SPLIT: An Automated Approach for Enterprise Product Line Adoption Through SOA

Carlos Parra* and Diego Joya
Heinsohn Business Technology - Colciencias
Bogotá, Colombia
{cparra, djoya}@heinsohn.com.co

Abstract

Nowadays, the software industry is faced with challenges regarding complexity, time to market, quality standards, and evolution. To face those challenges, two strategies that are gaining interest both in academy and industry are Service Oriented Architecture (SOA) and Software Product Lines (SPL). While SOA aims at building applications from an orchestration of services, SPL consists in building families of products instead of individual applications through the development of common core-assets. Adopting such approaches requires changes in the development process regarding existing software artifacts that must be transformed in order to respect an architecture that focus on modularity and reuse. This paper presents the Software Product Line Integration Tool (SPLIT), our strategy to such transformation in Heinsohn Business Technology (HBT). We propose a non-intrusive reverse engineering process for the development of modular services obtained automatically from existing software artifacts, and a variability-driven derivation process to assembly products out of such services. To validate our approach, we have implemented and tested SPLIT using real software artifacts from a framework of reusable components for several enterprise applications. The results show important benefits in terms of the development time and flexibility.

Keywords: Software Product Lines, Model-driven Engineering, Generative Programming, Service Oriented Architectures.

1 Introduction

In recent years the software industry has been faced with new challenges regarding complexity, time to market, quality standards and evolution. To face those challenges, two strategies that are gaining adepts both in academy and industry are Service Oriented Architecture (SOA) and Software Product Lines (SPL).

In an enterprise environment, as applications grow in complexity and size, they are typically divided into a set of independent modules that communicate with each other through provided and required services. SOA aims at modularization as a way to tame this complexity. By dividing big processes into smaller subprocesses, architects can design and implement separate modules and combine them to tackle different business applications.

On the other hand, Software Product Lines (SPL) aim at changing the focus from building individual applications to building families of products by exploiting commonalities and variabilities across the members of the product family [7]. This results in an increased reuse and reduced time to market as shown in several experiences (*e.g.,* [20, 11, 30]). An SPL is particularly useful in environments where multiple software products share an important set of common components.

While SPLs aim at identifying variability and commonalities in a product family, SOA aims at modularization and development of independent services that, when combined, form bigger software applications. One can notice that variability and commonalities in the SPL world can benefit from the

Journal of Internet Services and Information Security (JISIS), volume: 5, number: 1 (February 2015), pp. 29-52

*Corresponding author: Carrera 51 #106-86 Apto 303, Bogotá, Colombia, Tel: +57-3017819460, Secondary email: canesito@gmail.com

modularization and decoupling targeted in the SOA world. As a result, SPL and SOA might be complementary, as shown in several experiences that have already explored the possibilities of combining such approaches in the software development process [23, 22, 21].

Adopting such approaches within an organization involves important challenges with regard to existing software artifacts that must be transformed in order to respect an architecture focused on modularity and reuse. In this paper, which is an extended version of the SAC'2014 conference paper published in the Service Oriented Architectures and Programming (SOAP) track [28], we present in detail our approach to such adoption in Heinsohn Business Technology (HBT). HBT is a software development company specialized in financial, transportation, mortgage-backed securities, and pension-fund solutions. We propose a non-intrusive reverse engineering process for the development of modular services obtained automatically from existing software artifacts, and a variability-driven derivation process to assembly products out of such services.

We present an automated Software Product Line Integration Tool (SPLIT) that transforms current JEE artifacts into SPL assets that communicate with each other using services. SPLIT implements the two main processes of any software product line: domain engineering and application engineering [7]. For the domain engineering process, SPLIT provides the tools to analyze source code and create high-level core assets. For the application engineering process, SPLIT uses the assets to derivate products that respect an SOA architecture communicating with each other through services. SPLIT requires the code to be annotated with the provided annotations. After that, all the tasks for source code analysis, model transformations, code generation, and deployment, are fully automated and do not require any further manual development.

To evaluate the benefits of SPLIT we use several mature Java Enterprise Edition (JEE) artifacts developed in HBT for different projects covering financial, transportation, mortgage-backed securities, and pension-fund solutions. Such artifacts are used as input of the domain and application engineering processes that generate products. The results of using SPLIT reflect important benefits in terms of reduced derivation times due to the automated process of product derivation, and flexibility due to the freedom gained to combine loose coupled artifacts through contracts in various technologies.

The remainder of this paper is organized as follows. In Section 2 we introduce the product derivation problem and the motivation for modularization through services. In Section 3 we present our approach and introduce the slicing strategy. Section 4 presents the implementation of SPLIT in detail. Section 5 presents the results of our experimentation using real-world JEE artifacts. In Section 6 we discuss the limitations of our approach. Finally Section 7 presents the related work and Section 8 concludes the paper and points out several paths for future work.

2 Motivation

Reinforce the idea of software reuse across different software projects is a constant objective in software development companies. When adopting an SPL strategy, reuse is based on the development of core assets. According to Clements and Northrop [7], there are two ways to build such assets: from scratch, or from existing software artifacts. In the first case, every asset is built to fulfill the requirements of the SPL . In the second case, current artifacts are modified to be in accordance with the SPL architecture.

In our case, we start from a framework of software artifacts whose main goals are: (1) to ease the implementation of frequently used functionalities, and (2) to reduce the development time and avoid building similar artifacts multiple times. Such framework currently has several artifacts that vary in size from 2000 to 15000 lines of code (LoC), and keeps on growing with new common functionalities being added frequently. Current artifacts cover a wide area of requirements varying from common ones like security or file management to more business-related ones like accounting, people management,

or business commissions. This framework offers an ideal starting point for the development of core assets for the SPL. However, in order to transform such artifacts into core assets, we identify three main challenges described here below.

1. ***Loose coupled assets through SOA***

   Currently all the artifacts are developed under the *Java Enterprise Edition* (JEE) platform and their functionalities are exposed through *Enterprise Java Beans* (EJB). Additionally, there is a separation at the level of data which means that every artifact in the framework is responsible for its own data model. However, artifacts in the framework are tightly coupled and their distribution is error-prone. There are hard-coded dependencies that are resolved by the implementation platform which makes it difficult to automatically plug and unplug such artifacts.

   The first challenge refers to the automated generation of SOA assets without modifying current artifacts. Such assets must respect an SPL architecture with clear interfaces to describe provided and required services, and define dependencies through contracts instead of platform-dependent hard-coded attribute declarations.

2. ***Variability driven derivation***

   In [27], we have presented an analysis of the variability identified in HBT. For this analysis, we have used both the documentation of the artifacts and discussions with the architects and engineers responsible for the development of each artifact. Nevertheless, in order to trigger the derivation process from this model, a well-defined binding has to be established between the features in the model and one or more SPL assets. With assets based on an SOA architecture, products can be expressed in terms of features in the variability model, and derived through the assembly of their services. However, for the variability model to trigger the derivation process, it needs to be accurate with regard to current implementations.

   The second challenge refers to the enrichment of the variability model to represent updated information about current implementations of the artifacts available, and the implementation of a derivation process to build actual software products out of product configurations.

3. ***Automated product derivation***

   One last issue for the product line refers to actual derivation and deployment of products. Currently, each project can use one or more artifacts from the framework by following an installation process that includes tasks like: implementing several classes for the specific technology, creating several configuration files, running the scripts that prepare the data model in the database, and developing the client-code to consume the functionalities provided by the artifacts. Additionally, several parameters and configuration scripts may change depending on the selected application server and database vendor.

   The third challenge refers to the development of a simplified product derivation process by automating the time-consuming activities associated with implementation, setup, and deployment.

In Section 3 we present our approach to face these challenges by means of an analysis and development of assets from current artifacts, and a product derivation based on the assembly of such assets.

3 The SPL Adoption Strategy

Our strategy for SPL adoption is based in two main processes: (1) identifying and building core assets for each artifact found in current implementations, and (2) deriving products through the generation of glue

code and assembly of the obtained SOA components. Such processes correspond to the ones defined in [7] for any SPL: domain engineering, and application engineering.

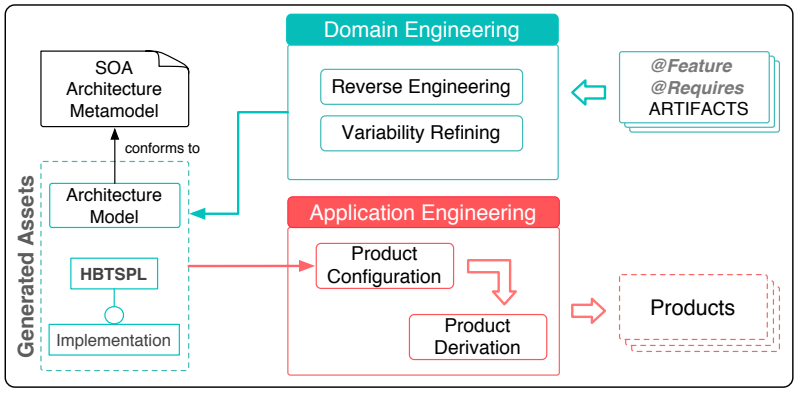

Figure 1: Approach and SPL processes.

Figure 1 illustrates our approach. As it can be seen, the processes of domain and application engineering go in opposite ways. First, during the domain engineering phase, the core assets are built from the existing software artifacts. Next, during the application engineering phase, such assets are used to obtain different products. The joining point in the middle of both processes is the variability model. This model is the result of an analysis to identify and gather the levels of variability inherent to applications developed in HBT. It is important to notice that the abstraction level changes as a result of each process. In the figure, from right to left, source code is analyzed to build high-level SOA models to represent current software artifacts. Conversely, from left to right, these models are used to generate glue code enabling products to be formed as an assembly of multiple SOA components. We start from the upper right part of the figure with current software artifacts in the framework, and generate: (1) high-level core assets defined as models that conform to an SOA architecture metamodel, and (2) an enriched variability model with updated information of current implementations. After the core assets have been built, the derivation process takes place. It starts with a product configuration expressed in terms of the features available in the implementation variability model. Each feature is linked to an asset that will communicate with each other through services. Currently we support three types of bindings for the services (EJB, Web-Services and REST). The selected features in the configuration are used to customize the architecture model which is the input for the generation of glue code to assembly the assets into functional software products. In this section we present in detail the variability analysis as well as the slicing strategy to obtain core assets from current software artifacts.

3.1 Variability analysis

In order to successfully build an SPL, it is necessary to take into account the multiple levels of variability in HBT applications. For the SPL adoption, we have defined a multi-level variability model [27] with three different levels: (1) the business level which represents the business requirements of the product being built, (2) the framework level which represents the different functionalities that are independent from any particular business, and (3) the platform level which represents the constraints of the specific execution platform in terms of software and infrastructure. To model each level, we use the Feature Oriented Domain Analysis (FODA) [16] terminology that distinguishes three types of features: (1) *mandatory* features (dark circles) which are always selected, (2) *optional* features (white circles), which can be both selected and deselected, and (3) *alternative* features (inverted arc), a special kind of optional features

where the choice is realized among a limited set of alternatives that can be exclusive (XOR) or not (OR). In addition to that, the diagram introduces two types of constraints among features: *requires* and *excludes*. The *requires* constraint states that for a given feature to be selected, the required feature has to be selected before. The *excludes* constraint states that for a given feature to be selected, the excluded feature has to be deselected. A simplified view of the model is illustrated in Figure 2 using a feature diagram.

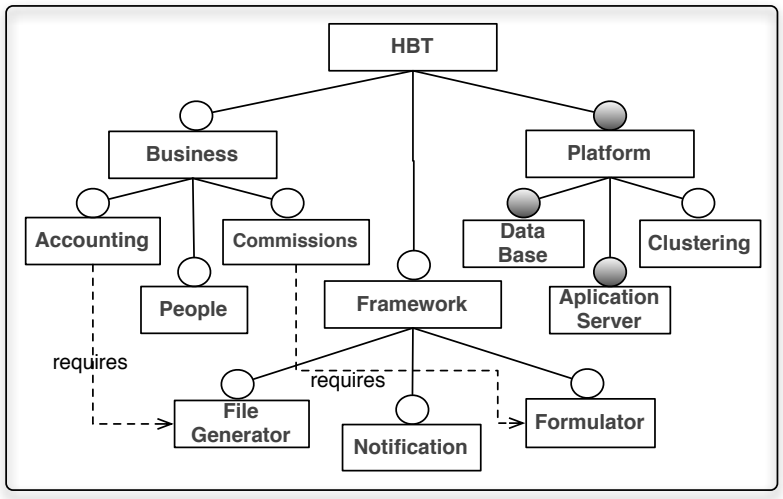

Figure 2: The variability model.

3.1.1 Business variability

Business variability refers to the different reusable functionalities that are tightly coupled with certain specific business. For example, Figure 2 presents three business features for: accounting, people, and commissions. As it can be noticed both the `Accounting` and `Commissions` features have a `requires` relationship towards `FileGenerator` and `Formulator` respectively, meaning that any configuration including one or both of the former features needs to include the latter ones to be valid. The features in this group target specific needs that vary for each client and business domain. As a consequence business features are smaller in number than functional features that are agnostic to the underlying businesses. Business features limit the product family size to target specific businesses. In the `HBT SPL` we focus mainly on financial and pension funds enterprise applications.

3.1.2 Framework variability

Framework variability refers to the different functionalities that support the business features. For instance, the `FileGenerator` feature represents the functionality for generating files with a specific formatting. It includes different options for file types, line grouping, validations, compression, encryption, and listeners. The `Notification` feature represents the functionality for sending a message to the final user, whenever various configurable events occur in the application. None of the features in this level are mandatory. They are independent of any business choice, and can be useful in many products. For instance, currently, most of the applications developed in the company provide some kind of file management and notifications. In this level, we have identified other features for functionalities

like: database mappings and exports, security, automated processes, formulae management, and digital signature among others.

3.1.3 Platform variability

Platform variability refers to the requirements of the platform where the software products are executed. The goal of this level is to allow architects to select business and functional features, and validate whether such features are supported on a specific platform. Features in this group have a different nature because they do not represent available software artifacts. These features correspond to external software and infrastructure needed by the product. Consider the platform subtree in Figure 2. The features correspond to database managers, application servers, and frameworks. Each feature includes different vendors and versions. Such decomposition allows us to have a granular management of the support offered by the assets in the previous levels. Certain features in the subtree are mandatory because are essential for the correct deployment and execution of every product in the `HBT` family (*e.g.,* databases, application servers, operating systems). Other optional features in this level include: caché, clustering, security, and deployment.

3.2 Slicing artifacts into assets

Once the features have been identified in the variability model, it is necessary to create the `SPL` assets that realize such features. As stated by Czarnecky and Antkiewicz [8], features in a feature model are merely symbols. Mapping features to other models such as behavioral or structural models is what gives them semantics. Different approaches can be followed to identify and build assets. In our case, since we start from the artifacts in the framework, we follow an *extractive* approach [19]. High-level core assets are created using as input the source of current implementations. An asset represents a set of software artifacts and as such, it enables us to create direct links between a feature in the variability model and multiple artifacts in the implementation. To model the `SPL` assets, we use the metamodel depicted in Figure 3, which is a simplified version of the metamodel presented in [26]. The metamodel defines a generic service-based structure similar to the Service-Component Architecture (SCA) [5]. It represents a set of rules that an artifact must respect in order to be a core asset of the `SPL`. These rules restrict component interfaces and contracts, and do not interfere with the way each component is implemented inside (*e.g.,* black-box), excluding aspects like transaction management or database model.

The root of the metamodel is the `Model` that may contain several `Components`. A `Component` is a single unit of composition and is linked to one feature in the variability model. It also offers a set of `Services` and requires a set of `References`. Both `Services` and `References` are connection points between components. A connection point also defines a `Contract`, which represents the interface, provided or required, depending on the type of connection point. Finally, a contract consists of a set of `Methods` that can have a set of `Parameters` and a return type (`returnType`).

The advantages of having this architecture metamodel are twofold: (1) it helps us to define a common structure to represent existing artifacts as `SOA` components, and (2) it can be used as a guide to model and build new core assets for the product line in different projects within the organization. For this reason, this architecture only restricts the way components expose and consume services. This allows us to model diverse artifacts, but at the same time, it allows us to build a generic derivation process that assemblies any combination of components.

The architecture metamodel becomes the target of the core asset generation process. To identify and generate assets from current artifacts, we define a source code analysis based on Java annotations. An annotation enables developers to add meta-data about the programs it annotates without modifying the structure or behavior. Using annotations, developers can indicate manually which parts of the original

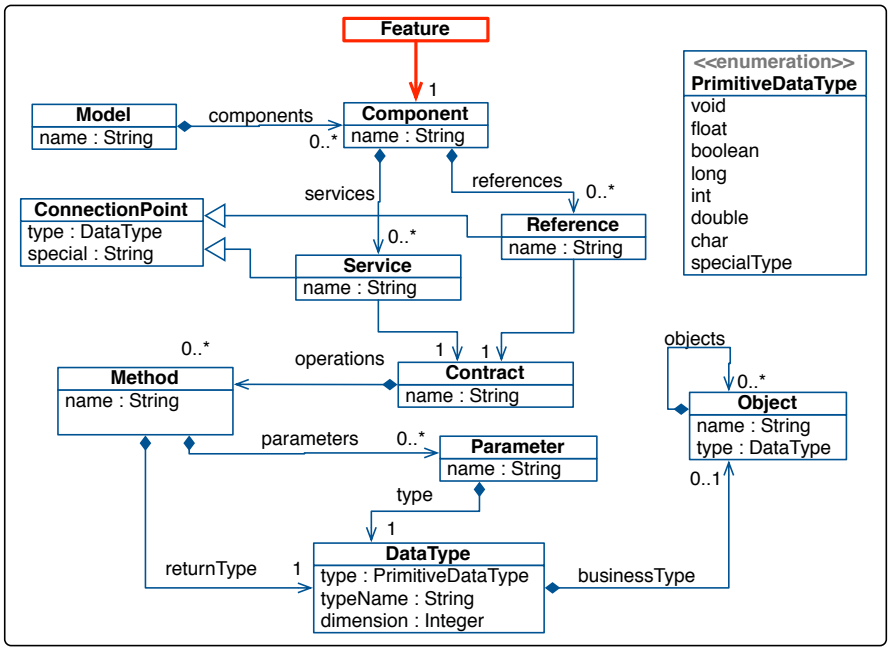

Figure 3: Generic architecture metamodel

artifacts can be exposed as services to be consumed externally (*e.g.,* by other components in the same deployment unit or by remote applications in a distributed environment). Two annotations are at the center of our approach: `@Feature` and `@Requires`.

3.2.1 Annotation `@Feature`

This annotation indicates what parts of the artifact are exposed as services and how they are linked to features in the variability model. It is used to annotate Java interfaces, classes, and methods. It has one attribute called `featureName` that indicates the feature in the variability model associated with the Java type being annotated. Using this attribute it is possible to establish a link between the annotated source code and a feature in the variability model presented in Section 3.1. The `@Feature` annotation is illustrated in the snippet of code in Figure 4. As it can be seen, the annotation is used to mark the interface (line 1) and the methods (line 4) that are going to be exposed as part of the service.

```
1 @Feature(featureName = "OfficeNotificationAdapter_LF")
2 public interface NotificationGenerator {
3
4     @Feature(featureName = "OfficeNotificationAdapter_LF")
5     public NotificationStatus buildNotification(Long notificationTypeID);
6
7     public void buildNotification(Long typeID, Map<String, Object> paramsDest);
8 }
```

Figure 4: The `@Feature` annotation.

3.2.2 Annotation `@Requires`

This annotation is used to break hard-coded dependencies that may exist between current software artifacts. In the architecture model, components must be independent from each other and communicate through services and references that follow a contract. Contracts specify the communication of a given component in terms of the services it offers or the services it requires.

The `@Requires` annotation is used on Java methods inside implementing classes and indicates that, within the body of such methods, there are one or more statements that invoke external services. These invocations must be transformed into references of components to obtain a loose coupled architecture. The `@Requires` annotation has a collection of `@Require` annotations inside. This is done because there might be multiple references to external artifacts within the body of a method, and it is not possible to use the same annotation type multiple times. This means that there will be as many `@Require` annotations as dependencies to external artifacts exist in the body of the method, and these annotations will be placed inside a single `@Requires` annotation. The `@Require` annotation has three attributes, `requiredFeatureName` to indicate the feature it requires, `providedFeatureName` to indicate the feature it provides (*i.e.,* the feature associated to the element where the method belongs), and eventually an interface name for the cases when multiple interfaces are associated with a single feature. An example of the `@Requires` annotation usage is illustrated in Figure 5. As it can be seen in the figure, we are indicating that the method `buildNotification` in the class `NotificationOfficeBean` belongs to the feature *OfficeNotification_LF* and uses the services exposed in the component associated with the feature *Notification_LF*.

```
1 public class NotificationOfficeBean {
2  //External service
3  NotificationGenerator notification;
4
5  @Requires({
6   @Require(
7     requiredFeatureName = "Notification_LF",
8     providedFeatureName = "OfficeNotification_LF",
9     interfaceName       = "NotificationGenerator")
10  })
11  public void buildNotification(NotificationDTO notDTO);
12   //dependency invocation
13   notification.buildNotification(
14         notDTO.getNotificationTypeId(),
15         notDTO.getDestinations());
16  }
17 }
```

Figure 5: The `@Requires` annotation.

So far we have presented the slicing strategy for the development of core assets. Annotations are used to mark the current software artifacts source code and slice them into assets that conform to an *SCA-like* architecture. The next section presents the implementation details for the processes of domain and application engineering. These processes fully automate the development of core assets as components in the architecture model, and the derivation of products using these components.

4 SPL Implementation

For the implementation of the SPL, we have developed a tool called SPLIT. As illustrated in Figure 1, SPLIT is separated in two different phases: domain and application engineering. SPLIT requires one manual input from the developer for each process respectively. For the domain engineering, it is necessary to annotate the code with the specific SPLIT annotations explained in Section 3. For the application engineering, the developer must create a product configuration where she selects the features for the product to be built. SPLIT automates every other task required to build a product like: source code analysis, model transformations, code generation, and deployment.

4.1 Domain Engineering

The reverse engineering process builds the core assets for the product line, using as input the annotated source code in the available software artifacts. The first step consist in manually annotating the artifacts using the annotations explained in Section 3.

In order to build a generic process for source code analysis, the Java code annotated with the `@Feature` must follow a set of rules as illustrated in Table 1. Three different kinds of Java types are supported in SPLIT: (1) stateless EJBs (that can be local or remote), (2) plain old java objects (POJO), and (3) classes with a singleton constructor. Each type has a specific way of being referenced by others. Stateless EJBs are referenced via a lookup using the EJB functionalities of the JEE container. Single classes are created through their constructor, and finally, when using the singleton pattern classes are built invoking a method (typically named `getInstance()`) that creates only one instance of the class. Since each case represents a different implementation in terms of the Java syntax used to obtain the object, SPLIT is limited to these types of instantiations to ease the process of automatically replacing them with invocations via contracts. For each case, the `@Feature` annotation may be placed on interfaces or classes. The `@Requires` annotation on the other hand is always on methods inside implementing classes so that the process can access inner statements of the method.

Input	**Description**
Stateless EJB	Local or Remote EJB interface and implementing class - `@Feature` is placed on the Interface and its methods - `@Requires` is placed on methods inside classes
POJO	Simple Java class - `@Feature` is placed on the class and its methods - `@Requires` is placed on methods inside classes
Singleton	Java Class with a private constructor and a singleton instance initialization - `@Feature` is placed on the class and its methods - `@Requires` is placed on methods inside classes

Table 1: SPLIT input types.

Once all the artifacts available have been annotated, the reverse engineering process analyses the source code and through two different transformations it obtains: (1) the SOA architecture model, and (2) an enriched variability model with updated information of current implementations. The former phase looks for the interfaces and methods associated with features and through a series of model-based transformations [18], it creates a model representation of the artifacts as SOA components where functionalities (provided and required) are described with explicit contracts. The latter one, creates an implementation sub-tree in the variability model, and verifies the constraints described among features have equivalent requires relationships at the level of code.

For the actual implementation of these phases we use Spoon [29] and the Eclipse Modeling Framework (EMF[1]) respectively. Spoon is a tool that analyses and transforms Java code using processors. A processor is a Java class that allows developers to look for and modify elements of an application with a high level of granularity. Spoon creates an abstract syntax tree of the code being analyzed and offers the API to navigate through the tree and eventually perform modifications. EMF on the other hand provides the functionalities to create and manipulate models.

4.1.1 Architecture model

Using Spoon and EMF, SPLIT analyses the source code to capture the annotated java types in the artifacts and build a model that conforms to the architecture metamodel. The analysis works using both annotations as follows:

- `@Feature:` this annotation is used to generate the components and services in the SOA architecture. For every `featureName` used in the annotated code, a component is created in the architecture model. Inside each component, there are as many services as interfaces or classes found using such feature name. Every service has a contract that contains the methods exposed. When the annotation is found on an interface or a class, it indicates that a service and its contract have to be created, and when it is placed on methods, it indicates that those methods are part of the contract. In practice, only the methods annotated are included in the contract and exposed as part of the service.

- `@Requires:` this annotation is used to generate the `References` in the architecture model. For every method annotated with the `@Requires` annotation, a reference is created inside the component whose name matches the `providedFeatureName` value. Similar to the services, the references have a contract formed by method, but in the references case, these methods are used within the body of the annotated method and must be provided by the service of a different component.

It is important to notice that the methods in the contracts, for both services and references, preserve detailed information about the data types used as parameters and return types in the original source code including: arrays, complex collections with parameterized types (*i.e.,* generics), and business objects. This information represents the data contracts that have to be respected when building the bindings between components in various technologies. Consider the component diagram of Figure 6. It illustrates the obtained architecture model after processing the code presented in snippets 4 and 5. We use the stereotype notation (<<>>) from UML to indicate the *conforms to* relationship between the elements in the model and its correspondent meta-class in the metamodel.

As it can be seen in the figure, two components are created for the features `OfficeNotification_LF` and `Notification_LF`. The `OfficeNotification_LF` component has a reference to consume the service provided by `Notification_LF`. The contract in the middle specifies the signature of the methods of the service provided. This model is used during the product derivation process to generate the glue code that allows the assembly of components.

4.1.2 Variability Refining

Besides the architecture model, the domain engineering process also refines the variability model with accurate information about current implementations. Three different levels have been identified manually in the variability model as explained in Section 3.1: business, framework, and platform. With the refining

[1] http://www.eclipse.org/modeling/emf/

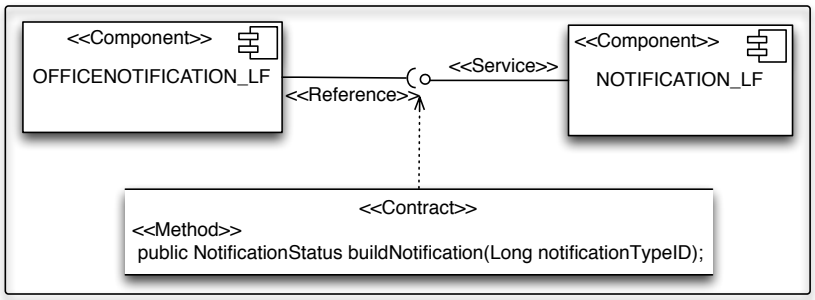

Figure 6: Sample architecture model

process, a fourth *implementation* level is added to the aggregated model. This level represents the actual implementations found in the annotated source code.

The input to the variability refining process is the annotated code. We defined a processor that generates a subtree in the variability model with the following structure: there is a root called `Implementation`. It has as many children as `featureName` values are found in the code. Below each child feature, the process creates a feature for each interface or class. To enable users to configure the technology used in the services to expose, the processor generates a subtree with three exclusive-alternative features (*i.e.,* only one of the three can be selected): EJB, WS and REST. Each feature in this level is required by its corresponding feature in the business or framework sub-trees. This constraint guarantees that for each functional or business feature selected, there will always be an implementation feature that is bounded to a component in the architecture model. Finally, thanks to the `@Requires` annotation, this process can also verify if the constraints expressed through annotations at the level of artifacts have equivalent *requires* constraints between the features in the variability model. If such constraints do not exist, they are added automatically. Figure 7 shows an example of the generated implementation subtree obtained as a result of the variability refining process for the `Notification` and `OfficeHelper` components. To the left of the figure, there are two features of the original framework level, and to the right of the figure, there is the generated implementation subtree (dashed lines) with the interfaces and classes for the features found in the annotated source code, and the subtree for the binding technology choice.

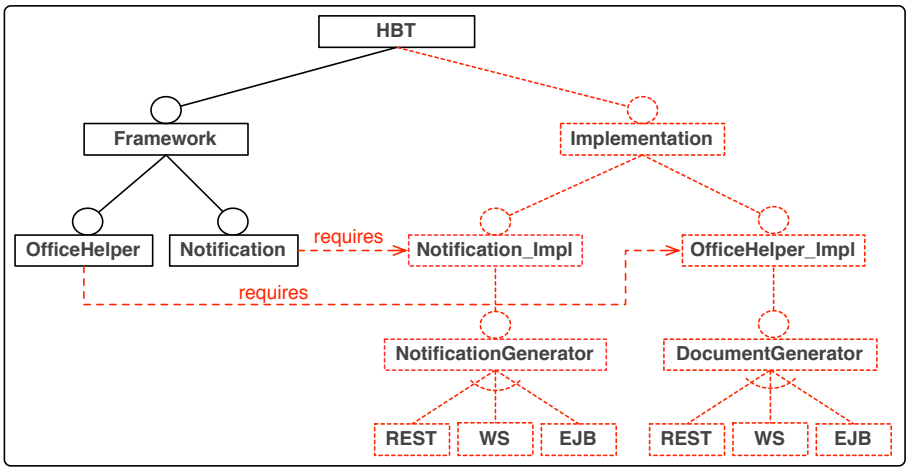

Figure 7: Enrichment variability result.

4.2 Application Engineering

The application engineering process builds a single product using a configuration expressed in terms of selected features from the variability model, and the architecture model. Three phases have been defined in the product derivation process: (1) configuration and composition, (2) server/client bindings generation, and (3) product deployment.

4.2.1 Configuration and Composition

The product configuration refers to the selection of a feature set to build a single product using the enriched feature model obtained after the variability refinement process. To represent the feature model and allow developers to configure a product, we use FeatureIDE [17]. FeatureIDE is a set of tools for variability modeling that enables one to create and edit feature diagrams. Furthermore, FeatureIDE provides a configuration tool to create and validate configurations with regard to the constraints defined in the variability model. Using FeatureIDE a developer can create product configurations and validate if such configurations respect the constraints expressed in the variability model.

Figure 8 illustrates a typical product configuration. We have chosen four features from the Framework level: (1) Notification, (2) OfficeHelper, (3) OfficeNotificationAdapter, (4) FileProcessing, and (5) MassiveLoader. Each feature selected in the Framework level has its correspondent implementation feature together with the chosen binding technology. In this case we have selected different bindings for each feature (e.g EJB, WS, REST).

$P_1 = \{$

Notification, Notification_IMP, NotificationGenerator_IMP_WS,
OfficeHelper, OfficeHelper_IMP, GenerateDocument_IMP_EJB,
OfficeNotificationAdapter, NotificationOffice_IMP, NotificationOffice_IMP_REST,
FileProcessing, FileProcessing_IMP, FileLoader_IMP_WS,
MassiveLoader, MassiveLoader_IMP, MassiveLoader_IMP_EJB

$\}$

Figure 8: Product Configuration.

Once the configuration is completed and validated via FeatureIDE, the selected features in the product configuration are used to find the corresponding components in the architecture model. Each product gets represented as a model that conforms to the architecture metamodel depicted in Figure 3. The product model only contains the components linked to the features chosen in the product configuration. Inside every component there is detailed information about services, references and contracts that are used in the subsequent code generation process.

4.2.2 Server and client binding generation

After the product model has been generated, the next step in the application engineering process is the code generation that transforms Java artifacts into SOA assets. To reduce the impact of this transformation, *bindings* are generated from the information about contracts in the product model. A *binding* encapsulates and exposes artifact functionalities through local or distributed SOA components. It represents an interface to provide (*i.e.,* server) or consume (*i.e.,* client) services within a given artifact.

Figure 9 illustrates the process of binding generation. The *bindings* are used to break hard-coded dependencies in current artifact implementations and provide entry points in three different technologies.

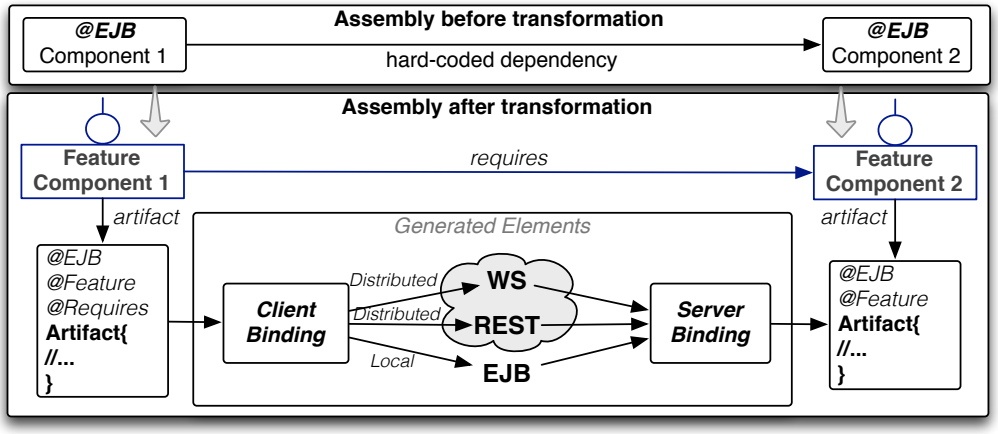

Figure 9: Binding generation.

As it can be seen in the figure, for every dependency found in the code, two types of bindings are generated, one at each side of the invocation. At the client side, the binding offers methods with the same signature as the ones found in the original server EJB (*e.g.,* Component 2) but internally, it redirects the invocation towards the server using the specific implementation technology. At the server side, the binding receives the invocations from the client side and transforms it to use the methods of the original artifact. Currently we support three types of bindings, `Web Service` [12, 15], `REST` [10] , and `EJB`.

1. *Web Services:* for the WS binding at the server side, we use the annotations provided by the JEE platform: `@WebService`, `@WebMethod`, and `@WebParam`. Inside the binding, code is generated to locate through JNDI the original EJB class in the artifact and invoke its methods. Every application server that complies the JEE specification is able to process such annotations and generate the `WSDL` [32] required for the remote invocation of the artifact functionalities via Web Services. At the client side the WSDL is used to generate the stub code. The client bindings are in charge of replacing the invocation to the original EJB into a distributed one that uses the classes in the stub code.

2. *REST:* for the REST binding we follow a similar approach than for Web Services using the annotations defined by the JEE specification (version 6 and above). The annotations indicate what methods are accessible and which `HTTP` request (*e.g.,* POST, GET, PUT, DELETE) is used for the invocation. Additionally, the `@Path` annotation indicates the relative URI where the resource can be located with regard to the application context. Finally, the annotations used at the level of methods `@Consumes` and `@Produces` define the `MIME` (Multipurpose Internet Mail Extensions) data types exchanged. In REST, data is typically exchanged using XML or JSON. We use XML to build data type contracts (XML Schemas) for complex types. Invocations in the REST bindings are performed using either POST or GET requests depending on the data types. POST requests are used for invocations where methods exchange either complex parameters or more than one parameter regardless of their type. A complex parameter can be a business object, typed collections (*e.g., generics*), or n-dimensional arrays. GET requests are used for invocations with one not-complex parameter (*i.e.,* primitive data types).

3. *EJB:* Finally, for EJB, we use the same strategy as for the previous cases. However, in this case the generated classes and interfaces have no extra annotations. All invocations are performed via

a JNDI lookup which takes place within the same application. This is the method that is currently used for interaction between EJBs in the framework. In this case, no XML transformations are required. However, only Java implementations can be used across all the invocations among the services. This integration mechanism becomes relevant when the annotated Java type depends on functionalities usually provided by the JEE container like security or transaction management. The EJB binding guarantees that such functionalities are preserved all along the invocation.

For the distributed bindings (*i.e.,* Web Service and REST), it is necessary to guarantee that information exchanged can be correctly represented in XML and sent via HTTP. All data types exchanged in distributed invocations are transformed into XML with the help of JAXB and XML Schemas as summarized in Table 2. A method in a client binding must transform Java data types to XML for the parameters to be sent in an invocation to the server, and it must recover a Java data type from the returned value received as XML from the server. Conversely, a method in a server binding must encode a return value in XML so that it can be sent to its client, and it must recover Java data types from XML data received as parameters.

	Client	**Server**
Parameters	Java to XML Data encoding to XML	XML to Java Data recovery from XML
Return	XML to Java Data recovery from XML	Java to XML Data encoding to XML

Table 2: XML data transformation.

In the cases where complex data types (*e.g.,* typed collections, n-dimensional arrays, and business objects) are used as parameters, it is mandatory to indicate the JAXB marshaller how such types are represented in the XML Schema. To guarantee the correct encoding of information, we generate *Bundle* classes that encapsulate complex data types and guarantee their correct representation in XML Schemas. In Web Services, a bundle is generated for each complex type found in the signature of methods and the WSDL takes care of parameter and return type definitions. In REST however, since the JEE REST implementation is based on HTTP, invocations are limited to one complex parameter (*e.g.,* HTTP body request). This means that every parameter to be sent in an invocation has to be part of a single HTTP body. In current `HBT` artifacts most interfaces have methods with multiple parameters. To overcome this limitation, we group all parameters in a bundle for each method. Like this, the bundle can be transformed to XML and sent in the body of the request. Bundles are used inside the generated bindings and guarantee that information exchanged between clients and servers can be manipulated at each side of the communication natively, and at the same time, it can be transformed and represented into XML to be sent through a communication channel in the case of distributed bindings without loosing its integrity.

The actual code generation is organized in three stages as illustrated in Figure 10. The first stage starts with a product configuration that is created by a developer using the enriched variability model (*model.xml*) obtained during the domain engineering phase. Using this configuration, a product model is created by selecting the associated components in the architecture model.

In the second stage, the product model is used as input for a *model to model* transformation towards the Java metamodel (*i.e.,* the Spoon metamodel). This transformation generates the bindings and bundles required. The bindings are interfaces with the annotations required depending on the selected technology (EJB, WS, and REST). Using Spoon the model is pretty printed (*i.e., model to text* transformation) to obtain the Java classes for the bindings that, at this point of the process are empty (*i.e.,* only method signatures are generated). For the REST and Web Service bindings, bundles are generated for complex

types as explained in Table 2. Finally for the Web Service case, it is necessary to generate the contract (*i.e.,* WSDL file) and the stub code that will be used by the clients to consume the services. To do this, we generate and execute the tasks WSGEN and WSIMPORT that generate both the contract, and the stub code from the contract that is used at the client side to consume the service.

Finally, in the third stage, bindings are completed by adding the method bodies. Afterwards, the process modifies the original client code, to replace the hard-coded dependencies with binding invocations. Like this, artifacts get separated from each other and only communicate through their bindings.

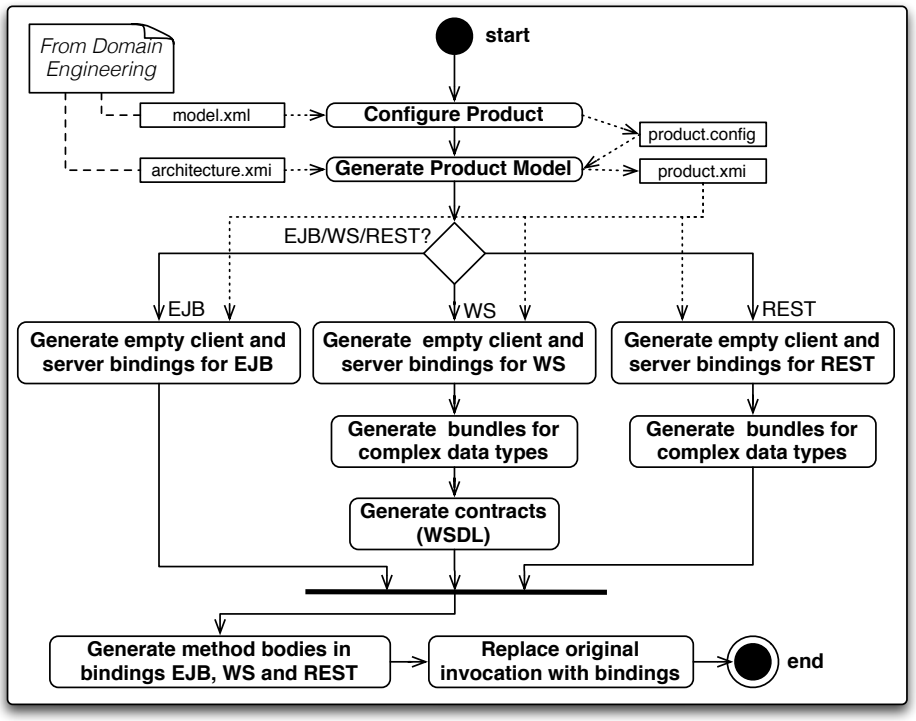

Figure 10: Code generation process.

4.2.3 Product Deployment

For the product to be completed, the application engineering provides a last task in charge of the product deployment. This phase combines the code in the original artifacts with the generated binding code to build the SPLIT version of each artifact. It has three inputs: (1) the generated code from bindings and bundles, (2) the configuration expressed in terms of selected features, and (3) the description of the artifacts in the framework in terms of what it contains (*e.g.,* configuration files, database initialization scripts, paths). The deployment merges the code in the artifacts with the bindings and bundles generated. Once the code has been merged, the deployment installs the new versions of the artifacts so that they can be referenced from the new product. Next, a deployment model is generated to describe the product in terms of the various artifacts it contains (*e.g.,* JAR files, XML files, properties, configuration scripts). Using as input this model and through a standard Maven[2] JEE archetype, a project (pom.xml) representing the final product is generated. This project includes as dependencies the SPL versions of the artifacts with the

[2]http://maven.apache.org

generated bindings and bundles. Using the deploy tasks included in the archetype, a self-contained Enterprise Archive (EAR) is generated including all the libraries required for its deployment and execution in an application server.

5 Validation

To validate our approach we have defined two different scenarios to measure and compare the derivation time using SPLIT, and the overhead in the actual execution due to the use of bindings for each artifact. In the first case, we list the available artifacts from the framework, annotate them, and compare the set up time using the `SPL` approach versus the manual process used in `HBT`. For the second scenario, we measure the overhead of the bindings. We define a product with five different components, and compare the execution time with and without the bindings in the three available technologies and in local and distributed environments.

All the tests have been performed in Windows 7 PCs with core i5 processors and 4 gigabytes of RAM. The deployment and execution have been tested in the JBoss[3] application server version 7.1. Additionally, we use Maven version 3.0 for project, dependencies, and source code management.

5.1 Derivation

For the derivation, we compare the time spent to set up the components manually and using SPLIT. Table 3 lists all the components available in the framework. The second column shows the number of services exposed by each component. The third column measures the average time spent by several different projects while manually doing the process. The fourth column shows the time spent using the `SPL` approach. Finally, the last column describes briefly the component functionalities.

Since SPLIT automates all the process of binding generation and deployment, our approach reduces the derivation and deployment time for all the artifacts where information of manual process is available (*i.e.,* for recent components time of manual deployment is not available yet). It is important to notice that the time presented on the SPLIT column is almost entirely dedicated to the manual processes of annotating the code and writing descriptors used by SPLIT to download the source code. This is only done once for any artifact when it is introduced in the `SPL`. Afterwards, the derivation time is reduced to an average of 1 minute per artifact, which represents an improvement in development time of more than 90% over the manual process.

5.2 Binding overhead

In the binding overhead test we measure the impact of adding local and distributed bindings in the size of each artifact, and the execution time of a product. For this test, we have used the product configuration presented in Figure 8. The product obtained of deriving this configuration corresponds to the assembly of components illustrated in Figure 11. The product includes the `MassiveLoader` component that loads data from a file to a data base using the `FileProcessing` component, and then it uses the service in the `OfficeNotificationAdapter` to send a notification with a configurable attachment via the services exposed by the `OfficeHelper` and `Notification` components.

Table 4 compares the average time spent when consuming the `MassiveLoader` service in 4 different scenarios: (1) using the original hard-coded EJB dependencies, (2) using EJB bindings, (3) using WS bindings, and (4) using REST bindings. For the latter two cases, we additionally measure the time when

[3] http://www.jboss.org

Component	Services	Manual (Hours)	SPLIT (Hours)	Description
Notifications	6	76,5	2	Provides the functionalities to send notifications with attachments via email
OfficeHelper	2	10	2	Enables to generate office documents using templates
FileGenerator	3	82	2	Provides mechanisms to generate plain files
FileProcessing	3	54	2	Provides mechanisms to load, validate and process plain files
Security	5	23	3	Provides user authentication and authorization functionalities.
AutomatedProcesses	4	30	3	Provides functionalities for scheduling and executing automated tasks
OfficeNotification	1	6	2	Combines the functionalities of the office helper and notification components
TaskManager	2	20	2	Provides functionalities for application tasks management.
MassiveLoader	1	6	2	Provides functionalities to load large amounts of information
Formulator	3	-	3	Allows one to build business process rules and formulae through a user friendly interface.
Accounting	1	-	2	Business component for accounting over specific periods

Table 3: Summary of available components and derivation time.

the components are deployed *locally* (*e.g.,* same application server) and *distributed* (*e.g.,* different servers on different machines).

The bindings size depends on the amount of methods in the original interface, and, for the distributed cases, it also depends on the amount of complex data types that need bundles. This can be noticed in the results of Table 4 that reflect a size increment of approximately 1000 *LoC* in the EJB case and more than 3000 *LoC* in the Web Services and REST cases. REST is bigger in the server side (*e.g.,* Notification and OfficeHelper) because the process generates bundle classes for almost every method in the original interface. Web Services is bigger in the client side (*e.g.,* OfficeNotificationAdapter), because it generates the stub code from the WSDL regardless of the clients whereas REST only generates bundles and methods that are required in the client. This can be noticed in the results for the components that consume other services (`MassiveLoader` and `OfficeNotificationAdapter`) whose size is bigger in the WebServices case. Despite the size increment, it is important to notice that the process of generating the extra code for the distributed cases is fully automated and does not require manual modifications of the original source code in the artifacts.

Regarding the execution time, it can be noticed that EJB binding have almost no impact on the execution time. In the local scenarios for Web Service and REST, the execution takes an extra 206 and 143 milliseconds on each case, which is due to the data type transformations required to and from XML. The slowest cases refer to distributed deployments, when time is slightly bigger ($\sim$100 milliseconds) than the original invocation in the local environment. This increase in time is an expected result and depends

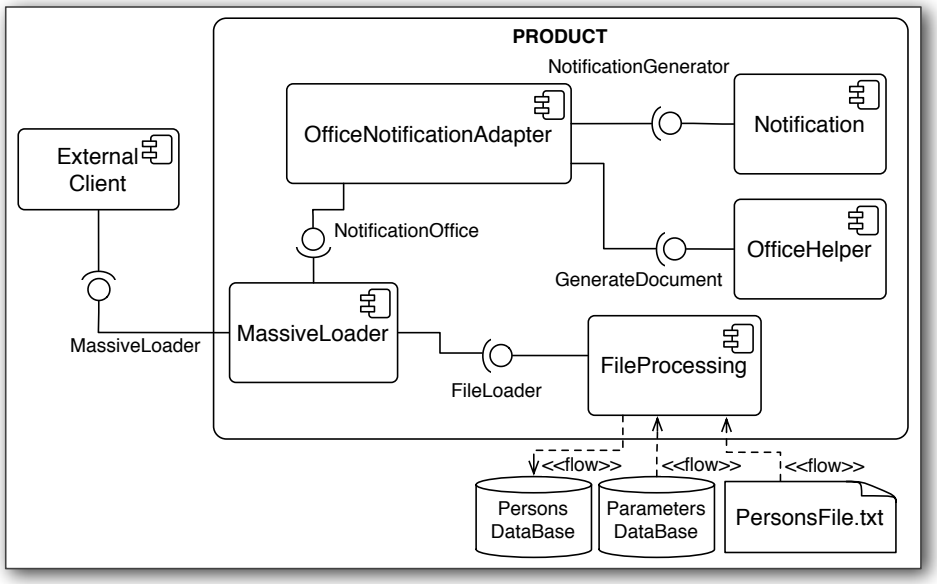

Figure 11: Product example.

Scenario	**Artifact**	**Size (LoC)**	**Time (seconds)**	
			Local	*Distributed*
No bindings	Notification	8630	0,372	-
	OfficeHelper	2188		
	OfficeNotificationAdapter	144		
	FileProcessing	13245		
	MassiveLoader	161		
EJB binding	Notification	9433	0,376	-
	OfficeHelper	2295		
	OfficeNotificationAdapter	258		
	FileProcessing	13743		
	MassiveLoader	275		
WS binding	Notification	11170	0,578	0,691
	OfficeHelper	2431		
	OfficeNotificationAdapter	2719		
	FileProcessing	14937		
	MassiveLoader	2507		
REST binding	Notification	15201	0,515	0,675
	OfficeHelper	2806		
	OfficeNotificationAdapter	413		
	FileProcessing	18784		
	MassiveLoader	340		

Table 4: Size and time binding overhead.

on network conditions. It is a small drawback in exchange for having modular SOA components capable of distributed communications.

With the evaluation we can observe important benefits from SPLIT in terms of derivation time and flexibility. Derivation times have been reduced in more than 90%, while execution time in products has no impact in EJB and small overheads in distributed bindings. This overheads however are largely compensated by the possibilities of having SOA communication across the assets of the product line.

6 Discussion

With SPLIT, we are able to transform currently developed software artifacts into SOA components, enabling the product line to assembly different products from a configuration expressed in terms of selected features from the variability model. SPLIT provides artifact management, automated deployment, and increases components evolution and reusability. Nevertheless, we have identified several challenges that are still open for further research. We summarize some aspects that we consider of highly importance for the next steps of the SPL adoption as follows:

- ***From Extractive to Reactive approach***
  Kreuger *et al.* [19] define three strategies for SPL adoption: *extractive*, *reactive*, and *proactive*. An extractive strategy allows the organization to start building SPL assets using existing software artifacts. In a reactive strategy components are built both from existing components and from scratch. In this case, the product base is built incrementally towards mass customization. Finally a proactive approach focuses on designing and implementing assets from scratch for the SPL architecture. SPLIT has been developed for an extractive adoption in order to work with existing software artifacts. A natural evolution for SPLIT consists in supporting a reactive strategy in which SPL assets are not only built from existing artifacts, but from scratch. As part of the future work for SPLIT, we intend to support *contract-first* components where the actual implementation does not exist and is developed only after the derivation process has taken place. In this way we expect to provide an incremental evolution of the actual asset base.

- ***Reusable artifacts vs Services***
  Despite the fact that building services through bindings around existing artifacts is an ideal strategy for product derivation, there are several artifacts that cannot be transformed. That is because their functionalities cannot be exposed through services. For instance, the framework provides an entity manager that creates all the code associated with a certain group of database entities. This is a practical functionality that is highly coupled with the business itself and which generates code that is only useful in the context of a particular business data model. Furthermore, a parameterization is needed in order to identify the set of business entities that will be processed by the artifact. However, this type of artifacts cannot be transformed into SOA assets. The only automation provided by SPLIT refers to the process of installation and deployment through Maven.

- ***Orchestration and Business Processes***
  The supported business processes are an important part of any application. An orchestration refers to the order of task executions and data flow between different actors in a business process. Some actors may correspond to services as the ones exposed by HBT assets. In our approach, small orchestrations exist between services, when *requires* relationships are defined between artifacts. With this information, it is possible to generate a generic process using the BPMN 2.0 specification [25] including invocations to any of the available assets selected in the configuration. However, for complex processes that involve not only services from available assets but also external services as

well as human interactions, there is not enough information to provide an automated development. For the time being, specific data about process properties, human tasks, execution flows, and data exchanged among others have to be added manually with tools like a BPMN modeler.

- ***Graphic User Interfaces for Components***
  An important part of any enterprise application is the graphic user interface (GUI). Every asset in the SPL may have several artifacts for this purpose. For the most part, these artifacts refer to dynamic HTML pages and the code that interacts with the business layer. However, these artifacts are hidden inside the `Element` in the metamodel, and are bounded to component's functionalities not exposed as services. For instance, when using the `FileGenerator` component, it includes a set of pages for parameterizing several configuration properties regarding the formatting for the files processed. These parts are not taken into account in the variability model, which means that, for the time being, developers cannot select and deselect them during the configuration and product derivation processes. The components are always deployed with all their inner GUI elements and no further configuration is possible beyond what has been implemented.

7 Related Work

In order to build software products, SPLIT automates the processes of reverse and application engineering processes. Several approaches in the literature face similar challenges for specific parts of each of these process. For this reason we classify related works in four different categories: (1) `SOA` and `SPL`, (2) reverse engineering, (3) product derivation, and (4) dynamic variability binding.

7.1 SOA and SPL

Even though `SPL` and `SOA` are independent approaches (one can build a product line without using services and vice versa), several works have tried to combine them. According to the SEI framework for product line engineering [24], when `SPL` and `SOA` are combined, services are typically considered as assets in the product line, and products are derived from the assembly of such services. Examples of such a combination can be found in the works of [33, 2, 14]. These approaches define assets as services and combine them to obtain software products which are the orchestration of the services selected for the features in the product configuration.

We follow a similar approach for variability and its realization through modular components. However, to enable this binding, we have to define a previous reverse engineering process where such assets are built from current EJB artifacts. Furthermore, in our case, existing artifact dependencies must be replaced with bindings that combined with the original code enable products to be assembled via services.

7.2 Reverse engineering

Reverse engineering has been explored in the context of `SPL` and service oriented applications. Acher *et al.* [1] present a reverse engineering process of the FraSCAti platform [31], a Service Component Architecture (SCA) implementation with reflective capabilities. Because of the modular design of SCA applications, one of their conclusions is the easiness to bind each feature with concrete components and then compose them together to build software products.

Unlike this approach, and since we aim at reuse of currently developed software artifacts, the reverse engineering in SPLIT does not start from actual SCA components that can be composed automatically. Instead, SPLIT input consist of Java classes and interfaces that get annotated by hand to slice them into

actual components in the architecture model. These models enable product derivation by establishing a link between features in the variability model and existing software artifacts.

7.3 Product derivation

Several approaches use features as the trigger of a product derivation and different strategies for the realization of such features. For instance Czarnecki and Antkiewicz [8] establish a mapping from feature models to application models. The idea is to allow the modeler to view directly the assets related to each feature and estimate the impact of selecting/deselecting a given feature. With a particular configuration, a template instance is obtained which represents the selection of the modeler. A template corresponds to design elements like UML diagrams.

Arboleda *et. al.* [4] propose a Software Product Line based on Models. Their approach uses variability and constraint models in combination with *Aspect Oriented Programming* (AOP) to derive products that integrate different concerns into a single product. All the operations to derive a product occur at design time (merging models and code generation).

We follow a similar approach by defining a variability model that is bound to elements defined in a loose coupled architecture to model components communicating through services. Unlike the first approach our transformation do not target UML diagrams but directly a Java metamodel to generate code. This is done because in our case, we do not need to generate the whole component implementation, but only the bindings that allow the artifact to communicate with others via contracts in different technologies. Regarding the second approach, we do not use aspects because current software artifacts are not cross cutting. Instead of weaving aspects with a core, we rather aim at having components that offer and consume services, so that they can be reused in different projects.

7.4 Dynamic variability binding

SOA and SPL have also been studied in the context of dynamic environments, and more specifically, in the context of dynamic software product lines (DSPL) [13], which are product lines where a product configuration can be changed at runtime, due to changes in the environment, and thus triggering an adaptation in the architecture or behavior of the product.

Baresi *et al.* [6] present an approach for modeling dynamic BPEL processes to reflect changes in variability at runtime. They focus on the possibilities of dynamically adapting a given product by modifying the process and the services associated.

Another way of achieving variability realization and eventually dynamic adaptations is through AOP. Apel *et al.* [3] present the aspectual feature modules. These combine feature oriented programing (FOP) and AOP to realize features. This approach uses classic feature modules for non cross-cutting concerns and aspects for special cases. Eventually these aspects could be weaved at runtime (dynamic AOP), making configurations possible at runtime as well. Dinkelaker *et al.* [9] propose a dynamic software product line using aspect models at runtime. They use aspect models to define features and feature constraints. Their approach mixes SPL principles of product derivation with the notion of dynamic variability. They distinguish static variability from dynamic variability, and for the latter one, they use dynamic AOP for the implementation. Their approach links what they call *dynamic features*, representing late variation points in an SPL, to dynamic aspects.

These approaches are well suited for dynamic environments where information only available at runtime can affect the architecture or behavior of the application. SPLIT target enterprise applications that do not require dynamic adaptations. For this reason, we focus on reusing existing software artifacts by means of an automated transformation towards components for a service oriented architecture. Consequently, unlike dynamic approaches, in SPLIT the product configuration is performed only once at the

beginning of the product derivation and stays unmodified during the deployment and execution of the product.

8 Conclusions

In this paper we have presented SPLIT, our automated tool for SPL adoption through SOA. We have identified three challenges regarding: (1) an architecture for service composition and reuse, (2) variability binding, and (3) automated product derivation. For the first challenge, we have created a reverse engineering process that generates SPL core assets using an architecture metamodel that represents assets as components that expose and consume services. For the second challenge, we have defined an enrichment process that automatically updates information of current implementations in the variability model, enabling developers to trigger the derivation of products by selecting features. Finally, for the third challenge, we have implemented a product derivation process that starts with core assets and generates the bindings to assembly a product for a given configuration. Furthermore, we have also defined an automated deployment process aware of the target platform that combines the code in the artifacts with the generated binding code, to obtain a complete product ready to be deployed in an application server.

To validate the approach we have implemented SPLIT using frameworks like Spoon, EMF, Java, JEE and JAXB. SPLIT supports communication between the generated services in Web Services, REST, and EJB. For the evaluation, we have used SPLIT to build products using real JEE artifacts. The results have shown improvements of more than 90% in derivation times, and more flexibility thanks to the different types of bindings supported. The evaluation has also shown an expected overhead in size (LoC) and execution time for the configurations, specially in distributed deployments where network latency affects performance. However, this minor drawback is largely compensated by the benefits in flexibility achieved through SOA bindings in local and distributed scenarios.

For future work, we intend to move towards a more reactive strategy for SPL adoption using the service-based architecture as a starting point for the development of new components from scratch, to provide an incremental evolution of the SPLIT asset base. We plan to include new assets developed not only by HBT but also by third parties. We also plan to improve the variability model, specially with regard to business constraints, so that new products target current market needs. Finally, we want to explore alternatives like multi-staged product configuration to include user interfaces and other technology-dependent decisions in the configuration and derivation processes.

9 Acknowledgments

This research is financed by the Colombian government through the *Francisco José de Caldas* fund of the department for science, technology, and innovation *Colciencias*. We would like to thank Santiago Gil and Catalina Acero from Heinsohn Business Technology for their feedback and help on the preparation of the final version of this article.

References

[1] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Reverse engineering architectural feature models. In *Proc. of the 5th European Conference on Software Architecture (ECSA'11), Essen, Germany, LNCS*, volume 6903, pages 220–235. Springer-Verlag, September 2011.

[2] S. Apel, C. Kaestner, and C. Lengauer. Research challenges in the tension between features and services. In *Proc. of the 2nd International Workshop on Systems Development in SOA Environments (SDSOA'08), Leipzig, Germany*, pages 53–58. ACM, May 2008.

[3] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[4] H. Arboleda, A. Romero, R. Casallas, and J.-C. Royer. Product derivation in a model-driven software product line using decision models. In *Proc. of Memorias de la XII Conferencia Iberoamericana de Software Engineering (CIbSE'09), Medellín, Colombia*, pages 59–72, April 2009.

[5] G. Barber. What is SCA?, August 2007. http://www.osoa.org/.

[6] L. Baresi, S. Guinea, and L. Pasquale. Service-oriented dynamic software product lines. *IEEE Computer*, 45(10):42–48, October 2012.

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

[8] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia, LNCS*, volume 3676, pages 422–437. Springer-Verlag, September-October 2005.

[9] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A Dynamic Software Product-Line Approach using Aspect Models at Runtime. In *Proc. of the 5th Domain-Specific Aspect Languages Workshop (DSAL'10), Rennes and Saint-Malo, France*, March 2010.

[10] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[11] R. Flores, C. Krueger, and P. Clements. Mega-scale product line engineering at General Motors. In *Proc. of the 16th International Software Product Line Conference (SPLC'12), Salvador, Brazil*, volume 1, pages 259–268. ACM, September 2012.

[12] D. Gisolfi. Web Services architect: Part 1. an introduction to dynamic e-business, April 2001.

[13] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[14] A. Helferich, G. Herzwurm, S. Jesse, and M. Mikusz. Software Product Lines, Service-Oriented Architecture and Frameworks: Worlds Apart or Ideal Partners? In *Revised Selected Papers from the 2nd International Conference on Trends in Enterprise Application Architecture (TEAA'07), Berlin, Germany, LNCS*, volume 4473, pages 187–201. Springer-Verlag, November-December 2007.

[15] N. Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[17] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *Proc. of the 31st International Conference on Software Engineering (ICSE'09), Vancouver, British Columbia, Canada*, pages 611–614. IEEE, May 2009.

[18] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[19] C. W. Krueger. Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering (PFE'01), LNCS, Bilbao, Spain*, volume 2290, pages 282–293. Springer-Verlag, October 2002.

[20] C. W. Krueger, D. Churchett, and R. Buhrdorf. Homeaway's transition to software product line practice: Engineering and business results in 60 days. In *Proc. of the 12th International Software Product Line Conference (SPLC'08), Limerick, Ireland*, pages 297–306. IEEE, September 2008.

[21] R. Krut and S. Cohen. Service-oriented architectures and software product lines - putting both together. In *Proc. of the 12th International Software Product Line Conference (SPLC'08), Limerick, Ireland*, page 383. IEEE, September 2008.

[22] B. Mohabbati, M. Asadi, D. Gašević, M. Hatala, and H. A. Müller. Combining Service-Orientation and Software Product Line Engineering: A Systematic Mapping Study. *Information and Software Technology*, 55(11):1845–1859, November 2013.

[23] E. Murugesupillai, B. Mohabbati, and D. Gašević. A preliminary mapping study of approaches bridging software product lines and service-oriented architectures. In *Proc. of the 15th International Software Product Line Conference (SPLC'11), Munich, Germany*, volume 2, pages 11:1–11:8, August 2011.

[24] L. M. Northrop, P. C. Clements, F. Bachmann, J. Bergey, G. Chastek, S. Cohen, P. Donohoe, L. Jones, R. Krut, R. Little, J. McGregor, and L. O'Brien. SEI A Framework for Software Product Line Practice, Version 5. http://www.sei.cmu.edu/productlines/framework.html, 2013.

[25] O. M. G. (OMG). Business Process Model and Notation (BPMN) Version 2.0. Technical report, Object Management Group (OMG), January 2011.

[26] C. Parra, X. Blanc, A. Cleve, and L. Duchien. Unifying design and runtime software adaptation using aspect models. *Science of Computer Programming*, 76(12):1247–1260, 2011. Special Issue on Software Evolution, Adaptability and Variability.

[27] C. Parra, L. Giral, A. Infante, and C. Cortés. Extractive SPL adoption using multi-level variability modeling. In *Proc. of the 16th International Software Product Line Conference (SPLC'12), Salvador, Brazil*, volume 2, pages 99–106. ACM, September 2012.

[28] C. Parra, D. Joya, L. Giral, and A. Infante. An SOA approach for automating software product line adoption. In *Proc. of the 29th Symposium on Applied Computing (SAC'14), Gjeoungju, South Korea,*. ACM, March 2014.

[29] R. Pawlak, C. Noguera, and N. Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA, May 2006.

[30] M. Schulze, J. Weiland, and D. Beuche. Automotive model-driven development and the challenge of variability. In *Proc. of the 16th International Software Product Line Conference (SPLC'12), Salvador, Brazil*, volume 1, pages 207–214. ACM, September 2012.

[31] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Proc. of the 6th International Conference on Service Computing (SCC'09), Bangalore, India*, pages 268–275. IEEE, September 2009.

[32] W3C. Web Services Description Language (WSDL) 1.1, 2001 March. `http://www.w3.org/TR/wsdl`.

[33] C. Wienands. Studying the common problems with service-oriented architecture and software product lines. In *Service Oriented Architecture (SOA) & Web Services Conference, Atlanta, Georgia, USA*, October 2006.

Author Biography

Carlos Parra is a researcher at Colciencias and Heinsohn Business Technology in Colombia. He received his B.S. in Computer Science from Francisco José de Caldas University at Bogotá in 2004, M.S. in Computer Science from the University of Los Andes at Bogotá in 2007, and Ph.D. in Computer Science from the University of Lille 1 and INRIA at Lille France in 2011. His research interests include Software Product Lines, Model-driven Engineering, Generative Programming, and Service Oriented Architectures.

Diego Joya is a software architect at Heinsohn Business Technology in Colombia. He received a B.S in Computer Science from the Central University at Bogotá in 2008, and a specialist degree in Software Development from the University of Los Andes at Bogotá in 2011. His interests include architectural styles and patterns for enterprise applications and software product lines.