GenArch – A Model-Based Product Derivation Tool

Elder Cirilo[1], Uirá Kulesza[1, 2], Carlos José Pereira de Lucena[1]

[1]Laboratório de Engenharia de Software – Departamento de Informática
Pontificia Universidade Católica do Rio de Janeiro (PUC-RIO), Brasil
{ecirilo,uira,lucena}@inf.puc-rio.br

[2]Departamento de Informática – Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa – Portugal

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.*

1. Introduction

Over the last years, many approaches for the development of system families and software product lines have been proposed [27, 7, 8, 14]. A system family [23] 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) [7] 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 [10] 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 [27, 7, 8, 14] motivate the definition of a flexible and adaptable architecture which 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 [1, 20], feature-oriented programming [5] and code generation [8]. 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 [28] 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 [13] and pure::variants [24] are two examples of 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 [17].

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 [25] platform and model-driven development toolkits available (such as EMF and oAW) at this platform were used as a base for the definition of our tool.

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. Section 5 presents a set of initial lessons learned from the implementation and use of the GenArch tool. Finally, Section 6 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) [8] 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 [4]. GP promotes the separation of problem and solution spaces, giving flexibility to evolve both independently. To provide this separation, Czarnecki & Eisenecker [8] propose the concept of a generative domain model. A generative domain model is composed of three basic elements: (i) *problem space* – which represents the concepts and features existent in a specific domain; (ii) *solution space* – which consists of the software architecture and components used to build members of a software family; and (iii) *configuration knowledge* – which 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 [4, 8] in the definition of a generative domain model. Common activities encountered in domain engineering methods are: (i) domain analysis – which 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 – which concentrates on the definition of a common architecture and components for this

domain; and (iii) domain implementation – which involves the implementation of architecture and components previously specified during domain design.

Two new activities [8] need to be introduced to domain engineering methods in order to address the goals of GP: (i) development of a proper means to specify individual members of the software family. Domain-specific languages (DSLs) must be developed to deal with this requirement; 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 [8]. 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 to automatically derive SPL members available in industry, such as Pure::variants and Gears. Pure::variants [24] is a SPL model-based product derivation tool. Its modeling approach comprises three models: features, family and derivation. The feature model contains the product variabilities and solution architectures are expressed in a family model. Both models are flexibly combined to define a SPL. Since a product line specification in this tool can consist of any number of models, the “configuration space” is used to manage this information and captures variants. The features are modeled graphically in different formats such as trees, tables and diagrams. Constraints among features and architecture elements are expressed using first order logic in Prolog and uses logic expression syntax closely related to OCL notation. This tool permits the use of an arbitrary number of feature models, and hierarchical connection of the different models. The pure::variants does not require any specific implementation technique and provides integration interfaces with other tools, e.g. for requirements engineering, test management and code generation.

Gears [13] allows the definition of a generative model focused on automatic product derivation. It defines three primary abstractions: feature declarations, product definitions, and variation points. Feature declarations are parameters that express the variations. 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 your software and map the feature declarations to choices at these variation points. The language for expressing constraints at feature models is propositional logic instead of full first-order logic.

3. Approach Overview

In this section, we present an overview of our product derivation approach based on the use of the 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 which enables the mainstream software developer community to use the concepts and foundations of the SPL approach in the development of software systems

and assets, such as, frameworks and customizable libraries, without the need to understand complex concepts or models from existing product derivation tools.

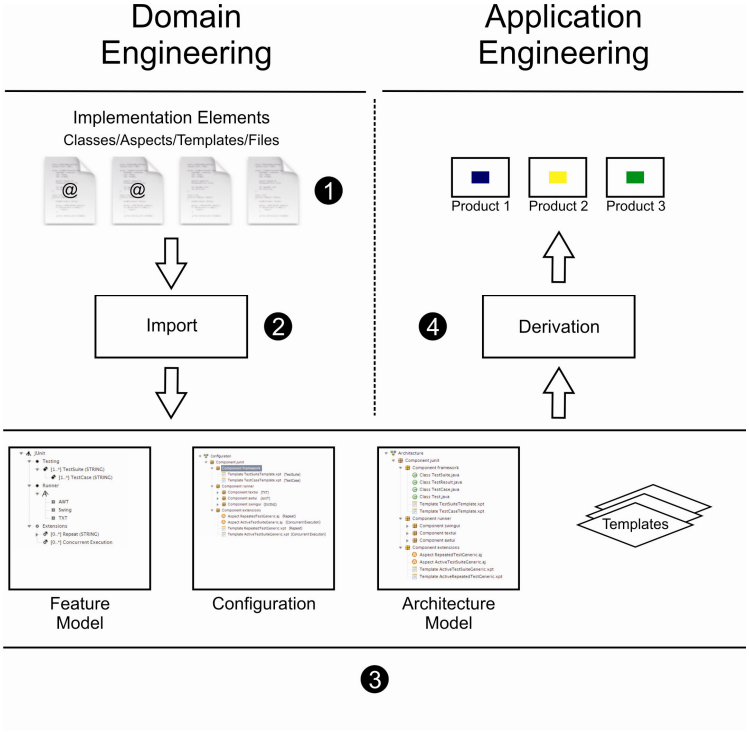


Figure 1. Approach Overview

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[1] to be inserted in the implementation elements (classes, interfaces and aspects) of SPL architectures. 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 is automatically derived 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 which have variabilities to be solved during application engineering. Initial versions of code

[1] 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 provides support to the definition of its annotations and models.

templates are automatically created in the architecture models based on GenArch annotations (see details in Section 4.2).

Feature models [16] 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 [9]) 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 [8], 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.

After all models are refined and 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 is mandatory. If an implementation 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 and 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 the 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 [25] using different technologies available at this platform. New model-driven development toolkits, such as Eclipse Modeling Framework (EMF) [6] and openArchitectureWare (oAW) [22] were

used to specify its models and templates, respectively. Figure 2 shows the general structure of the GenArch architecture based on Eclipse platform technologies. Our tool uses the JDT (Java Development Tooling) API [25] 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.

The feature, configuration and architecture model of GenArch tool are specified using EMF. EMF is a Java/XML framework which enables the building of MDD based 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 plugin, called FMP (Feature Modeling Plugin) [3]. It allows modeling the feature model proposed by Czarnecki et al [8], which supports modeling mandatory, optional, and alternative features, and their respective cardinality. The FMP also works with EMF models.

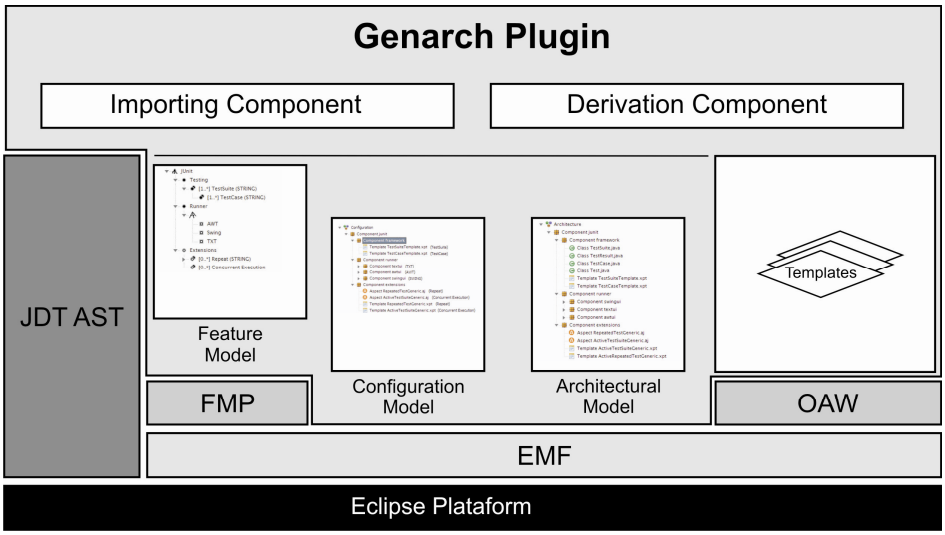


Figure 2. Genarch Architecture

The openArchitectureWare (OAW) plug-in [22] 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, 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. Annotating Java Code with Feature 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 PL 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 to specify 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.

Table 1. GenArch Annotations and their Attributes

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

```
@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

Along the next sections, we will use the JUnit testing framework to illustrate the GenAch functionalities. In particular, we are considering the following components of this framework:

(I) Core – 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;

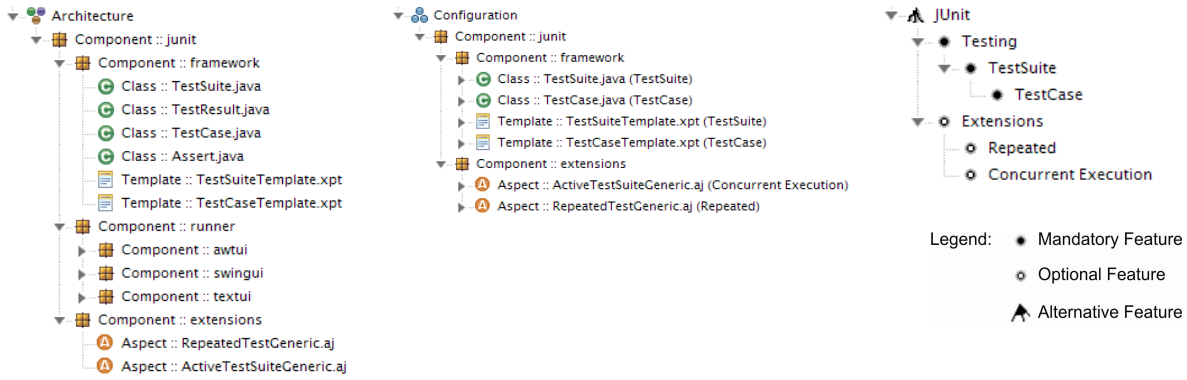
(II) Runner – this component is responsible for offering an interface to start and track the execution of test cases and suites. JUnit provides three alternative implementations of test runners, as follows: a command-line based user interface (UI), an AWT based UI, and a Java Swing based UI; and, finally,

(III) Extensions – responsible for defining functionality extending the basic behavior of the JUnit framework. Examples of available extensions are: a test suite to execute test suites in separate threads and a test decorator to run test cases repeatedly.

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 and 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.

4.3. 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 [25] 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.



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

Figure 4. The JUnit GenArch Models – Initial Versions

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 [25]. 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 which 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 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.

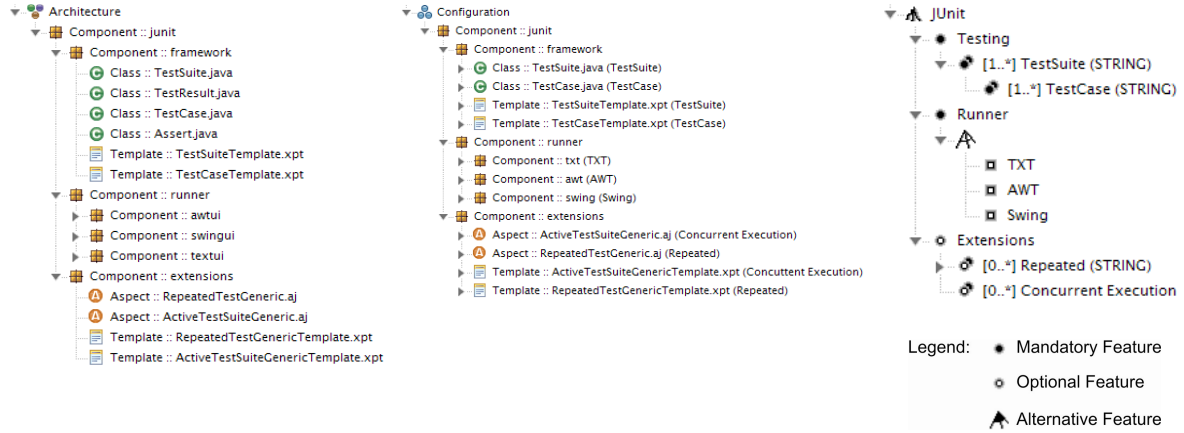
The GenArch configuration model defines a set of mapping relationships. Each mapping relationship links a specific implementation element from the architecture model to one or more features from the feature model. An initial version of the configuration model is created based on the `@Feature` annotations 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) which 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[2] depends explicitly on the Repeated feature. It represents a mapping relationship between these elements, 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 the 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.

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

[2] In this paper, we are using the implementation of the JUnit framework, which was refactored using the AspectJ language. Two features of the Extension component were implemented as aspects in this version: (i) the repetition feature – which allows to repeat the execution of specific test cases; and (ii) the active execution feature – which addresses the concurrent execution of test suites. Additional details about it can be found in [18, 20].

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 the JUnit.



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

Figure 5. The JUnit GenArch Models – Final Versions

4.4. Implementing Variabilities with Templates

The GenArch tool adopts the XPand language from the oAW plugin [22] 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 which 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 featuremodel»
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 `featuremodel` 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.

4.5. 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 using the FMP plug-in which 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 needs to be instantiated to constitute the final application requested. The selected implementation elements are then loaded in a specific source folder of an Eclipse Java project. The complete algorithm used by GenArch tool can be found in [18, 19, 20]

5. 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 SPL game [18]. Although these examples are not so complex, they allow to illustrate and exercise all the tool functionalities. Additional details about the models and code generated for these case studies will be available at [29].

Synchronization between Code, Annotations and Models. In the current version of the GenArch tool, there is no available functionality to synchronize the SPL code, annotations and respective models. This is fundamental to guarantee the compatibility of the SPL artifacts and to avoid inconsistencies during the process of product derivation. Besides, it is also important to allow that specific changes in the code, models or

annotations will be reflected on the related artifacts. At this moment, we are focusing at the implementation of a synchronization functionality which tries to solve automatically the inconsistencies between models, code and annotations, and if it is not possible it only shows the inconsistency to the product line engineer, as a GenArch warning or error. The following inconsistencies are planned to be verified by our tool: (i) removing of features from the feature model which are not longer used by the configuration model or annotation; (ii) removing of mapping relationships in the configuration model which refer to non-existing features or implementation elements; (iii) removing of implementation elements from the architecture model which do not exist anymore; and (iv) automatic creation of annotations in implementation elements based on information provided by the configuration model. Finally, the implementation of our synchronization functionality can also enable the automatic generation of implementation elements with annotations, from existing GenArch models.

Integration with Refactoring Tools. Application of refactoring techniques [12, 21] 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 [2], 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 [2], 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 [17] 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 5.1) in the GenArch tool, because it can involve complex scenarios of merging products.

Architecture Model Specialization. The architecture model supported currently by GenArch tool allows to represent 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 framework [15]. It is a specialization of our current architecture model. It will allow to work not only with Java and AspectJ elements, but also with Spring beans and their respective configuration files.

6. 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, but we also validated it in the context of a J2ME Games SPL. As a future work, we plan to evolve the GenArch functionalities to address the following functionalities: (i) synchronization between models, code and annotations (as described in Section 5.1); (ii) to extend the GenArch parsing functionality to allow the generation of template structure based on existing AspectJ abstract aspects; (iii) to address the aspect-oriented generative model proposed in [20] in order to enable the customization of pointcuts from feature models.

Acknowledgments. The authors are supported by ESSMA/CNPq project under grant 552068/2002-0. Uirá is also partially supported by European Commission Grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

[1] V. Alves, P. Matos, L. Cole, P. Borba, G. Ramalho. “Extracting and Evolving Mobile Games Product Lines”. Proceedings of SPLC'05, pp. 70-81, September 2005.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C. Lucena. Refactoring Product Lines, Proceedings of the GPCE´2006, ACM Press: Portland, Oregon, USA.

[3] M. Antkiewicz, K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004.
[4] G. Arrango. Domain Analysis Methods. In Software Reusability, New York, pp. 17-49, 1994.
[5] Y. Smaragdakis, D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM TOSEM, 11(2): 215-255 (2002).
[6] F. Budinsky, et al. Eclipse Modeling Framework. Addison-Wesley, 2004.
[7] P. Clements, L. Northrop. Software Product Lines: Practices and Patterns. 2001: Addison-Wesley Professional.
[8] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[9] K. Czarnecki, S. Helsen, U. Eisenecker. Staged Configuration Using Feature Models. In Proceedings of the Third Software Product-Line Conference, September 2004.
[10] K. Czarnecki, S. Helsen. Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal, 45(3), 2006, pp. 621-64.
[11] K. Czarnecki, M. Antkiewicz, C. Kim. "Multi-level Customization in ApplicationEngineering". CACM, Vol. 49, No. 12, pp. 61-65, December 2006.
[12] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison, 1999.
[13] Gears/BigLever, URL: http://www.biglever.com/, January 2007.
[14] J. Greenfield, K. Short. Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. 2005: John Wiley and Sons.
[15] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, C. Sampaleanu. Professional Java Development with the Spring Framework, Wrox, 2005.
[16] K. Kang, et al. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Pittsburgh, PA, November 1990.
[17] C. Krueger. "Easing the Transition to Software Mass Customization". In Proceedings of the 4th PFE, pp. 282–293, 2001.
[18] U. Kulesza, et al. Mapping Features to Aspects: A Model-Based Generative Approach. Early Aspects@AOSD´2007 Post-Workshop Proceedings, LNCS 4765, Springer-Verlag.
[19] U. Kulesza, C. Lucena, P. Alencar, A. Garcia. Customizing Aspect-Oriented Variabilites Using Generative Techniques. Proceedings of SEKE'06, July 2006.
[20] U. Kulesza. Uma Abordagem Orientada a Aspectos para o Desenvolvimento de Frameworks. Rio de Janeiro, 2007. Tese de Doutorado, DI/PUC-Rio, Abril 2007.
[21] W. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
[22] openArchitectureWare, URL: http://www.eclipse.org/gmt/oaw/,
[23] D. L. Parnas. On the Design and Development of Program Families. IEEE Transactions on Software Engineering (TSE), 1976. 2(1): p. 1-9.
[24] Pure::Variants, URL: http://www.pure-systems.com/, January 2007.
[25] S. Shavor, et al. The Java Developer's Guide to Eclipse. Addison-Wesley, 2003.
[26] T. Stahl, M. Voelter. Model-Driven Software Development: Technology, Engineering, Management. 2006: Wiley.
[27] D. Weiss, C. Lai. Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley Professional, 1999.
[28] S. Deelstra, M. Sinnema, J. Bosch. Product derivation in software product families: a case study. Journal of Systems and Software 74(2): 173-194, 2005.
[29] GenArch – Generative Architectures Plugin, URL: http://www.teccomm.les.inf.puc-rio.br/genarch/.