Integrating Component and Product Lines Technologies

Elder Cirilo[1], Uirá Kulesza[2,3], Roberta Coelho[1],
Carlos J.P. de Lucena[1], and Arndt von Staa[1]

[1] PUC-Rio, Computer Science Department, Rio de Janeiro, Brazil
{ecirilo,roberta,lucena,arndt}@inf.puc-rio.br
[2] Recife Center for Advanced Studies and Systems – C.E.S.A.R., Recife, Brazil
uira@cesar.org.br
[3] CITI/DI/FCT, New University of Lisbon, Lisboa, Portugal

Abstract. In this paper, we explore the integration of product line and component technologies in the context of the product derivation process. In particular, we propose new extensions to our existing model-based product derivation tool, called GenArch, in order to address the new abstractions and mechanisms provided by the Spring and OSGi component models. The GenArch extensions enable the automatic instantiation of product lines and applications - implemented using these component technologies. Moreover, it also enables different levels of customization, from fine-grained configuration of component properties to the automatic selection of components that will compose the final product.

1 Introduction

A software product line (SPL) [5] can be seen as a system family that addresses a specific market segment. A system family [10] is a set of programs that shares common functionalities and maintain specific functionalities that vary according to specific systems being considered. Software product lines and system families are typically specified, modeled and implemented in terms of common and variable features. A feature [13] is a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in SPLs.

Many approaches for SPL development [21,5,6,10] propose the definition of an architecture which comprises their common and variable features. This architecture is typically defined in a process called *domain engineering*. Different technologies can be adopted to implement the code artifacts of SPLs architectures, for instance: object-oriented frameworks and design patterns [8,3,9], aspect-oriented programming [1,15], feature-oriented programming [18], conditional compilation [1] and code generation [6]. Each one of them brings benefits and drawbacks to the modularization of common and variable features (*variabilities*) of SPL. Therefore, it is common to combine two or more of these technologies to implement different code assets of typical SPL architectures.

In the *application engineering* stage of SPL development [6], the core assets produced during the *domain engineering* stage [6] are composed and integrated to generate an instance (product) of the SPL architecture. This process is also known as *product derivation* [7]. Recent proposed approaches, such as Generative Programming [6] and

H. Mei (Ed.): ICSR 2008, LNCS 5030, pp. 130–141, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Software Factories [10] motivate the definition of mechanisms to support automatic product derivations. Such mechanisms can improve the productivity and quality of the derivation process. Domain-specific languages (DSLs) and code generators are the main technologies adopted by them. Several product derivation tools based on the feature model [13] or DSLs [6] have already been used in industry.

Over the last years, new component infrastructure technologies have been proposed. The main goal of them is to offer a unified model to allow the adequate management (i.e., assembling, adapting and connecting) of components and their configuration. Two important examples of such technologies based on Java platform are Spring and OSGi. The Spring framework [19] is a widely adopted Java/J2EE application framework. It offers a model to build applications as a collection of simple components (called beans) which can be connected or customized using dependency injection and aspect-oriented technologies. The OSGi [16] technology provides an infrastructure to manage the life-cycle of application components. The OSGi applications are structured as a set of *bundles* [16] - a *bundle* represents an application component that provides services to the end-user or other components. In the context of SPL development, these component infrastructure technologies can be combined with the programming techniques mentioned above to allow a better management of the SPL features.

This paper explores the integration of product line and component technologies in the context of the *product derivation process*. In particular, we propose new extensions to our existing model-based product derivation tool, called GenArch [4], in order to address the abstractions and mechanisms provided by the Spring and OSGi component models. The proposed extensions enable the automatic instantiation of product lines and applications implemented using these mainstream component technologies. Moreover, the GenArch extensions also provide different levels of customization from fine-grained configuration of component properties to the automatic selection of components that will compose the final product generated.

The remainder of this paper is structured as follows. Section 2 briefly describes GenArch tool in the context of an illustrative *product derivation* scenario - each subsection (Sections 2.1 to 2.3) presents one step in the generative approach supported by the tool. Section 3 details the GenArch extensions that integrate Spring (Section 3.1) and OSGi (Section 3.2) technologies to support automatic *product derivation*. Section 4 presents discussions and lessons learned while extending the tool and using it on *product derivation* scenarios. Finally, Section 5 presents our conclusions and directions for future work.

2 GenArch – A Model-Based Derivation Tool

GenArch [4] is a model-based tool which enables the mainstream software developer community to use the concepts and foundations of the SPL approach in the product derivation process [7] without the need to understand complex concepts or models from existing product derivation tools. This section presents an overview of the GenArch through an illustrative example of a *product derivation* scenario in JUnit framework.

2.1 Annotating Java Code with Annotations

The first step of the *domain engineering* consists in the creation of a domain model which defines which features exist in a specific domain and which of them are mandatory, optional and alternative features. Our approach starts at the end of the domain engineering stage, when the engineers annotates the existing code (classes, interfaces and aspects) of SPL architectures using GenArch specific annotations. These annotations map product line *features* and *variabilities*, defined in the domain model, to implementation elements of the SPL architecture. Two kinds of annotations are 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, optional) being implemented and its respective feature parent if exists; and

(ii) @Variability - it indicates that the annotated element represents an extension point (e.g. a hotspot framework class) in the SPL architecture.

Figure 1 shows an example of the use of the GenArch annotations in the context of JUnit framework. The `TestCase` class is a framework hotspot in JUnit that needs to be extended in order to create specific test cases. Thus, according to our approach, the `TestCase` class was annotated with two GenArch annotations (see Figure 1). The @Feature annotation indicates that this class implements the **Test Case** feature, which is mandatory. This means that every instance of the JUnit framework requires the implementation of this class. The @Variability annotation specifies that the **Test Case** is an extension point of the JUnit framework. It represents a hotspot that needs to be specialized when creating instances of the framework. Although the **Test Case** is a variation point, it is also as a mandatory feature since all JUnit instances must have at least one instance of `TestCase` class.

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

Fig. 1. TestSuite Class Annotated

Next subsection shows how GenArch annotations are processed to generate the initial version of the derivation models.

2.2 Generating and Refining the Approach Models

The GenArch approach encompasses three models: (i) the *product line implementation model*; (ii) the *feature model*; and (iii) the *configuration model*. These models must be specified in our approach to enable the automatic derivation of SPL members. The *product line implementation* model defines a visual representation of the

SPL implementation elements (i.e., classes, aspects, templates, configuration and extra files) in order to relate them to feature models. *Feature models* [6,13] are used in our approach to represent the variabilities of SPL architectures. The configuration model is responsible for defining the mapping between features and implementation elements.

After the developer annotates the source code, the GenArch tool processes these annotations and generates initial versions of the models. The models are automatically derived by parsing the directory that contains the implementation elements. In this parsing step, each @Feature annotation demands the creation of a new feature in the feature model, and the creation of a mapping relationship between the created feature and the respective annotated implementation element in the configuration model. The GenArch tool also generates code templates based on the @Variability annotation. After the generation of the initial versions of GenArch models, the domain engineer can refine them - including, modifying or removing any feature, implementation element or mapping relationship.

The JUnit product line implementation model contains all JUnit implementation elements and templates. The configuration model specifies the mapping relationships between JUnit implementation elements and features from the JUnit feature model. Some mapping relationships can be created automatically based on GenArch annotations, such as: (i) the mapping between `TestCase` class and **Test Case** feature can be created based on the @Feature annotation from the `TestCase` class (Figure 1); and (ii) the mapping between `TestCaseTemplate` and **Test Case** feature can be created based on the @Variability annotation from the `TestCase` class (Figure 1). On the other hand, the mappings between some components need to be created manually. For instance, the mapping between the runner components (awtui, swingui and txtui) and runner features (**TXT**, **AWT** and **Swing**). The runner components are responsible to starting and tracking the execution of test cases and suites. JUnit provides three alternative implementations of test runners: command-line based user interface; an AWT based interface; and a Swing based interface. These mapping was not created automatically because it is not possible to annotate Java libraries.

2.3 Product Derivation Process in GenArch

The derivation process supported by GenArch, demands the specification of a feature model instance (also called a configuration) in which product variabilities are chosen and configured. The GenArch tool supports the SPL architecture customization by deciding which implementation elements need to be instantiated to constitute the final application requested and by customizing classes, aspects or configuration files. Each element that must be customized is represented by a template. The customization of each template is accomplished by GenArch tool using information collected by the *feature* and *product line implementation models*.

The last step of the derivation process in GenArch is characterized by the selection of existing implementation elements and the template-based code generation. The implementation elements that were selected and generated are then included in a source folder of a specific Eclipse Java project. The complete algorithm used by GenArch tool can be found in [4, 14].

3 Extending GenArch with Component-Based Technologies

In this section, we present the GenArch extensions that enable the automatic instantiation of product lines (and applications) implemented using Spring and OSGi component technologies. We use a web application to illustrate the proposed extensions. The web application is a simple shopping store that allows the management of customers' orders. Its main features are: (i) registering customers' orders; (ii) presenting administration reports - such as the number of orders by customer, orders that contain expensive products and list of expensive products; and (iii) logging of application´s operations, database queries, and exceptions thrown inside the application. In this application, we assume that the reports generation and the logging are optional features. Additionally, the logging feature also offers two alternative ways of persistence: database and xml files, respectively. Figure 2 shows the feature model of this web application.

The shopping store application is structured according to the Layer pattern [9], following the traditional web architecture of three layers [22]: web (front-end), business and data access. It is organized in terms of six main components: (i) `web` – it specifies the Java classes responsible to process the user web requests; (ii) `service` – defines the base business services offered by the application; (iii) `data` – defines the classes that implement the database access; (iv) `reports` – aggregates the business classes the implement the application reports; (v) `logging` – provides different implementations of the logging crosscutting feature (**Query**, **Exception** and **Operation**); and (vi) `util` – it is composed of the utility classes.

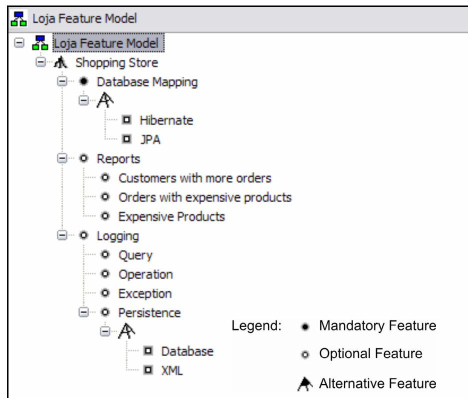


Fig. 2. Shopping Store Feature Model

Next subsections illustrate how this web application and its variabilities can be automatically instantiated using the GenArch extensions provided to support the Spring and OSGi component models.

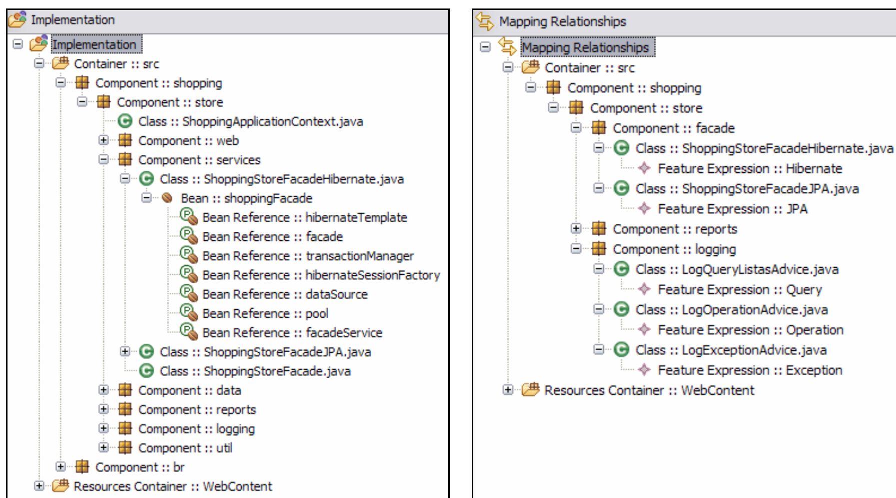
3.1 Spring Framework

Spring [19] is an open-source framework created to address the complexity of Java enterprise application development. Spring enables the development through use of components, called POJOs (*Plain Old Java Objects*) [19]. Each POJO contains only *business logic*. The Spring framework is responsible for addressing additional features (e.g., transaction, security, logging), which increment the base functionality provided by POJOs with characteristics required to build enterprise applications.

Spring makes it possible to use Java Beans component model to address the design of Java based enterprise applications in a flexible way, as opposed to complex component models like Enterprise Java Beans [15]. However, the usefulness of Spring is not limited

to server-side development. Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling. The simplicity and loose coupling is reached by the inversion of control principle [12] (IoC), also called dependency injection [12], and the aspect-oriented container provided by the Spring framework. In the IoC technique, the objects are passively given their dependencies instead of creating or looking for dependent objects for themselves. A component expresses its dependency on other components by exposing setter methods or through constructor arguments. Due to this approach, Spring components are simpler to write and maintain. The Spring AOP (Aspect-Oriented Programming) framework provides a flexible solution for addressing crosscutting enterprise concerns, such as transaction management, logging and security. Spring container uses a XML configuration file to specify the dependency injection on application components. This file contains one or more bean definitions which typically specify: (i) the class that implements the bean, (ii) the bean properties and (iii) the respective bean dependencies. Additionally, this configuration file also defines which aspects will be applied to each bean (component) of the application.

In this work, we developed an extension to GenArch tool that enables the use of Spring technology in the implementation of SPL architectures. It allows the automatic instantiation of applications during product derivation by helping the decision of which Spring components (beans) will integrate the final product. In our implementation, we extend the GenArch product line implementation model to incorporate the Spring Bean abstraction. In this new model version, each Java class (a POJO in Spring terminology) can be associated with a bean abstraction, which can be related with other beans. Based on this description, GenArch can choose which beans will compose the final application and automatically generate a specific Spring configuration file for this final application.



(a) Product Line Implementation Model (b) Configuration Model

Fig. 3. Shopping Store GenArch Models

Figure 3(a) shows the product line implementation model of the shopping store web application with some associated Spring beans. The `ShoppingStoreFacadeHibernate` class that implements the Facade design pattern (see Figure 3(a)), is associated with the `shoppingFacade` bean. It means that this class implements a Spring Bean, called `shoppingFacade`. This bean depends on different beans, marked with the `Reference` abstractions in the GenArch implementation model, such as: `hibernateTemplate`, `facade`, `hibernateSessionFactory`, `pool` and `facadeService`, `transaction-Manager`, `dataSource`. The Spring beans definitions can be created manually or automatically in the product line implementation model. The automatic creation is based on the Spring configuration file defined by domain engineers during the SPL implementation. The GenArch tool also parses the configuration files while processing the annotations and automatically generate the derivation models (Section 2.1).

The configuration model that incorporates the Spring extension, keeps the same characteristics of the GenArch original version (Section 2). The domain engineers must create mapping relationships between the features and the implementation elements in the configuration model (Figure 3(b)). If a specific Java class, which is also a Spring bean, is marked with a @Feature annotation, the mapping relationship between that class and the feature specified in the annotation is automatically created in the configuration model.

```
01.«IMPORT br::pucrio::inf::
02.      les::genarch::models::feature»
03.«EXTENSION br::pucrio::inf
04.      ::les::genarch::models::Model»
05.«DEFINE Main FOR Model»
06. [...]
07. <beans>
08.<bean id="shoppingFacade"
09.      class=" shopping.store.data
10.      .ShoppingStoreFacadeHibernate">
11.
12.      <property name="template">
13.      <ref bean="hibernateTemplate" />
14.      </property>
15. </bean>
16. [...]
17. <bean id="facadeService"
18.      class="org.springframework.
19.      aop.framework.ProxyFactoryBean">
20. <property name="proxyInterfaces">
21.      <value>shopping.store.
22.      services.ShoppingStoreFacade</value>
23. </property>
24. <property name="interceptorNames">
25.      <list>
26.      «LET feature("exception",
27.      featureElements) AS e»
28.      «IF e.isSelected »
29.      <value>ExceptionAdvise</value>
30.      «ENDIF»
32.      «ENDLET»
33.      [...]
34.      </list>
36. </property>
37. [...]
38. </bean>
39. [...]
28. </beans>
```

Fig. 4. Spring Configuration File Template

Each Spring application configuration file is defined as an XPand template [17] in our extension. These templates are processed by GenArch tool in two steps: (i) customization of the bean tags; and (ii) choice of the beans tags that will compose the final application configuration file. Figure 4 shows the template used to generate Spring configuration file of the shopping store application. During the product derivation process, this template is processed to customize its respective variabilities. For example, the property called `interceptorNames` need to be configured in agreement with selected logging policies. In this template (lines 26-32), the Spring AOP Proxy Interface, which are responsible to intercept methods and weave advises, only weave the `ExceptionAdvise` if the feature with `id` **exception** (line 26) was

selected (line 28). After the previous configuration, the GenArch chooses, based in the feature model instance and the mappings on the configuration model, which Spring beans will compose the final application.

At the end of the derivation process, the Java classes (representing beans) and the customized configuration file are then loaded in a specific source folder of an Eclipse Java project that represents the product.

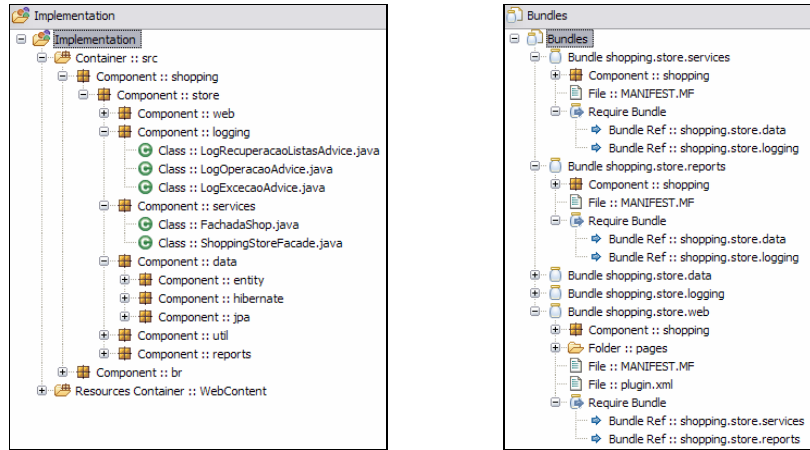
3.2 OSGi Technology

The Open Services Gateway Initiative (OSGi) [16] is a consortium of approximately eighty companies from around the world that collaborate to create a platform and infrastructure to enable the deployment of services from wide area networks to local networks and devices. The OSGi specifies an open, common architecture to develop, deploy and manage services in a coordinated fashion. According to the OSGi specification, Java applications are structured into a set of bundles. Each bundle represents an application component that provides services to the end-user or other components. A bundle is defined as the only entity responsible for deploying Java applications. It is typically deployed as a Java .jar archive file that contains, besides other implementation resources (e.g., classes, aspects, pictures), the manifest file which comprises information about the bundle. This information includes the location of a class, called the activator, that is called when the installed bundle is started or stopped. The manifest also contains other interesting information, such as package dependencies, used and provided services and additional general information about the bundle.

In the present study, we developed a GenArch extension that enables the customization and configurable deployment of OSGi bundles through a feature model. Our extension enables the customization in two levels: (i) the definition of resources (classes, files, etc) that compose the bundles; and (ii) the definition of bundles that will be part of the final application.

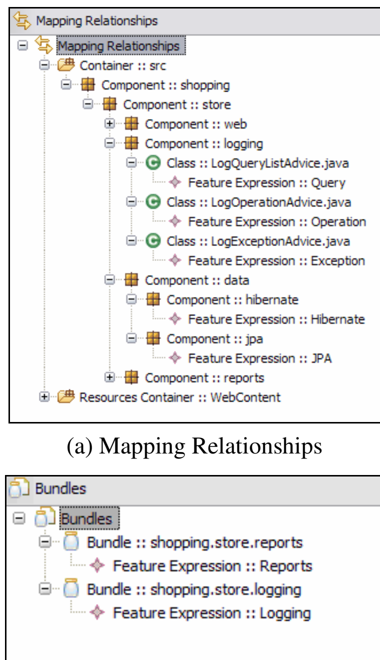
Figure 5 shows the product line implementation model of our shopping store case study using the OSGi extension developed to GenArch tool. Figure 5(a) shows the implementation elements in the traditional view of the product line implementation model. This view was already supported in the base version of GenArch (Section 3). Figure 5(b) shows the deployment OSGi view of GenArch considering the shopping store case study. It specifies the bundles that implement the application or product line. The `shopping.store.service` bundle implements the application facade component. It requires two other bundles: (i) `shopping.store.data` – that implements the database access; and (ii) `shopping.store.logging` – that implements the logging crosscutting feature. The `shopping.store.web` bundle implements the application web interface. It requires the services implemented by the `shopping.store.service` and `shopping.store.reports` bundles.

The OSGi technology requires the definition of a manifest file (MANIFEST.MF) for each bundle of an application. The main information about bundles is described in the following properties: `Bundle-Name`, `Require-Bundle`, and `Export-Package`. In our OSGi extension, GenArch is responsible for customizing this file during *product derivation* in order to include the specification of bundles as part of the final generated application. More specifically, these three properties are customized based on information provided by the derivation models (feature, configuration and implementation).



(a) Implementation Elements (b) Bundles

Fig. 5. OSGi Product Line Implementation Model



(a) Mapping Relationships

(b) Bundles Relationships

Fig. 6. OSGi Configuration Model

Two levels of configuration are supported by our GenArch OSGi extension. Figure 6 illustrates these levels by showing the different views of the configuration model for an OSGi based product line. In the first level, the domain engineer can define fine-grained configurations by creating mapping relationships of specific implementation elements to any feature. Figure 6(a) shows, for example, that each aspect of the logging component depends on a specific logging optional feature from the feature model. The `LogQueryListAdvice` aspect, for example, depends on the `Query` feature. The definition of such mapping relationships (in the configuration model) enables our tool to decide which elements will compose the final bundles of a specific product.

In the second level, the domain engineer can define mapping relationships between bundles and any feature. Figure 6(b) illustrates a new view provided by the GenArch OSGi extension to the configuration model. It allows the definition of specific bundles of a product line, according to the features selected to be included in the product (during *application engineering*). The `shopping.store.logging` bundle, for example, depends on the occurrence of the **Logging** feature.

The information provided by the OSGi configuration model enables the GenArch tool to decide which bundles will compose the final product, based on the feature selection. During the product derivation, our tool proceeds in the following way to generate each bundle of the final product: (i) it creates an Eclipse plug-in project; (ii) it loads the selected implementation elements and template generated elements in this project; and, finally, (iii) it customizes the `Bundle-Name`, `Require-Bundle` and `Export-Package` fields in the OSGi manifest file. The fields of the OSGi manifest file are customized based on the information available in the product line implementation model and feature model instance. These models work as DSLs that provide custom information for the template processing. Differently from previous versions of GenArch, which demands the derivation of only one Eclipse Java project, the OSGi extension generates one Eclipse project for each bundle, because the OSGi implementation requires the definition of one project per bundle.

4 Discussions and Lessons Learned

Integrating Spring and OSGi. Some recent works [20] have emphasized the combined adoption of Spring and OSGi as complementary component technologies. While the Spring framework offers a flexible and effective component model to manage static and more fine-grained component dependencies of both crosscutting and non-crosscutting services, the OSGi provides a dynamic runtime infrastructure that allows the management of components in runtime. The GenArch extensions presented in this work already take into consideration the possibilities and benefits for integration of both technologies. The developers can create and manipulate a series of Spring beans in the product line implementation model, and after that they can assign to a specific OSGi bundle the Java classes that implement the Spring beans. Although our tool already addresses these scenarios, it has not considered the implementation of the Spring OSGi module [20], which is currently under development. This module is responsible for providing a smooth integration between Spring and OSGi frameworks by allowing an OSGi application to import and export Spring packages and services. We are currently investigating this new Spring support to OSGi in order to provide support to it in our product derivation tool.

Runtime Customization of Product Lines. Many of the component infrastructure technologies developed over the last years have emphasized the need to provide support to dynamic customization of applications. J2EE technology, for example, enables the dynamic deployment of enterprise beans components. The deployment of new components or the management (e.g, update or removal) of existing ones is supported by means of mechanisms provided by the application servers, such as, JBoss. OSGi technology also allows flexible dynamic management of components. However, it has not been much explored in the context of Java server-side applications. The Spring OSGi [20] module is an initiative in this way. The Spring and OSGi extensions to the GenArch tool proposed in this work already represent an advance in the use of these technologies to implement product lines architectures. They can enable the dynamic customization of product lines. However, this customization is accomplished mainly based on the product line components. One interesting direction to investigate is to

explore the dynamic customization of product lines based on the feature model. In this kind of approach, feature based tools would drive the customizations based on the selection of features. The feature selection performed by the application engineer would demand the automatic deployment (e.g., removal or updating) of several components associated to the selected features.

5 Conclusions and Future Work

In this paper, we presented two extensions to a product derivation tool which address the integration of Spring and OSGi mainstream component-based technologies. Our extensions consider the use of these technologies in the implementation of SPL architectures by incorporating their abstractions and mechanisms to the derivation models adopted by GenArch tool [4]. Automatic mechanisms are used to generate partial version of these models based on the specific artifacts (configuration and manifest files) and abstractions (beans, aspects, bundles, dependencies, etc) of Spring and OSGi technologies. During the product derivation process, the GenArch tool enables the automatic instantiation of product lines (or applications implemented using the mechanisms available in Spring and OSGi) by selecting and customizing components based on a set of selected features.

As a future work, we intend: (i) to apply and evaluate the proposed extensions in the context of complex component based product lines; (ii) to address the support to the under development Spring OSGi module; (iii) to investigate the use of feature models in the dynamic customization of product lines.

Acknowledgments. The authors are supported by LatinAOSD/Prosul Project - CNPq/Brazil. Uirá is also partially supported by European Commission Grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

[1] Alves, V., Matos, P., Cole, L., Borba, P., Ramalho, G.: Extracting and Evolving Mobile Games Product Lines. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 70–81. Springer, Heidelberg (2005)

[2] Anastasopoulos, M., Muthig, D.: An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In: Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, pp. 141–156. Springer, Heidelberg (2004)

[3] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, A System of Patterns, vol. 1. Wiley, Chichester (1996)

[4] Cirilo, E., Kulesza, U., Lucena, C.: GenArch: A Model-Based Product Derivation Tool. In: Proceedings of Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2007), Campinas - Brazil (August 2007)

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

[6] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)

[7] 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)
[8] Fayad, M., Schmidt, D., Johnson, R.: Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons, Chichester (1999)
[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, vol. 395. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1995)
[10] Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. John Wiley and Sons, Chichester (2005)
[11] Harold, E., Means, W.: XML in a Nutshell. O'Reilly, Sebastopol (2004)
[12] Johnson, R.: Expert One-on-One J2EE Design and Development. Worx (2002)
[13] Kang, K., et al.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Pittsburgh, PA (November 1990)
[14] Kulesza, U.: An Aspect-Oriented Approach to Framework Development, PhD Thesis, Computer Science Department (in Portuguese), PUC-Rio, Brazil (April 2007)
[15] Monson-Haefel, R.: Enterprise JavaBeans. O'Reilly, Sebastopol (2001)
[16] OSGi, http://www.osgi.org
[17] openArchitectureWare, http://www.eclipse.org/gmt/oaw/
[18] Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM TOSEM 11(2), 215–255 (2002)
[19] Spring Framework, http://www.springframework.org
[20] Spring OSGi, http://www.springframework.org/osgi
[21] Weiss, D., Lai, C.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley Professional, Reading (1999)
[22] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, Reading (2002)