An SOA approach for Automating Software Product Line Adoption

Carlos Parra, Diego Joya, Leonardo Giral, Alvaro Infante
Heinsohn Business Technology
Cra 13 No 82-77
Bogotá, Colombia
{cparra, djoya, ogiral, ainfante}@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 a set of core-assets and a derivation strategy based on such assets. Adopting such approaches involves important challenges with regard to existing software artifacts that must be transformed in order to respect an architecture that focus on modularity and reuse. This paper presents an industrial experience of such transformation. 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 the reverse engineering and derivation processes using real software JEE artifacts from a component framework of reusable functionalities in several different enterprise applications. The results show important benefits in terms of the development time and flexibility.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Experimentation

Keywords

Service Oriented Architecture, Software Product Lines, Model-driven Engineering, Reverse Engineering

1. INTRODUCTION

In the 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 the 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 services offered and required. 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 obtain different results and satisfy customers needs.

On the other hand, Software Product Lines (SPL) aim at changing the focus from building individual software applications to building families of products by exploiting commonalities and variabilities across a set of software applications [5]. This results in an increased reuse and reduced time to market as shown in several experiences (*e.g.* [16, 9, 22]). An SPL is particularly useful in environments where multiple software products share common parts and can be partially derived from them.

While SPLs aim at identifying variabilities and commonalities in a product family, SOA aims at modularization and development of independent services that combined form bigger software applications. As a result, SPL and SOA can be complementary approaches. Several experiences have already explored the possibilities of combining such approaches in the software development process [19, 18, 17].

However, for organizations with large and stable software systems, adopting these paradigms is not a straightforward process. One of the biggest challenges to face is the transformation and reuse of current software artifacts. In order to minimize the impact of such adoption, organizations typically choose an *extractive approach* [15] in which new assets and services are built from current software artifacts.

In this paper we present an approach to SPL adoption by means of an SOA modular architecture that bridges features in the product line with existing artifacts in a framework of reusable functionalities in several different enterprise applications. This is achieved through the definition of components that communicate with each other through services. We define two processes: (1) a domain engineering where SPL assets are built out of existing artifacts, and (2) an application engineering process where products are built from such assets. The domain engineering process consists of a

SAC'14 March 24-28, 2014, Gyeongju, Korea.

http://dx.doi.org/10.1145/2554850.2554987

reverse engineering phase that uses annotations to analyze the software artifacts, and generates: (1) the implementation variability model and (2) the architecture model. For the application engineering process, we present a fully automated derivation of products that uses as input a configuration of features, and assemblies an architecture model. From this model, the process generates the missing *wrapper* code and contracts required for the correct deployment and communication of current artifacts as modular SOA-capable components.

We evaluate our approach using several mature Java Enterprise Edition (JEE) artifacts developed for different sectors like 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 show that building SOA components around existing artifacts eases the product derivation by establishing bindings between features in the variability model and current software artifacts in the implementation level.

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 in detail. Section 4 presents the results of our experimentation using several JEE artifacts. In Section 5 we discuss the limitations of our approach. Finally Section 6 presents the related work and Section 7 concludes the paper and points out several paths for future work.

2. MOTIVATION

For organizations, it is important to reinforce the idea of software reuse across different software projects. In our case, we work with 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 more than 20 artifacts of around 2000-15000 lines of code, and keeps on growing as new common functionalities are identified and developed in different software projects. Existing functionalities 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.

The framework offers an ideal starting point for the development of core assets for the SPL. According to Clements and Northrop [5], there are two ways to build such assets: from scratch, or from existing software artifacts. In the first case, every asset is built from scratch to fulfill the requirements of the SPL architecture. In the second case, current artifacts are modified to be in accordance with the SPL architecture. Since we start from the artifacts in the framework, we follow the second approach. However, in order to transform such artifacts into core assets, we identify two main challenges to face:

1. ***Generic architecture with clear interfaces***

   Currently all the artifacts are developed under the 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 cannot be distributed. There are hard-coded dependencies that are resolved by the implementation platform.

   The first challenge refers to the transformation of every artifact into an SOA component with clear interfaces to describe the services it provides and requires, and thus removing hard-coded dependencies. To do this, it is necessary to define an architecture based on the assembly of loose coupled components.

2. ***Variability driven derivation***

   Additionally, the architecture must enable one to represent products as a given selection of features. Like this, products can be expressed in terms of features, and derived through the assembly of the associated modular components. The second challenge refers to the definition of a variability model and the implementation of a derivation process to build actual software products out of SOA components.

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

3. THE SPL ADOPTION STRATEGY

For the SPL adoption we follow the processes defined in [5]: domain engineering, and application engineering. We have defined a core-asset building process for the domain engineering, and a product derivation process for the application engineering. Figure 1 illustrates these processes.

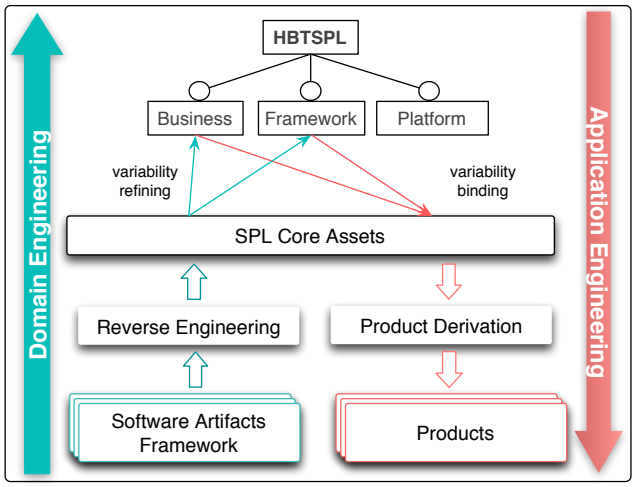

Figure 1: The SPL processes.

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. It is important to notice that the level of abstraction changes for each process. In the figure, as the arrow goes up, the current artifacts are used to build high-level models to represent them. Conversely, as the arrow goes down, the models are

used to generate the glue code that enables one to assembly multiple artifacts and produce individual software products. Since the adoption strategy is extractive, we start from the bottom left of the figure with current software artifacts in the framework, and go towards high-level core assets represented as models. After the core assets have been built, the derivation process takes place. The joining point in the middle of both processes is the variability model. This model gathers the multiple levels of variability that are inherent to the applications. We have manually identified three variability levels: (1) the business level that models business requirements of the product family, (2) the functional level which models different functionalities in the framework that support the business requirements, and (3) the platform level that represents the constraints of the specific execution platform in terms of software and infrastructure (*e.g.* databases, application servers, APIs, etc). This model is used to configure products and trigger the application engineering process. As follows, we describe the domain and application engineering processes.

3.1 Domain Engineering

The domain engineering phase creates high-level core assets using as input the source of the current implementations. Two phases are implemented to achieve this: a reverse engineering phase that creates the core-assets, and a variability refining phase that obtains an enriched variability model.

3.1.1 Reverse Engineering

The main goal of the reverse engineering process is to build core assets for the product line in an automated way, using as input the current implementations of the software artifacts.

An asset bridges a feature in the variability model and one or more artifacts in the implementation. As stated by Czarnecky and Antkiewicz [6], features in a feature model are merely symbols. Mapping features to other models such as behavioral or structural models is what gives them semantics. To reduce the impact on the current implementations, this process does not modify current artifacts. Instead, it generates *wrappers* that encapsulate and expose artifact functionalities through SOA components. Currently all the artifacts are implemented in the Java language and are executed in the JEE platform. For this reason, and to avoid modifications of the source code, we use Java annotations. An annotation enables developers to add meta-data about the programs it annotates without interfering in their structure or behavior. Through annotations, we indicate manually which parts of the original artifacts can be transformed into services to be used externally (*e.g.* by other modules in the same deployment unit or by remote applications in a distributed environment).

Unlike current artifact implementations, services are independent from each other, which means the dependencies that exist either directly coded in the implementation or built by the specific container have to be removed. To do this, we generate contracts. A contract specifies the communication of a given component in terms of services offered and required. To build such contracts in an automated way, we define two annotations `@Feature` and `@Requires`. The `@Feature` annotation is used to annotate Java interfaces and its inner methods. It indicates that the functionalities represented by the methods in such interface are considered as a feature in the variability model and must be exposed as a service for other artifacts or clients to consume. The `@Requires` annotation on the other hand, is used on Java methods inside implementing classes and indicate that within the body of such methods there are invocations to external services associated with a different feature.

The `@Feature` annotation is used on java interfaces and has two attributes: `featureName` to indicate the feature in the feature model associated with the code annotated, and an `implementation` to indicate the technology used to expose its functionalities. The `@Requires` annotation is used in java methods and has three attributes, `requiredFeatureName` to indicate the feature it requires, `providedFeatureName` to indicate the feature it provides (*i.e.* the component where the method belongs), and eventually an interface name for the cases when several interfaces are associated with the same feature.

The first part of the domain engineering consist in manually annotating the artifacts in the framework. Once the artifacts have been annotated, the reverse engineering process takes place. The process looks for the interfaces and methods associated with features. Next, through a series of model-based transformations [14], the process creates a model representation of the artifacts as SOA components, where functionalities provided and required are described through clear interfaces (*e.g.* contracts). To model this architecture, we use the metamodel depicted in Figure 2.

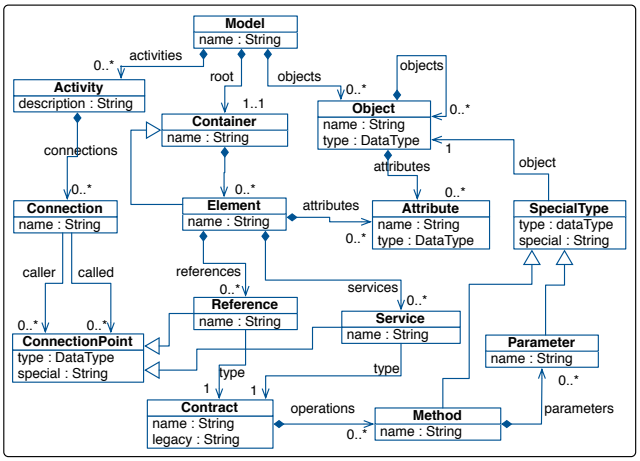

Figure 2: Generic architecture metamodel

The metamodel defines a generic service based structure similar to the Service-Component Architecture (SCA) [3]. In the metamodel, a single unit of composition is called an `Element`, which has a set of `Attributes`. An element might contain other elements in which case, it is defined as a `Container`. The `Element` can offer a set of `Services` and require a set of `References`. Both `Services` and `References` are connection points between elements. A set of connection points forms an `Activity`. A connection point also defines a `Contract`, which represents the interface, provided or required, depending on the type of connection point. A contract consists of a set of `Methods` that may have a set of `Parameters` and a return type (`SpecialType`).

The architecture metamodel represents a set of rules that

any new asset must respect in order to be considered a core asset of the SPL. These rules refer mainly to component interfaces, and do not affect the way each component is implemented inside (*e.g.* black-box), including its transaction management or database model. The advantages of having this architecture metamodel are twofold: (1) it helps us to define a common structure to represent existing artifacts as services, 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 is generic enough so that, it can model diverse artifacts, but at the same time, restrictive enough so that we can build the derivation mechanisms to assembly any asset.

For the code analysis and model transformations we use Spoon [21] and EMF[1]. Spoon is a tool that analyzes 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. We have used Spoon to capture the elements in the artifact annotated with the `@Feature` and `@Requires` annotations, as well as EMF-based java transformations to build a model that conforms to the architecture metamodel. Every artifact annotated and analyzed in this process gets represented by an `Element` that exposes its functionalities through contracts. Such contracts define the services the component offers and requires to automate the assembly with other services or external clients.

3.1.2 Variability Refining

In addition to the generation of a generic architecture model, the reverse engineering process also refines the variability model with accurate information about the current implementations. Three different levels have been identified manually in the variability model: Business, Functional, and Platform. With the refining process, a fourth *Implementation* level is added to the aggregated model. This level represents the actual implementations for every component. Figure 3 depicts the input and output of the variability refining phase.

The input of the process is the annotated interface. In this case, we present the interface of a component that offers services for sending notifications via different technologies. In the interface (lines 1, 4, 7 and 10), the `@Feature` annotation is used to mark both the interface and its methods. With this information, the processors generate a subtree with the following structure: each featureName is transformed into a Feature and has as many child features as interfaces annotated with `@Feature` exist with the same featureName. Below each child feature, there are new mandatory features for all the methods exposed by the interface, so that, from the tree, developers can have an idea of the functionalities that will be available once the product is deployed. Each implementation feature is required by its corresponding feature in the business or functional sub-trees. This constraint guarantees that for each functional or business feature selected, there will always be an implementation feature that is bounded to an element 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

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

```
1 @Feature(featureName = "Notification",
      implementation = "WS")
2 public interface Notification {
3
4   @Feature(featureName = "Notification",
        implementation = "WS")
5   public void notifyByEmail(String address);
6
7   @Feature(featureName = "Notification",
        implementation = "WS")
8   public void notifyBySMS(int number);
9
10  @Feature(featureName = "Notification",
        implementation = "WS")
11  public void notifyByDataBase(String id_Client);
12 }
```

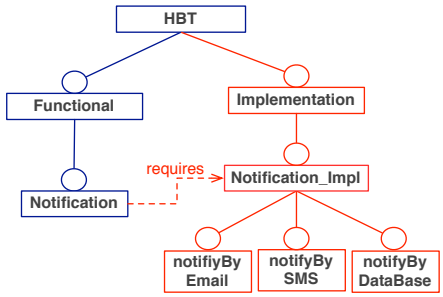

Figure 3: Feature annotation process.

between the features in the variability model. If such constraints do not exist, they are added automatically.

3.2 Application Engineering

The application engineering process builds a single product using a configuration expressed in terms of selected features from the implementation subtree previously described. Three phases have been defined in the application engineering process: (1) product configuration and composition, (2) server and client wrapper generation, and (3) product deployment.

3.2.1 Product configuration and composition

The product configuration refers to the selection of a given set of features to derive a single product. To configure a product we use the enriched feature model obtained after the refinement process. To represent the feature model and configure a product, we use FeatureIDE [13]. FeatureIDE is a set of tools for variability modeling that enables the creation and edition of feature diagrams. Furthermore, FeatureIDE provides a configuration tool to create and validate configurations with regard to the constraints defined in the model. Using FeatureIDE a developer can create product configurations and validate if such configurations respect the constraints expressed in the variability model. Next, to derivate a product, the selected features are used to find their corresponding implementation feature. Each implementation feature is named after an `Element` in the architecture model. A product becomes an assembly of the elements associated with every feature in the configuration. Every element defines a set of services and references that it provides and/or requires. At the end of the composition, there is a model with an assembly of the different elements associated with each implementation feature selected.

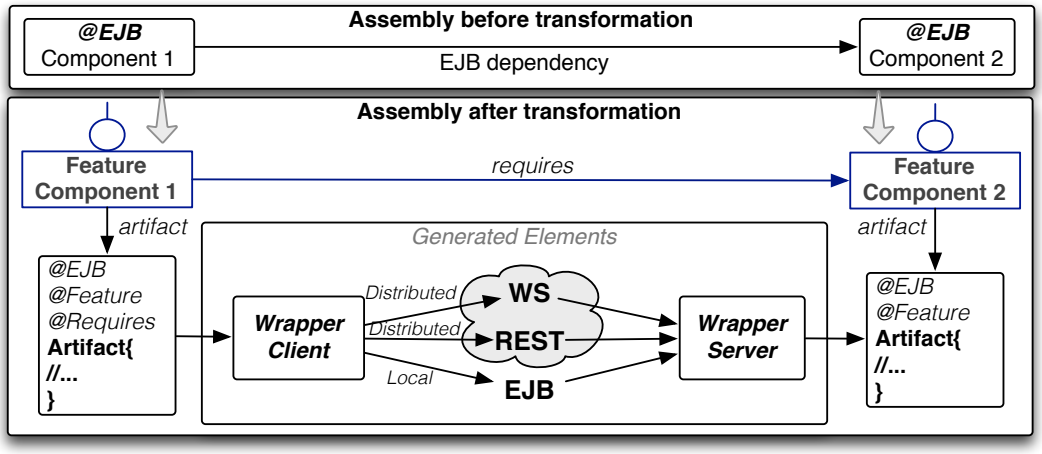

Figure 4: Wrapper generation

3.2.2 *Server and client wrapper generation*

A wrapper represents an interface to provide (*i.e.* server) or consume (*i.e.* client) functionalities within a given artifact. It works as an interface to the original artifacts in the framework, avoiding the modification on their implementation. Currently we support three types of wrappers, `Web Service` [10, 12], `REST` [8], and local `EJB`. The *wrappers* are generated from the contracts inside every service and reference in the architecture model. Figure 4 illustrates the process of wrapper generation.

As it can be seen, there are two types of wrappers generated for each side of the invocation. In the server side, the wrapper exposes the same functionalities as the original artifact, and adds the required technology-specific details so that invocations can be made via Web Services, REST, or EJB. Internally, the wrapper implementation invokes the methods of the original artifact through a java naming and directory interface (`JNDI`) lookup. In the client side, the wrapper replaces the direct mapping towards the original EJB with an invocation to the server wrapper. The client wrapper acts as a proxy, by exposing the same functionalities in the original artifact, but performing the invocation according to the chosen technology. In any case, a wrapper is a combination of an interface and an implementing class. As follows we describe each of the supported wrapper technologies.

1. *Web Services:* For Web Service wrappers, we use the annotations provided by the JEE platform. In this case, both the interface and the methods are annotated with the annotations `@WebService`, `@WebMethod`, and `@WebParam`. The implementation of this interface is a class that locates through JNDI the original EJB 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` [23] 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 wrappers in web services are in charge of transforming the original invocation into a new one that uses the classes in the stub code. At both ends of the communication, code is generated to transform business objects into the WSDL compliant counterparts (XML Schema types) and vice versa, to avoid incompatible types being exchanged.

2. *REST:* For REST wrappers, we follow a similar approach than for Web Services. However, in this case we use the annotations defined for the JEE (version 6 and above). The annotations indicate which methods are accessible and which `http` method (*i.e.* POST, PUT, GET, DELETE) is used for the exchange of information. Additionally, the `@Path` annotation indicates the relative URI where the resource can be located with regard to the application context. This annotation also allows one to add arguments to the URI. Finally, the annotations used at the level of methods `@Consumes` and `@Produces` define the `MIME` (Multipurpose Internet Mail Extensions) parameters and return type. In REST, data is typically exchanged using XML or JSON, however we only use XML responses so that we can build data type contracts (XML Schemas) for business types used as parameters or return types. At the client side, the wrappers transform the original java invocations into REST invocations via POST and GET methods. Similar to the Web Services case, for the data types, we generate the code to transform business objects into JAXB counterparts and vice versa, based on the schema defined by the server side. This guarantees that the XML data files exchanged between components can be correctly marshalled and interpreted at both ends of the communication.

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 components.

3.2.3 Product deployment

The last part of the derivation process consists in deploying the product. No matter the configuration selected, the product is represented by an aggregation of the components implementing the features selected. The product deployment creates a Maven[2] project (*.pom). It has two inputs: (1) the configuration expressed in terms of selected features, and (2) the description of the artifacts in the framework. This description has been implemented through a set of configuration files that include the main pieces of information that characterize each artifact (*e.g.* configuration files, database initialization scripts, paths) required for the derivation process. The product deployment performs two main tasks: (1) it customizes the artifacts according to the platform variability, and it generates a maven project that includes as dependencies the components associated with the features in the configuration.

The Maven project includes the list of customized files and scripts for each artifact selected, as well as the components and their wrappers whose combination forms the final product.

4. VALIDATION

To validate our approach we have defined two different scenarios to compare the derivation time, and the overhead due to the use of wrappers for each artifact. In the first case, we take several artifacts from the framework, annotate them, and compare the set up time using the `SPL` approach versus the original process. For the second scenario, we measure the overhead of the wrappers. In this case we use a process that consumes three different services from the framework, and compare the execution time with and without the wrappers generated.

All the tests have been performed in a Windows 7 PC with a core i5 processor and 8 gigabytes of RAM. The deployment and execution have been tested in the JBoss application server version 6.1[3] . Additionally, we use Maven version 3.0 for project, dependencies, and source code management.

4.1 Derivation

For the derivation, we compare the time spent to set up various components with and without the `SPL` automated process. We have chosen 5 different components in the framework as follows: (1) Notifications, (2) OfficeHelper, (3) FileGenerator, (4) FileProcessor, and (5) Security. Table 1 summarizes the results of setting up and deploying each component. The left column measures the average time spent by several different projects while manually doing the process. The right column shows the time spent using the `SPL` approach.

Since we have automated all the process of wrapper generation and deployment, our approach reduces the derivation and deployment time for all the artifacts in the framework. Furthermore the time presented on the SPL derivation column is almost entirely dedicated to the manual process of annotating the code, which is only done once for any artifact when it is included as part of the assets of the `SPL`. Otherwise the derivation time is reduced to an average of 1 minute per artifact.

[2]http://maven.apache.org
[3]http://www.jboss.org

Component	**Manual Process** (Hours)	**SPL Derivation** (Hours)
Notifications	76,5	2
OfficeHelper	10	2
FileGenerator	82	2
FileProcessor	54	2
Security	22,8	3

Table 1: Derivation times before and after the SPL.

4.2 Wrapper overhead

In the wrapper overhead test, we measure the impact of adding local and distributed wrappers in the size of each artifact in lines of code (LoC), and the execution time of a small process that involves the generated SOA components. For this test, we have defined a scenario when a component `OfficeNotificationAdapter` consumes the services in the `OfficeHelper` component to prepare a parameterized document, and then uses the `Notification` component to build a notification including the document and send it via email to a group of users.

Table 2 compares the average time to execute 100 times the same task in 4 different scenarios: (1) using the original hard-coded EJB dependency, (2) using an EJB wrapper, (3) using a local WS wrapper, and (4) using a REST wrapper. For the latter two cases, we additionally measure the time when the components are deployed in the same application server (*local*) and in different application servers located in different cities (*distributed*).

Scenario	**Size** (LoC)	**Time elapsed** (average per seconds)
No wrappers	11699	0,342
EJB wrappers	12698	0,372
WS wrappers	20685	
- Local		0,78
- Distributed		1,00
REST wrappers	17114	
- Local		0,75
- Distributed		0,94

Table 2: Size and time wrapper overhead.

The wrappers size depends on the original interfaces and methods, and for the distributed cases, it also depends on the amount of special business data types that need to be transformed into simpler types to guarantee the correct XML marshalling and unmarshalling process. This can be noticed in the results of Table 2 that show an increase in the combined components of approximately 1000 *LoC* in the EJB case and more than 6000 *LoC* in the Web Services and REST cases. However, 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 to the original source in the artifacts.

Regarding the execution time, it can be noticed that EJB wrappers add a minimal overhead of around 30 milliseconds. However, when it comes to Web Service and REST wrappers, the execution time is almost twice as the original invocation in the local environment and almost three times in the fully distributed one. This increase in time is an expected result when separating invocations that were orig-

inally developed in a local environment. It is a drawback in exchange for having modular components, and distributed SOA communication across the assets in the product line.

5. DISCUSSION

With our approach, we are able to transform current artifacts of the framework into SOA components. This makes possible to assembly different products. However, there are still challenges open for further research. We summarize two aspects that we consider of highly importance for the next steps of our `SPL` as follows:

- ***Reusable artifacts vs Services***
  Despite the fact that building services through wrappers around existing artifacts is an ideal strategy for the variability binding and assembly, there are several artifacts that cannot be transformed into services. 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 useful only 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. For the time being, for this type of artifacts, our approach cannot transform them into components, the only automation we provide refers to the process of installation and deployment through Maven.

- ***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 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.

6. RELATED WORK

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 [20], 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 [24], [2], [11]. We follow a similar approach for variability and its realization through modular components. However, to enable this binding, we first define an SOA architecture model from current EJB artifacts, and break hard-coded dependencies through wrappers that combined with the original code enable products to be assembled via services.

SOA and SPL have also been studied in the context of dynamic environments. For instance Baresi *et al.* [4] 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. Unlike the authors, we currently do not work on dynamic adaptations, and concentrate on the reuse of software artifacts as SOA components and the derivation process that assemblies such components.

Regarding product derivation, Dhungana *et al.* [7] present an approach to deal with large real-world systems by organizing product lines as a set of interrelated model fragments defining the variability of particular parts of the system. They focus on the evolution of product lines and the benefits of using loosely coupled model fragments to deal with variability changes over time. We follow a similar approach by defining a variability model bound to elements defined in a loose coupled architecture to model components communicating through services.

There have also been reverse engineering experiences for `SPL` in the context of service oriented applications. For instance, Acher *et al.* [1] presents a reverse engineering process of the FraSCAti platform, 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. In our case, we do not start from actual components, instead, we aim at building them through the generation of wrappers on top of current software artifacts.

7. CONCLUSIONS

In this paper we have presented our approach for `SPL` based on SOA services. We have identified two challenges related to generic service architecture and variability binding. For the first challenge, we have created a reverse engineering process that generates `SPL` core assets through a model where artifacts are represented as components that expose and consume services. This model reinforces the idea of components and contracts with clear interfaces that define services provided and required. To face the second challenge, we have implemented a refining process that adds an implementation subtree to the variability model. This model triggers a derivation process that generates the glue code to assemby multiple components communicating with each otherin local and distributed environments through services.

To validate the approach we have implemented the reverse engineering and derivation processes. We create model representations of artifacts as core assets with Spoon and EMF model transformations. The product derivation process uses these models to derive a functional product for a given product configuration. The communication between components is implemented via Web Services, REST, or EJB. The results have shown important benefits in the derivation time, and improved flexibility by allowing the distribution of the services in the product with a small overhead in the execution time.

For future work, we intend to move towards a more proactive adoption approach using the service-based architecture

as a starting point for the development of new components from scratch. We also plan to improve the variability model, specially with regard to business constraints, to define products that match the current needs of the market. Finally, we want to explore alternatives like multi-staged product configuration to include the GUI, and other technology-dependent decisions in the configuration and derivation processes.

8. 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*.

9. REFERENCES

[1] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Reverse engineering architectural feature models. In *Proceedings of the 5th European conference on Software architecture*, ECSA'11, pages 220–235, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] S. Apel, C. Kaestner, and C. Lengauer. Research challenges in the tension between features and services. In *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 53–58, New York, NY, USA, 2008. ACM.

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

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

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

[6] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE*, pages 422–437, 2005.

[7] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122, July 2010.

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

[9] R. Flores, C. Krueger, and P. Clements. Mega-scale product line engineering at general motors. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 259–268, New York, NY, USA, 2012. ACM.

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

[11] A. Helferich, G. Herzwurm, S. Jesse, and M. Mikusz. Software product lines, service-oriented architecture and frameworks: Worlds apart or ideal partners? In *Trends in Enterprise Application Architecture*, volume 4473/2007, pages 187–201. Lecture Notes in Computer Science, Springer, 2007.

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

[13] 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 *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society.

[14] 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.

[15] 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, pages 282–293, London, UK, UK, 2002. Springer-Verlag.

[16] C. W. Krueger, D. Churchett, and R. Buhrdorf. Homeaway's transition to software product line practice: Engineering and business results in 60 days. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 297–306, Washington, DC, USA, 2008. IEEE Computer Society.

[17] R. Krut and S. Cohen. Service-oriented architectures and software product lines - putting both together. *Software Product Line Conference, International*, page 383, 2008.

[18] 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*, June 2013.

[19] E. Murugesupillai, B. Mohabbati, and D. Gašević. A preliminary mapping study of approaches bridging software product lines and service-oriented architectures. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 11:1–11:8, New York, NY, USA, 2011. ACM.

[20] 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. S.e.i. a framework for software product line practice, version 5. http://www.sei.cmu.edu/productlines/framework.html, 2013.

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

[22] M. Schulze, J. Weiland, and D. Beuche. Automotive model-driven development and the challenge of variability. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 207–214, New York, NY, USA, 2012. ACM.

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

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