Journal of Universal Computer Science, vol. 14, no. 8 (2008), 1344-1367
submitted: 15/11/07, accepted: 25/3/08, appeared: 28/4/08 © J.UCS

A Product Derivation Tool Based on Model-Driven Techniques and Annotations

Elder Cirilo
(Pontifical Catholic University of Rio de Janeiro, Brazil
ecirilo@inf.puc-rio.br)

Uirá Kulesza
(CITI/DI/FCT - New University of Lisbon – Portugal
Recife Center for Advanced Studies and Systems – Brazil
uira@di.fct.unl.pt, uira@cesar.org.br)

Carlos José Pereira de Lucena
(Pontifical Catholic University of Rio de Janeiro, Brazil
lucena@inf.puc-rio.br)

Abstract: In this paper, we present a model-based tool for product derivation. Our tool is centered on the definition of three models (feature, architecture and configuration models) which enable the automatic instantiation of software product lines (SPLs) or frameworks. The Eclipse platform and EMF technology are used as the base for the implementation of our tool. A set of specific Java annotations are also defined to allow generating automatically many of our models based on existing implementations of SPL architectures. We illustrated the use and validation of our tool in the preparation of the automatic derivation of the JUnit framework and a J2ME games product line.

Key Words: Software Product Lines, Product Derivation Tools, Generative Programming, Model-Driven Development
Categories: D.2.3, D.2.13

1 Introduction

Over the last years, many approaches for the development of system families and software product lines have been proposed [Weiss and Lai, 1999] [Clements and Northrop, 2001] [Czarnecki and Eisenecker, 2000] [Greenfield and Short, 2005]. A system family [Parnas, 1976] is a set of programs that shares common functionalities and maintain specific functionalities that vary according to specific systems being considered. A software product line (SPL) [Clements and Northrop, 2001] can be seen as a system family that addresses a specific market segment. Software product lines and system families are typically specified, modeled and implemented in terms of common and variable features. A feature [Czarnecki et al., 2006] is a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in SPLs.

Most of the existing SPL approaches [Weiss and Lai, 1999] [Clements and Northrop, 2001] [Czarnecki and Eisenecker, 2000] [Greenfield and Short, 2005] motivate the definition of a flexible and adaptable architecture that addresses the

common and variable features of the SPL. These SPL architectures are implemented by defining or reusing a set of different artifacts, such as object-oriented frameworks and software libraries. Recently, new programming techniques have been explored to modularize the SPL features, such as, aspect-oriented programming [Alves et al., 2005] [Kulesza, 2007], feature-oriented programming [Smaragdakis and Batory, 2002] and code generation [Czarnecki and Eisenecker, 2000] [Stahl and Voelter, 2006]. Typical implementations of SPL architectures are composed of a set of different assets (such as, code artifacts), each of them addressing a small set of common and/or variable features.

Product derivation [Deelstra et al., 2005] refers to the process of constructing a product from the set of assets specified or implemented for a SPL. Ideally the product derivation process would be accomplished with the help of instantiation tools to facilitate the selection, composition and configuration of SPL code assets and their respective variabilities. Over the last years, some product derivation tools have been proposed. Gears [Gears, 2008] and Pure::variants [pure::variants, 2008] are two examples of modern industrial tools developed in this context. Although these tools offer a set of useful functionalities for the product derivation, they are in general complex and heavyweight to be used by the mainstream developer community, because they incorporate a lot of new concepts from the SPL development area. As a result, they suffer from the following deficiencies: (i) difficult to prepare existing SPL architecture implementations to be automatically instantiated; (ii) definition of many complex models and/or functionalities; and (iii) they are in general more adequate to work with proactive approaches [Krueger, 2003].

In this context, this paper proposes a model-driven product derivation tool, called GenArch, centered on the definition of three models (feature, architecture and configuration). Initial versions of these three models can be automatically generated based on a set of code annotations that indicate the implementation of features and variabilities in the code of artifacts from the SPL. After that, a domain engineer can refine or adapt these initial model versions to enable the automatic product derivation of a SPL. The Eclipse [Shavor et al., 2003] platform and other model-driven development toolkits available, such as EMF and oAW, were used as a base for the definition of our tool. We illustrated the use of GenArch tool in the instantiation of two case studies: (i) the JUnit testing framework; and (ii) a J2ME games product line.

The remainder of this paper is organized as follows. Section 2 presents background of generative programming and existing product derivation tools. Section 3 gives an overview of our product derivation approach based on the combined use of models and code annotations. Section 4 details the GenArch tool by describing its architecture, annotations, models and derivation process. The JUnit framework is used to illustrate the main functionalities of GenArch along the Section 4. Section 5 describes the use of the tool in a J2ME games product line. Section 6 presents a set of lessons learned from the implementation and use of the GenArch tool in the case studies. Finally, Section 7 concludes the paper and provides directions for future work.

2 Background

This section briefly revisits the generative programming approach (Section 2.1). Our SPL derivative approach is defined based on its original concepts and ideas. We also give an overview of existing product derivation tools (Section 2.2).

2.1 Generative Programming

Generative Programming (GP) [Czarnecki and Eisenecker, 2000] addresses the study and definition of methods and tools that enable the automatic generation of software from a given high-level specification language. It has been proposed as an approach based on domain engineering [Arrango, 1994]. GP promotes the separation of problem and solution spaces, giving flexibility to evolve both independently. To provide this separation, Czarnecki & Eisenecker [Czarnecki and Eisenecker, 2000] propose the concept of a generative domain model. A generative domain model is composed of three basic elements: (i) problem space – represents the concepts and features existent in a specific domain; (ii) solution space – consists of the software architecture and components used to build members of a software family; and (iii) configuration knowledge – defines how specific feature combinations in the problem space are mapped to a set of software components in the solution space. GP advocates the implementation of the configuration knowledge by means of code generators.

The fact that GP is based on domain engineering enables us to use domain engineering methods [Arrango, 1994] [Czarnecki and Eisenecker, 2000] in the definition of a generative domain model. Common activities encountered in domain engineering methods are: (i) domain analysis – is concerned with the definition of a domain for a specific software family and the identification of common and variable features within this domain; (ii) domain design – concentrates on the definition of a common architecture and components for this domain; and (iii) domain implementation – involves the implementation of architecture and components previously specified during domain design.

Two new activities [Czarnecki and Eisenecker, 2000] need to be introduced to domain engineering methods in order to address the goals of GP, such as: (i) development of a proper means to specify individual members of the software family through the specification of domain-specific languages (DSLs); and (ii) modeling of the configuration knowledge in detail in order to automate it by means of a code generator.

In a particular and common instantiation of the generative model, the feature model is used as a domain-specific language of a software family or product line. It works as a configuration DSL. A configuration DSL allows to specify a concrete instance of a concept [Czarnecki and Eisenecker, 2000]. Several existing tools adopt this strategy to enable automatic product derivation (Section 2.2) in SPL development. In this work, we present an approach and a tool centered on the ideas of the generative model. The feature model is also adopted by our tool as a configuration DSL which expresses the SPL variabilities.

2.2 Existing Product Derivation Tools

There are many tools available in industry to automatically derive SPL members. The main available tools based on feature models are Pure::variants and Gears. Pure::variants [pure::variants, 2008] is a SPL model-based product derivation tool. Its modeling approach comprises mainly two models: the feature and the family model. The feature model contains the product variabilities (features). The family model describes the internal structure of the individual components and their dependencies on the features. The family model is structured in several levels. The highest level is formed by the components. Each component represents one or more functional features of the solutions and consists of logical parts of the software (classes, objects, functions, variables, documentation). The physical elements can be files that already exist, files that will be created and actions that will be performed based on the variant knowledge. The feature model can be viewed graphically in different formats such as trees, tables and diagrams. Constraints among features and architecture elements are expressed using first order logic in Prolog using expression syntax closely related to Object Constraint Language (OCL) notation. This tool permits the use of an arbitrary number of feature models, and hierarchical connection of the different models. Product derivation is done by selecting the desired feature in a variant model. Pure::variants checks the selection interactively, and solves problems or reports them if they cannot be solved automatically. A product line specification in this tool can consist of any number of models, a "configuration space" is used to manage this information and captures variants. This space provides an overview over variabilities and commonalities between product variants. The Pure::variants does not require any specific implementation technique and provides powerful integration interfaces, through XML-based exchange format, with other tools, e.g. for requirements engineering, test management and code generation.

Gears [Gears, 2008] provides the infrastructure and a development environment for creating a product line, allowing the definition of a generative model focused on automatic product derivation. It is based on an incremental approach that allows start small – with or two products, subsystems or assets, teams or individual - enabling an ease transition, into manageable increments, to product line engineering, This approach comprises three key abstractions: Software Assets, Product Feature Profiles and Gears Configurator. The Software Assets are configurable software artifacts (source code, requirements, and test cases) engineered to be reused across the product line. Each product in the portfolio is modeled in the Product Feature Profiles. The Product Feature Profiles has three major elements: feature declarations, product definitions, and variation points. Feature declarations are parameters that express the variations. It enables the product line modeling in terms of optional and varying feature. The language for expressing constraints at feature models is propositional logic instead of full first-order logic. Product definitions are used to select and assign values to the feature declaration parameters for the purpose of instantiating a product. Variation points encapsulate the variations in the SPL and map the feature declarations to choices at these variation points. Guided by the Product Feature Profiles, the Gears Configurator automatically assembles and configures the software assets to produce the products.

3 Approach Overview

In this section, we present an overview of our product derivation approach based on the use of GenArch tool. Next section details the tool by showing its architecture, adopted models and supporting technologies. Our approach aims to provide a product derivation tool that enables the mainstream software developer community to use the concepts and foundations of the SPL approach, without the need to understand complex concepts or models from existing product derivation tools.

Figure 1 gives an overview of our approach. Initially (step 1), the domain engineers are responsible to annotate the existing code of SPL architectures (e.g. an object-oriented framework). We have defined a set of Java annotations to be inserted in the implementation elements (classes, interfaces and aspects) of SPL architectures. Although the current version of GenArch tool has been developed to work with Java technologies, our approach is neutral with respect to the technology used. It only requires that the adopted technologies provide support to the definition of GenArch annotations and models. The purpose of our annotations is twofold: (i) they are used to specify which SPL implementation elements correspond to specific SPL features; and (ii) they also indicate that SPL code artifacts, such as an abstract class or aspect, represent an extension point (hot-spot) of the architecture.

After that, the GenArch tool processes these annotations and generates initial versions of the derivation models (step 2). Three models must be specified in our approach to enable the automatic derivation of SPL members: (i) an architecture model; (ii) a feature model; and (iii) a configuration model. The architecture model defines a visual representation of the SPL implementation elements (classes, aspects, templates, configuration and extra files) in order to relate them to feature models. It can be automatically created by parsing an existing directory containing the implementation elements (step 2). Code templates can also be created in the architecture model to specify implementation elements that have variabilities to be solved during application engineering. Initial versions of code templates are also automatically created in the architecture models based on GenArch annotations (see details in Section 4.2).

Feature models [Kang et al., 1990] are used in our approach to represent the variable features (variabilities) from SPL architectures (step 3). During application engineering, application engineers create a feature model instance (also called a configuration [Czarnecki et al., 2004]) in order to decide which variabilities are going to be part of the final application generated (step 4). Finally, our configuration model is responsible to define the mapping between features and implementation elements. It represents the configuration knowledge from a generative approach [Czarnecki and Eisenecker, 2000], being fundamental to link the problem space (features) to the solution space (implementation elements). Each annotation embedded in an implementation element is used to create in the configuration model, a mapping relationship between an implementation element and a feature.

The initial versions of the derivation models, generated automatically by GenArch tool, must be refined by domain engineers (step 3). During this refinement process, new features can be introduced in the feature model or the existing ones can be reorganized. In the architecture model, new implementation elements can also be introduced or they can be reorganized. Template implementations can include

additional common or variable code. Finally, the configuration model can also be modified to specify new relationships between features and implementations elements. In the context of SPL evolution, the derivation models can be revisited to incorporate new changes or modifications according to the requirements or changes required by the evolution scenarios.

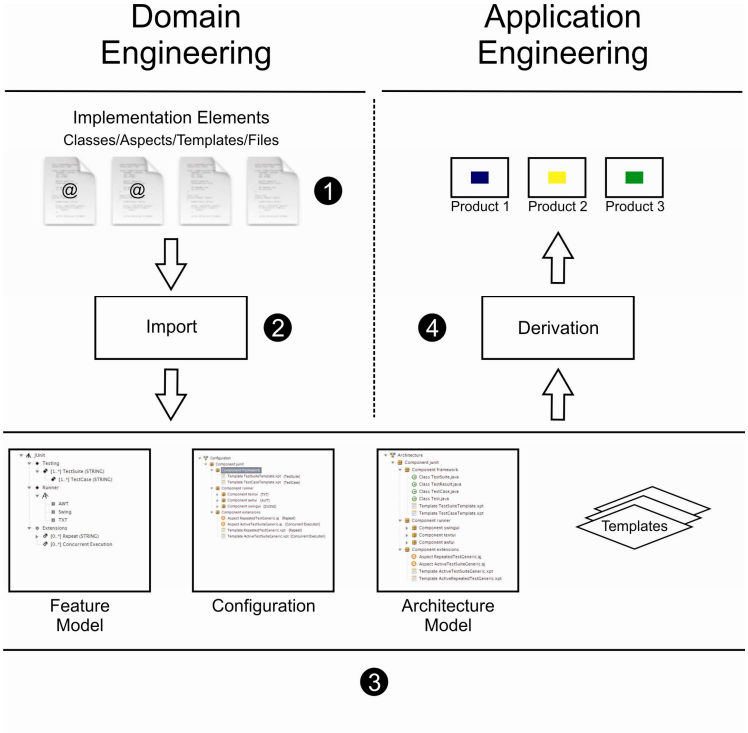


Figure 1: Approach Overview

After all models are refined to represent the implementation and variabilities of a SPL architecture, the GenArch tool uses them to automatically derive an instance/product of the SPL (step 4). The tool processes the architecture model by verifying if each implementation element depends on any feature from the feature model. This information is provided by the configuration model. If an implementation element does not depend on a feature, it is automatically instantiated since it can be seen as a mandatory feature. If an implementation element depends on a feature, it is only instantiated if there is an occurrence of that specific feature in the feature model instance created by the application engineer. Every template element from the architecture model, for example, must always depend on a specific feature. The information collected by that feature is then used in the customization of the template. The GenArch tool produces, as result of the derivation process, an Eclipse/Java project containing only the implementation elements corresponding to the specific

configuration expressed by the feature model instance, that was specified by the application engineers.

4 GenArch – Generative Architecture Tool

In this section, we present the architecture, adopted models and technologies used in the development of GenArch tool. Following subsections detail progressively the functionalities of the tool by illustrating its use in the instantiation of the JUnit framework.

4.1 Architecture Overview

The GenArch tool has been developed as an Eclipse plug-in [Shavor et al., 2003] using different technologies available at this platform. New model-driven development toolkits, such as Eclipse Modeling Framework (EMF) [Budinsky et al., 2004] and openArchitectureWare (oAW) [openArchitectureWare, 2008] were used to specify its models and templates, respectively. Figure 2 shows the general structure of GenArch architecture based on Eclipse platform technologies. Our tool uses the JDT (Java Development Tooling) API [Shavor et al., 2003] to browse the Abstract Syntax Tree (AST) of Java classes in order to: (i) parse the Java elements to create the architecture model; and (ii) to process the GenArch annotations.

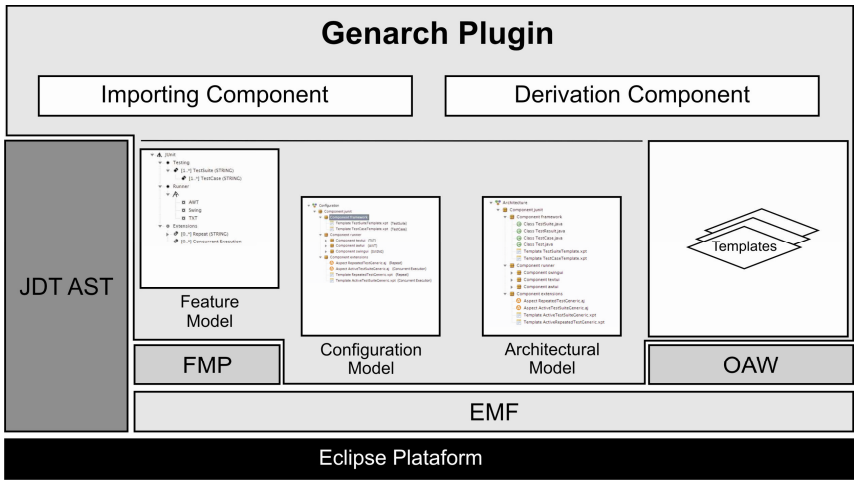


Figure 2: Genarch Architecture

The feature, configuration and architecture models of GenArch tool were specified using EMF. EMF is a Java/XML framework that enables the building of Model Driven Development tools based on structured data models. It allows generating a set of Java classes to manipulate and specify visually models. These classes are generated based on a given meta-model, which can be specified using XML Schema, annotated Java classes or UML modeling tools (such as Rational

Rose). The feature model used in our tool is specified by a separate plug-in, called FMP (Feature Modeling Plug-in) [Antkiewicz and Czarnecki, 2006]. It allows modeling the feature model proposed by Czarnecki and Eisenecker [Czarnecki and Eisenecker, 2000], which supports modeling mandatory, optional, and alternative features, and their respective cardinality. The FMP is also based on the EMF technology.

The openArchitectureWare (OAW) plug-in [openArchitectureWare, 2008] proposes to provide a complete toolkit for model-driven development. It offers a number of prebuilt workflow components that can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then, finally, for generating code. oAW is also based on EMF technology. Currently, the GenArch plug-in has only adopted the XPand language of oAW to specify its respective code templates (see details in Section 4.4).

4.2 JUnit Framework

Along the next sections, we use the JUnit testing framework to illustrate the GenAch functionalities. The main purpose of JUnit framework is to allow the design, implementation and execution of unit tests in Java applications. In particular, we are considering the following components of this framework:

(I) Framework – defines the framework classes responsible for specifying the basic behavior to execute test cases and suites. The main hot-spot classes available in this component are `TestCase` and `TestSuite`. The framework users need to extend these classes in order to create specific test cases to their applications;

(II) Runner – is responsible for offering an interface to start and track the execution of test cases and suites. JUnit provides three alternative implementations of test runners, as follows: a command-line based user interface (UI), an AWT based UI, and a Java Swing based UI; and, finally,

(III) Extensions – responsible for defining functionality extending the basic behavior of JUnit framework. We are using the implementation of the JUnit that was refactored using the AspectJ language. Aspects were used in this implementation to provide a better modularization of some of the JUnit optional features. Two functionalities of the `extensions` component were implemented as aspects in this version: (i) the repetition – allows repeating the execution of specific test cases; and (ii) the active execution – addresses the concurrent execution of test suites. Additional details about it can be found in [Kiczales et al., 2001] [Kulesza et al., 2007].

@Feature	
Attributes	
Name	Name of feature
Parent	The parent of the feature
Type	alternative, optional or mandatory
@Variability	
Attributes	
Type	hotspot or hotspotAspect
Feature	Contains the feature associated with the variability

Table 1: GenArch Annotations and their Attributes

4.3 Annotating Java Code with Features and Variabilities

Following the steps of our approach described in Section 3, the domain engineer initially creates a set of Java annotations in the code of implementation elements (classes, aspects and interfaces) from the SPL architecture. The annotations are in general embedded in the code of implementation elements representing the SPL variabilities. Table 1 shows the two kinds of annotations supported by our approach: (i) `@Feature` – this annotation is used to indicate that a particular implementation element addresses a specific feature. It also allows specifying the kind of feature (mandatory, alternative, or optional) being implemented and its respective feature parent if exists; and (ii) `@Variability` – it indicates that the implementation element annotated represents an extension point (e.g. a hotspot framework class) in the SPL architecture. In the current version of GenArch, there are three kinds of implementation elements that can be marked with this annotation: abstract classes, abstract aspects, and interfaces. Each of them has a specific type (hotspot or hotspot-aspect) defined in the annotation.

```
@Feature(name="TestCase",parent="TestSuite",
                                type=FeatureType.mandatory)
@Variability(type=VariabilityType.hotSpot,
                                feature="TestCase")
public abstract class TestCase extends Assert implements
Test {
    private String fName;

    public TestCase() {
           fName= null;
    }
    public TestCase(String name) {
        fName= name;
    }
    ...
}
```

Figure 3: `TestCase` class annotated

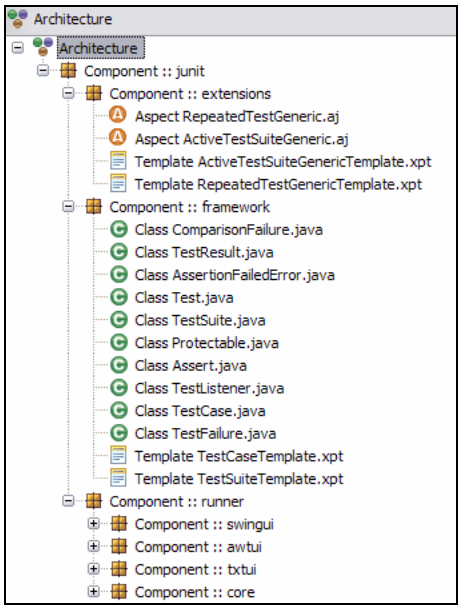
Figure 3 shows the `TestCase` abstract class from the JUnit framework marked with two GenArch annotations. The `@Feature` annotation indicates that the `TestCase` class is implementing the TestCase feature, which has the TestSuite as feature parent. It also shows that this feature is mandatory. This means that every instance of the JUnit framework requires the implementation of this class. The `@Variability` annotation specifies that the `TestCase` class is an extension point of the JUnit framework. It represents a hot-spot class that needs to be specialized when creating test cases (a JUnit framework instantiation) for a Java application. Next section shows how GenArch annotations are processed to generate the initial version of the derivation models.

4.4 Generating and Refining the Approach Models

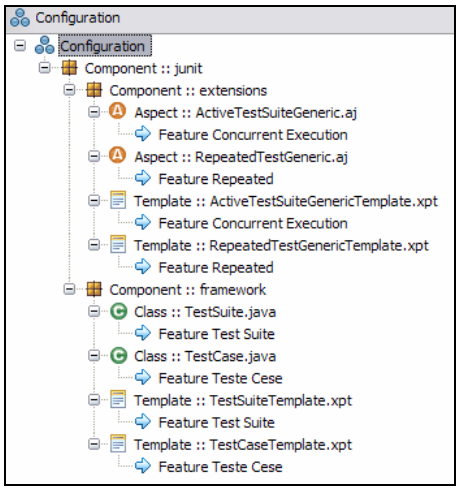
In the second step of our approach, an initial version of each GenArch model is produced. The architecture model is created by parsing the Java project or directory that contains the implementation elements of the SPL architecture. The Eclipse Java Development Tooling (JDT) API [Shavor et al., 2003] is used by our plug-in to parse the existing Java code. During this parsing process, every Java package is converted to a component with the same name in the architecture model. Each type of implementation element (classes, interfaces, aspects or files) has a corresponding representation in the architecture model. Figure 4(a) shows, for example, the initial version of the JUnit architecture model. Every package was converted to a component and every implementation element was converted to its respective abstraction in the architecture model. As we mentioned before, architecture models are created only to allow the visual representation of the SPL implementation elements in order to relate them to a feature model.

The GenArch annotations are used to generate specific elements in the feature, configuration and architecture models. These elements are also processed by the tool using the Eclipse Java Development Tooling (JDT) API [Shavor et al., 2003]. The JDT API allows browsing the Abstract Syntax Tree (AST) of Java classes to read their respective annotations. The `@Feature` annotation is used to create different features with their respective type in the feature model. Every `@Feature` annotation demands the creation of a new feature in the feature model. If the `@Feature` annotation has a `parent` attribute, a parent feature is created to aggregate the new feature. Figure 4(c) shows the partial feature model generated for the JUnit framework. It aggregates, for example, the TestSuite and TestCase features, which were generated based on the `@Feature` annotation presented in Figure 3.

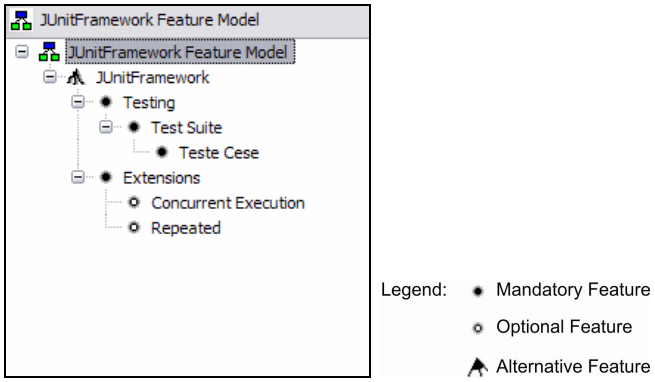
On the other hand, each `@Variability` annotation demands the creation of a code template that represents concrete instances of the extension implementation element that is annotated. The architecture model is also updated to include the new template element created. Consider, for example, the annotated `TestCase` class presented in Figure 3. The `@Variability` annotation of this class demands the creation of a code template (`TestCaseTemplate`) which represents `TestCase` subclasses, as we can see in Figure 4(a). This template will be used in the derivation process of the JUnit framework to generate `TestCase` subclasses for a specific Java application under testing. Only the structure of each template is generated based on the respective implementation element (abstract class, interface or abstract aspect) annotated. Empty implementations of the abstract methods or pointcuts from these elements are automatically created in the template structure. Next section shows how the code of every template can be improved using information collected by a feature model instance.



(a) Architecture Model



(b) Configuration Model



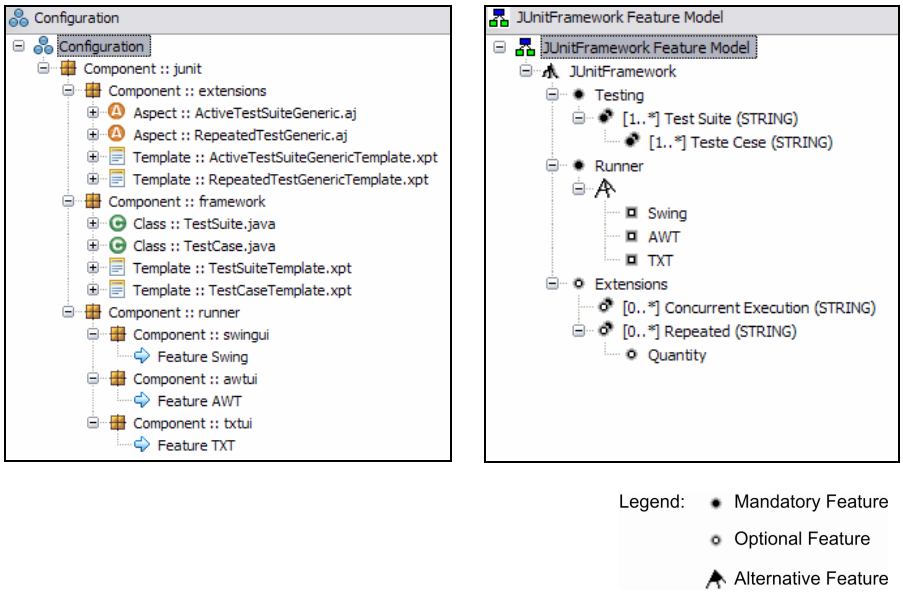
(c) Feature Model

Figure 4: The JUnit GenArch Models – Initial Versions

The GenArch configuration model defines a set of mapping relationships. Each mapping relationship links a specific implementation element from the architecture model to one or more features from the feature model. An initial version of the configuration model is created based on the `@Feature` annotations with attribute `type` equals to `optional` or `alternative`. When processing these annotations, the GenArch tool adds a mapping relationship between the feature created and the

respective implementation element annotated. The current visualization of our configuration model shows the implementation elements in a tree (similar to the architecture model), but with the explicit indication of the feature(s) that each of them depends on. Figure 4(b) shows the initial configuration model of the JUnit framework based on the processed annotations. As we can see, the `RepeatedTestAspect` aspect depends explicitly on the Repeated feature. It represents a mapping relationship between these elements (implementation and feature), created because the `RepeatTestAspect` is marked with the `@Feature` annotation and its attributes have the following values: (i) `name` equals to `Repeated`; and (ii) `type` equals to `optional`.

In GenArch tool, every template must depend at least on a feature. The `@Variability` annotation specifies explicitly this feature. This information is used by the tool to update the configuration model by defining that the template generated depends explicitly on a feature. Figure 4(b) shows that the `TestCaseTemplate` depends explicitly on the TestCase feature. It means that during the derivation/instantiation of the JUnit framework, this template will be processed to every TestCase feature created in the feature model instance.



(a) Configuration Model **(b) Feature Model**

Figure 5: The JUnit GenArch Models – Final Versions

After the generation of the initial versions of GenArch models, the domain engineer can refine them by including, modifying or removing any feature, implementation element or mapping relationship. Figure 5 shows a partial view of the final versions of the JUnit framework models. As we can see, they were refined to completely address the features and variabilities of JUnit. The runner component that

implements the execution interface of JUnit, were not configured automatically by GenArch. This happened because in the current version of GenArch, it is not possible to create any kind of feature annotatation in Java packages. Thus, in the feature model, shown in Figure 5(b), the Runner feature and its respective child alternative features (TXT, AWT and Swing) were created manually. Each one of alternative features represents a type of execution interface of the JUnit. In the configuration model, illustrated in Figure 5(a), mappings between components (Java packages), that implements the execution interface, and their respective features (TXT, AWT and Swing) also were created manually. Besides of these, an attribute of the type String was created in the features TestSuite, TestCase and Repeated. These attributes are used to provide information to templates during the derivation process. Cardinalities have been also associated to some features, such as TestSuite, TestCase and Repeated. It enables specifying several instances of these features during application engineering.

4.5 Implementing Variabilities with Templates

The GenArch tool adopts the XPand language from the oAW plug-in [openArchitectureWare, 2008] to specify the code templates. XPand is a very simple and expressive template language. In our approach, templates are used to codify implementation elements (classes, interfaces, aspects and configuration files) which need to be customized during the product derivation. Examples of implementation elements that can be implemented using templates are: concrete instances of hot-spots (classes or aspects) and parameterized configuration files. Every GenArch template can use information collected by the feature model to customize its respective variable parts of code.

```
01 «IMPORT br::pucrio::inf::les::genarch::models::feature»
02 «DEFINE TestSuiteTemplate FOR Feature»
03   «FILE attribute + ".java"»
04    package junit.framework;
05    public class «attribute»  {
06           public static Test suite() {
07           TestSuite suite = new TestSuite();
08           «FOREACH features AS child»
09           suite.addTestSuite(«child.attribute».class);
10           «ENDFOREACH»
11           return suite;
12    }
13  }
14  «ENDFILE»
15 «ENDDEFINE»
```

Figure 6: The TestCaseTemplate

Figure 6 shows the code of the `TestSuite` template using the XPand language. It is used to generate the code of specific JUnit test suites for Java applications. The `IMPORT` statement allows using all the types and template files from a specific

namespace (line 1). It is equivalent to a Java import statement. GenArch templates import the `br::pucrio::inf::les::genarch::models::feature` namespace, because it contains the classes representing the feature model. The `DEFINE` statement determines the template body (line 2). It defines the name of the template as well as the meta-model class whose instances will be used in its customization. The `TestSuiteTemplate` template, for example, has the name `TestSuiteTemplate` and uses the `Feature` meta-model class to enable its customization. The specific feature to be used in the customization of each template is defined by the mapping relationship in the configuration model. Thus, the `TestSuiteTemplate`, for example, will be customized using each TestSuite feature specified in a feature model instance by the application engineer (see configuration model in Figure 5(b)).

Inside the `DEFINE` tags (lines 3 to 14) are defined the sequence of statements responsible for the template customization. The `FILE` statement specifies the output file to be written the information resulting from the template processing. Figure 6 indicates that the file name resulting from the template processing will have the name of the feature being manipulated (`attribute`) plus the Java extension (`.java`). The following actions are accomplished in order to customize the `TestSuiteTemplate`: (i) the name of the resulting test case class is obtained based on the feature name (`attribute`); and (ii) the test cases to be included at this test suite are specified based on the feature names (`child.attribute`) of the feature child (`child`). The `FOREACH` statement allows the processing of the child features of the TestSuite feature being processed for this template.

4.6 Synchronization between Code, Annotation and Models

The synchronization between code, annotation and models is fundamental to guarantee the compatibility of the SPL artifacts and to avoid inconsistencies during the product derivation process. Besides, it is also important to allow that specific changes in the code, models or annotations will be reflected on the related artifacts. The current version of GenArch addresses synchronization functionality by trying to solve automatically inconsistencies between models, code and annotations. The following inconsistencies are automatic resolved by our tool: (i) removing of features from the feature model that are not longer used by the configuration model or some annotation in the code; (ii) removing of mapping relationships in the configuration model elements that refer to non-existing features or implementation elements; (iii) removing of implementation elements from the architecture model which do not exist anymore; (iv) automatic creation of `@Feature` annotations in implementation elements based on the existence of dependency relationships between a feature and an implementation element in the configuration model; (v) changes in the original implementation element path (because it was moved to another directory, for example) implies in the same changes in the architecture model related element.

The functionality of synchronization is implemented in GenArch tool by the Synchronizer module. Four listeners are responsible to reporting the changes that occur in the Eclipse workspace. One of them is responsible for observing changes in the project resources, and the other ones observe changes in the architecture, configuration and feature models editors. The Synchronization component receives

the change reports and based on that, it performs the verification of inconsistencies and updates the models.

4.7 Generating SPL Instances

In the fourth and last step of our approach (Section 3), called product derivation, a product or member of the SPL is created. The product derivation is organized in the following steps: (i) choice of variabilities in a feature model instance – initially the application engineer specifies a feature model instance (also called a configuration) using the FMP plug-in that determines the final product to be instantiated; (ii) next, the application engineer provides to GenArch tool, the architecture and configuration model of the SPL and also the feature model instance specified in step (i); and (iii) finally, the GenArch tool processes all these models to decide which implementation elements need to be instantiated to constitute the final application requested.

During the process of product generation (step iii), the GenArch traverses the architecture model and processes each implemented element encountered verifying in the configuration model if it depends on any special feature or a logical feature expression. In such case, the GenArch only instantiates that element (and processes respective sub-elements in the component element case), if there is an occurrence of that specific feature in the feature model instance, or if the logical feature expression evaluates to true. The evaluation of logical feature expression is doing as follow: (i) for each feature in the expression, its occurrence in the feature model instance is checked, if the feature occurs in the feature model instance its value is “true”, if not its value is “false”; and (ii) then, the logical expression is evaluated.

In our tool, the implementation elements, which represent the product under instantiation, are created in a new Eclipse project. The components are instantiated by creating a correspondent Java package. For the other implementation elements (classes, interfaces, aspects, files), one copy is created in the respective related package. The template elements are processed for every occurrence of their associated features. During the template processing, the information about the feature (and respective sub-features), which the template depends, is used to support the template customization.

5 J2ME Games Product Line

In this section, we describe the adoption of the GenArch tool in the preparation and instantiation of a J2ME Games Product Line. Next subsections present the case study description and respective GenArch models produced to enable its instantiation.

5.1 Case Study Description

J2ME games are mainstream mobile applications of considerable complexity [Alves et al., 2005]. Many of the J2ME games can be seen as a product line because they need to be adapted and ported to different devices in order to address new market segments. Since the devices vary in terms of the resources available, such as processor and memory, it is necessary to adapt the games according to the specific properties of each device. However, the mandatory functionalities of each game still

remains, it is defined by a game engine. It basically defines a state machine whose state change is driven by elapsed time and user input through the device keypad. State change impacts the game objects (such as actors and environment) and how they interact. Game object redrawing typically occurs after every state change.

The game involved in our case study, the Rain of Fire, is a commercial project developed by Meantime Mobile Creations[1]. It offers the following variabilities to be addressed: (i) optional images – some images of the game (e.g. clouds rolling in the background) can be made optional because they are not essential to play the game. It allows optimizing the performance of the game for resource-constrained devices; (ii) proprietary drawing images API – the game images are drawn at various locations and are transformed (rotated, flipped) in specific situations using device proprietary drawing API; (iii) image loading – the games images can be loaded using the following two policies: loading on demand when there is a screen changing and loading of all images at once during game startup; (iv) language used (English, Portuguese) in the game; (v) cell model being considered; and (vi) key mapping of the device (such as, Motorola or Nokia).

Most of the variabilities of the Rain of Fire game were originally implemented using conditional compilation. In order to improve the modularization and management of these variabilities, they were refactored to be implemented using aspect-oriented programming [Alves et al., 2006]. The implementation of the game variabilities with aspects brought many benefits such as: (i) the simplification of the game core with the extraction of a lot of intricate pieces of code resulting from the use of conditional compilation; and (ii) the capacity to plug/unplug the aspects from the SPL core implementation. Next section details how the game code was annotated to derive automatically its GenArch models.

```
@Feature(name="On Demand",parent="Image Loading",
                        type=FeatureType.alternative)
public privileged aspect LoadImgOnDemand {
    after() : ResourcesEvents.loadingImages() {
          Resources.loadGameImages();
    }
    ...
}
```

```
@Feature(name="On Init",parent="ImageLoading",
                        type=FeatureType.alternative)
public privileged aspect LoadImgOnInit {
    after() : ResourcesEvents.loadingImages() {
          Resources.loadGameImages();
    }
    ...
}
```

Figure 7: LoadImgOnDemand and LoadImgOnInit aspects with GenArch annotations.

[1] http://www.meantime.com.br/en/

5.2 Annotating Features in the J2ME Games Product Line

The first step to prepare the Rain of Fire game product line to be automatically instantiated was to annotate its respective variabilities. We inserted the `@Feature` annotation inside all the aspects and classes that modularizes the game variabilities. The annotation was created according to the type of variability (optional or alternative) being addressed. Figure 7 shows, for example, the code of the LoadImgOnDemand and LoadImgOnInit aspects aggregating the `@Feature` annotations. As we can see in the annotations, these aspects represent alternative implementations (On Demand and On Init features) for the Image Loading feature. Most of the aspects were annotated in a similar way, because they represent alternative implementations of the Rain of Fire variable features. Some of the aspects also represent optional features. This was the case, for example, of the `Cloud` aspect, which implements the clouds images rolling in the background of the game.

5.3 Generating the Rain of Fire Models

After the insertion of annotations in the source code, the GenArch tool generates the initial versions of the feature, architecture and configuration models representing the product line. Figures 8, 9 and 10 show each of these models generated for the Rain of Fire. The feature model includes each of the variabilities with their respective type (Figure 10). The Cloud optional feature represents optional images to be or not included in the game screens. Several alternative features were codified to address the variety of devices that can execute the game, such as: Cell Model, Flip, Key Mapping and Image Loading. The Image Loading feature, for example, defines the two alternative loading policies (On Demand, On Init) of the device based on the memory available.

Figure 8 illustrates the architecture model of the Rain of Fire game. All classes/interfaces codified in Java and aspects codified in AspectJ were imported to the architecture model to facilitate the creation of dependency relationships with the feature model. The `rain` component aggregates all the classes that implement the mandatory features of the game. These classes represent the game engine. Each of the variabilities was codified through a set of aspects and was organized in separate components/packages. Examples of such components are: `bright`, `cell`, `croma`, `flip`, `keymapping` and `loadimg`. The only exception, it was the `language` component which was implemented as classes (one class per language). In this case study, it was not required to create code templates because most of the variabilities are already modularized as alternative or optional aspects, thus there is no need to customize such variabilities during application engineering.

Finally, the configuration model of the Rain of Fire was also generated by the GenArch tool. It is depicted in Figure 9. It shows the dependency relationships between the implementation elements and features for the case study. Only the implementation elements that have any dependency with any feature are shown. The `LoadImgOnDemand` and `LoadImgOnInit` aspects, for example, modularize the alternative implementations for the image loading policy. Because of that, each of them has a dependency relationship with the On Demand and On Init subfeatures, respectively, of the Image Loading feature. It means each of the aspects will be instantiated during application engineering only if those features are selected.

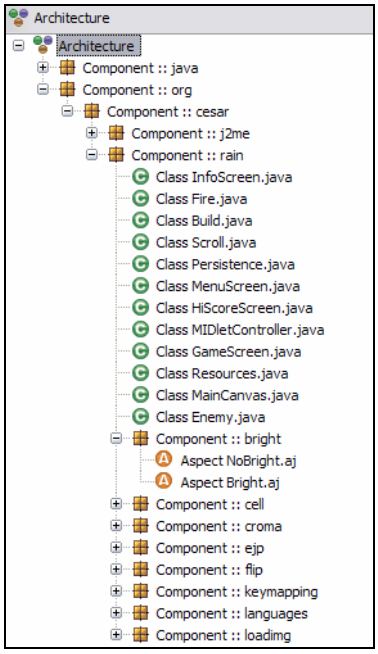


Figure 8: Rain of Fire - Architecture Model

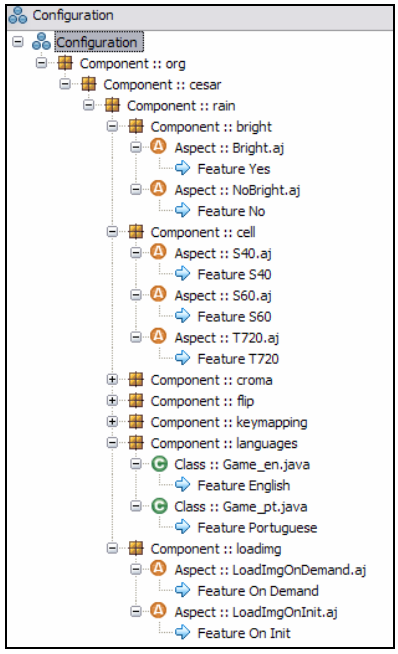


Figure 9: Rain of Fire - Configuration Model

In the Rain of Fire case study, it was not necessary to create additional features, implementation elements or dependency relationships in the feature, architecture and configuration models generated by GenArch tool. It happened because all the implementation choices for the variabilities of the game are already codified and available, similar to a black box framework [Roberts et al., 1998]. Thus, the simple use of GenArch annotations over the implementation elements that modularize the variabilities was enough to create automatically complete versions of the models. It shows how efficient the GenArch tool can be when working in the context of software product lines which have most of their variabilities already implemented.

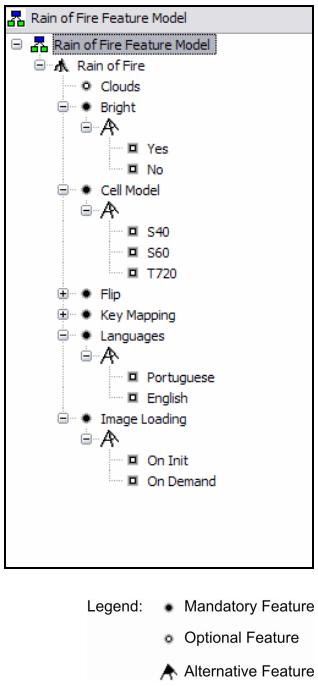


Figure 10: Rain of Fire - Feature Model

6 Lessons Learned and Discussions

This section provides some discussions and lessons learned based on the initial experience of use the GenArch tool to automatically instantiate the JUnit framework and a J2ME game SPL [Kulesza et al., 2007]. Although these examples are not so complex, they allow illustrating and exercise all the tool functionalities. Additional details about the models and code generated for these case studies will be available at [GenArch, 2008].

Integration with Refactoring Tools. Application of refactoring techniques [Fowler, 1999] [Opdyke, 1992] is common nowadays in the development of software systems. In the development of SPLs, refactoring is also relevant, but it assumes a different perspective. In the context of SPL development, refactoring technique needs to consider [Alves et al., 2006], for example: (i) if changes applied to the structure of existing SPL implementation elements do not decrease the set of all possible configurations (products) addressed by the SPL; and (ii) complex scenarios of merging existing programs into a SPL. Although many existing proposed refactorings introduce extension points or variabilities [Alves et al., 2006], the refactoring tools available are not strongly integrated with existing SPL modeling or derivation tools. It

can bring difficulties or inconsistencies when using both tools together in the development of a SPL. The integration of GenArch with existing refactoring tools involves several challenges, such as, for example: (i) to allow the creation of `@Feature` annotations to every refactoring that exposes or creates a new variable feature in order to present it in the SPL feature model to enable its automatic instantiation; and (ii) refactorings that introduce new extension points (such as, abstract classes or aspects or an interface) must be integrated with GenAch to allow the automatic insertion of `@Variability` annotations. Also the functionality of synchronization of models, code and annotations (discussed in Section 5.1) is fundamental in the context of integration of GenArch with existing refactoring tools, because it guarantees that every refactoring applied to existing SPL implementation elements, which eventually cause the creation of new or modification of existing annotations, will be synchronized with the derivation models.

SPL Adoption Strategies. Different adoption strategies [Krueger, 2003] can be used to develop software product lines (SPLs). The proactive approach motivates the development of product lines considering all the products in the foreseeable horizon. A complete set of artifacts to address the product line is developed from scratch. In the extractive approach, a SPL is developed starting from existing software systems. Common and variable features are extracted from these systems to derive an initial version of the SPL. The reactive approach advocates the incremental development of SPLs. Initially, the SPL artifacts address only a few products. When there is a demand to incorporate new requirements or products, the common and variable artifacts are incrementally extended in reaction to them. Independent from the SPL adoption strategy adopted, a derivation tool is always needed to reduce the cost of instantiation of complex architectures implemented to SPLs.

We believe that GenArch tool can be used in conjunction with proactive, extractive or incremental adoption approaches. In the proactive approach, our tool can be used to annotate the implementation elements produced during domain engineering in order to prepare those elements to be automatically instantiated during application engineering. Also, the extractive approach can demand the introduction of GenArch annotations in classes, interfaces or aspects, whenever new extension points are exposed in order to gradually transform the existing product in a SPL. Finally, the reactive approach requires the implementation of the synchronization functionality (Section 4.5) in GenArch tool, because it can involve complex scenarios of merging products. Both extractive and reactive approaches require the use of refactoring tools to help the extraction and merging of features [Alves et al., 2006], because of that the integration of GenArch with refactoring tools is fundamental to address these SPL adoption strategies.

Architecture Model Specialization. The architecture model supported currently by GenArch tool allows representing SPL architectures implemented in the Java and AspectJ programming languages. However, our architecture model is not dependent of Java technologies, only the GenArch functionalities responsible to manipulate the implementation elements were codified to only work with Java and AspectJ implementation elements. Examples of such functionalities are: (i) the parser that imports and processes the implementation elements and annotations; and (ii) the

derivation component that creates a SPL instance/product as a Java project. In this way, the GenArch approach, as presented in Section 3, is independent of specific technologies. Currently, we are working on the definition of specializations of the architecture model. These specializations have the purpose to support other abstractions and mechanisms of specific technologies. In particular, we are modifying the tool to work with a new architecture model that supports the abstractions provided by the Spring [Johnson et al., 2005] and OSGi frameworks. This new model [Cirilo et al., 2008] is a specialization of our current architecture model. It will allow working not only with Java and AspectJ elements, but also with Spring beans, OSGi bundles and their respective configuration files.

Aspect-Oriented Variatilities Instantiation. Aspect-oriented software development has been recently investigated as a prominent technique to improve the modularization and management of features in software product lines [Zhang and Jacobsen, 2004] [Kulesza et al., 2006] [Kulesza, Coelho et al., 2006] [Alves et al., 2006] [Apel and Batory, 2006] [Mezini and Ostermann, 2004]. Aspects contribute to better modularize crosscutting optional and integration features in software family architectures. As a result, it also allows plugging and unplugging these features from the core of the product line architecture. Additionally, it can simplify the complexity of the implementation of the SPL architecture. Aspects can also define specific variabilities through the definition of extension behavior or specific joinpoints to affect. Several mechanisms are available in current languages/platforms to implement these variabilities, such as, the definition of abstract aspects, pointcuts and methods using AspectJ [Kiczales et al., 2001]; and specification of pointcuts directly in configuration files in Spring [Johnson et al., 2005]. We are currently incorporating mechanisms in GenArch tool to provide support to manage and instantiate automatically these kinds of variabilities [Kulesza, 2007] [Kulesza et al., 2006]. It basically involves: (i) the definition of crosscutting and joinpoint features in the feature model; and (ii) the specification of mapping of the crosscutting relationships in problem space (features) to the respective ones in solution space (pointcuts in aspects) in the configuration model.

Documentation and Tracing of Variabilities. The feature and configuration models generated by GenArch can be seen, respectively, as a useful documentation of the existing variabilities of the SPL and of the mapping of those variabilities to implementation elements. The new functionality of model synchronization of GenArch guarantees that these documentations are always kept updated and synchronized. The model synchronization is also a first step to address the tracing of mandatory and variable features in product lines, from feature models to the architecture model and code artifacts. The tracing of features in SPL development can help engineers with the following activities: analysis of feature covering; impact analysis of changing features or requirements; and improved and consistent variability management. As part of the traceability functionality, we are planning to develop new views for GenArch that process and use information provided by the feature, architecture and configuration models. Finally, we also plan to integrate the GenArch models with the traceability mechanisms between feature and use case models provided by another approach under development [Alferez et al., 2008].

7 Conclusions and Future Work

In this paper, we presented GenArch, a model-based product derivation tool. Our tool combines the use of models and code annotations in order to enable the automatic product derivation of existing SPLs. We illustrated the use of our tool using the JUnit framework and a J2ME games product line. As a future work, we plan to evolve the GenArch functionalities to address the following functionalities: (i) to extend the GenArch parsing functionality to allow the generation of template structure based on existing AspectJ abstract aspects; (ii) to allow the customization of aspect variabilities and libraries considering the AspectJ and Spring technologies; (iii) to integrate our tool with existing refactoring tools available ; (iv) to offer an specialization of the architecture model [Cirilo et al., 2008] that supports the Spring [Johnson et al., 2005] and OSGi technologies; and (v) to allow the composition of other DSLs (activity and state models) with the feature model in order to enable the complete generation of more complex variabilities.

Acknowledgments

We would like to thank Meantime Mobile Creations company for making the Rain of Fire implementation available. The authors are supported by ESSMA/CNPq project under grant 552068/2002-0 and LatinAOSD/CNPq-Prosul project. Uirá is also partially supported by European Commission Grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

[Alferez et al., 2008] Alferez, M., Kulesza, U., Moreira, A., Araújo, J., Amaral, V..: "A Model-Driven Approach for Software Product Lines Requirements Engineering". In Proceedings of the 20th International Conference on Software Engineering and Knowledge (SEKE´2008), San Francisco, July 2008.

[Alves et al., 2005] Alves, V., Matos, P., Cole, L., Borba, P., Ramalho, G.: "Extracting and Evolving Mobile Games Product Lines". In Proceedings of the 9th International Software Product Line Conference (SPLC'05), LNCS 3714, Springer-Verlag, 70-81, September 2005.

[Alves et al., 2006] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: "Refactoring Product Lines". In Proceedings of the 5th ACM International Conference on Generative Programming and Component Engineering (GPCE'06). ACM Press, October 2006.

[Antkiewicz and Czarnecki, 2006] Antkiewicz, M., Czarnecki, K.: "FeaturePlugin: Feature modeling plug-in for Eclipse". In Proceeding of the OOPSLA'04, Eclipse Technology eXchange (ETX) Workshop, 2004.

[Apel and Batory, 2006] Apel, S., Batory, D.: "When to Use Features and Aspects? A Case Study". In Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06), Portland, Oregon, October 2006.

[Arrango, 1994] Arrango, G.: "Domain Analysis Methods" In Software Reusability, 17-49, New York, 1994.

[Budinsky et al., 2004] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: "Eclipse Modeling Framework". Addison-Wesley, 2004.

[Cirilo et al., 2008] Cirilo, E., Kulesza, U., Coelho, R., Lucena, C., Staa, A.: "Integrating Component and Product Line Technologies". In Proceedings of the 10th International Conference on Software Reuse (ICSR'2008), China, May 2008.

[Clements and Northrop, 2001] Clements, P., Northrop, L.: "Software Product Lines: Practices and Patterns", Addison-Wesley Professional, 2001.

[Czarnecki and Eisenecker, 2000] Czarnecki, K., Eisenecker, U.: "Generative Programming: Methods, Tools, and Applications", Addison-Wesley, 2000.

[Czarnecki et al., 2004] Czarnecki, K., Helsen, S., Eisenecker, U.: "Staged Configuration Using Feature Models". In Proceedings of the 3rd Software Product-Line Conference (SPLC'04), September 2004.

[Czarnecki et al., 2006] Czarnecki, K., Helsen, S.: "Feature-Based Survey of Model Transformation Approaches", IBM Systems Journal, 45, 3, 621-64, 2006.

[Deelstra et al., 2005] Deelstra, S., Sinnema, M., Bosch, J.: "Product derivation in software product families: a case study". Journal of Systems and Software, 74, 2, 173-194, 2005.

[Fowler, 1999] Fowler, M.: "Refactoring: Improving the Design of Existing Code", Addison, 1999.

[Gears, 2008] Gears/BigLever, retrieved at http://www.biglever.com/, January 2008.

[GenArch, 2008] GenArch – Generative Architectures Plugin, retrieved at http://www.teccomm.les.inf.puc-rio.br/genarch/, January 2008.

[Greenfield and Short, 2005] Greenfield, J., Short, K.: "Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools", John Wiley and Sons, 2005.

[Johnson et al., 2005] Johnson, R., Hoeller, J., Arendsen, A., Risberg, T., Sampaleanu, C.: "Professional Java Development with the Spring Framework", Wrox, 2005.

[Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: "Feature-oriented domain analysis (FODA) feasibility study", Technical Report CMU/SEI-90-TR-021, SEI, Pittsburgh, PA, November 1990.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: "Getting Started with AspectJ", Communication ACM, 44, 59-65, 2001.

[Krueger, 2003] Krueger, C.: "Easing the Transition to Software Mass Customization". In Revised Papers From the 4th international Workshop on Software Product-Family Engineering, October 2003. Linden, F., Ed. Lecture Notes In Computer Science, 2290. Springer-Verlag, London, 282-293.

[Kulesza et al., 2007] Kulesza, U., Alves, V., Garcia, A., Neto, A., Cirilo, E., Lucena, C., Borba, P.: "Mapping Features to Aspects: A Model-Based Generative Approach". Early Aspects@AOSD´2007 Post-Workshop Proceedings, LNCS 4765, Springer-Verlag 2007.

[Kulesza, 2007] Kulesza, U.: "An Aspect-Oriented Approach to Framework Development", PhD thesis, Computer Science Department, PUC-Rio, April 2007 (in Portuguese).

[Kulesza et al., 2006] Kulesza, U., Lucena, C., Alencar, P., Garcia, A.: "Customizing Aspect-Oriented Variabilites Using Generative Techniques" In Proceedings of 18th International Conference on Software Engineering and Knowledge Engineering (SEKE'06), San Francisco, Skokie, IL, Knowledge Systems Institute, 17-22, July 2006.

[Kulesza, Coelho et al., 2006] Kulesza, U., Coelho, R., Alves, V., Neto, A., Garcia, A., Lucena, C., Staa, A., Borba, P.: "Implementing Framework Crosscutting Extensions with EJPs and

AspectJ". In Proceedings of the 20th Brazilian Symposium on Software Engineering (SBES'2006), 177-192, October 2006.

[Mezini and Ostermann, 2004] Mezini, M., Ostermann, K.: "Variability Management with Feature-Oriented Programming and Aspects". In Proceedings of the Foundations of Software Engineering (FSE-12), ACM SIGSOFT, pp. 127-136, November 2004.

[Opdyke, 1992] Opdyke, W.: "Refactoring Object-Oriented Frameworks" University of Illinois at Urbana-Champaign, 1992.

[openArchitectureWare, 2008] openArchitectureWare, retrieved at http://www.eclipse.org/gmt/oaw/, January 2008.

[Parnas, 1976] Parnas, D.: "On the Design and Development of Program Families", IEEE Transactions on Software Engineering (TSE), 2, 1, 1-9, 1976.

[pure::variants, 2008] Pure::Variants, retrieved at http://www.pure-systems.com/, January 2008.

[Roberts et al., 1998] Roberts, D., Johnson, R.: "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks" In Martin, R., Riehle, D., Buschmann, F.: "Pattern Languages of Program Design", Addison-Wesley, 3, 471-486, 1998.

[Shavor et al., 2003] Shavor, S., D'Anjou, J., Fairbrother, S., Kehn D., Kellerman, K., McCarthy, P.: "The Java Developer's Guide to Eclipse" Addison-Wesley, 2003.

[Smaragdakis and Batory, 2002] Smaragdakis, Y., Batory, D.: "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs". ACM Transaction on Software Engineering and Methodology. 11, 2, 215-255, April 2002.

[Stahl and Voelter, 2006] Stahl, T., Voelter, M.: "Model-Driven Software Development: Technology, Engineering, Management", Wiley, 2006.

[Weiss and Lai, 1999] Weiss, D., Lai, C..: "Software Product-Line Engineering: A Family-Based Software Development Process", Addison-Wesley Professional, 1999.

[Zhang and Jacobsen, 2004] Zhang, C., Jacobsen, H.: "Resolving Feature Convolution in Middleware Systems". In Proceedings of ACM SIGPLAN Object Oriented Programming Systems and Applications Conference 2004 (OOPSLA'2004), 188-205, 24-28, 2004, Vancouver, BC, Canada, October 2004.