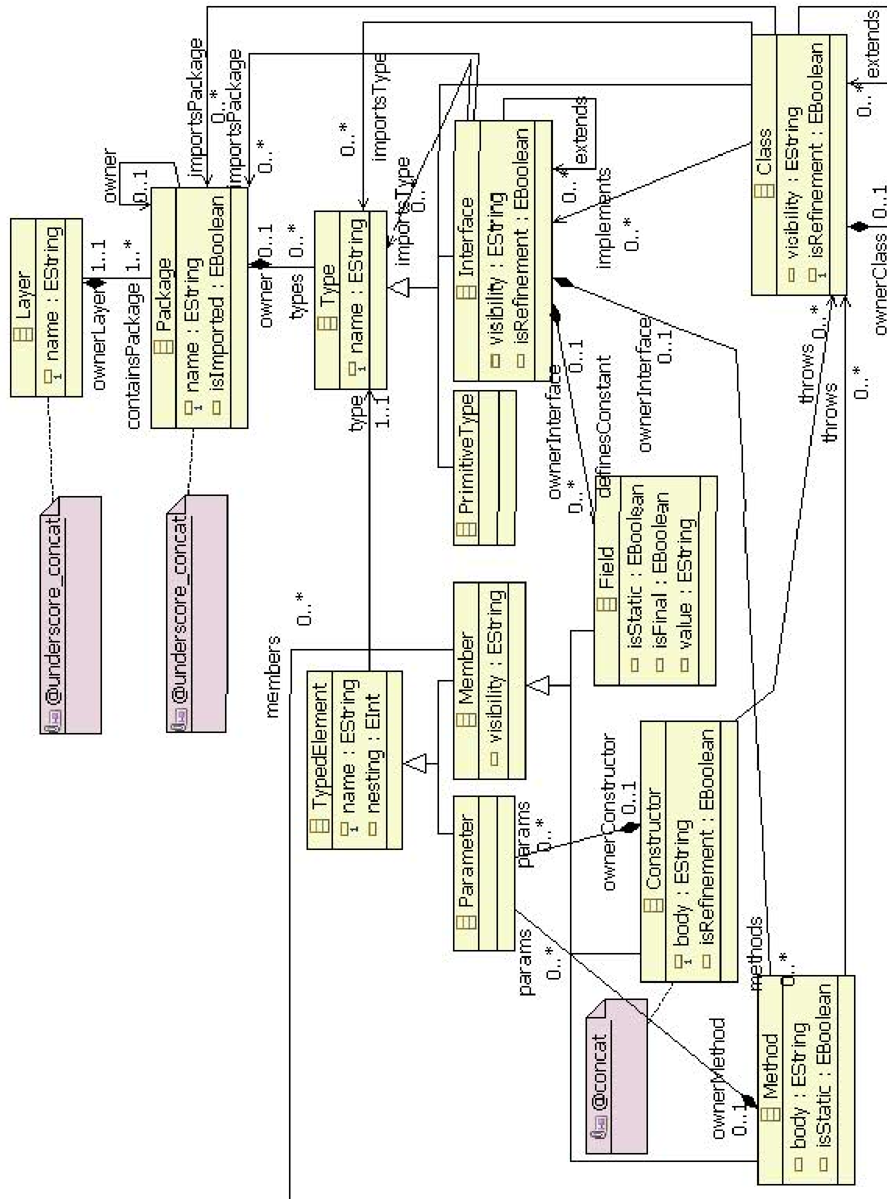


Figure 5.12: Annotated Jak Delta Metamodel

of classes, attributes and references of each metamodel and the distribution of their composition types are listed. Note that Jak uses a large number of domain-specific composition algorithms. It was interesting to note that several of these algorithms worked on objects and not attributes.

The *Graph Product Line (GPL)* and the *ATS Product Line (APL)*, which are two product lines written in Jak, were used as case studies. GPL is a family of classical graph applications [LHB01]. A graph is either `Directed` or `Undirected`. Edges can be `Weighted` with non-negative numbers or `Unweighted`. Every graph application requires at most one search algorithm (`BreadthFirst` or `DepthFirst`) and one or more graph algorithms (e.g., `VertexNumbering`, `CycleChecking`, `MinimumSpanningTree`).

A much larger example is the *AHEAD Tool Suite (ATS)*, a set of tools to support FOP [BSR04]. ATS was refactored into the *ATS Product Line (APL)*. That is, ATS bootstraps itself by composing features of APL [TBD06]. Over time, ATS has grown to 24 different tools comprising over 200K LOC Java. In addition to code, there are makefiles, regression tests, documentation, and program specifications, all of which are fragmented into features. In our case study, we focused only on the model representations of the code that realizes the features.

We created a tool that translates the APL or GPL features written in Jak (i.e., the EBNF technical space) into model deltas that conform to the Jak delta metamodel (i.e., to the Ecore technical space). Finally, we wrote a model-to-text transformation from Jak models back to code to verify the result after a composition was performed.

For illustrative purposes, Table 5.2 lists the features of GPL, their size in objects (a.k.a *Objs*), their references (a.k.a *Refs*), and these statistics for particular GPL programs that we composed.

In the same way, Table 5.3 compares the average size of CSSPL and APL features with their size in objects and references. The reason to present average values is that APL contains 96 features. It also presents the average size for particular products we composed. Note that APL model

GPL	Objs	Refs	GPL	Objs	Refs
base	25	53	MSTPrimPrepGR	31	88
Benchmark	24	60	Number	18	42
BFS	31	83	Prog	9	18
Connected	22	54	StronglyConnected	31	88
Cycle	34	90	Transpose	9	21
DFS	28	72	UndirectedGENR	78	221
DirectedGenR	70	197	UndirectedGNR	70	196
DirectedGnR	64	179	UndirectedGR	64	178
DirectedGR	59	164	WeightedGENR	20	47
MSTKruskal	16	43	WeightedGNR	35	91
MSTKruskalPrepGnR	27	70	WeightedGR	29	75
MSTKruskalPrepGR	37	100	Product 1	135	382
MSTPrim	21	55	Product 2	147	418
MSTPrimPrepGnR	32	88	**AVG**	**43,2**	**117,5**

Table 5.2: GPL Statistics

	Objs	Refs
CSSPL Features	22	21
APL Features	109	332
CSSPL Products	63	62
APL Products	3035	9394

Table 5.3: CSSPL and APL Statistics

deltas are on average four times larger than the arrows of CSSPL, and APL products are over fifty times larger than CSSPL products.

Again, one of the advantages of the Jak case studies was to demonstrate our composition principles hold across different technical spaces. Another advantage was that we could verify the correctness of our compositions. Model deltas were composed to yield a certain product and then transformed into code. The same product was obtained by directly composing its code features and both results were compared using source equivalence to test for equality[4].

5.7 Discussion

This chapter describes how the generic composition that was defined in Chapter 4 is realized. Additionally, domain-specific composition seman-

[4]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.

tics are captured through annotations in the domain metamodel. This improves declaractiveness. The benefits include:

- *Automatization*. The composition algorithm can be automatically generated. As a proof of concept, we showed the implementation that generates EML rules.
- *Understandability*: Anchoring composition annotations on the metamodel, permits designers to focus on the *what* rather than on *how* the composition is achieved.
- *Maintenance*: Additional composition algorithms can be added or removed with minimal effort. This could be of interest in the context of metamodel evolution where new metaclasses/attributes can be added that require customized composition algorithms [Wac07].

5.8 Related Work

There are a number of model delta composition tools, some using *Aspect Oriented Modeling (AOM)*, that are now available. Examples of such are XWeave [VG07] and Kompose [FBFG07]. However, they support only generic composition, although the latter has some limited support for domain specificities in the form of pre-merge and post-merge directives. Other approaches, e.g., SmartAdapters [MBJ08], VML* [ZSS+09] and FeatureHouse [AJTK09], provide support for domain-specific composition. The first two require engineers to explicitly indicate how deltas are to be composed, while we strive to promote reuse by automatically generating as much as possible from the metamodel using metamodel annotations. FeatureHouse also accounts for domain specificities. However, it is restricted to tree composition, and domain-specific composition is limited to tree leaves. By contrast, our approach considers graphs rather than trees and composition can be simultaneously specified at different points within models.

5.9 Conclusions

Realizing arrows (i.e., features) as model deltas entails that a composition algorithm for model deltas needs to be specified. In this chapter, we explained how the composition implementation can be automatically generated from the domain metamodel. Additionally, we described how the domain metamodel can be annotated to indicate composition semantics. Moreover, we presented a generalization of the annotation mechanism that permits developers to write their own annotations that can be reused in other domains. In this way, composition implementation details are almost shielded from developers, thus increasing reuse and lowering the entry barrier for domain experts.

Parts of this chapter have been previously presented:

- Maider Azanza, Don Batory, Oscar Díaz, and Salvador Trujillo. Domain-Specific Composition of Model Deltas. In *3rd International Conference on Model Transformations (ICMT 2010)*, Malaga, Spain, 2010.

Chapter 6

The Assembly Process in *MDPLE*

"You do not really understand something unless you can explain it to your grandmother."
– *Albert Einstein*

6.1 Overview

As presented in previous chapters, *Model Driven Product Line Engineering* promises to decrease the cost of software development through systematic reuse. *MDPLE*, nevertheless, reduces the cost of coding software at the expense of increasing assembling complexity, i.e., the process of coming up with the final end product. To alleviate this problem, this chapter advocates for a new discipline inside the general software development process, i.e., *Assembly Plan Management*, that permits to face complexity in assembly processes. A non-trivial case study is used for illustrative purposes.

6.2 Assembly Plan Management Overview

A main premise of *SPLE* is that the expense required to develop reusable artifacts during domain engineering is outweighted by the benefits obtained when deriving the individual products during application engineering [DSB04]. Consequently, most of the efforts are geared towards variability management whereby an infrastructure of core assets and variability mechanisms are prepared during domain engineering so that application engineers can come up with the right product in a cost-effective way.

The claim of this chapter is that the combined use of *MDE* and *SPLE* increases the burden of software development in general, and software assembly in particular (see Chapter 3). Therefore, additional effort should be dedicated to create an infrastructure (i.e., core assets) that facilitates assembly during application engineering. The effort to build such infrastructure will payoff by streamlining the assembly process.

Our aim is to automate the process of realizing the assembly process. Such automation is achieved through the so-called *Assembly Machine Tool*. In product manufacturing a machine tool is a powered mechanical device, typically used to fabricate metal components of machines. Machine tools that operate under automatic control are also known as computerized numerical control machines where the machine tool is fed with a program that dictates the process that constructs the desired item.

This notion of numerical control machine tool is here used to describe the assembly infrastructure. Such machine tool is realized through a library. The numerical control program is supported through an *Assembly Program*. This assembly program can, in turn, be specified using an *Assembly Equation*, which is a declarative specification that embodies a partially ordered set of transformations and model delta compositions. Given an assembly equation, the assembly program will be automatically generated. Enacting this assembly program will deliver the product (i.e., the set of *code* artifacts) that exhibit the desired features. This process is supported by the *GROVE Tool Suite (GROVE TS)*, which we implemented to

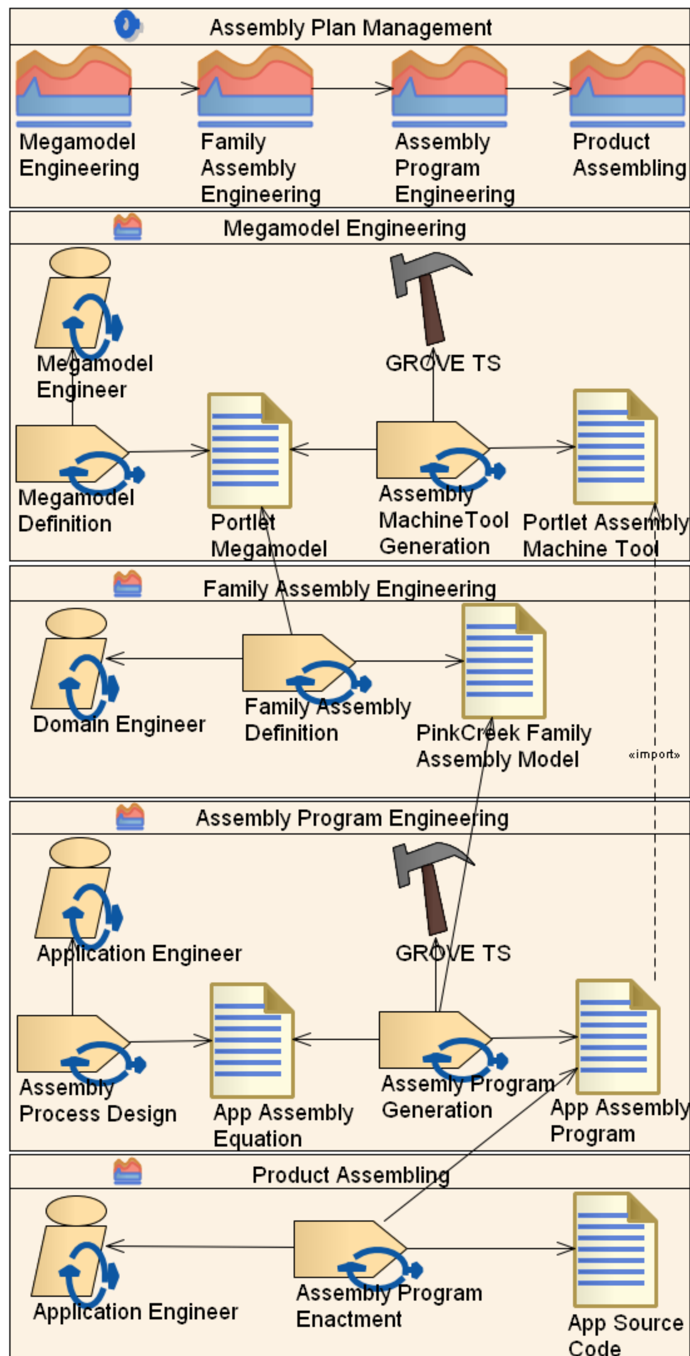


Figure 6.1: SPEM Diagram of the Assembly Plan Management Discipline

assist in the assembly plan management. To describe this process, we resort to the *Software Process Engineering Metamodel (SPEM)* [OMG08b], an initiative of the *Object Management Group (OMG)* for software process modeling. It is a methodology-independent language based on UML. Hereafter, SPEM terminology is used to specify the tasks, artifacts and roles that produce software.

According to SPEM a software development process is defined as a collaboration between abstract active entities called *process roles* that perform operations called *tasks* on concrete, tangible entities called *work products*. A *discipline* partitions tasks within a process according to a common theme. This work introduces a new discipline in the development of a model driven product line: the *Assembly Plan Management*, which splits along four phases: *Megamodel Engineering, Family Assembly Engineering, Assembly Program Engineering* and *Product Assembling*. Figure 6.1 outlines the different roles, work products and tasks supporting this endeavour.

Next sections delve into the details of each phase. For each of them the process is explained, the models that take part are described and the transformations that are used are specified. Each section ends with a description of how the phase is applied for the *PinkCreek* flight booking portlet case study (see Chapter 3 for a description).

6.3 Megamodel Engineering Phase

This phase sets the *conceptual framework* to support the model driven product line: which metamodels to use (e.g., statecharts vs. state-transition diagrams), how variability is supported (e.g., collaboration vs. aspects), which transformation language is to be used (e.g., QVT vs. ATL), which are the exogenous transformations that map between different metamodels and so on.

Starting from a megamodel, the goal of this phase is to generate the *Assembly Machine Tool*, i.e., a reusable library that provides operations

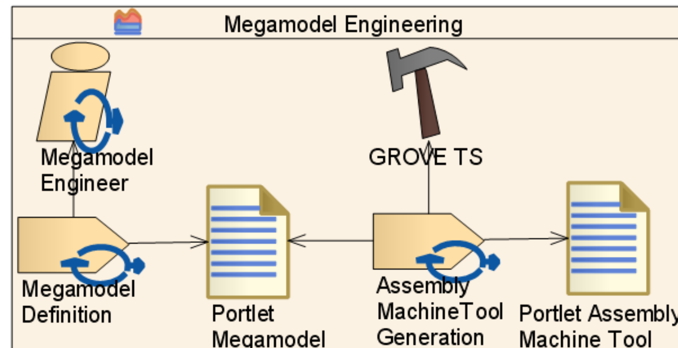


Figure 6.2: Megamodel Engineering Phase

for model transformations within the scope set by the megamodel. This assembly machine tool will be used later by assembly programs when a product is specified. Figure 6.2 depicts the SPEM diagram of this phase. Next, the process, models and transformations that take part in the phase are detailed.

6.3.1 Process

The *Megamodel Engineering Phase* is divided into two activities. First, the Megamodel Engineer defines the megamodel, which contains the set of metamodels and transformations defined for a certain domain [BJV04]. We denoted this activity as `Megamodel Definition`. The megamodel groups the set of transformations needed to obtain a product from its abstract specification, that is, it contains the set of steps needed to *assemble* a product from an input model.

Second, the GROVE TS performs the `Assembly Machine Tool Generation`. Taking the previously defined megamodel as input, our tool generates a reusable machine tool. Realized as a library, the machine tool contains the implementation able to enact the transformations that were defined in the megamodel. More to the point, it can be used by several product lines, as long as they are defined in the same domain (i.e.,

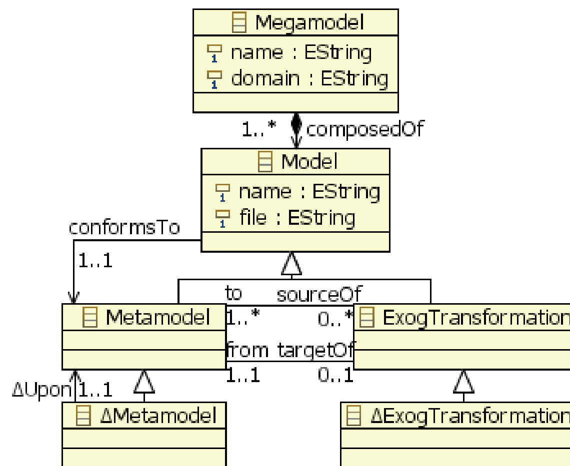


Figure 6.3: Assembly Megamodel Metamodel

they use the same metamodels and transformations). So, reuse is fostered by grouping together the code the transformations of a particular domain and making it available for all the product families defined in one particular domain.

6.3.2 Models

Two are the models introduced in this phase: the *Megamodel*, defined by the Megamodel Engineer and presented in Figure 6.2, and the *Assembly Machine Tool Model*, and intermediate model generated by the GROVE TS during the transformation process. The former is an abstract representation that gathers together the metamodels and the transformations between them defined in the domain. We transform this model into the `Assembly Machine Tool Model`, which acts as an intermediate model that is later transformed into the library code.

Figure 6.3 shows the *Assembly Megamodel Metamodel*. Megamodeling is the activity of modeling in the large, that is, of taking a global view on the considered artifacts (i.e., how models are related to each other, how they can be transformed into other models and so on), related to other pos-

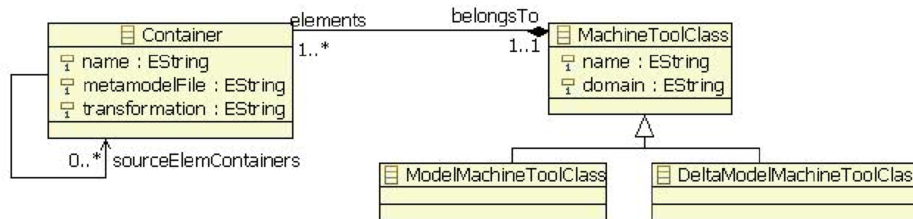


Figure 6.4: Assembly Machine Tool Metamodel

sible local views handled in various other contexts [BJV04]. In our case, the megamodel gives a global view on the metamodels and transformations defined for the domain, which will allow to perform program assembly later on.

Each `Megamodel` has two attributes, its `name` and the `domain` at hand. It contains a set of `Models`, as can be seen in Figure 6.3. For each model, its `name` and the path to its `file` are specified. This way, the assembly program will later be able to locate them. These models can be `Metamodels` or `Exogenous Transformations` (i.e., transformations where the input and the result models conform to different metamodels). These metamodels and transformations operate with complete model instances, model deltas are dealt with separately[1].

As described in Chapter 4, model deltas do not conform to the domain metamodel. Model deltas conform to `Delta Metamodels`. In the same way, `Delta Exogenous Transformations` are defined between two different delta metamodels. Therefore, delta exogenous transformations are transformations where the source and target metamodels are delta metamodels (see Figure 6.3).

The second model involved in this phase is the `Assembly Machine Tool Model` (Figure 6.4 presents the metamodel it conforms to). It aims at modeling the concepts that will permit to automatically gener-

[1]In order to simplify the prototype implementation, endogenous transformations do not appear in the megamodel. The premise is that AHEAD Tool Suite will be the transformation engine for such transformations [Bat]. This brings no lack of generality as other tools can be easily integrated with it.

ate the repetitive code that previously had to be written by hand. It defines a set of containers, where a `Container` is an element that encapsulates models of a certain type and is able to perform operations on them. For that purpose, each container defines its `name`, the `metamodelFile` that defines which type of instances it can store and the `(exogenous) transformation` from which models of that type are created, if applicable. That is, for every metamodel in the megamodel, a container is created, with the ability to transform models into instances of that metamodel.

As mentioned above, we are dealing with two types of models: complete models and model deltas. For each of them a separate `Machine Tool Class` is created, which will be a *Model Machine Tool Class* or a *Delta Model Machine Tool Class* depending on the type of models at hand. It contains its `name` and the `domain` it is defined on. A *Machine Tool Class* groups together the transformation sequence defined in the metamodel that, when performed, will obtain the corresponding implementation from a model or from a delta model.

6.3.3 Transformations

The second activity in this phase is the *Assembly Machine Tool Generation*. This activity, performed by the GROVE TS, consists of the execution of two consecutive transformations whose result will be the assembly machine tool implementation.

The first of the transformations is the *Megamodel2AssemblyMachineToolModel* transformation. It is a model-to-model transformation that allows us to move from the megamodels to the assembly machine tools.

Table 6.1 shows the mapping from the *Megamodel* to the *Machine Tool Model*. The first row shows how two machine tool classes are created from the original megamodel: one is the model machine tool class (`modelMTC`), which deals with transformations between complete models, and the second one is the delta model machine tool class (`deltaModel`

Megamodel	**Machine Tool Model**
megamodel:Megamamodel	modelMTC:ModelMachineToolClass deltaModelMTC:DeltaModelMachineToolClass
metamodel.name	modelMTC.name deltaModelMTC.name
megamodel.domain	modelMTC.domain deltaModelMTC.domain

metamodel.composedOf	for every elem in megamodel.composedOf
	if elem.type = DeltaModel then
	add to deltaModelMTC.elements
	else
	add to modelMTC.elements

mm:Metamodel	c:Container
mm.name	c.name
mm.targetOf.name	c.transformation
mm.targetOf.from	c.sourceElemContainers

Table 6.1: *Megamodel2AssemblyMachineToolModel* Mapping

MTC), which works with delta models. Their names and domains will be created from the megamodel's name and domain. Each megamodel is composedOf a set of models that can be metamodels or exogenous transformations. For each metamodel a container will be generated, where deltaModelMTC will group the containers generated from the delta metamodels and modelMTC will group the rest. The name of the container will be the name of the metamodel it was created from, to indicate that the container will hold instances of that metamodel. Moreover, the container will also store the transformation of which that metamodel is the target and what the source of such transformation is (mm.targetOf.name and mm.targetOf.from).

Being the assembly machine tool metamodel close to the code, the second transformation, *AssemblyMachineTool2JavaCode* is a model-to-text transformation.

For every machine tool class a Java class is created. The class will have an attribute for every container that belongs to it and it will have a set of methods that encapsulate the transformations defined in the domain. This way, these methods will be called by the *assembly program* when performing a transformation.

Another Java class is created from every container in the metamodel. These classes encapsulate the management of the instances of the meta-

model from which they are created. They only contain a default constructor. If the metamodel is the target of a transformation, the constructor calls the transformation that will create the instance. Otherwise, the file that contains the model is passed as a parameter.

It is important to note that the generated code extends a set of classes that are provided with the GROVE TS. These classes contain the code that is common to every domain, such as the feature composition method, which is implemented as a call to the AHEAD Tool Suite [Bat].

These transformations fulfill the goal of automating the creation of repetitive code. Moreover, this code is defined generically for the domain, so it will be reusable for every product family defined in such domain.

6.3.4 Case Study

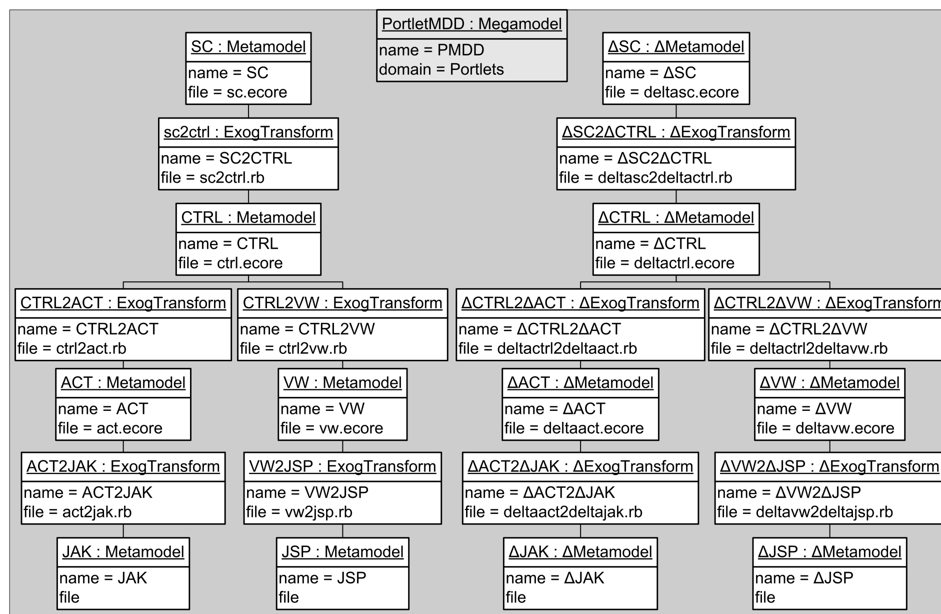


Figure 6.5: Portlet MDD Assembly Megamodel

The model driven domain of PinkCreek was modeled using *Portlet MDD*, which is a model driven approach that automates portlet implementation by defining a set of metamodels and transformations [Tru07]. Figure

6.5 presents the megamodel that accounts for those metamodels and transformations.

Recall from Chapter 3 that when applying *MDE* to portlets, the chosen most abstract metamodel is a `State Chart (SC)`, which provides a platform independent model for representing the flow of computations in a portlet. 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.

This State Chart is transformed into a `Ctrl` that defines a controller for a portlet. This in turn is transformed into `Act` and `View`, that define the actions to be performed and the views to be rendered during the controller execution respectively. Finally, more platform specific implementation details are to be given. The action model is transformed into `Jak` code. Jak(arta) is a superset of the Java language, where class and method refinements can be declared [BSR04]. Views, in turn, are transformed into `Java Server Pages (JSPs).`

Then, the same is done for *Delta Metamodels* and *Delta Exogenous Transformations*. The right half of Figure 6.5 sketches the delta metamodels and delta exogenous transformations defined in *Portlet MDD*. In this case the figure is symmetrical, i.e., an delta metamodel and an delta exogenous transformation exist for every metamodel and every exogenous transformation. It defines the transformations that will permit to obtain a $\triangle Jak$ and a $\triangle Jsp$ from a $\triangle SC$ instances.

This megamodel is then transformed into the assembly machine tool model shown in Figure 6.6. For each metamodel in the megamodel a container is generated (e.g., the `scContainer` is created from the `SC`). Moreover, for both complete models and delta models an assembly class is generated that gathers together the transformations needed to obtain the implementation (`Jak` and `JSP`) from the platform independent model (`SC`).

On the last step, this machine tool model is transformed into Java code. Figure 6.7 shows snippets of the code generated for *Portlet MDD*. Figure

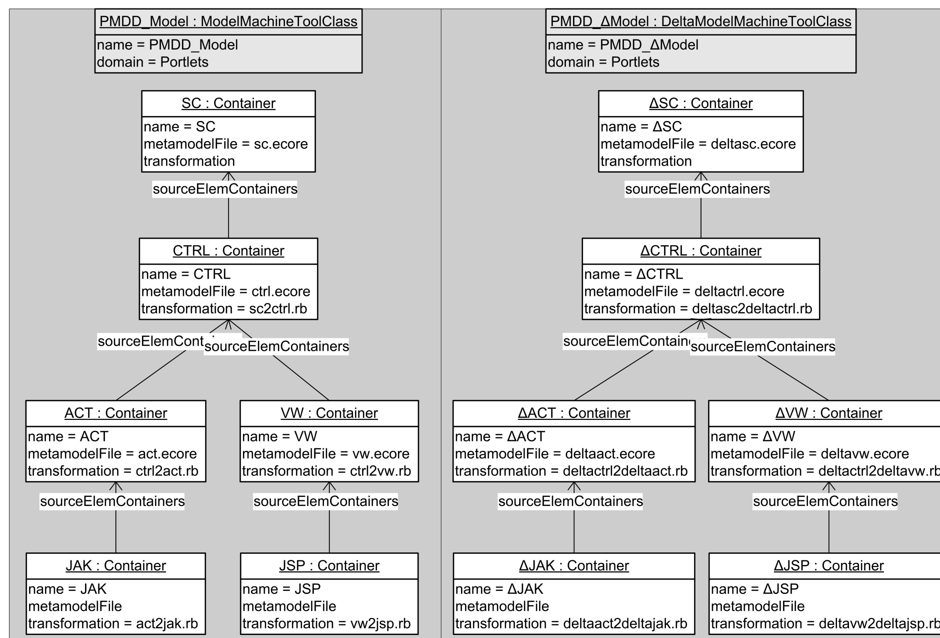


Figure 6.6: Portlet MDD Assembly Machine Tool Model

6.7a shows the assembly class for models. Note that it contains one method for every exogenous transformation defined in the megamodel. Figure 6.7b and 6.7c show examples of the code generated for containers. The first is the code generated for `SC`. Being the `State Chart` the topmost metamodel, from which all the rest are generated, the class takes the file that contains the model as input, as there is no transformation that has the `SC` as output. The second shows the code generated for `Ctrl`. The constructor calls the transformation that, with a `SC` as a parameter, generates the corresponding `Ctrl`.

Note that the generated code extends other classes (e.g., FeatureImp in Figure 6.7a). These classes are part of the GROVE TS and provide the functionality that is common for any given domain. In this way, transformations only have to generate the domain specific code.

It is important to highlight that there is nothing in this generated code that links it to a certain family of portlets. The methods defined in Figure 6.7a can be called by any family of portlets developed using *Portlet MDD*,

```
package org.PMDD;
import java.io.File;
import java.util.Collection;
import org.ctahead.impl.FeatureImpl;

public class PMDD_Model extends FeatureImpl {
  /*Create objects for elements*/
  public Sc elemSc =null;
  public Ctrl elemCtrl =null;
  public Act elemAct =null;
  public Jak elemJak =null;
  public Vw elemVw =null;
  public Jsp elemJsp =null;
  /*Create constructor method*/
  public PMDD_Model (file fSc) {
    /*for each raw element, create new from file*/
   elemSc=new Sc(fSc);}
  /*Create transformation steps*/
  public boolean t_sc2ctrl() {...}
  public boolean t_sc2ctrl() {...}
  public boolean t_sc2ctrl() {...}
  public boolean t_sc2ctrl() {...}
  public boolean t_sc2ctrl() {...}
}
```
a

```
package org.PMDD;
import java.io.File;
import org.ctahead.IObject;
import org.ctahead.impl.ObjectImpl;

public class Sc extends ObjectImpl implements IObject {
  public Sc (File fSc) {
     super(fSc, "Sc"); {
}
```
b

```
package org.PMDD;
import java.io.File;
import org.ctahead.IObject;
import org.ctahead.impl.ObjectImpl;

public class Ctrl  extends ObjectImpl implements IObject {
  public Ctrl (Sc elemSc) {
     this.setType("Sc");
     File fSc=elemSc.getFile();
     String sCtrl=fSc.getParent()+File.separator+"gen-"+
        +this.getType+"."+fSc.getName();
     File fCtrl=new File(sCtrl);
     this.setFile(fCtrl);
     Vector vParams=new Vector();
     vParams.add(0,fSc.toString());
     vParams.add(1,this.getFile().toString());
     this._or.processHandle("targett_sc2ctrl",vParams);}
}
```
c

Figure 6.7: Portlet Assembly Machine Tool (Class Examples)

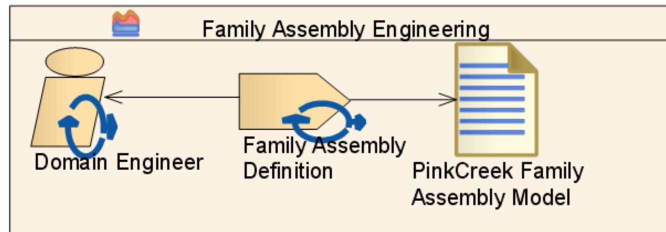


Figure 6.8: Family Assembly Engineering Phase

thus fostering reuse.

6.4 Family Assembly Engineering Phase

The previous phase generates the *Assembly Machine Tool*, realized as a library to be reused when assembling specific products. It defines the set of model transformations needed to obtain the desired product from an abstract specification (i.e., the *MDE perspective*). The next step is *Family Assembly Engineering*, where the focus of the development shifts from a single product to a family of them. The aim of this phase is to model features (i.e., the *SPLE perspective*), rather than code generation. As a consequence, note that this phase does not include transformations since its goal is only to model the assets that belong to the product family. Figure 6.8 depicts the SPEM diagram of this phase.

6.4.1 Process

There is a only one activity in this phase: the *Family Assembly Definition.* This activity models the features available in a family and the corresponding deltas that implement them, which will later be needed when assembling any member of the family. The SPL Domain Engineer is in charge of this activity.

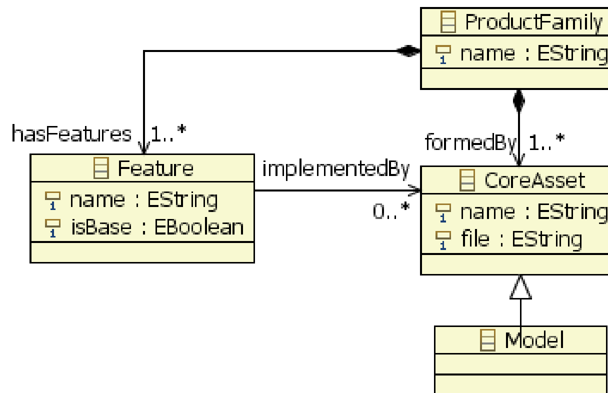


Figure 6.9: Family Assembly Metamodel

6.4.2 Models

The *Family Assembly Model* results from the family assembly definition activity. This model is intended to represent the realization in terms of features of the family.

Figure 6.9 shows its corresponding metamodel. A `ProductFamily` is characterized by a set of `Features`. These are implemented by `Core-Assets` of different types. These assets can be code artifacts, documentation, production plans and so on. Core assets also have two attributes: their `name` and `file`, i.e., the path where the core asset can be found.

The cause to specialize `CoreA`
`ssets` into `Models` is that these particular assets are relevant from an *MDE* perspective. The model also distinguishes between base and delta features. Remember that the former represents entire assets and the latter represents deltas. `Features` are modeled with two attributes to represent previous distinction: their `name` and `isBase`.

Overall, when assembling any product of the family, this model enables to locate each of its constituent core assets. This avoids the need to manually specify the path of every core asset every time a product is assembled.

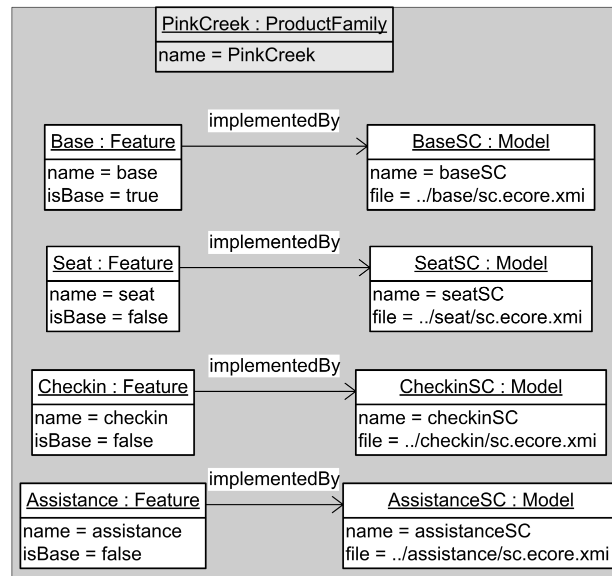


Figure 6.10: PinkCreek Family Assembly Model (simplified)

6.4.3 Case Study

The *Family Assembly Engineering Phase* is conducted once the *traditional* Domain Engineering has already been carried out [BSR04, CN01]. In the case of *PinkCreek*, this means that both the feature model of the flight-booking portlet family and their implementing model deltas were defined beforehand. Hence, the Domain Engineer instantiates the *PinkCreek Family Assembly Model*, that gathers together information about the core assets that realize the product family. It also includes where they can be located. Figure 6.10 shows a simplification of such model for PinkCreek, which conforms to the metamodel defined above. It sketches the `PinkCreek Product Family`, defined by four features, namely `base`, `seat`, `checkin` and `assistance`. Each of the features is implemented by a `SC model` (recall Figure 6.5, where SC models are the only ones that cannot be obtained from transformations). Each model contains the path to the file where it is stored. In this way, when assembling a certain portlet, the assembly program will be able to locate its constituent features.

As a result, the information about the core assets that belong to *PinkCre-*

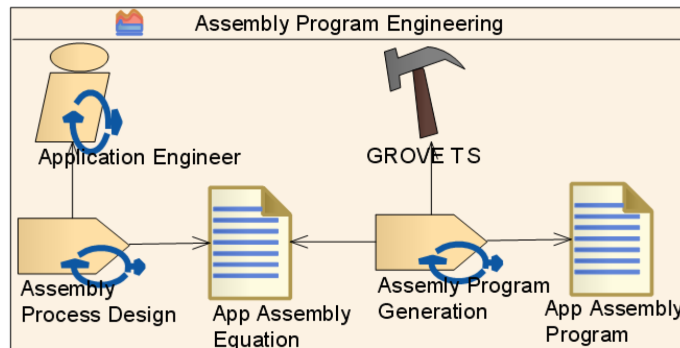


Figure 6.11: Assembly Program Engineering Phase

ek's features is centralized. Therefore, it can be accessed by any assembly program that requires it. This prevents the need to manually repeat this information every time a certain portlet is to be built.

6.5 Assembly Program Engineering Phase

During this phase the Application Engineer declaratively describes the assembly process as an *Assembly Equation*. For instance, she decides to add the *bountyFees* feature to the *baseModel* and then, to move down to code by applying transformation *sc2ctrl*, *ctrl2act* and *act2Jak*. The $app = act2jak \bullet ctrl2act \bullet sc2ctrl \bullet bountyFees \bullet baseModelSC$ equation reflects this decision.

The goal of this phase is to create the actual *Assembly Program* from the previous *Assembly Equation*. This phase is the core of the GROVE approach and the two previous phases provide the grounds for it. More precisely, they are used to build the reusable infrastructure needed to generate the assembly program. Figure 6.11 presents the SPEM diagram of this phase.

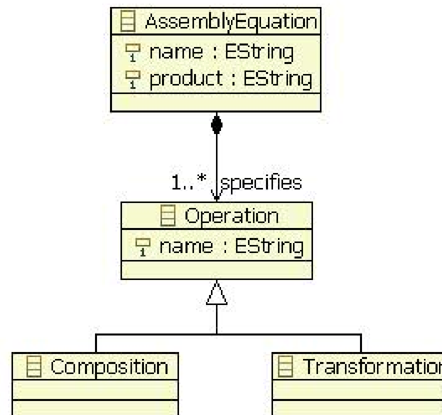


Figure 6.12: Assembly Equation Metamodel

6.5.1 Process

The Assembly Engineering Phase is divided into two activities. First, the Application Engineer performs the `Assembly Equation Definition`, producing the assembly equation as a result. This equation is a model that contains the operations needed to assemble the final product.

Second, the GROVE TS performs the `Assembly Program Generation`. Taking the previously defined equation as input, the tool generates the assembly program implementation.

6.5.2 Models

The result of the first activity is an equation model, where model transformation and composition operations are intertwined. Such model conforms to the metamodel presented in Figure 6.12.

An `Assembly Equation` contains a sequence of operations which accomplish the steps that need to be performed to obtain the final product. Each equation has two attributes, its `name` and the `product` it will lead to. In previous work (e.g., AHEAD Tool Suite), an equation only specifies the set of features that distinguishes the program [BSR04]. In our case these equations had to be enriched with model transformations, making

our equations a sequence of both compositions and model transformations. Figure 6.12 shows the metamodel for these assembly equations.

The transformation from this metamodel into code is detailed in the next subsection.

6.5.3 Transformations

The assembly machine tool created during the *Megamodel Engineering Phase* makes the transformation from the previous metamodel into Java code straightforward. Each operation is already realized as a method of a machine tool class. Therefore, this transformation involves creating a Java code that invokes the methods that already exist in the assembly line library.

This transformation has two input models: the *Assembly Equation* and the *Family Assembly Model*. The former defines the sequence of operations defined by the application engineer and the latter specifies the location of the models on which the operations will be performed. In order to generate the assembly program, a model-to-text transformation first creates an instance of the corresponding assembly class for each composition operation defined in the equation. The input files for each assembly class are located using the *Family Assembly Model* defined in the previous phase. Then, the sequence of operations is implemented by writing calls to the corresponding methods defined in the assembly classes.

6.5.4 Case Study

To build a certain portlet, the Application Engineer, when performing the *Assembly Equation Definition* activity, decides to add the *seat*, *checkin* and *assistance* features to the *base*. Apart from that, she decides that the best way to build the product is to first perform the composition of all features and then to transform the result to obtain the final code. Figure 6.13 shows the textual representation of such equation. Taking it as input, the GROVE

```
assembly_equation 'product1' do
  composition 'base'
  composition 'seat'
  composition 'checkin'
  transformation 'sc2ctrl'
  transformation 'ctrl2act'
  transformation 'act2jak'
  transformation 'ctrl2vw'
  transformation 'vw2jsp'
end
```

Figure 6.13: Assembly Equation Example for PinkCreek

```
1  package org.product1;
2  import java.io.File;
3  import org.PMDD;
4
5  public class AssemblyProgram {
6    public static void main(String [] args) {
7     /*Build objects for equation features*/
8     File fbaseSC = new File ("../base/sc.ecore.xmi");
9     PMDD_Model featurebase=new PMDD_Model (fbaseSC);
10    File fseatSC = new File ("../seat/sc.ecore.xmi");
11    PMDD_DeltaModel featureseat=new PMDD_Model (fseatSC);
12    File fcheckinSC = new File ("../checkin/sc.ecore.xmi");
13    PMDD_DeltaModel featurecheckin=new PMDD_Model (fcheckinSC);
14    /*Operations in equation*/
15    PMDD_Model result0=featurebase;
16    PMDD_Model result1=featureSeat.Dot(result0);
17    PMDD_Model result2=featurecheckin.dot(result1);
18    featurebase.t_sc2ctrl();
19    featureseat.t_sc2ctrl();
20    featurecheckin.t_sc2ctrl();
21    result0.t_sc2ctrl();
22    result1.t_sc2ctrl();
23    result2.t_sc2ctrl();
24    featurebase.t_ctrl2act();
25    /*Content omitted*/
26    result2.t_ctrl2act();
27    /*Content omitted*/
28    result2.t_act2jak();
29    /*Content omitted*/
30    result2.t_ctrl2vw();
31    /*Content omitted*/
32    result.t_vw2jsp();          /*This is the result*/
32    }
33 }
```

Figure 6.14: Assembly Program Example for PinkCreek

TS performs the transformation defined in the previous subsection to obtain the assembly program (see Figure 6.14).

For every composition operation in the assembly equation, an instance of an assembly class, which was generated in the *Megamodel Engineering*

Phase, needs to be created. First, the files where the models are stored are located using the the Family Assembly Model. Line 8 of Figure 6.14 shows how the `SC model` for the `base` feature is located. Then, an instance of the machine tool class is created. Line 9 in Figure 6.14 show this for the same feature. Notice that the machine tool class is different depending on the feature type. If it is the base feature, a `PMDD_Model` class is created. This class contains the transformations defined for complete models, while the `PMDD_DeltaModel` class, which is instantiated for the rest of the features, contains the transformations defined for delta models.

Next, the methods that were previously generated are called for every operation defined in the *Assembly Equation*. Lines 16 and 17 of Figure 6.14 show the implementation of the compositions. The rest of the program implements the sequence of model transformations. Note that transformations are applied to every machine tool class instance (e.g., lines 18 to 23 of Figure 6.14 present the calls to the *sc2ctrl* transformation). The reason for this behavior comes from the fact that when performing a composition, both machine tool classes need to be at the same *abstraction level* i.e., if one class has already generated a `Ctrl` model but the other only has the `SC` model, the composition cannot be performed. This ends in line 32, where after executing all the transformations, the result will be obtained.

6.6 Product Assembling Phase

The last task of the Application Engineer is the `Assembly Program Enactment`. Figure 6.1 depicts the SPEM diagram of this phase, which builds upon the results from the previous phases. Specifically, it reuses the code infrastructure generated from the megamodel and the assembly program generated from the equation to assemble a certain product. The Application Engineer will just execute the assembly program to obtain a family member.

The aim of *PinkCreek* product line is to yield individual portlets. Basically, the application engineer only needs to enact the assembly program

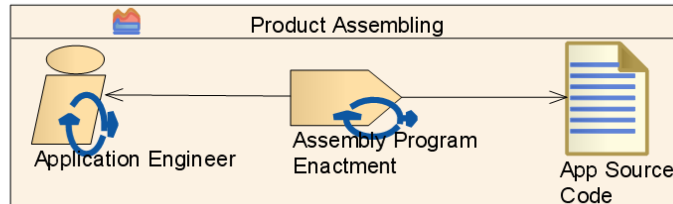


Figure 6.15: Product Assembling Phase

in order to get the end-product. In the example shown in Figure 6.14, this product would implement the *seat*, *checkin* and *assistance* features.

6.7 Discussion

This work is based on some perceived liabilities in product assembly in the context of *MDPLE*, namely: complexity (i.e., assembly programs become larger), design choices (i.e., there can be more than one possible assembly programs), and reuse opportunities (i.e., some assembly tasks are repetitive which leads to code snippets being repeated). The expected benefit of our approach is the improvement of this situation. Next, we evaluate to which extent this aim is achieved.

Handling Complexity. Before this work, an average assembly program to create a single portlet in *PinkCreek* accounted for 500 LOC of batch processes using 300 LOC of ANT makefiles and 2 KLOC of Java code. More important, it took around 4 people/day to complete. Now, an assembly program to build a portlet with 5 features is specified with an equation of 12 LOC. This is providing that the *Megamodel Engineering Phase* and the *Family Assembly Engineering Phase* have already been performed, where each one took around one hour and were only carried out once for all products. This is an improvement of at least one order of magnitude in the effort needed to develop the assembly program.

Handling Design Choices. Assembly programs are now specified at a higher abstraction level instead of being directly implemented. This al-

lows the application engineer to concentrate on the design of the assembly process, pondering the advantages and trade-offs of each decision, without worrying about implementation details. Currently the only decision available is the order of the operations when assembling a product (in domains where the order is unimportant). However, the equation could be enriched with more decisions related to the assembly process (e.g., whether to use other composition mechanisms apart from AHEAD, whether to execute certain tests and so on). These choices in the *Assembly Process* can be captured as part of the variability of the production plan [DTA05].

Handling Reuse Opportunities. As assembly gets more complex, an increase in the number of clones, i.e., chunks of code that are repeated in distinct parts of the program, is observed. This was the reason for stating that assembly programs have reuse opportunities. These opportunities call for a reuse mechanism that departs from traditional *"clone&own"* practices. This is precisely what we achieved by abstracting away from code and moving to a higher level where clones are generated through transformations from the abstract model and where reuse is accomplished in a systematic manner using model driven techniques. Model transformations were only the first mechanism provided to foster reuse. We also introduced the notion of the *Assembly Machine Tool*, a reusable library that provides operations for model transformation within the scope set by a megamodel, and which is reusable by every product family defined in such domain.

6.8 Related Work

Software Factories bring a perspective change in how software is developed where one of the changes is the importance of application assembly. Greenfield and Short state that in the near future most development will be component assembly, involving customization, adaptation and extension [GS03]. Having experienced this ourselves, we motivated the need for assembly processes to be designed and described the *Assembly Plan Management* as the means to fulfill that need.

Model management, of which application assembly is part, has been identified as one of the challenges that need to be addressed if *MDE* is to succeed [FR07]. In this line, megamodels establish and use global relationships on models, metamodels and transformations, ignoring the internal details of these global entities [Béz04]. Megamodels, however, only reflect structural relationships. We believe this should be complemented with the corresponding processes in order to achieve the *MDE* vision. Lately, different tools have aimed at this problem [RRGLR+09, VAB+07].

As an example, UniTI is a model-based approach to reuse and compose transformations in a technology-independent fashion [VAB+07]. UniTI focuses on transformation execution to yield a single product, but it does not consider product families. Our work faces the complexity that arises when *MDE* and *SPLE* are combined. This shows up the increasing complexity of program assembling, and raises some issues that do not appear when *MDE* transformation are considered in isolation.

Production planning defines how programs are built. It provides the strategical vision of the production process [CM02]. GROVE specifically concentrates on the assembling process part of this managerial plan.

6.9 Conclusions

Although its benefits are substantial, *MDPLE* also brings disadvantages, being increased product assembling complexity one of them. Along these lines, this chapter motivated the need for a new discipline in software development: the *Assembly Plan Management*. This discipline promotes the assembly process as a *first-class* citizen inside the software development process. Essentially, the *Assembly Plan Management* introduces a model driven approach to the generation of assembly programs in *MDPLE*.

The main contribution of this chapter is that assembly processes are no longer implemented, but are modeled and thus designed, raising the abstraction level. This makes the reuse of assembly processes possible. We evaluated our ideas with a case study.

Parts of this chapter have been previously presented:

- Maider Azanza, Oscar Díaz and Salvador Trujillo. Software Factories: Describing the Assembly Process. In *4th International Conference on Software Process (ICSP 2010)*, Paderborn, Germany, 2010.
- Salvador Trujillo, Maider Azanza and Oscar Díaz. Generative Metaprogramming. In *6th International Conference on Generative Programming and Component Engineering (GPCE 2007)*, Salzburg, Austria, 2007.

Chapter 7

Conclusions

"All I know is that I know nothing"
– *Socrates*

7.1 Overview

Model Driven Engineering and *Software Product Line Engineering* are two powerful paradigms for software development. Their combination in *Model Driven Product Line Engineering* brings together the advantages of both. Nevertheless, *MPLE* has implications on both core assets, where models become the primary focus, and processes, which need to blend the requirements of *SPLE* and *MDE*. This dissertation addressed these challenges by following the *Feature Oriented Software Development* paradigm for the former and by presenting a solution for assembly processes as an example of the latter. Different non trivial case studies were used to assess the applicability of the presented ideas.

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

7.2 Results

The first part of this dissertation was dedicated to the core asset implications of *MDPLE*. Here, the members of a product family of models are developed following a FOSD approach. In FOSD the members of the product family are created by composing the arrows that represent the features those products exhibit. Specifically:

- *Chapter 4* specified arrows as model deltas. Advantages of using models to realize arrows include declarativeness, being able to check certain constraints at delta building time (as model deltas conform to their own metamodel, which is related to the domain one) and the possibility to use existing work in incremental consistency management to check constraints at delta composition time. However, defining arrows as model deltas entails the need to specify how such deltas are composed. Generic composition was identified as the default, metamodel independent composition, which is sufficient in most cases. Besides generic composition, the need for domain specific composition was motivated. Two different case studies, namely *Questionnaires* and *UML Interaction Diagrams*, were used to evaluate the approach.

- *Chapter 5* defined a metamodel annotation mechanism to specify model delta composition. This brought increased reuse and productivity gains as delta composition implementation was automatically generated from the annotated metamodel. The benefits of such approach also include gains in automatization, understandability and better maintenance of the delta composition algorithm. Similarities of the presented ideas with other technical spaces were explored to assess their generality.

The second part of this dissertation delved into the impact of *MDPLE* on the assembly process. The work was aimed at increasing reuse by applying

MDE to assembly programs and by leveraging on the previous activities at every step of the process. More precisely:

- In *Chapter 6* the *Assembly Plan Management* was presented as a discipline inside the general software development process that permits to face the complexity brought by *MDPLE* to assembly processes. The benefits of such plan include a reduction of the assembly process implementation complexity, that the possibility to evaluate design choices is introduced and increased reuse. This plan specifies assembly processes at a higher abstraction level, from which its implementation is automatically generated. A non-trivial case study of a family of flight booking portlets was presented.

7.3 Publications

Part of the work presented in this thesis has already been discussed and presented in different peer-reviewed forums. The list of publications to which the author has contributed are listed below:

Selected Publications

- Maider Azanza, Don Batory, Oscar Díaz, and Salvador Trujillo. Domain-Specific Composition of Model Deltas. In *3rd International Conference on Model Transformations (ICMT 2010)*, Malaga, Spain, 2010 [ABDT10]. Rank B in the *ERA Conference Ranking*.

- Maider Azanza, Oscar Díaz and Salvador Trujillo. Software Factories: Describing the Assembly Process. In *4th International Conference on Software Process (ICSP 2010)*, Paderborn, Germany, 2010 [ADT10]. Rank A in the *ERA Conference Ranking*.

- Don Batory, Maider Azanza and João Saraiva. The Objects and Arrows of Computational Design. In *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS*

2008), Toulouse, France, 2008 [BAS08]. Rank B in the *ERA Conference Ranking*.

International Conferences/Workshops

- Roberto Lopez-Herrejon, Alexander Egyed, Salvador Trujillo, Josune de Sosa, and Maider Azanza. Using Incremental Consistency Management for Conformance Checking in Feature-Oriented Model-Driven Engineering. In *4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010)*, Linz, Austria, 2010 [LHET$^+$10].

- Maider Azanza, Josune De Sosa, Salvador Trujillo and Oscar Díaz. Towards a Process-Line for *MDPLE*. In *2nd International Workshop on Model-Driven Product Line Engineering (MDPLE 2010),* Paris, France, 2010 [AdSTD10].

- Salvador Trujillo, Maider Azanza and Oscar Díaz. Generative Metaprogramming. In *6th International Conference on Generative Programming and Component Engineering (GPCE 2007)*, Salzburg, Austria, 2007 [TAD07]. Rank B in the *ERA Conference Ranking*.

- Salvador Trujillo, Maider Azanza, Oscar Díaz and Rafael Capilla, Exploring Extensibility of Architectural Design Decisions. In *2nd Workshop on SHAring and Reusing architectural Knowledge - Architecture, rationale, and Design Intent (SHARK/ADI 2007)*, Minneapolis, Minnesota, USA, 2007 [TADC07].

- Oscar Díaz, Arantza Irastorza, Maider Azanza and Felipe M. Villoria: Modeling Portlet Aggregation Through Statecharts. In *7th International Conference on Web Information Systems Engineering (WISE 2006)*, Wuhan, China, 2006 [DIAV06b]. Rank A in the *ERA Conference Ranking*.

Spanish Conferences/Workshops

- Oscar Díaz, Arantza Irastorza, Maider Azanza and Felipe M. Villoria. Modelado de la Agregación de Portlets por medio de Statecharts. In *XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006)*, Sitges, Spain, 2006 [DIAV06a].

Others

- Maider Azanza, Salvador Trujillo and Oscar Díaz. Towards Generative Metaprogramming. In *2nd Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Braga, Portugal, 2007 [ATD07].

7.4 Research Stages

Any research work is enriched by the influence of different perspectives. This statement is even more true in the case of a Ph.D, which is, in essence, a learning process. This dissertation was enriched by the results of two research stages. First, the author visited the *Automated Software Design Research Group* at the University of Texas at Austin, USA, headed by Prof. Dr. Don Batory from January to April 2008 and again from June to September of the same year. Then, the author visited the *Software and Systems Modelling Team* at the University of York, UK, under the supervision of Prof. Dr. Richard Paige from June to August of 2009. Both visits fostered discussion, broadened horizons and greatly helped to improve this work.

7.5 Assessment and Future Research

This dissertation motivated and proposed a solution for two of the challenges that *Model Driven Product Line Engineering* poses. A unbiased

assessment of the work reveals some limitations and raises issues that provide an area for future research.

Core Asset Implications

- *Generalization.* Currently, our definition of model deltas only permits additions and updates of existing model elements. In certain cases, generalizing this definition to allow deletions might be necessary [KBA09]. Studying in which cases deletion is required and what its drawbacks are is an area for future work.

- *Aspects.* Model deltas are homogeneous extensions of models [ALS08]. In contrast, aspects may offer an alternative, where pointcuts identify one or more targets for rewriting (a.k.a. advising) [KH01]. However, the generality of aspects is by no means free: it comes at a cost of increased complexity in specifying, maintaining, and understanding concerns [ALS08]. An assessment of their advantages and the frecuency in which they are needed compared to the cost at which their generality comes is required.

- *Safe Composition of Models.* When developing a family of models incrementally, we want to guarantee that all legal feature compositions (i.e., all feature compositions that correspond to a product) yield models that satisfy all constraints of the metamodel, *but without enumeration*. Our work paves the way to this goal by allowing to check delta constraints and composition constraints. Nevertheless, the question of how to check deferred constraints to guarantee that every product conforms to its metamodel still remains open. Existing work using propositional formulas and SAT-solvers suggests a direction in which to proceed [CP06].

- *Transformation of Model Deltas.* In an *MDE* setting deltas are only half of the picture. As described in Chapter 3, composing model

deltas and then transforming the result product to yield the corresponding implementation is only one way of assembling the product. Deltas can also be transformed and then composed together, which can bring benefits such as validation and optimization [BAS08, TBD07, UGKB08]. To obtain these gains, we are interested in transformations that map a delta that is an instance of a metamodel to a delta that is an instance of another. Being deltas and models closely related, we are also interested in how these transformations are related to the ones that transform complete models.

- *Model Delta Decomposition.* When an error is detected in one product, the corresponding core asset should be corrected so that the rest of the products that contain such core asset can benefit. Although manually editing the core asset is possible, being able to decompose the product into its original (and now corrected) core assets increases efficiency [BSR04]. Model deltas can be seen as models that gather together the changes a feature makes to one or more models. Hence, recent work on selective undoing of model changes [GE10] suggests a interesting path to obtain the goal.

- *Case Studies.* This dissertation presented three case studies to prove the validity of the approach: the *Crime and Safety Survey Product Line (CSSPL)*, the *Game Product Line (GPL)* and the *AHEAD Product Line (APL)*. However, more case studies (preferably from industry) are needed.

- *Tool Support.* We implemented the ANDROMEDA tool as a proof of concept of our ideas. Nevertheless, more work is needed if we want to convert it in a tool available for others to use.

Process Implications - The Assembly Process

- *Generalization:* Currently, GROVE TS only supports transformations. A generalization is required to incorporate other model man-

agement operations.

- *Variability of the Assembly Process.* According to previous work, the production plan is subject to variability [DTA05]. Hence, variability of the assembly process needs to be defined and the *Assembly Plan Management* needs to be enriched to cope with such variability.

- *Specification of a process for MDPLE.* If *MDPLE* is to be spread in industry, an explicit process that details the necessary activities, roles, tasks and workproducts is needed [AdSTD10, MD08].

- *Variability of the MDPLE Development Process.* Not only is the assembly process subject to variability, the same holds for the software development process that applies *MDPLE* [AdSTD10].

- *Tool Support.* GROVE Tool Suite supports the *Assembly Plan Management*. Nevertheless, more work is needed to generalize it with other paradigms. Furthermore, the assembly is only a part of the development process. More tools are needed to support developers following *MDPLE*.

7.6 Conclusions

Programs are structures that software engineers create. They use tools to transform, manipulate, and analyze them. Object orientation uses methods, classes, and packages to structure programs. Compilers transform source structures into byte-code structures. Refactoring tools transform source structures into other source structures, and metamodels of *MDE* define the allowable structures of model instances: transformations map metamodel instances to instances of other metamodels for purposes of analysis and synthesis [BAS08].

The future of software design and development lies in automation — the mechanization of repetitive tasks to free programmers from mundane

activities so that they can tackle more creative problems. We are entering the age of *Computational Design (CD)*, where both program design and program synthesis are computations [Win06], and *MDPLE* lies at the forefront of it, as the combination of two paradigms that promote systematic reuse and increase automation.

Nevertheless, there are implications that need to be resolved if *MDPLE* is to reach its full potential. This dissertation presented two essential issues in this endeavor, namely the core asset and process implications. Viable solutions for the case of feature oriented development of models and assembly processes were presented, which were validated using case studies of different sizes. We believe this work will help in making the vision of *MDPLE* possible, thus providing its numerous benefits.

Bibliography

[ABDT10] Maider Azanza, Don Batory, Oscar Díaz, and Salvador Trujillo. Domain-Specific Composition of Model Deltas. In *3rd International Conference on Model Transformations (ICMT 2010), Malaga, Spain*, volume 6142 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2010.

[AdSTD10] Maider Azanza, Josune de Sosa, Salvador Trujillo, and Oscar Díaz. Towards a Process-Line for MDPLE. In *2nd International Workshop on Model-Driven Product Line Engineering (MDPLE 2010), Paris, France*, volume 625 of *CEUR Workshop Proceedings (CEUR-WS.org)*, pages 3–12. CSun SITE Central Europe (CEUR), 2010.

[ADT07] Felipe I. Anfurrutia, Oscar Díaz, and Salvador Trujillo. On Refining XML Artifacts. In *7th International Conference on Web Engineering (ICWE 2007), Como, Italy*, volume 4607 of *Lecture Notes in Computer Science*, pages 473–478. Springer, 2007.

[ADT10] Maider Azanza, Oscar Díaz, and Salvador Trujillo. Software Factories: Describing the Assembly Process. In *4th International Conference on Software Process (ICSP 2010), Paderborn, Germany*, volume 6195 of *Lecture Notes in Computer Science*, pages 126–137. Springer, 2010.

[AJTK09] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. Model Superimposition in Software Product Lines. In *2nd International Conference on Theory and Practice of Model Transformations (ICMT 2009), Zurich, Switzerland*, volume 5563 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2009.

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. FEATUREHOUSE: Language-Independent, Automated Software Composition. In *31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada*, pages 221–231. IEEE, 2009.

[ALS08] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[AMW] Atlas Model Weaver (AMW). Online at http://www.eclipse.org/gmt/amw/. Last accessed: September 2010.

[ASE] 25th IEEE/ACM International Conference on Software Engineering (ASE 2010), Antwerp, Belgium. Online at http://soft.vub.ac.be/ase2010/. Last accessed: September 2010.

[ATD07] Maider Azanza, Salvador Trujillo, and Oscar Díaz. Towards Generative Metaprogramming. In *2nd Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007), Braga, Portugal*, 2007.

[BAS08] Don Batory, Maider Azanza, and João Saraiva. The Objects and Arrows of Computational Design. In *11th International Conference on Model Driven Engineering Lan-*

guages and Systems (MoDELS 2008), Toulouse, France, volume 5301 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2008.

[Bat] Don Batory. AHEAD Tool Suite (ATS). Online at http://www.cs.utexas.edu/users/schwartz/ATS.html. Last accessed: September 2010.

[BBF+06] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios S. Kolovos, Ivan Kurtev, and Richard 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, 2006*, volume 4066 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2006.

[BCRL07] Artur Boronat, José A. Carsí, Isidro Ramos, and Patricio Letelier. Formal Model Merging Applied to Class Diagram Integration. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:5–26, 2007.

[Ber03] Philip A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *1st Biennial Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, California, USA*, 2003.

[Béz04] Jean Bézivin. In Search of a Basic Principle for Model-Driven Engineering. *UPGRADE, The European Journal for the Informatics Professional, Special Issue on UML and Model Engineering*, 5(2):21–24, 2004.

[Béz05] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSyM)*, 4(2):171–188, 2005.

[BFK+99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In *Symposium on Software Reusability (SSR 1999), Los Angeles, California, USA*, pages 122–131. ACM, 1999.

[BGL+06] Krishnakumar Balasubramanian, Aniruddha S. Gokhale, Yuehua Lin, Jing Zhang, and Jeff Gray. Weaving Deployment Aspects into Domain-specific Models. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 16(3):403–424, 2006.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, Canada*, 2004.

[BK06] Jean Bézivin and Ivan Kurtev. Model-Based Technology Integration with the Technical Space Concept. In *5th Metainformatics Symposium (MIS 2006)*, 2006.

[BLW05] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context - Motorola Case Study. In *8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005), Montego Bay, Jamaica*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491. Springer, 2005.

[Boe05] Barry W. Boehm. The Future of Software Processes. In *International Software Process Workshop (SPW), Beijing, China, Revised Selected Papers*, pages 10–24, 2005.

[Bro87] Frederick Brooks. No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

[BSR04] Don Batory, Jacob Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[CA05] Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *4th International Conference Generative Programming and Component Engineering (GPCE 2005), Tallinn, Estonia*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CG98] Gianpaolo Cugola and Carlo Ghezzi. Software Processes: A Retrospective and a Path to the Future. *Software Process: Improvement and Practice*, 4(3):101–123, 1998.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[CM02] Gary Chastek and John D. McGregor. Guidelines for Developing a Product Line Production Plan. Technical report, Carnegie-Mellon University/ Software Engineering Institute, June 2002. CMU/SEI-2002-TR-06.

[CMT06] Jesús Sánchez Cuadrado, Jesús García Molina, and Marcos Menárguez Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *2nd European Conference on*

Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain, volume 4066 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2006.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2001.

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *5th International Conference on Generative Programming and Component Engineering (GPCE 2006), Portland, Oregon, USA*, pages 211–220. ACM, 2006.

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology (JOT)*, 6(9):165–185, 2007. Special Issue: TOOLS EUROPE 2007.

[Dav87] Stanley M. Davis. *Future Perfect*. Addison-Wesley, 1987.

[DIAV06a] Oscar Díaz, Arantza Irastorza, Maider Azanza, and Felipe M. Villoria. Modelado de la Agregación de Portlets por Medio de Statecharts. In *XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006), Sitges, Spain*, pages 453–462. Centro Internacional de Métodos Numéricos en Ingeniería (CIMNE), 2006.

[DIAV06b] Oscar Díaz, Arantza Irastorza, Maider Azanza, and Felipe M. Villoria. Modeling Portlet Aggregation Through Statecharts. In *7th International Conference on Web Information Systems Engineering (WISE 2006), Wuhan, China*,

volume 4255 of *Lecture Notes in Computer Science*, pages 265–276. Springer, 2006.

[DS97] Michael Diaz and Joseph Sligo. How Software Process Improvement Helped Motorola. *IEEE Software*, 14(5):75–81, 1997.

[DSB04] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Experiences in Software Product Families: Problems and Issues During Product Derivation. In *3rd International Software Product Line Conference (SPLC 2004), Boston, Massachusetts, USA*, volume 3154 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2004.

[DTA05] Oscar Díaz, Salvador Trujillo, and Felipe I. Anfurrutia. Supporting Production Strategies as Refinements of the Production Process. In *9th International Software Product Line Conference (SPLC 2005), Rennes, France*, pages 210–221, 2005.

[DTP07] Oscar Díaz, Salvador Trujillo, and Sandy Pérez. Turning Portlets into Services: The Consumer Profile. In *16th International Conference on World Wide Web (WWW 2007), Banff, Alberta, Canada*, pages 913–922. ACM, 2007.

[ECO] 24th European Conference on Object-Oriented Programming (ECOOP 2010), Maribor, Slovenia. Online at http://ecoop2010.uni-mb.si/. Last accessed: September 2010.

[Egy06] Alexander Egyed. Instant Consistency Checking for the UML. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China*, pages 381–390. ACM, 2006.

[Egy07] Alexander Egyed. Fixing Inconsistencies in UML Design Models. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA*, pages 292–301. IEEE Computer Society, 2007.

[Eps] Epsilon. Online at http://www.eclipse.org/gmt/epsilon/. Last accessed: September 2010.

[Fav04] Jean-Marie Favre. Towards a Basic Theory to Model Driven Engineering. In *3rd Workshop in Software Model Engineering (WISME 2004) at at the 7th International Conference on the Unified Modeling Language (UML 2004), Lisbon, Portugal*, 2004.

[FBFG07] Franck Fleurey, Benoit Baudry, Robert B. France, and Sudipto Ghosh. A Generic Approach for Automatic Model Composition. In *Models in Software Engineering, Workshops and Symposia at the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007), Nashville, Tennessee, USA, Reports and Revised Selected Papers*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer, 2007.

[FBL08] Greg Freeman, Don Batory, and R. Greg Lavender. Lifting Transformational Models of Product Lines: A Case Study. In *1st International Conference on Theory and Practice of Model Transformations (ICMT 2008), Zürich, Switzerland*, volume 5063 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.

[FHL+98] Eckhard Falkenberg, Wolfgang Hesse, Paul Lindgreen, Björn Nilsson, J. L. Han Oei, Colette Rolland, Ronald Stamper, Frans J. M. Van Assche, Alexander A. Verrijn-Stuart, and Klaus Voss. A Framework of Information System Concepts-The FRISCO Report. Technical Report 1,

International Federation for Information Processing (IFIP), 1998.

[FKN+92] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L. Finkelstein, and Michael Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 2(1):31–57, 1992.

[FR07] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Workshop on the Future of Software Engineering (FOSE 2007), at the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA*, pages 37–54, 2007.

[FS04] Frédéric Fondement and Raul Silaghi. Defining Model Driven Engineering Processes. In *3rd International Workshop in Software Model Engineering (WiSME 2004), at the 7th International Conference on the Unified Modeling Language (UML 2004), Lisbon, Portugal*, 2004.

[Fug00] Alfonso Fuggetta. Software Process: a Roadmap. In *22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, pages 25–34. ACM, 2000.

[FV04] Lidia Fuentes and Antonio Vallecillo. An Introduction to UML Profiles. *UPGRADE, The European Journal for the Informatics Professional, Special Issue on UML and Model Engineering*, 5(2):5–13, 2004.

[GE10] Iris Groher and Alexander Egyed. Selective and Consistent Undoing of Model Changes. In *13th International Conference on Model Driven Engineering Languages and Sys-*

tems (MODELS 2010), Oslo, Norway, volume 6395 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2010.

[GS03] Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *Companion of the 18th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), Anaheim, California, USA*, pages 16–27. ACM, 2003.

[GS04] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Models and Tools*. Wiley, 2004.

[ICM] International Conference on Model Transformation (ICMT). Online at http://www.model-transformation.org/. Last accessed: September 2010.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming (SCP)*, 72(1-2):31–39, 2008.

[KBA09] Martin Kuhlemann, Don S. Batory, and Sven Apel. Refactoring feature modules. In *11th International Conference on Software Reuse (ICSR 2009), Falls Church, Virginia, USA*, volume 5791 of *Lecture Notes in Computer Science*, pages 106–115. Springer, 2009.

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University/Software Engineering Institute, November 1990.

[Ker] Kermeta. Online at http://www.kermeta.org/. Last accessed: September 2010.

[KH01] Gregor Kiczales and Erik Hilsdale. Aspect-Oriented Programming. In *8th European Software Engineering Conference held jointly with 9th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE 2001), Vienna, Austria*, page 313. ACM, 2001.

[KKL+98] Kyo Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[KPP06a] Dimitrios Kolovos, Richard Paige, and Fiona Polack. Merging Models with the Epsilon Merging Language (EML). In *9th International Conference in Model Driven Engineering Languages and Systems (MoDELS 2006), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2006.

[KPP06b] Dimitrios Kolovos, Richard Paige, and Fiona Polack. Model Comparison: a Foundation for Model Composition and Model Transformation Testing. In *International Workshop on Global Integrated Model Management (GaMMa 2006), Shangai, China*, pages 13–20. ACM, 2006.

[KPP08] Dimitrios Kolovos, Richard Paige, and Fiona Polack. Detecting and Repairing Inconsistencies across Heterogeneous Models. In *International Conference on Software Testing, Verification, and Validation (ICST 2008), Lillehammer, Norway*, pages 356–364. IEEE Computer Society, 2008.

[KR03] Vinay Kulkarni and Sreedhar Reddy. Separation of Concerns in Model-Driven Development. *IEEE Software*, 20(5):64–69, 2003.

[Kur05] Ivan Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, 2005.

[LHB01] Roberto Lopez-Herrejon and Don Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001), Erfurt, Germany*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer, 2001.

[LHET+10] Roberto Lopez-Herrejon, Alexander Egyed, Salvador Trujillo, Josune de Sosa, and Maider Azanza. Using Incremental Consistency Management for Conformance Checking in Feature-Oriented Model-Driven Engineering. In *4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2010), Linz, Austria*, volume 37 of *ICB-Research Report*, pages 93–100. Universität Duisburg-Essen, 2010.

[LMÁ09] Francisco J. Lucas, Fernando Molina, and José Ambrosio Toval Álvarez. A Systematic Review of UML Model Consistency Management. *Information and Software Technology (IST)*, 51(12):1631–1645, 2009.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The Generic Modeling Environment. In *9th International Workshop on Intelligent Signal Processing (WISP 2001), Budapest, Hungary*, 2001.

[LWK10] Philip Langer, Manuel Wimmer, and Gerti Kappel. Model-to-Model Transformations By Demonstration. In *3rd International Conference on Theory and Practice of Model Transformations (ICMT 2010), Malaga, Spain*, volume 6142 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2010.

[MAP] 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010), collocated with the 14th International Software Product Line Conference (SPLC 2010), Jeju Island, South Korea. Online at http://lero.ie/maple2010. Last accessed: September 2010.

[Mat04] Mari 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*, pages 127–136. IEEE Computer Society, 2004.

[MBJ08] Brice Morin, Olivier Barais, and Jean-Marc Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *2nd International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2008), Duisburg-Essen, Germany*, pages 53–62, 2008.

[MD08] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008), Berlin, Germany*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443. Springer, 2008.

[MDP] 2nd International Workshop on Model-driven Product Line Engineering (MDPLE 2010), held in conjunction with the

6th European Conference on Modelling Foundations and Applications (ECMFA 2010), Paris, France. Online at http://www.lero.ie/mdple2010/. Last accessed: September 2010.

[MFB09] Pierre-Alain Muller, Frédéric Fondement, and Benoit Baudry. Modeling Modeling. In *12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, Colorado, USA*, volume 5795 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2009.

[MKBJ08] Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. A Generic Weaver for Supporting Product Lines. In *13th International Workshop on Early Aspects (EA 2008), co-located with the 7th International Conference on Aspect-Oriented Software Development (AOSD 2008), Leipzig, Germany*, pages 11–18. ACM, 2008.

[ML97] Marc H. Meyer and Alvin Lehnerd. *The Power Of Product Platforms: Building Value And Cost Leadership*. Free Press, 1997.

[Nor04] Linda M. Northrop. Software Product Line Adoption Roadmap. Technical Report CMU/SEI-2004-TR-022, Carnegie-Mellon University/Software Engineering Institute, 2004.

[OAC+06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[OMG] OMG. OMG Model Driven Architecture (MDA). Online at http://www.omg.org/mda/. Last accessed: September 2010.

[OMG02] OMG. Meta Object Facility (MOF) Specification - Version 1.4. Adopted Specification, April 2002. Online at http://www.omg.org/spec/MOF/1.4/PDF/. Last accessed: September 2010.

[OMG08a] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.0. Formal Specification, April 2008. Online at http://www.omg.org/spec/QVT/1.0/PDF/. Last accessed: September 2010.

[OMG08b] OMG. Software Process Engineering Metamodel Specification. Formal Specification, April 2008. Online at: http://www.omg.org/spec/SPEM/2.0/PDF/. Last accessed: September 2010.

[OMG09] OMG. Unified Modeling Language (UML), v2.2. Formal Specification, February 2009. Online at http://www.omg.org/spec/UML/2.2/Superstructure/PDF/. Last accessed: September 2010.

[Ost87] Leon J. Osterweil. Software Processes Are Software Too. In *9th International Conference on Software Engineering (ICSE 1987), Monterey, California, USA*, pages 2–13. IEEE Computer Society Press, 1987.

[Par76] David Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering (TSE)*, 2(1):1–9, 1976.

[PB03] Rachel Pottinger and Philip Bernstein. Merging Models Based on Given Correspondences. In *29th International Conference on Very Large Data Bases (VLDB*

2003), Berlin, Germany, pages 862–873. VLDB Endowment, 2003.

[PBvdL06] Klaus Pohl, Günter Bockle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer, 2006.

[Pes03] Linda Pesante. Software Engineering Institute (SEI). In *Encyclopedia of Computer Science*, pages 1611–1613. John Wiley and Sons Ltd., Chichester, UK, 2003.

[PWCC93] Mark Paulk, Charles Weber, Bill Curtis, and Mary Beth Chrissis. Capability Maturity Model for Software, Version 1.1. Technical Report CMU/SEI-93-TR-024, Carnegie Mellon University/Software Engineering Institute, 1993.

[RBJ07] Rodrigo Ramos, Olivier Barais, and Jean-Marc Jézéquel. Matching Model-Snippets. In *10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007), Nashville, Tennessee, USA*, volume 4735 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2007.

[RPKP08] Louis Rose, Richard Paige, Dimitrios Kolovos, and Fiona Polack. The Epsilon Generation Language. In *4th European conference on Model Driven Architecture Foundations and Applications (ECMDA-FA 2008), Berlin, Germany*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[RRGLR+09] Jose Rivera, Daniel Ruiz-González, Fernando López-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL Model Transformations. In *1st International Workshop on Model Transformation with ATL (MtATL*

2009), at the 10th Libre Software Meeting (LSM 2009), Nantes, France, pages 34–46, 2009.

[RV08] José Rivera and Antonio Vallecillo. Representing and Operating with Model Differences. In *46th International Conference on Objects, Components, Models and Patterns (TOOLS EUROPE 2008), Zurich, Switzerland*, volume 11 of *Lecture Notes in Business Information Processing*, pages 141–160. Springer, 2008.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. Addison-Wesley, 2008.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[SPLa] Software Product Line Conferences (SPLC). Online at http://splc.net/. Last accessed: September 2010.

[SPLb] Software Product Line Hall of Fame. Online at http://www.splc.net/fame.html. Last accessed: September 2010.

[TAD07] Salvador Trujillo, Maider Azanza, and Oscar Díaz. Generative Metaprogramming. In *6th International Conference Generative Programming and Component Engineering (GPCE 2007), Salzburg, Austria*, pages 105–114. ACM, 2007.

[TADC07] Salvador Trujillo, Maider Azanza, Oscar Díaz, and Rafael Capilla. Exploring Extensibility of Architectural Design Decisions. In *2nd Workshop on SHAring and Reusing architectural Knowledge - Architecture, rationale, and Design Intent (SHARK/ADI 2007) at the 29th International*

Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA, 2007.

[TBD06] Salvador Trujillo, Don Batory, and Oscar 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*, pages 191–200. ACM, 2006.

[TBD07] Salvador Trujillo, Don Batory, and Oscar Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA*, pages 44–53. IEEE Computer Society, 2007.

[TGLH+10] Salvador Trujillo, Jose Miguel Garate, Roberto Erick Lopez-Herrejon, Xabier Mendialdua, Albert Rosado, Alexander Egyed, Charles W. Krueger, and Josune de Sosa. Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy. In *6th European Conference on Modelling Foundations and Applications (ECMFA 2010), Paris, France*, volume 6138 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 2010.

[Tru07] Salvador Trujillo. *Feature Oriented Model Driven Product Lines*. PhD thesis, School of Computer Sciences, University of the Basque Country, March 2007. Online at http://www.struji.com.

[UGKB08] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. Testing Software Product Lines Using Incremental Test Generation. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008),*

Seattle/Redmond, Washington, USA, pages 249–258. IEEE Computer Society, 2008.

[UNKC08] Muhammad Usman, Aamer Nadeem, Tai-hoon Kim, and Eun-suk Cho. A Survey of Consistency Checking Techniques for UML Models. In *International Conference on Advanced Software Engineering and Its Applications (ASEA 2008), Hainan Island, China*, pages 57–62. IEEE Computer Society, 2008.

[VAB+07] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. UniTI: A Unified Transformation Infrastructure. In *10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007), Nashville, USA*, volume 4735 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.

[Val10] Antonio Vallecillo. On the Combination of Domain Specific Modeling Languages. In *6th European Conference on Modelling Foundations and Applications (ECMFA 2010), Paris, France*, volume 6138 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2010.

[VAM] 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), Linz Austria. Online at http://www.vamos-workshop.net/2010/. Last accessed: September 2010.

[VG07] Markus Völter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *11th International Conference on Software Product Lines (SPLC 2007), Kyoto, Japan*, pages 233–242. IEEE Computer Society, 2007.

[VN96] Michael VanHilst and David Notkin. Using Role Components to Implement Collaboration-Based Designs. In *11th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1996), San José, California, USA*, pages 359–369. ACM, 1996.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *21st European Conference on Object-Oriented Programming (ECOOP 2007), Berlin, Germany*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer, 2007.

[Win06] Jeannette Wing. Computational Thinking. *Communications of the ACM*, 49(3):33–35, 2006.

[Wir83] Niklaus Wirth. Program Development by Stepwise Refinement. *Commununications of the ACM*, 26(1):70–74, 1983.

[WJE+09] Jon Whittle, Praveen K. Jayaraman, Ahmed M. Elkhodary, Ana Moreira, and João Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. *Transactions on Aspect-Oriented Software Development VI, Special Issue on Aspects and Model-Driven Engineering*, 6:191–237, 2009.

[WKK+10] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In *3rd International Conference on Theory and Practice of Model Transformations (ICMT 2010), Malaga, Spain*, volume 6142 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2010.

[WL99] David Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: a Family-Based Software Development Process*. Addison-Wesley, 1999.

[Wri05] Kevin Wright. Researching Internet-Based Populations: Advantages and Disadvantages of Online Survey Research, Online Questionnaire Authoring Software Packages, and Web Survey Services. *Journal of Computer-Mediated Communication*, 10(3):article 11, 2005.

[ZSS+09] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. VML* - A Family of Languages for Variability Management in Software Product Lines. In *Second International Conference on Software Language Engineering (SLE 2009), Denver, Colorado, USA, Revised Selected Papers*, volume 5969 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2009.

Acknowledgments

There is an african proverb that says *"If you want to go quickly, go alone. If you want to go far, go together."* A doctorate is a long journey and many people have travelled with me, helping me reach this point. This lines intend to thank all of them wholeheartedly.

First and foremost, I would like to thank my supervisor, Prof. Oscar Díaz. From the very beginning, when I was uncertain whether a Ph.D was the right choice for me, he has always encouraged me. There is so much I have learnt from him.

Oscar leads the Onekin Research Group at the University of the Basque Country. The above proverb summarizes perfectly its spirit. Collaboration has always been one of its hallmarks. This work has been greatly influenced and enriched by the Onekin members. In particular, this work follows the path opened by Salvador Trujillo in his dissertation, who was always ready to engage in endless and very fruitful discussions. Gorka Puente eagerly participated in long debates over all the aspects of MDE and Josune de Sosa helped greatly on the process side. Arantza Irastorza provided moral support in several occasions. I would also like to express my gratitude to the remaining present and past members of Onekin: Luis M. Alonso, Cristóbal Arellano, Iker Azpeitia, Oscar Barrera, Sergio Fernández, Jokin García, Felipe Ibáñez, Jon Iturrioz, Jon Kortabitarte, Felipe Martín, Itziar Otaduy, Iñaki Paz, Sandy Pérez and Juanjo Rodríguez. We shared many hours, many talks, many laughs...

During these years I enjoyed a doctoral grant from the Basque Goverment under the *"Researchers Training Program"*. This financial support

allowed me to focus entirely on my Ph.D and also gave me the invaluable opportunity of visiting the University of Texas at Austin and the University of York. The six months I spent in Austin working with Prof. Don Batory were a turning point in this work. I am deeply grateful for all time and effort he dedicated to it. I was very enriched by it. I also thank Prof. Richard Paige and all the members of the Software and Systems Modelling Team for welcoming me among them during my time in York.

One of the most rewarding parts of this journey has been the opportunity to discuss and share experiences with relevant people in the field. In particular, I would like to thank Martin Kuhlemann from the University of Magdeburg, Roberto E. López Herrejón from the Johannes Kepler University, Beatriz Pérez Lamancha from the University of Castilla-La Mancha, Jesús Sánchez Cuadrado from the University of Murcia and João Saraiva from the University of Minho. I would also like to thank the anonymous reviewers and the members of the dissertation committee, who helped improve this work substantially.

Although sometimes it was difficult to believe, there is a world outside the walls of the lab.

I would like to thank Laura, who was also making her way through a doctorate, for always being there. The questionnaire case study was born in one of our *therapy sessions*. Also Maika, Mari and Sonia, three English majors and great friends, who by now can give a talk in *MDPLE* and fool almost anyone. Thanks for everything, I owe you one. To Aines and Txon, for their help with the artistic part.

My gratitude to Juan, for his support, for the music, for so much more.

Thanks also to all my friends and relatives, for never forgetting to inquire about *"those little things you put together to make bigger things"*, for taking my mind off the thesis and for making me smile.

To my father and brother, who have always supported and encouraged me. Also to my mother because, although she is not here to see this day, she taught me that where there's a will there's a way. Eskerrikasko bihotz bihotzez.

Epilogue

A Ph.D is a long term goal. During these years there have been both good and bad moments, including ones where it seemed unachievable. Nevertheless, the experience has been unforgetable and a learning ground for many more things aside from what you have in your hands.

In perspective, a doctorate provides some of the necessary skills and knowledge but it only constitutes the first steps in the research world. An exciting journey lays ahead, let it begin.