A MODEL-DRIVEN APPROACH TO VARIABILITY MANAGEMENT IN PRODUCT-LINE ENGINEERING

ANDRÉ L. SANTOS[1]*, KAI KOSKIMIES[1], ANTÓNIA LOPES[2]
[1]*Institute of Software Systems, Tampere University of Technology*
P.O. Box 553, 33101 Tampere, Finland
`{andre.santos,kai.koskimies}@tut.fi`
[2]*Department of Informatics, Faculty of Sciences, University of Lisbon*
Campo Grande, 1700 Lisboa, Portugal
`mal@di.fc.ul.pt`

Abstract. Object-oriented frameworks play an essential role in the implementation of product-line architectures (PLAs) for product families. However, recent case studies reveal that deriving products from the shared software assets of a product-line is a time-consuming and expensive activity. In this paper, we present a model-driven approach for product derivation in framework-based product-lines. This approach aims to alleviate the aforementioned problems by bridging the gap between domain and application engineering activities in product-line-based development. Our approach is centered on a layered model embracing different artifacts ranging from models for conceptual and architectural variability to models for the realization of variation points. The approach provides mechanisms for enforcing the variation rules throughout the product derivation process and for documenting the variability issues in frameworks and the derived applications. We demonstrate the approach using an existing Java GUI framework.

ACM CCS Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques — *Computer-aided software engineering (CASE), Object-oriented design methods*; D.2.13 [**Software Engineering**]: Reusable Software — *Domain engineering, Reuse models*; D.2.11 [**Software Engineering**]: Software Architectures — *Domain-specific architectures*

Key words: product-lines, frameworks, variability management, model-driven engineering

1. Introduction

Software product-lines have emerged as a central paradigm for systematic, large-scale software reuse. A *software product-line* embodies the processes, tools, and software assets that can be used to derive applications sharing similar structure and functionality [Krueger 2006]. The applications derived from a product-line constitute a product family. A key asset of a product-line is the *product-line architecture* (PLA) – the shared architecture of the product family [Bosch 2000]. Products, i.e. members of the product family, are built according to the PLA using the support provided by the product-line. The development process responsible for

*On leave from (2) with the support of the Portuguese *Fundação para a Ciência e Tecnologia* establishing a product-line is known as *domain engineering* whereas *application engineering* refers to the actual product derivation process.

A *variation point* refers to a well-defined location in a software artifact where some variation can occur among different products. A variation point can be located at different levels of abstraction, ranging from requirements to source code. A variation point is expressed in terms of *variability mechanisms*, allowing the attachment of the chosen variant at the variation point. An overview of existing variability mechanisms for product-lines is presented in [Gacek and Anastasopoulos 2001].

Domain engineers have to find appropriate variability mechanisms for each variation point. In addition, each variation point should be augmented with relevant documentation on how to use the variability mechanism, in order to achieve the desired effect in a product. The specification, realization and usage of the variation points are key activities in product-line processes, commonly referred to as *variability management* (e.g. [Bosch *et al.* 2002]). Figure 1 illustrates the described concepts.

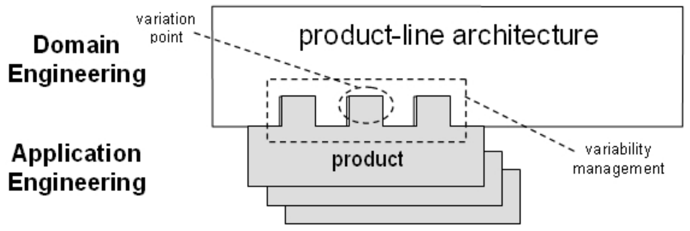

Fig. 1: Domain and application engineering in software product-lines.

Development strategies based on product-lines have proven to be adequate for achieving large-scale software reuse and reduced time-to-market [Bosch 1999]. One of the most common and successful techniques to realize a product-line architecture is to use *(object-oriented) frameworks* [Johnson and Foote 1988; Fayad *et al.* 1999; Bosch 2000]. A framework is a set of classes that constitutes a skeletal software product: it contains holes (or *hot-spots* [Pree 1995]) which must be filled using product-specific code. Thus, a framework can serve as a product-line platform, providing the variation points as hot-spots. In the context of frameworks, both the derived product itself and the product derivation process are often called *specialization*.

Industrial case studies on product-lines [Deelstra *et al.* 2005] report that application engineering can be a time-consuming and expensive activity. This is partly related with the well-known difficulties on framework-based development such as learning the framework [Moser and Nierstrasz 1996], or typical problems in organizations, such as the non-existence of appropriate and up-to-date framework documentation [Bosch 1999; Deelstra *et al.* 2004].

This paper presents a model-driven approach for product derivation in framework-based product-lines. In this approach, in addition to the development of the framework, domain engineers are required to explicitly represent the variation points of the framework through *conceptual variation models* (CVMs) and to use *application-independent models* (AIMs) to describe the variation points within the framework. The former define the set of possible valid configurations by *feature modeling* [Kang *et al.* 1990], while the latter are concise and rigorous representations of the framework hot spots. An application-independent model may include references to elements of the conceptual model. When deriving a product, once the features are defined at the conceptual model, our approach allows to univocally obtain a set of models describing how the framework should be specialized, identifying elements to be developed by application engineers in *application-specific models* (ASMs). We demonstrate the approach using JHotDraw, a popular GUI framework written in Java [SourceForge 2006].

Our approach does not intend to support a full-scale MDA-like (Model Driven Architecture [OMG 2003]) technique, where the application is generated by successive model transformations. In particular, we do not rely on model transformation technologies (like QVT [OMG 2005]), but rather on the much simpler concept of template instantiation. Although some parts of the product-specific code may be automatically generated, we expect that typically the application developer writes the code for product-specific behavior under the guidance of the ASMs. On the other hand, with respect to variability, our approach is more general than MDA: instead of just concentrating on platform variation, we consider any type of variability. Rather than trying to provide a high-level degree of automation, we aim at systematic specification and documentation of the variation points, in a way that it can be directly exploited by an application engineer in the development of a correct specialization.

The main contributions of this paper are (i) a comprehensive model-driven technique for variability management, (ii) the usage of (UML 2.0 [OMG 2004]) template instantiation mechanism in model specialization, and (iii) a demonstration of the technique using a real framework.

The remainder of this paper is organized as follows. Section 2 presents an overview of our approach for model-driven variability management. Section 3 introduces an example of a product-line for the purpose of illustrating the application of the approach in the following sections. In Section 4 we focus on the domain engineering activities and in Section 5 we take the perspective of the application engineer. Section 6 presents related work and we conclude, in Section 7, by discussing the benefits of the proposed approach and future work.

2. Approach overview

This section gives an overview of our approach to variability management. Subsection 2.1 introduces the concept of *specialization template*. Subsection 2.2 describes a layered model for the representation of variability at different abstraction levels and different dimensions within those levels. Subsection 2.3 presents the variability management processes supported by the approach.

2.1 Specialization templates

A central modeling mechanism of our approach is based on the concept of *specialization template*. A specialization template consists of a package containing a class diagram with concrete and template classes, analogously to the concept of *template package* used in the Catalysis approach [D'Souza and Wills 1998] and UML 2.0 [OMG 2004]. A concrete class represents an actual class, while a template class (that is, a generic parameter of the template) is just a placeholder for an actual class. One or more actual classes may be *bound* to a template class, depending on the cardinality value associated with the template class. The cardinality of a template class is by default equal to 1, unless a different value is specified in an attached UML note. This mechanism for multiple bindings of template classes is not included in UML 2.0 and should be considered as an extension to it.

Possible relationships, operations and attributes attached to template classes are interpreted as constraints on the actual classes that are bound to the template classes: the actual classes must have the same attributes and operations, and corresponding relationships. We omit here a more detailed description of the interpretation of such constraints, which falls beyond the scope of this paper.

The classes in a specialization template fall into two categories: *domain* and *specific*. The former belong to the framework, whereas the latter are application-specific. Domain classes (both template and concrete) are denoted with stereotype <<framework>> (as proposed in the UML-F notation [Pree *et al.* 2002]). Classes that are bound to *specific* template classes are stereotyped with the name of the template class, whereas bindings of *domain* template classes are not.

A *template instance* of a specialization template is a model that binds its template classes to actual classes. Figure 2 illustrates an example of a specialization template and a possible template instance of it, using the notation of UML 2.0 as realized by the Rational Software Architect tool [IBM 2006]. On the left-hand side of the figure, there is a specialization template with one concrete class, DomainC, and two template classes, DomainT and SpecificT. The latter is a specific class with cardinality equal to 2, while DomainT is a domain class. On the right-hand side of the figure, there is a template instance that binds DomainT to Z and SpecificT to both X and Y.

Although in this example all the template classes are bound to actual classes, template instances may just bind a part of the template classes. If this is the case, the template instance is a specialization template itself, since it still has template classes.

2.2 Layered variability model

Our approach involves modeling activities at three levels of abstraction supported by different types of models, conducted by different stakeholders: *Conceptual Variation Model* (CVM), *Application-Independent Model* (AIM) and *Application-Specific Model* (ASM). In addition, a fourth layer contains the *Application-Specific Code* (ASC). Figure 3 outlines the relationship between these different artifacts, from the viewpoint of domain and application engineering.

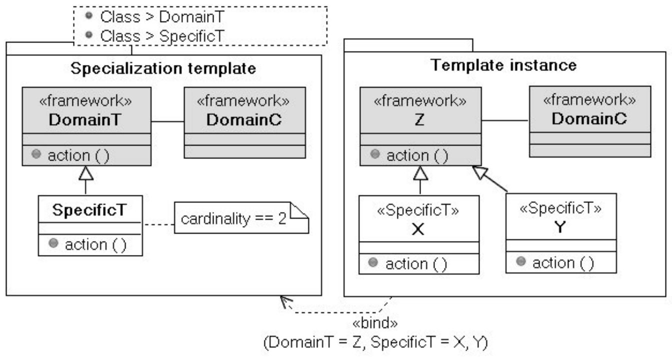

Fig. 2: Example of a specialization template (left) and a possible template instance (right).

- *Conceptual Variation Model* — This type of model is intended to describe the variability supported by a framework at a conceptual level, i.e. independently of the variability mechanisms. CVMs support the description of the variability within a PLA in terms of its features by means of feature modeling (e.g. [Kang *et al.* 1990; Czarnecki *et al.* 2005; Gurp *et al.* 2001; Clauss 2001; Ziadi *et al.* 2003]), giving a high-level view of the variability. Developing CVMs is a task of domain engineers.
- *Conceptual Variation Model Instance* — This type of model supports the description of feature configurations of a CVM. Application engineers define CVM instances for specific products by selecting the appropriated features of the CVM provided by the domain engineers.
- *Application-Independent Model* — This type of model is intended to be used by domain engineers to describe fragments of the architectural variability of a PLA, after having produced the CVM. Each AIM is associated with a CVM feature, implying that if that feature is selected in a CVM instance, the corresponding AIM has to be taken into account in the product derivation. An AIM consists of a specialization template which can have both *domain* and *specific* template classes. For each *domain* template class, the AIM has to specify binding rules associated with the selection of CVM features. Given a CVM instance, these rules must allow to univocally obtain a template instance, where all the *domain* template classes of the AIM are bound.
- *Application-Independent Model Instance* — This type of model describes how the framework should be specialized in the derivation of a given product. Thus, an AIM instance guides the application engineer in achieving certain variability. An AIM instance is a template instance of an AIM that is be automatically obtained using a CVM instance. The binding rules imply the AIM instances to contain only *specific* template classes, which represent application-specific elements that need to be developed by application engineers. Since it has template classes, an AIM instance is a specialization

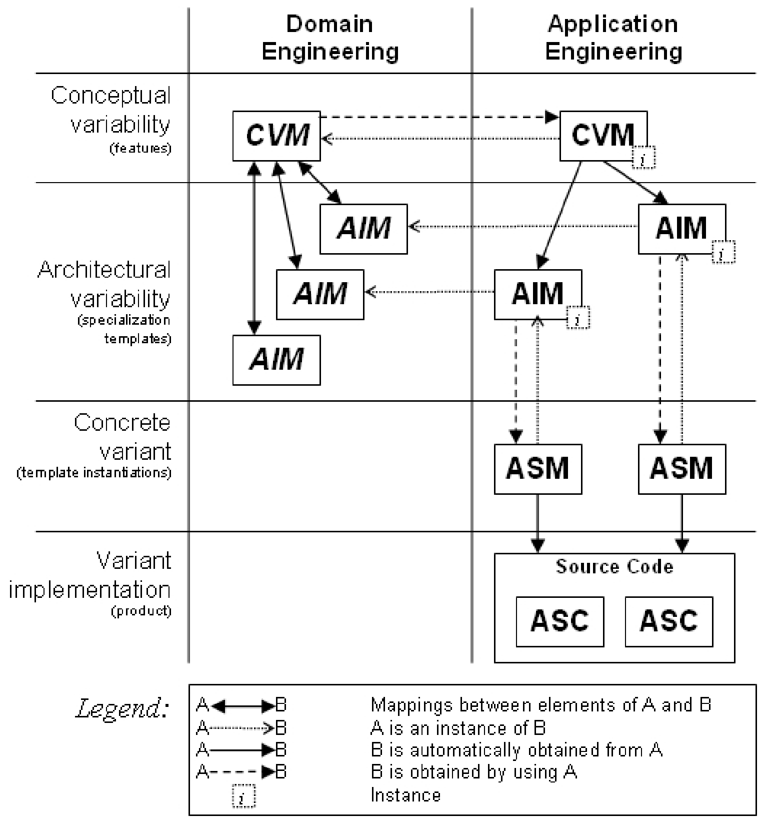
Domain Engineering
Application Engineering
Conceptual variability (features)
Architectural variability (specialization templates)
Concrete variant (template instantiations)
Variant implementation (product)
CVM
CVM
AIM
AIM
AIM
AIM
AIM
ASM
ASM
Source Code
ASC
ASC
Legend:
Mappings between elements of A and B
A is an instance of B
B is automatically obtained from A
B is obtained by using A
Instance

Fig. 3: Model-driven approach for PLA specialization.

template itself. However, it differs from an AIM since it has no *domain* template classes.

◦ *Application-Specific Model* — This type of model is intended to be used by application engineers to describe fragments of a product architecture. An ASM is a template instance of an AIM instance, i.e. a model where all the *specific* template classes are bound to application-specific elements. In a given product derivation, an ASM has to be developed for each of the AIM instances obtained from the CVM instance. These models are intended to be able to generate the skeleton of the product's implementation.

◦ *Application-Specific Code* — This corresponds to specific source code that application engineers have to develop in a product derivation, since ASMs are not intended to generate the complete implementation of a product. ASC should be added at well-identified locations in the source code generated from ASMs. Typically, these locations correspond to method bodies, which were generated with an empty or default implementation.

2.3 Variability management processes

In our approach, activities related to variability management are located at different levels of abstraction, starting from the construction of a CVM and ending with the production of the executable code of a software product. Figure 4 presents the different phases of the variability management process, assuming that a PLA, developed by domain engineers, is already available.

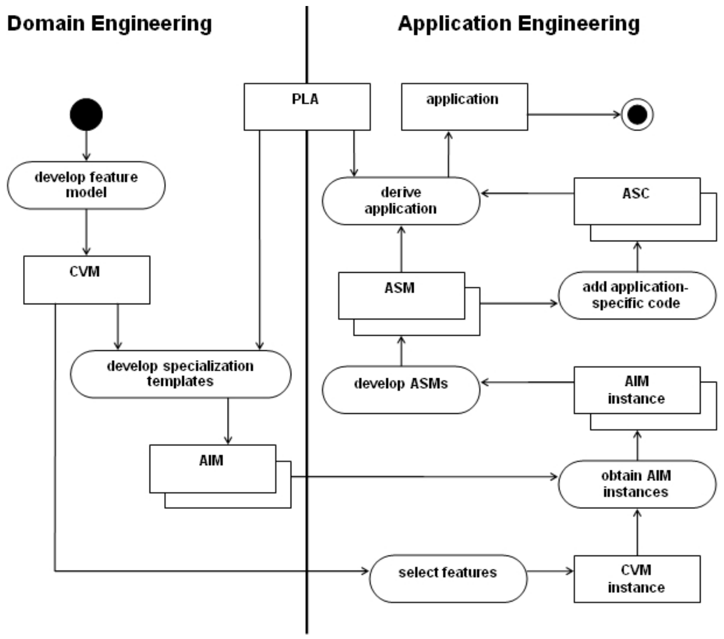

Fig. 4: The processes of specifying PLA variability (domain engineering), and product derivation (application engineering).

Domain engineers develop a CVM describing the features that the product-line supports. Then, they develop a set of AIMs describing specialization templates for the PLA.

From the application engineering viewpoint, the CVM is used to define a CVM instance by selecting the desired features. Given the feature configuration (CVM instance) and the set of AIMs, a set of AIM instances is automatically obtained. Application engineers can then introduce application specific elements by developing an ASM for each of the AIM instances. The result is then a set of ASMs, used for the partial generation of the product's implementation. Finally, the full product can be obtained by adding ASC in the proper locations.

3. Example of framework-based product-line

We will use an existing, well-known framework to demonstrate our approach. The framework is JHotDraw [SourceForge 2006], a Java GUI framework for applications that deal with technical and structured graphics. Its design relies heavily on some design patterns (e.g. GoF catalog [Gamma *et al.* 1995]), and it is considered by the academia as an example of good design. For the sake of simplification, some details of the framework are omitted or simplified in the remainder of the paper.

The left-hand side of Figure 5 illustrates the typical UI layout of the applications developed using JHotDraw, where some variable parts are marked with a circle. The illustrated variable parts are the left-hand side *palette*, which can be customized with application-specific *tools*, and the *menu bar* where application-specific *menus* can be included. In addition, consider also that an application can vary between having a single drawing pane or multiple sub-windows.

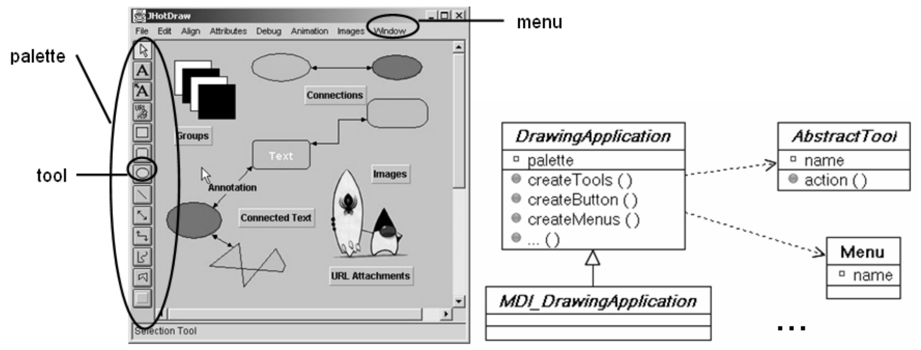

Fig. 5: Examples of variable parts of the JHotDraw framework (left) and relevant classes (right).

The right-hand side of Figure 5 presents some of JHotDraw's classes that are relevant for the adaptation of the variable parts in specializations. Considering the JHotDraw framework as a PLA realization, let us assume the following list of application engineering goals and the tasks required for achieving these goals:

(1) *To have either a single or multi-window application.* The former requires the application's main class to be an extension of DrawingApplication, whereas the latter requires an extension of MDI_DrawingApplication.

(2) *To have an application-specific type of tool in the palette.* This requires a concrete class that extends AbstractTool, and the extension of DrawingApplication (main class) to override the default method createTools() for including the tool.

(3) *To have an application-specific menu in the menu bar.* This requires the application's main class to override the default method createMenus(), in order to plug instances of Menu.

These goals are used in the following sections for describing the product derivation using CVMs, AIMs, and ASMs.

4. Domain engineering

This section illustrates our model-driven approach from a domain engineering viewpoint. As explained previously, domain engineers are responsible for developing a CVM and a set of AIMs for the PLA. Using the JHotDraw example, a CVM is first given in subsection 4.1, and two AIMs are given in subsection 4.2.

4.1 Conceptual Variation Model (CVM)

We adopted a notation for CVMs based on cardinality-based feature models [Czarnecki *et al.* 2005]. A feature model may have several valid configurations, which can be inferred by analyzing it having the root feature as a starting point. Figure 6 presents an example of a CVM for JHotDraw, considering the variable parts that were described previously.

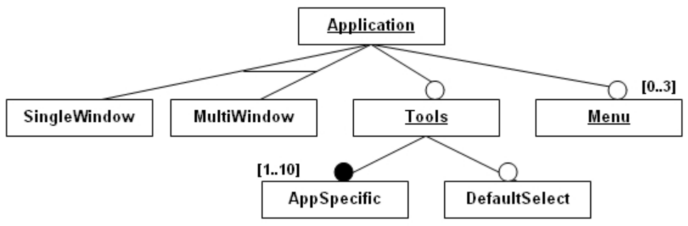

Fig. 6: CVM for the JHotDraw.

Application is the root feature. *SingleWindow* and *SingleWindow* form a group of features. Groups have associated constraints that constrain the possible selection of their features. In this case, a constraint (denoted by the link that joins the two links) implies that exactly one of the associated features can be selected in a feature configuration (*SingleWindow* or *SingleWindow*, but not both). An unfilled circle denotes that a feature is optional (e.g. *Menu*), whereas a filled circle denotes a compulsory feature (e.g. *AppSpecific*). Features have an associated cardinality, which is by default equal to [0..1] for optional features and equal to [1] for compulsory features. Having knowledge about the domain and reading the diagram, we infer that (i) an application may be either single window or multi-window, (ii) there is the possibility of having tools, (iii) if there are tools, there must exist at least one application-specific tool, (iv) the default select tool is optional, and (v) an application may have a maximum of three application-specific menus.

An underlined feature denotes that it has an associated AIM. For simplicity, we assume that the AIM has the same name as the feature.

4.2 Application-Independent Models (AIMs)

For the CVM of Figure 6, we consider in this subsection two AIM templates associated with the features *Application* and *Tools*.

An AIM is a specialization template described by a UML package with stereotype <<AIM>> and name equal to the feature that it is associated to.

Associated with the CVM root, the AIM *Application* is presented in Figure 7. We can see two template classes, Type (domain) and Application (specific). Type refers to the main class that implements a generic application (which can be either single or multi window), and it has a default value defined in the signature of the template parameter. A UML note is attached to Type, containing a binding rule stating that if *SingleWindow* is a selected feature then the concrete class DrawingApplication should be bound to it. Otherwise if *MultiWindow* is selected, the default class is bound instead. Application refers to the main class for implementing the application.

Take into account that several binding rules involving a same CVM feature can be attached to different classes, implying that the selection of a single feature affects more than one class in a product derivation.

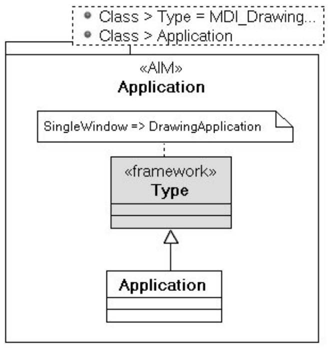
Class > Type = MDI_Drawing...
Class > Application
«AIM»
Application
SingleWindow => DrawingApplication
«framework»
Type
Application

Fig. 7: AIM *Application*.

Given a CVM instance, Type can be automatically bound. When having all the domain template classes bound, we are in the presence of an AIM instance. This issue is covered in next section.

Figure 8 presents the AIM associated with the feature *Tools*. The class name in the form *"AIM-name::AIM-element"* denotes a composition relationship. For instance, in the case of Application::Type, it means that this element is not an element itself, but instead, it is a reference to the element Type of the AIM *Application*. Composition relationships imply that the same concrete class is bound to the original element and its references. References are also made explicit in template parameters. However, their binding can be automatic as soon as the element that is being referenced is bound. This issue is illustrated with more detail in the next section.

The template class SpecificTool has a template integer card. We can see a binding rule *"AppSpecific.card => card"*, stating that card should be bound to an integer equal to the cardinality of the feature *AppSpecific*.

We also allow domain-specific dependencies in the specialization templates. In the example (Figure 8), there is a domain-specific dependency <<add>>, expressing that for each binding of SpecificTool there must exist a call in createTools() for adding an instance of the bounded class (denoted by $SpecificTool$) in the application palette. Here we simply assume that an attached UML note defines the semantics of the domain-specific dependency.

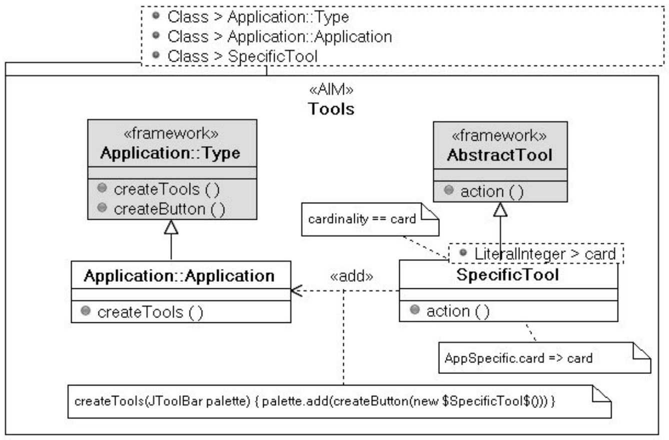

Fig. 8: AIM *Tools*

5. Application engineering

The previous section presented the models developed by domain engineers for a part of JHotDraw. In this section, the development tasks of application engineers are detailed, using the CVM and both AIMs of the previous section. First, a CVM instance is presented in subsection 5.1, while subsection 5.2 presents both the AIM instances that result from the selected features and an example of possible ASMs. Finally, subsection 5.3 presents the code that is able to be generated from the ASM models.

5.1 CVM instance

The application engineer uses the CVM to obtain a CVM instance, i.e. a configuration of features for a particular product. Figure 9 presents an instance of the CVM of Figure 6, where the darker elements indicate the selected features. Notice also that the cardinality of the feature *AppSpecific* is defined. Having this configuration for a specialization, we can infer that two AIMs corresponding to the features *Application* and *Tools* are going to be included in the product derivation (notice the underlined features).

Having a CVM instance, a set of AIM instances can be automatically obtained, in order that application engineers can proceed to the development of ASMs. The next subsection details these issues.

5.2 AIM instances and Application-Specific Models (ASMs)

In the sequel, we explain the mechanisms for obtaining AIM instances from the AIMs, and the development of ASMs. More precisely, we will derive two ASMs (for *Application* and *Tools*) corresponding to the feature choices represented by the CVM instance given in the previous subsection (Figure 9). Figure 10 illustrates

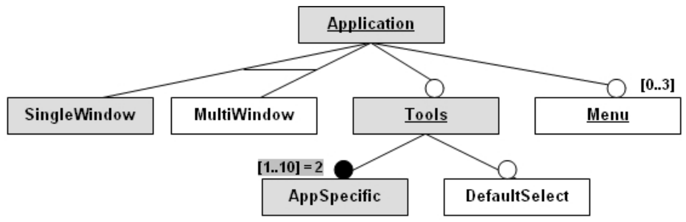

Fig. 9: CVM instance (possible feature configuration for the CVM of Figure 6)

both cases, including the (simplified) AIMs defined by domain engineers. Recall that application engineers only deal with AIM instances.

Considering first the *Application* feature, we can see the AIM instance *Application* obtained from the corresponding AIM, with the template class Type bound to DrawingApplication, according to the CVM instance (*SingleWindow* is selected in Figure 9). The AIM instance leaves the template class Application to be bound in the ASM, which we can see below, binding it to the application-specific class MyApplication.

Considering the *Tools* feature, we can see the AIM instance *Tools*, where the template class reference Application::Type is automatically bound to DrawingApplication, i.e. the same element that is bound to the referenced element, and the template integer card of SpecificTool bound to 2, according to the cardinality defined for the feature *AppSpecific*. Finally, we can see an ASM *Tools* for the AIM instance, where the template class reference Application::Application is bound automatically to MyApplication. This ASM introduces the application-specific elements StarTool and ArrowTool, and as required, the domain-specific dependencies <<add>> are defined. The semantics of this type of dependency is defined in the corresponding AIM, and is transparent to the application engineers.

Having the presented models, the specialization process would be complete in terms of modeling. The ASMs resulting from a specialization plus the associated AIMs, are intended to be able to generate the application's structural code (like class skeletons), as well as behavioral code imposed by the domain-specific dependencies.

5.3 Application-Specific Code (ASC)

ASC corresponds to code that application engineers have to give in addition to the code that can be generated from the modeling abstractions. Figure 11 shows the source code (in Java) that could be generated from the given specialization example. Annotation-like comments indicate the parts generated from the models. In the case when the ASMs of the product derivation have domain-specific dependencies, code can be filled in certain method bodies, according to what is specified in the AIMs. For instance, notice the statements that were included in the method createTools(...), resulting from the domain-specific dependencies <<add>> of ASM *Tools* (Figure 10), according to the semantics defined in AIM *Tools* (Figure 8).

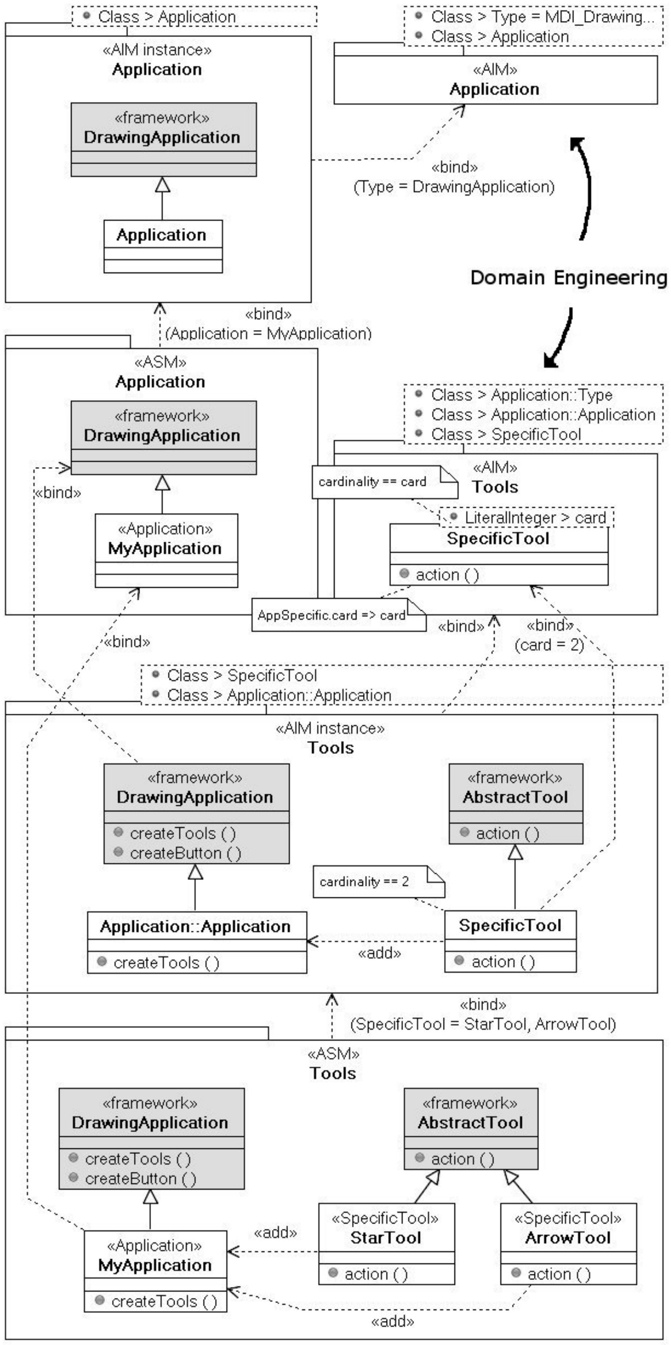

Fig. 10: AIM instances and ASMs for the features *Application* and *Tools*.

The *TODO* comments mark the locations in the generated code where the application engineer should add ASC.

```
@Application
public class MyApplication extends DrawingApplication {

   @Tools
   public void createTools(JToolBar palette) {
      palette.add(createButton(new StarTool()));
      palette.add(createButton(new ArrowTool()));
   }
}

@Tools
public class StarTool extends AbstractTool {
   public void action() {
       // TODO: ASC
   }
}

@Tools
public class ArrowTool extends AbstractTool {
   public void action() {
       // TODO: ASC
   }
}
```

Fig. 11: Specialization code resulting from the example product derivation.

6. Related work

A feature modeling notation was originally proposed in the Feature-Oriented Domain Analysis (FODA) method [Kang *et al.* 1990], while was posteriorly extended and adapted in other approaches such as *cardinality-based feature models* [Czarnecki *et al.* 2005], *feature graphs* [Gurp *et al.* 2001], and *UML-based feature models* [Clauss 2001; Ziadi *et al.* 2003]. The decision of adopting cardinality-based feature modeling was mainly motivated by the possibility to specify cardinalities, which are useful in our approach.

The work in [Hautamäki and Koskimies 2005] introduces the concept of *specialization pattern*. This type of patterns are a variant of design patterns, especially intended for describing framework extension points related to high-level specialization goals. This approach supports the development of framework specializations at the code level, using a tool for specifying patterns and posteriorly provide assistance for pattern instantiation in a task-based manner.

In [Selonen and Xu 2003], an approach for validating UML models against *architectural profiles* is presented. Architectural profiles are extended UML profiles for describing architectural constraints and rules for a given domain. The main goal of this approach is to check conformance of UML class diagrams against architectural profiles, and it is argued that the approach is helpful for enforcing conventions of product-line architectures.

A generalization of [Selonen and Xu 2003] was proposed in [Hammouda *et al.* 2005], introducing the concept of *design profile*. Design profiles are composed by *design forms*, *composition forms* and *global forms*. A design form, given as UML class diagram, is a pattern-like description of structural properties that must be satisfied by models that are intended to conform to the design profile. Since the binding of pattern roles from different design forms may overlap classes, there is the mechanism of composition forms, which defines role overlapping relationships. In this way, different pattern roles that have a composition relationship are bound to the same concrete class in different design form instantiations. We are currently analyzing the suitability of this approach as a basis for tool support for product derivations using our layered variability model.

In [Czarnecki and Antkiewicz 2005], a template-based approach is presented for mapping feature models to representations of variability in UML models. Our approach also applies a template-based mechanism in AIMs, however, focusing on exploiting the specific characteristics of framework specialization patterns.

The work in [Antkiewicz and Czarnecki 2006], proposes the definition of *Framework-Specific Modeling Languages (FSMLs)* on top of object-oriented frameworks. The approach is based on the definition of mappings of the abstract syntax of a FSML to the framework API. Similarly to the binding rules and domain-specific dependencies of our approach, these mappings are intended to produce only a part of the specializations' code. This approach is also focused on the difficulty of framework usage, and can be seen as an alternative to our approach.

COVAMOF [Sinnema *et al.* 2004] is an approach for managing the variability provided by a software product-line, allowing the representation of variability at different abstraction layers. COVAMOF focuses on the configurability of the architecture assets, having for instance specific abstractions for modeling dependencies. Our approach focuses on the specifics of framework-based product-lines and their specialization mechanisms, and intends to support variability which implies that specializations develop application-specific components for extending the PLA. COVAMOF does not seem to address these issues as a primary concern.

OMG's MDA initiative [OMG 2003] aims to introduce a paradigm shift in software development, where a system is built using the following types of models: the Computation Independent Model (CIM), the Platform Independent Model (PIM), and Platform Specific Models (PSMs). A CIM describes the system from a computation-independent viewpoint – it does not show details of the structure of the system, and it is close to the application domain concepts. A PIM describes the implementation of a system, independently of the technology to be adopted. In turn, a PSM describes the implementation of a system considering a specific technology (e.g. J2EE, .NET). PSMs for different platforms are obtained from the PIM through transformation rules.

MDA concentrates on one aspect of variability – the variation of the implementation platform. Our approach to product-line engineering addresses the problem of handling other types of variations, namely the variable parts within the product family. Our work is an MDA approach in the sense that it allows model-based variability management of PLAs, considering different feature dimensions that are relevant to specializations. We propose a layered model that could be situated between MDA's CIM and PIM, focusing on the architectural variation of a product-line.

Our approach could "fit" in MDA's model layering, in such a way that the sets of ASMs would play the role of the PIM. The platform variability could be then achieved by using those ASMs to obtain PSMs for different platforms. On the other hand, since many framework-based PLAs are already dependent on a technology (e.g. Java, C++, C#), platform variation through PIM to PSMs transformations does not seem to be applicable in these cases. However, for instance when we have a framework that has different implementations in several technologies (e.g. CORBA), deriving different types of PSMs in order to achieve platform variability in specializations becomes appealing. The benefits of combining MDA with configurable product families are outlined in [Deelstra *et al.* 2003].

7. Discussion and further work

Our approach does not aim to provide full-scale MDA technique, since ASC has to be developed for having complete implementation of specializations. However, it addresses the fundamental problems related to PLA learning and development of (correct) specializations.

A main benefit of our approach is that it makes explicit the specialization interface of a framework by modeling the variation points on a higher abstraction level. Thus, we expect that this approach facilitates and speeds up the specialization process. Even without tool support, this approach can be used to systematically document the variation supported by a framework. Although the precise form of the binding rules and domain-specific dependencies are still on-going work, the approach can be applied manually as shown in the example. The concept of a specialization template is basically simple, closely matching the intuitive idea of a generic structure familiar to many programmers.

Another important benefit is the support for correct specializations, assuming that there is suitable tool support. For example, enforcing ASMs to be template instances of AIM instances is advantageous for reducing incorrect specializations and imposing certain architectural rules. A tool can also assist in the development of ASMs. Since AIMs are formal descriptions, the required bindings of template classes can be inferred from the model and ASM elements can be generated semi-automatically, leaving the variable issues such as class names to be defined by the application engineers. Tool support for achieving this is currently under development.

In terms of documentation, CVMs can be simultaneously a documentation artifact which gives a conceptual overview of the variability offered by the product-line, and a development artifact for specializations. Nevertheless, a CVM instance also constitutes specializations' documentation that describes the adopted product-line features.

AIMs can constitute a significant part of PLA documentation for using its variability mechanisms, and serve simultaneously as a development artifact for application engineers. On the other hand, the resulting ASMs of the modeling activities serve as development artifacts and constitute a significant part of the documentation of a specialization.

References

Antkiewicz, Michal and Czarnecki, Krzysztof. 2006. Framework-specific modeling languages with round-trip engineering. In *MoDELS*.

Bosch, Jan. 1999. Product-line architectures in industry: a case study. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*.

Bosch, Jan. 2000. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co.

Bosch, Jan, Florijn, Gert, Greefhorst, Danny, Kuusela, Juha, Obbink, J. Henk, and Pohl, Klaus. 2002. Variability issues in software product lines. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*.

Clauss, Matthias. 2001. Modeling variability with UML. In *Net.ObjectDays*.

Czarnecki, Krzysztof and Antkiewicz, Michal. 2005. Mapping features to models: a template approach based on superimposed variants. In *GPCE*.

Czarnecki, Krzysztof, Helsen, Simon, and Eisenecker, Ulrich W. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice 10*, 1, 7–29.

Deelstra, Sybren, Sinnema, Marco, and Bosch, Jan. 2004. Experiences in software product families: problems and issues during product derivation. In *Proceedings of the Third International Software Product Line Conference (SPLC)*.

Deelstra, Sybren, Sinnema, Marco, and Bosch, Jan. 2005. Product derivation in software product families: a case study. *Journal of Systems and Software 74*, 173 – 194.

Deelstra, Sybren, Sinnema, Marco, van Gurp, Jilles, and Bosch, Jan. 2003. Model driven architecture as approach to manage variability in software product families. In *Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications*.

D'Souza, Desmond and Wills, Alan Cameron. 1998. *Objects, components and frameworks with UML: the Catalysis approach*. Addison-Wesley.

Fayad, Mohamed E., Schmidt, Douglas C., and Johnson, Ralph E. 1999. *Building application frameworks: object-oriented foundations of framework design*. John Wiley & Sons, Inc.

Gacek, Critina and Anastasopoulos, Michalis. 2001. Implementing product line variabilities. In *SSR '01: Proceedings of the 2001 Symposium on Software Reusability*.

Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc.

Gurp, Jilles Van, Bosch, Jan, and Svahnberg, Mikael. 2001. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*.

Hammouda, Imed, Pussinen, Mika, Ruokonen, Anna, Koskimies, Kai, and Systä, Tarja. 2005. Design profiles: specifying and using structural patterns in UML. In *NWUML '05: The 3rd Nordic Workshop on UML and Software Modeling*.

Hautamäki, Juha and Koskimies, Kai. 2005. Finding and documenting the specialization interface of an application framework. *Software: Practice and Experience (Electronic version)*, DOI 10.1002/spe.728.

IBM. 2006. Rational Software Architect. http://www-306.ibm.com/software/awdtools/architect/swarchitect/.

Johnson, Ralph E. and Foote, Brian. 1988. Designing reusable classes. *Journal of Object-Oriented Programming 1*, 22–35.

Kang, K., Cohen, S., Hess, J., Nowak, W., and Peterson, S. 1990. Feature-oriented domain analysis (FODA) feasibility study. Tech. report, Carnegie Mellon University.

Krueger, Charles W. 2006. Introduction to software product lines. http://www.softwareproductlines.com/.

Moser, Simon and Nierstrasz, Oscar. 1996. The effect of object-oriented frameworks on developer productivity. *Computer 29*, 45–51.

OMG. 2003. *MDA Guide Version 1.0.1*.

OMG. 2004. *UML Superstructure Specification, v2.0*.

OMG. 2005. *MOF 2.0 QVT : Queries / Views / Transformations*.

Pree, Wolfgang. 1995. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co.

Pree, Wolfgang, Fontoura, Marcus, and Rumpe, Bernhard. 2002. Product line annotations with UML-F. In *Proceedings of the Second International Conference on Software Product Lines (SPLC)*.

Selonen, Petri and Xu, Jianli. 2003. Validating UML models against architectural profiles. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of Software Engineering*.

Sinnema, Marco, Deelstra, Sybren, Nijhuis, Jos, and Bosch, Jan. 2004. COVAMOF: a framework for modeling variability in software product families. In *Proceedings of the Third International Software Product Line Conference (SPLC)*.

SourceForge. 2006. JHotDraw framework. http://www.jhotdraw.org.

Ziadi, Tewfik, Hélouët, Loïc, and Jézéquel, Jean-Marc. 2003. Towards a UML profile for software product lines. In *PFE: 5th International Workshop Software Product-Family Engineering*.