An Assessment on Technologies for Implementing Core Assets in Service-Oriented Product Lines

Heberth Braga G.
Ribeiro and Silvio
Romero de Lemos
Meira
Informatics Center,
Federal University of
Pernambuco (CIn-
UFPE)
Recife, PE, Brazil
{hbgr,
srlm}@cin.ufpe.br

Eduardo Santana de
Almeida
Computer Science
Department, Federal
University of Bahia
(DCC-UFBA)
Salvador, BA, Brazil
{esa}@dcc.ufba.br

Daniel Lucrédio and
Alexandre Álvaro
Computer Science
Department, Federal
University of São
Carlos (DC-UFSCAR)
São Carlos/Sorocaba,
SP, Brazil
{daniel,
alvaro}@dc.ufscar.br

Vander Alves
Computer Science
Department,
University of Brasília
(DC-UnB)
Brasília, DF, Brazil
{valves}@unb.br

Vinicius Cardoso
Garcia
Reuse in Software
Engineering Group
(RiSE)
Recife, PE, Brazil
{vcg}@rise.com.br

Abstract—Software Product Lines (SPL) and Service-Oriented (SO) are two strategies that focus on different goals, but they both use software reuse as their main principle. SPL encompass a family of software systems developed from reusable assets, while SO is a way of developing service-based applications, providing reusable business functionalities. In the combination of SPL and SO, a key aspect is the understanding about the use of variability mechanisms for realizing variabilities in different levels of granularity (e.g., *components*, *services*, *service-orchestrations*) addressed in a technology specific way and improve modularity, complexity, and Separation of Concerns (SoC) of reusable assets. In this paper, we present a quantitative case study for evaluating the *modularity*, *complexity*, *SoC* and *stability* of reusable core assets using distinct technologies for implementing them, and an initial decision model based to guide software engineers on choosing suitable technologies.

Keywords—decision model; software product lines; service-oriented development; software metrics; service technologies;

I. INTRODUCTION

Software reuse is a key factor for enterprises interested in productivity gains, reduced development costs and increased software quality. In this sense, SPL and SO are two strategies that focus on distinct goals, but use software reuse as their main principle. In the context of software reuse, SPL explores commonality and variability among related products, while SO is a way of developing service-based applications, which has in its roots the Service-Oriented Architecture (SOA). SOA is a model with the support of *designing*, *developing*, *deploying*, and *managing* systems in which services provide reusable business functionality [1]. Furthermore, applications or other service consumers are built using functionalities from available services, and the SOA infrastructure enables the *discovery*, *composition*, and *invocation* of services [1].

In this way, SPL and SO encourage organizations to reuse existing services and components instead of redeveloping them repeatedly from scratch [2]. The development of a SPL using software architectures and implementation supports systematic planned reuse, high software flexibility and contract-based communication. Moreover, it produces a common, managed set of core assets

for rapidly developing customized products for specific customers. On the other hand, a SOA implementation exposes standard interfaces to make services available for authorized service consumers to use in a variety of ways [1].

However, reasoning about how to combine SPL and SOA concepts, variability implementation and service technological concerns become a very challenging task [1, 2]. The challenges consist of more complex implementation activities, since it is important to understand the variability mechanisms for realizing variabilities in different levels of granularity (e.g., *components*, *services*, *service-orchestrations*) at runtime and feature types (e.g., *alternative*, *optional*, and *OR-feature*) [3]. Moreover, it becomes more critical when these aspects are addressed in a technology-specific way. For instance, the need of mapping these aspects with the interaction and life cycle of services, making the assembly of distinct services easier to perform, since it is expected that it is under SOA quality attributes, such as *loose coupling*, *interface-based design*, *composability*, and *discoverability*.

In the literature, many studies [1, 4, 5] have discussed the relationship between SPL and SOA concepts. In particular, these work focus on solutions for variability implementation in service-oriented product lines. Nevertheless, they focus on services as core assets in SPL in which the services handle variabilities and they address the relationship between SOA and SPL from a conceptual level but do not go into details about the implementation. Consequently, they do not provide decisions for recommending specific variability mechanisms according to granularity, modularity and SoC principles. Furthermore, they do not provide relationship between variability and technological concerns, such as the maintenance of reusable services as SPL core assets through a transparent life cycle mechanism, neither address a quantitative analysis.

In order to understand and mitigate this scenario, this paper provides a quantitative case study for comparing service technologies with the support of variability mechanisms to implement core assets in service-oriented product lines. According to the results of the case study analysis, we have constructed an initial decision model based on the analysis and some discussion with respect to technological and variability

concerns. In this sense, given a variability mechanism and some criteria (e.g., *modularity*, *complexity*, *understandability*, and *stability*), the decision model is able to indicate which technology is suitable for supporting, and consequently, implementing them.

II. VARIABILITY IN SERVICE-ORIENTED PRODUCT LINES

The main goal of SOA and SPL is that both approaches encourage an organization to reuse existing assets and capabilities rather than to develop them from scratch. SPL explores commonalities and variabilities among a set of related products. On the other hand, SOA enable orchestration, assembly and maintenance of enterprise solutions to quickly react to changing business requirements [2]. Reusing services can be improved by creating a product line that satisfies several variability requirements from different customer applications.

In this way, variabilities are included in the product line architecture following SOA concepts and implemented using services or components. A specific configuration could select the component or service, depending on the specific functional or quality features needed by the application and satisfied by each option [2]. This configuration can be applied in the context of service technologies responsible to implement the services and components and variability mechanisms to customize specific and variable features.

Variability mechanisms choice must be based on a systematic analysis of the context where the mechanism will be used, the variation point and the binding time that is supported by the variability. In this context, a variability mechanism is responsible for implementing a *variation point*. A variation point is a place in software system where choices are made as to which *variant* to use. Moreover, variant is a specific decision that is bound to a variation point for each core asset.

In service-oriented product lines, variation points can be employed during the *service invocation*, that is the activity in which a service invokes other services to accomplish its goals [4]. In this way, variability in the following cases may occur: messages can be exchanged in a *synchronous* or *asynchronous* way; the *protocol* used for *communication* can vary; and the *service provider* selection can be made at *compile-time* or *runtime*, hard-coded in the source code or delegated to a specific entity, such as a *service registry* through the use of an appropriate service technology (e.g., Web Services, OSGi, etc.).

Thus, for this study purpose, we decide to select variability mechanisms decisions that provide support to map variability in order to cover infrastructure usage patterns and best practices based on runtime configuration required in service-oriented product lines environment. The variability mechanisms are briefly explained as follows:

- *Binding variants within services*: In service-oriented applications, the binding time of services are usually defined at runtime, without the need to explicit dependencies among them. It allows the loose coupling, since they do not need to know who will implement its functionalities. Thus, enabling what will be exposed and what will be hidden in a modular way. In this context, *Dependency Injection* (DI) [6] may be applied to bind

different services considering specific customer's requirements to implement variability. As a consequence, such approach may be useful as a mechanism to implement SPL variabilities at runtime.

- *Parameters*: It is a simple variation mechanism in which parameters are used to invoke different variations of a service [1]. In this context, Configuration Files [3, 7] can be useful to separate the source code from the definition of the parameters that may change, facilitating its understandability and modularity.
- *Strategy Pattern*: In a SOA context, a product line may support dynamic variations through the combination of mechanisms that can handle such dynamic behavior. In the context of the Strategy Pattern [8], we can dynamically decide which strategy to choose but the choice is still limited by the set of strategies defined at compile-time. In this sense, a technology responsible for controlling the modules at run-time may be applied to enable its full dynamicity.

III. SERVICE TECHNOLOGIES

In a service-oriented context, considering orchestrating and designing applications, it is expected that it must be under SOA quality attributes, such as *loose coupling*, *interface-based design*, *composability*, and *discoverability* [9]. However, this scenario increases with the management of variability in service-oriented product lines, since the development of services as core assets must be produced with a good modularity, well-defined interfaces and granularity concerns in variabilities identification. In this case, granularities concerns are services decomposition as possible variants under an approach that focus on its life cycle control. This approach relies on service technologies, which are useful to provide support for service implementation and to allow the configuration (e.g., *addition*, *exclusion* or *replacement*) of services at runtime.

Indeed, there are many service technologies that can fit well in this context, such as Open Services Gateway Initiative (OSGi), Apache Tuscany and JAX Web Services. Such technologies are generally coupled with a configuration environment to allow the adequate management (e.g., *assembly*, *adaptation* and *connection*) of core assets. Moreover, core assets configuration allows discoverability and composability for different customers or application at runtime. These characteristics are the core of both technologies, which will be briefly described next.

OSGi [10] is an interface specification that defines an open, common architecture to develop, deploy and manage services in a coordinated fashion, which enables the capability of applications to dynamically discover and use services provided by other applications running inside the same OSGi platform [10]. Moreover, when it comes down to service-oriented product lines context, it is suitable for implementing variation points, by providing an easy way to dynamically perform changes among different implementations [11]. In this scenario, new services and components can be included without the need to recompile the whole system, thus, facilitating the upgrading and replacement of single services in a dynamic way.

Apache Tuscany is an open source project at the Apache Software Foundation that aims to develop SOA solutions by providing an infrastructure for its development and management. It provides a lightweight infrastructure that implements Service Component Architecture (SCA) [12], which has the capability for composing applications from business services resulting in an assembly model. The assembly model defines an eXtensible Markup Language (XML) language for assembling services components into applications at runtime. The use of such assembly model can be suitable for selecting services and components depending on specific features selected for distinct service-oriented product lines applications.

Java API for XML Web Services (JAX-WS) [13] is a programming model that simplifies application development through support of a standard, *annotation-based* model to develop Web Service applications and clients. The JAX-WS specification aligns itself with the current industry trend towards a more document-centric messaging model introducing support for annotating Java classes with meta-data to indicate that such class is a Web service. Using annotations within the Java source simplifies the development of Web services allowing flexibility for creating a delegation model that can be used to support variability mechanisms decisions to design customized service-based applications.

IV. CRITERIA FOR THE ASSESSMENT

This section presents the criteria that will be useful to define the case study proposed in this paper, allowing the collection and interpretation of the data in the context of service-oriented product lines projects. The idea behind service-oriented product lines is to develop and consume reusable service modules from a common base and map its variations. In the context of service-orientation, the central purpose is to achieve improved SoC through the decomposition of modules into services/service-orchestration in a high-level and components in a low-level, affecting attributes such as *instability*, *coupling*, *cohesion*, and *size*.

We believe that such scenario could be significantly enhanced when a service technology is properly tailored to a domain, culminating with activities that focus on providing and consuming reusable services. Furthermore, this scenario increases when SPL takes place with the definition and mapping of variability within the service technology. Consequently, the use of variability implementation mechanisms becomes unavoidable, making the decisions on the implementation more complex.

However, such *complexity* in addressing variability implementation mechanisms within a technology has a positive side, since it may modularize critical concerns as services and components, reducing the replication of code, the size of the system, and making concerns more stable to be consumed by others. On the other hand, the inappropriate use of a service technology and variability implementation mechanisms together may affect negatively these attributes and increase the complexity of the application domain, making the assets hard to maintain and reuse. According to Briand et al. and Sant'Anna et al. [14, 15], software metrics are the most effective way to supply empirical evidence that may improve the understanding of different dimensions of the software reusability and maintainability.

The metrics are more effective when they are associated with some measurement framework, allowing software engineers to understand and interpret the meanings of the collected data [15]. In this sense, creating a metric suite is important to capture information about the code generated in service-oriented product lines projects in terms of attributes such as *size*, *coupling*, *cohesion*, *instability* and *SoC*. This work defines a suite that reuses and refines classical and current metrics defined in [16, 17, 18, 19, 15, 20, 21, 11, 22, 23]. The criteria for the selection and refinement of these metrics were based on theoretical and practical demands and will be described in Section V. In addition, they were chosen and refined to reflect new abstractions introduced by the combination of SOA and SPL.

Our suite is composed of six code metrics grouped according to the attributes they measure, for example: size, coupling, cohesion, instability, and separation of concerns. Furthermore, the description of each metric emphasizes how it satisfies the measurement requirements defined. According Sant'Anna et al. [15], measuring particular internal attributes, such as cohesion, is useful if it is related to a measure of some external attribute of the object of study, for example, reusability or maintainability. In this sense, we developed a measurement framework to capture the understanding of the size, coupling, cohesion, instability, and separation of concerns attributes in terms of their usefulness as predictors of the maintainability and reusability qualities. Fig. 1 depicts the measurement framework defined for this study's purposes.

Figure 1. Measurement Framework.

As it can be seen in the Figure, the framework is composed of three different elements: *qualities*, *factors* and *attributes*. The qualities are the attributes that must be observable in the system (reusability and maintainability). The factors (complexity, understandability, stability, and modularity) are secondary quality attributes that influence the primary

qualities. The attributes are related to well-established software engineering principles, which are essential in SOA and SPL development.

V. AN ASSESSMENT ON TECHNOLOGIES FOR IMPLEMENTING CORE ASSETS IN SERVICE-ORIENTED PRODUCT LINES

This section presents the case study for analyzing how specific technologies support variability implementation mechanisms using different feature types in the context of service-oriented product lines projects. In order to conduct this case study, we organized the section following concepts proposed by [24, 25, 26]. Moreover, the output of the case study is the use of a decision model for representing the relationships among technologies and variability mechanisms, providing useful guidance for software engineers in further projects.

A. Definition

The definition of the case study was based on the GQM (Goal/Question/Metric) paradigm [27]. According to this paradigm the *goal* of the case study is:

Goal: Analyze *technologies* (OSGi, Apache Tuscany and JAX-WS) to implement core assets for the *purpose* of *evaluation* with respect to *reusability, maintainability, complexity, stability, modularity and understandability* from the *point of view* of the software engineers and researchers in the *context* of service-oriented product lines projects.

The *questions* and *metrics* that must be answered and interpreted by the case study are:

Q1. *How complex are the services and components developed using a particular technology?* Structural dependencies between services/components providers and consumers have considerable influence on system complexity. In this context, the following metrics were defined to help answering this question:

M1. Weighted Operations per Component or Service (WOCS). WOCS will measure the complexity of the service or component, in terms of its operations (methods) that will be required by other services or components. Hence, consider a Component or Service C with operations $O_1, \dots, O_n$. Let $c_1, \dots, c_n$ be the complexity of the operations. Then: $WOCS = c_1 + \dots + c_n$. The metric indicates that operations with many formal parameters are more likely to be complex than operations in which require less parameter. In this sense, the complexity of operation O_k , c_k can be defined as follows: $c_k = \alpha_k + 1$, where α_k denotes the number of formal parameters of O_k . According to Chidamber and Kemerer [17], a value greater than 10 is considered a too high value.

M2. Cyclomatic Complexity (CC). Cyclomatic Complexity [16] is measured in terms of the number of paths (e.g., loops and conditional statements) through a program, and can be useful to keep the size of the operations provided by services or components manageable. This measure is calculated from a connected graph of the services or component as follows: $CC = E - N + p$, where: E = the number of edges of the graph; N = the number of nodes of the graph;

and, p = the number of connected services or components. According to McCabe [16], this metric indicates that a unit with CC between 1 and 10, it is a simple program, without much risk. A value between 11 and 20 represents a more complex program, with moderate risk. Values ranging between 21 and 50 represent a complex program, with high risk. Finally, a complexity greater than 50 presents a high risk.

Q2. *How stable are the services and components developed using a particular technology?* A single change may begin a cascade of changes of independent services or components making the design and development fragile and difficult to reuse. In order to answer this question, we define one metric:

M3. Instability Metric for Service or Component (IMSC). This metric is supported by *fan.in* and *fan.out* metrics, reported by the Metric Advisor from C-DAC¹. Pereplechikov et al. [28] defined the IMSC to measure the interaction between a service or component through sending and receiving messages as follows: $IMSC = \frac{fan.out}{fan.in + fan.out} \times 100\%$. If the value of the instability is low, the level of dependency of service is low whereas others depend on it higher [28]. Thus, according to Pereplechikov et al. [28], $IMSC = 0$ means that the stability of the service is very high. On the other hand, $IMSC = 1$ means that the service is very instable, difficulty the maintenance.

Q3. *Do the technologies produce services and components with low coupling?* It is important to measure the extent to which interdependencies exist between services/components providers and consumers. A well-modularized design and development of a SOA application brings forth potential in multiple aspects, such as acceleration of development, reduction of maintenance cost, as well as the enhanced flexibility and reusability. We define one metric to understand this question:

M4. Coupling Between Components or Services (CBCS). This metric is calculated based on the number of relationships between one service A, for example, and other services in an application. It can be mathematically described as follows:

$$CBCS = \frac{\sum_{i \neq j=1}^n A_i B_j}{n}$$

In which: n is the number of services in application; $A_i B_j = 0$ if A_i does not connect to B_j and $A_i B_j = 1$ if A_i connects to B_j . For a service A, the larger the value of CBCS metric, the tighter the relationship with other services is. In other words, service A depends much more on others [23]. This metrics is based on the Coupling between Objects (CBO) metric from C K metrics [17] and Coupling between Components (CBC) defined in [15, 29] work. A CBCS value greater than 9 is considered a too high value.

Q4. *Do the technologies produce services and components with high cohesion?* In the context of SOA design and development, low cohesion between services or components decreases software modularity, thereby increasing the probability of errors during the development process. In order

¹ <http://www.cdac.in/html/ssdgbllr/metric.asp>

to understand and help answering this question, we define the following metric:

M5. Lack of Cohesion over Operations (LCOO).

According to Sant'Anna et al. [15], this metric can be measured by counting the number of pairs of operations that share attributes, minus the number of pairs of operations that do not share any attribute. This metric can be represented mathematically as follows:

- If a component C_1 has n operations (methods) $O_1, \dots, O_n$ then $\{I_j\}$ is the set of instance variables used by operation O_j :

a) Let $|P|$ be the number of method pairs which access no common attribute of C_1 .

b) Let $|Q|$ be the number of method pairs which access at least on common attribute of C_1 .

Then: $LCOO = |P| - |Q|$, if $|P| > |Q|$, $LCOO = 0$, otherwise.

A high LCOO (more than 20) value could indicate that the implementation of the component or service is poor (i.e., it might be worthwhile to split both into two or more modules).

Q5. How easy is to separate concerns using a particular technology? The decomposition of service-oriented product lines applications into services and components may allow the SoC in specific self-contained and replaceable building blocks [30]. In order to understand and answering it, the following metric will be used:

M6. Concern Diffusion over Components or Services (CDCS). This metric was adapted to the context of service-oriented product lines context using services and components to implement possible concerns identified. Thus, the CDCS will count the number of services or components whose main purpose is to contribute to the implementation of concern. A high value for CDCS metric indicates that a concern implementation might be scattered.

B. Hypotheses

Defining the hypotheses correctly is both the most difficult and the most important part of defining a case study. According to Wohlin et al. [26], the *null hypotheses* are those who the experimenter wants to reject, while the *alternative hypotheses* are those that the experimenter wants to confirm.

Null hypotheses. In this study, the null hypotheses state that there is no difference among the technologies for implementing core assets in service-oriented product lines projects.

$$\begin{aligned}
 H_0: \quad & \mu_{\text{Tuscany WOCS}} = \mu_{\text{JAX-WS WOCS}} = \mu_{\text{OSGi WOCS}} \\
 & \mu_{\text{Tuscany CC}} = \mu_{\text{JAX-WS CC}} = \mu_{\text{OSGi CC}} \\
 & \mu_{\text{Tuscany IMSC}} = \mu_{\text{JAX-WS IMSC}} = \mu_{\text{OSGi IMSC}} \\
 & \mu_{\text{Tuscany CBCS}} = \mu_{\text{JAX-WS CBCS}} = \mu_{\text{OSGi CBCS}} \\
 & \mu_{\text{Tuscany LCOO}} = \mu_{\text{JAX-WS LCOO}} = \mu_{\text{OSGi LCOO}} \\
 & \mu_{\text{Tuscany CDCS}} = \mu_{\text{JAX-WS CDCS}} = \mu_{\text{OSGi CDCS}}
 \end{aligned}$$

Alternative Hypotheses. In this study, the alternative hypotheses state that there are differences among the

technologies for implementing variabilities in service-oriented product lines projects.

$$\begin{aligned}
 H_1: \quad & \mu_{\text{Tuscany WOCS}} \cdot \mu_{\text{JAX-WS WOCS}} \cdot \mu_{\text{OSGi WOCS}} \\
 & \mu_{\text{Tuscany CC}} \cdot \mu_{\text{JAX-WS CC}} \cdot \mu_{\text{OSGi CC}} \\
 & \mu_{\text{Tuscany IMSC}} \cdot \mu_{\text{JAX-WS IMSC}} \cdot \mu_{\text{OSGi IMSC}} \\
 & \mu_{\text{Tuscany CBCS}} \cdot \mu_{\text{JAX-WS CBCS}} \cdot \mu_{\text{OSGi CBCS}} \\
 & \mu_{\text{Tuscany LCOO}} \cdot \mu_{\text{JAX-WS LCOO}} \cdot \mu_{\text{OSGi LCOO}} \\
 & \mu_{\text{Tuscany CDCS}} \cdot \mu_{\text{JAX-WS CDCS}} \cdot \mu_{\text{OSGi CDCS}}
 \end{aligned}$$

C. Planning

After the definition of the case study, the planning was started. The case study was conducted in *off-line* (not industrial software development) in the context of a service-oriented product line project.

Pilot Project. The case study presented was based on the development of a SPL in the domain of libraries, herein named the *Rental for All Software Factory* (R4ll). The R4ll was developed as part of a postgraduate course at the Federal University of Pernambuco, Brazil. The SPL project development involved all life cycle phases ranging from scoping to testing concerns. For validation purposes, the architecture reference and the feature model of R4ll were analyzed and adapted in order to support service-oriented product lines concepts, such as services that encapsulate variabilities in different levels of granularity (service operations, and components) [30]. The feature model was defined for capturing commonalities and variabilities of the domain, which resulted in *fifty-nine* features, and *sixteen* variabilities could be identified. In this way, three critical variabilities (*Authentication*, *Fine*, and *Item*) were defined, since they are usually used in any library application. Moreover, they can be encapsulated as possible services, since a system may track authentication and fine operations from distinct and third-party services. The entire source used to perform this case study is available for consultation in <http://www.cin.ufpe.br/~hbgr/>.

Subjects. Only one subject was chosen to perform the case study, and he is a master student at the Informatics Center in the Federal University of Pernambuco (CIn-UFPE). The decision to choose only one subject is further detailed in the *internal validity* section.

Training. Initially, the subject read the technologies specifications. Next, examples were implemented, and other sources, such as papers and technical reports were considered to complement the study. Moreover, the subject had previous experience in the development of one software product lines project. Thus, it was assumed that the subject was already familiarized with concepts related to variability and domain engineering, then, facilitating the understanding of the domain without the need of training.

Quantitative analysis mechanism. The quantitative analysis was divided as *complexity*, *modularity*, *stability*, and *understandability* for the implementation steps. It is important to highlight that these quality factors are predictors for *reusability* and *maintainability*, and thus it will be analyzed as

well. Furthermore, it will also use descriptive statistics, such as *mean*, *maximum (max)*, *minimum (min)*, and *standard deviation (std.)* to analyze the data generated by the case study. Regarding the hypotheses defined for the case study, it will be used the ANalysis Of VAriance (ANOVA) test [26]. The ANOVA can be used to analyze case studies (or experiments) from a number of different designs. In this case study, the ANOVA was applied with its simplest form, i.e., the one factor for three or more different treatments.

Case study design. In order to provide more quality in what will be assessed, we decide to divide the assessment into three *treatments*: (i) the first stage encompassed the implementation of the R4ll artifacts (service modules, components, and classes) using Apache Tuscany as the main technology and three variability mechanisms (*Binding variability within services*, *Parameters*, and *Strategy Design Pattern*); (ii) in the second stage, the subject implemented the R4ll artifacts using JAX-WS technology and the three variability mechanisms aforementioned; and, (iii) in the third and last stage, the R4ll artifacts were implemented using OSGi. This separation of treatments was implemented in this manner to reduce the possible confounding factors regarding the interference of using a more complex technology and activities aimed at learning its usage within the use of a less simple technology.

Data collection procedures. The case study was performed during March-July 2009, at the Federal University of Pernambuco, Brazil. In addition, it was developed around 580 hours and 45 minutes, including all the implementation. The project was implemented with the support of three technologies (*OSGi*, *Apache Tuscany* and *JAX-WS*), and for each technology, three variability mechanisms were applied (e.g., *Binding variability within services*, *Parameters* and *Strategy Design Pattern*). For each of the technologies implementation, several metrics data were collected, such as *complexity*, *modularity*, *stability*, and *understandability* metrics. The metrics data were collected using the following tools: *metrics*² and *Borland Together*³. For the *SoC* metric, the collection was made following the guidelines proposed by [21].

Internal validity. This case study has five dimensions of expertise, such as *software product lines*, *service-orientation concepts*, *variability implementation mechanisms*, the *combination between service-orientation and software product lines*, and *R4ll domain*. Present knowledge in these areas might take considerable time, one year at least. Consequently, as the deadline to perform the case study was shorter, the subject team formation and training required could take quite some effort and time. In this sense, we chose to use only one subject with considerable expertise in these areas. This might lead with mistreatments with the data collected, since different acquirements are not considered. Nevertheless, the participation of the subject in the R4ll development is fundamental to justify this case study execution with a single

subject. Furthermore, it is important to state and we encourage that this case study might be replicated using subject teams providing thus more accurate data. Moreover, the small domain considered may have also influence in the internal validity. However, trying to solve these possible issues, critical variations were chosen, and, as the subject have been participated on the entire R4ll development life cycle it is acceptable that it might be controlled.

External validity. Due to time constraints, it will not be possible to apply the study in other research groups. Nevertheless, the external validity of the study is considered sufficient, since it aims to select suitable technology that fits well in the context of service-oriented product lines. Therefore, it may be used as baseline for comparison in further studies developed in the same context of this one.

Conclusion validity. The analysis and interpretation of the results of this case study will be described using descriptive statistics, which are appropriate to the data type collected during the case study.

D. Analysis

This section involves analyzing the data collected after the implementation. The output of the analysis is a decision model for recommending technologies to implement core assets in service-oriented product lines projects. The following subsections were organized based on the quantitative analysis mechanisms defined in the planning phase. In this sense, descriptive statistics will be applied, as well as the hypotheses testing using ANOVA.

Complexity Analysis. As can be seen in Table 1, in the *Binding variants within services* mechanism analysis, the WOCS values are higher for the Apache Tuscany, because it was necessary to define a method to instantiate the *UserService*, and for OSGi and JAX-WS it was not necessary. However, the values are also higher for JAX-WS because it was important to define *onCycleDetected* methods in the variant represented by *Item* and *Category* to avoid the circular reference issue. In the context of the *Parameters* mechanism, a *Factory Method Pattern* [8] was defined to instantiate the correct feature, which is reflected for the higher values for CC and WOCS concerning to Apache Tuscany and JAX-WS. It happens because extra operations for instantiate the variants that must be created for *PenaltyForDelay* and *PenaltyForViolation*, and processed by the web services. In the context of the *Strategy Pattern* mechanism, the values of OSGi for WOC metric is lower when compared to the Apache Tuscany and JAX-WS. In the OSGi implementation, the *Fine* variant defines only four methods that will be overridden by the concrete classes. On the other hand, for the Apache Tuscany and JAX-WS were defined six methods.

Table 2 summarizes the results of the ANOVA test for the CC and WOCS measures. In the context of the ANOVA analysis, it is important to verify if the treatments have different mean values. Thus, we have to observe the F_0 and F_{CRITIC} values. If $F_0 > F_{CRITIC}$, the null hypotheses are rejected, concluding that have differences in the means among the treatments. Hence, as can be seen in Table 2, the ANOVA did not reject the null hypothesis for both *Binding variants with services*, *Parameters* and *Strategy Pattern* mechanisms ($F_0 <$

² <http://metrics.sourceforge.net/>

³ <http://www.borland.com/br/products/together/>

F_{CRITIC}) for CC. In this sense, it means that there is no difference among the technologies. However, they can be classified in different priority levels, according to the points discussed in the descriptive statistics analysis. It is important to state these assumptions, since it will help in the construction of our decision model. For WOCS, the hypotheses cannot be rejected, since $F_0 < F_{CRITIC}$.

TABLE I. RESULTS FOR THE COMPLEXITY ANALYSIS.

Technology	Mechanism	Measure	Statistic			
			Mean	Max	Min	Std.
OSGi	Binding Variants	CC	8	13	2	4.6
		WOCS	11.2	17	3	5.6
	Parameters	CC	7.2	12	3	3.5
		WOCS	9.6	16	2	5.3
	Strategy Pattern	CC	7	12	2	3.8
		WOCS	10	16	2	5.2
Apache Tuscany	Binding Variants	CC	11	20	3	7
		WOCS	13.6	26	3	9.4
	Parameters	CC	10.3	18	3	6.1
		WOCS	12.8	24	3	8.6
	Strategy Pattern	CC	10.6	18	2	6.5
		WOCS	13.5	24	2	8.8
JAX-WS	Binding Variants	CC	11	20	2	7.3
		WOCS	16	36	6	11.6
	Parameters	CC	11.1	19	3	6
		WOCS	12.8	25	6	7.9
	Strategy Pattern	CC	10.8	19	2	6.7
		WOCS	13.1	25	2	8.8

TABLE II. ANOVA RESULTS FOR COMPLEXITY.

ANOVA	Binding variants		Parameters		Strategy Pattern	
	CC	WOCS	CC	WOCS	CC	WOCS
F_0	0.8145	0.4766	0.774	0.320	0.7	0.368
F_{CRITIC}	3.738	3.738	3.73	3.738	3.74	3.738

Stability Analysis. As can be seen in Table 3, the values are higher for OSGi and a little close for Apache Tuscany. In the OSGi implementation side, it must be defined the *bundle context* and additional code for variants discovery. Consequently, due to the dynamic behavior of OSGi, we cannot separate the persistence from the Bundle, creating a high coupling among them. This impact was reflected for the *Binding variants within services* mechanism values. Moreover, such condition is reflected also in the two other variability mechanisms (*Parameters* and *Strategy Pattern*).

As can be seen in Table 4, the ANOVA test did not reject the null hypotheses for *Parameters* and *Strategy Pattern* mechanisms ($F_0 < F_{CRITIC}$). On the other hand, it rejects the null hypothesis for *Binding variants within services* mechanism ($F_0 > F_{CRITIC}$). In the context of the null hypothesis rejected, JAX-WS is a well suitable technology. In contrast, OSGi was the technology with the highest value followed, respectively, by the Apache Tuscany.

Modularity Analysis. Table 5 presents the results for CBCS and LCOO metric. For modularity reasons required for OSGi, it was necessary to expose interfaces for *Item* and *Category* variants, reflecting in low values for both metrics (CBCS and LCOO). In contrast, for Apache Tuscany and JAX-WS, both entities were implemented as simple *JavaBean* classes, reflecting in a high coupling, and low cohesion, since they cannot exist independently. In the context of the *Parameters* mechanism, a *Factory Method Pattern* was

defined to instantiate the *PenaltyForDelay* and *PenaltyForViolation* variants. In this way, it can be observed that the values for CBCS metric are lower for OSGi and higher for JAX-WS and Apache Tuscany.

TABLE III. RESULTS FOR THE STABILITY ANALYSIS.

Technology	Mechanism	Measure	Statistic			
			Mean	Max	Min	Std.
OSGi	Binding Variants	IMSC	0.517	0.6	0.49	0.05
	Parameters	IMSC	0.517	1	0.49	0.21
	Strategy Pattern	IMSC	0.575	0.68	0.48	0.09
Apache Tuscany	Binding Variants	IMSC	0.42	0.66	0.13	0.22
	Parameters	IMSC	0.38	0.75	0.08	0.24
	Strategy Pattern	IMSC	0.34	0.37	0.12	0.33
JAX-WS	Binding Variants	IMSC	0.25	0.33	0.12	0.07
	Parameters	IMSC	0.35	0.52	0.13	0.17
	Strategy	IMSC	0.28	0.66	0.13	0.2

TABLE IV. ANOVA RESULTS FOR STABILITY.

ANOVA	Binding variants	Parameters	Strategy Pattern
	IMSC	IMSC	IMSC
F_0	4.704526	2.933618	2.203263
F_{CRITIC}	3.738892	3.738892	3.738892

In the OSGi implementation, it was defined an infrastructure for discovering variants explicitly in the code, contrasting the JAX-WS implementation that defines such infrastructure in a XML file. As can be seen in the Table, the CBCS values are quite similar for OSGi, Apache Tuscany and JAX-WS concerning the *FineService*. However, the user can choose JAX-WS or OSGi if he prefers to define additional infrastructure that increases the coupling value for the *PenaltyForDelay* class, or, he should use Apache Tuscany that does not need to define such infrastructure.

Table 6 summarizes the results of the ANOVA test for the CBCS and LCOO measures. In the analysis of the CBCS metric, the ANOVA did not reject the null hypotheses for *Binding variants within services* and *Strategy Pattern* mechanisms ($F_0 < F_{CRITIC}$). On the other hand, it rejects the null hypothesis for *Parameters* mechanism ($F_0 > F_{CRITIC}$). In the context of the null hypotheses rejected, OSGi is a well suitable technology. In contrast, Apache Tuscany was the technology with a middle value followed by a higher value for Apache Tuscany, respectively. Moreover, it also presents that the hypotheses cannot be rejected for the results of the LCOO metric, since $F_0 < F_{CRITIC}$.

Understandability Analysis. In order to measure the CDCS metric, we considered the following concerns: *penalty for delay*, *penalty for violation*, *comments*, and *authentication by password, fingerprint, and card*. These concerns are related to variants that are spread in two or more places in the application. Table 7 presents the values with respect to the CDCS metric. In the *Binding variants within services* mechanism, the *same two classes and the spring file in OSGi spread the PasswordAuthentication concern*.

TABLE V. RESULTS FOR THE MODULARITY ANALYSIS.

Technology	Mechanism	Measure	Statistic			
			Mean	Max	Min	Std.
OSGi	Binding Variants	CBCS	3.4	6	2	1.5
		LCOO	33.4	83	1	33.5
	Parameters	CBCS	3.6	7	2	1.9
		LCOO	23.4	55	1	23.8
	Strategy Pattern	CBCS	3.4	7	1	2.2
		LCOO	24.6	55	0	23.4
Apache Tuscany	Binding Variants	CBCS	4.8	8	2	2.3
		LCOO	32.8	86	1	31.7
	Parameters	CBCS	4.1	8	1	2.7
		LCOO	34.8	80	1	32.8
	Strategy Pattern	CBCS	4.3	8	1	2.7
		LCOO	36.5	80	0	32.8
JAX-WS	Binding Variants	CBCS	4.1	7	1	2.2
		LCOO	34.6	87	1	33.6
	Parameters	CBCS	8.3	13	3	4.2
		LCOO	34.1	83	2	34
	Strategy Pattern	CBCS	3.5	7	1	2
		LCOO	40	83	0	33.7

TABLE VI. ANOVA RESULTS FOR MODULARITY.

ANOVA	Binding variants		Parameters		Strategy Pattern	
	CBCS	LCOO	CBCS	LCOO	CBCS	LCOO
F ₀	0.644	0.007	3.741	0.222	0.2714	0.366
F _{CRITIC}	3.738	3.738	3.738	3.738	3.738	3.738

The Apache Tuscany value is lower because the *PasswordAuthentication* concern is spread over two classes (*Authentication* and *PasswordAuthentication*) and the spring *tuscany-content.xml* file. On the other hand, for JAX-WS, it is spread over three classes and the spring *beans.xml* file. In the *Parameters* mechanism, what changes is that instead of mapping concerns in the spring files, here it will be mapped in a class responsible for load the configuration file (the *PropertiesFactory* class). However, when we compare the three technologies, OSGi and Apache Tuscany has a quickly lower value with respect to each other and the JAX-WS. In the *Strategy Pattern*, only we have to do is map the concerns in a context class. In the Apache Tuscany and JAX-WS side, the higher value (5) for CDCS occurs because we decided to adapt the Strategy, creating a *Data Structure Type* for defining the *PenaltyForDelay* variant due to constraints for web services implementation.

As can be seen in Table 8, the ANOVA test did not reject the null hypotheses for *Parameters* and *Strategy Pattern* mechanisms ($F_0 < F_{CRITIC}$). On the other hand, it rejects the null hypotheses for *Binding variants within services* mechanism ($F_0 > F_{CRITIC}$). In the context of the null hypotheses rejected, Apache Tuscany is well suitable. In the context of the hypotheses that was not rejected, priority levels will be defined according to the points discussed, for each technology.

E. Decision Model

The motivation for the decision model relies on the management of variability identified in the core assets, and how complex is the mapping of such approach in a service-oriented environment. Nevertheless, differently of the existing models [3, 31], our decision model is based on a quantitative analysis. In the work reported in [29], they constructed a decision model in the product lines context for recommending only variability mechanisms for three criteria. In contrast, our approach has the focus on the service-oriented product lines context, which implies in the analysis of different criteria and

metrics for recommending service technologies combined with variability mechanisms.

TABLE VII. RESULTS FOR THE UNDERSTANDABILITY ANALYSIS.

Technology	Mechanism	Measure	Concerns				
			PasswordAuthentication	CardAuthentication	FingerPrintAuthentication	Comments	PenaltyForDelay
OSGi	Binding Variants	CDCS	4	4	4	4	4
	Parameters	CDCS	4	4	4	4	4
	Strategy Pattern	CDCS	4	4	4	4	4
Apache Tuscany	Binding Variants	CDCS	3	3	3	4	4
	Parameters	CDCS	3	3	3	5	5
	Strategy Pattern	CDCS	3	3	3	5	5
JAX-WS	Binding Variants	CDCS	4	4	4	5	5
	Parameters	CDCS	4	4	4	5	5
	Strategy	CDCS	4	4	4	5	5

TABLE VIII. ANOVA RESULTS FOR UNDERSTANDABILITY.

ANOVA	Binding variants	Parameters	Strategy Pattern
	CDCS	CDCS	CDCS
F ₀	7.5	1	1
F _{CRITIC}	3.68232	3.68232	3.68232

Since our decision model is variability mechanism centric, the **first input** consists of *variability mechanisms* (e.g., *Binding variants within services*, *Parameters*, and *Design Patterns*). The **second input** is some *criteria* used to compare the technologies. The criteria are quality factors (e.g., complexity, and modularity). Finally, the **last input** consists of a *catalog of technologies* (e.g., *OSGi*, *Apache Tuscany*, etc).

Figure 2. An Application of the Decision Model.

As exemplified in Fig. 2, the output is represented by a *tuple* that contains besides the technologies recommended, the variability mechanisms and a particular criterion. The tuple depicted in Fig. 5 is an example of the result. It is important to highlight that not always will be possible to decide which technology fits better to a set of criteria. For instance, if *complexity* criterion is considered and all the technologies are similar according to this, both technologies are recommended.

However, in some cases we leave the decision for the user. This decision is based on the points discussed in the analysis section of the case study.

For a better understanding of the model and due to the size limitation, we decide to split the model in three parts as depicted in Fig. 3, Fig. 4, and Fig. 5. However, the three parts can be considered as a unique decision model as a whole. The “**lamp (idea)**” icon indicates the result of the answer about the criterion question (represented by the “?” icon), i.e., the technology(ies) recommended. The answer is performed in two ways: **yes** or **no**. The “**yes**” icon indicates that the technologies satisfy a particular criterion.

Figure 3. Binding variants within services mechanism.

Figure 4. Parameters mechanism.

The “**no**” icon concerning the *complexity* criterion indicates that the technologies provide core assets with a low complexity. On the other hand, the “**no**” concerning the other criteria is a negative aspect. Moreover, as can be seen in the Figures, priorities were defined. These priorities are based on the analysis discussion, in particular when the hypotheses were not rejected. Consequently, each of the technology was marked according to a specific priority that differentiates each one.

For instance, as can be seen in Fig. 3, OSGi, Apache Tuscany, and JAX-WS were considered suitable to be applied using the *Binding variants within services mechanism* in a requirement that needs low complexity. In this example, OSGi was defined with the Priority 1, since according to the analysis of the values, OSGi is suitable in some situations that provide an implementation with low complexity.

Figure 5. Strategy Pattern mechanism.

VI. CONCLUDING REMARKS AND FUTURE WORK

This paper presented a preliminary decision model based on a quantitative case study performed in the academy. The goal of the case study was to evaluate service technologies (OSGi, Apache Tuscany and JAX-WS) for implementing services as SPL core assets with the support of variability mechanisms (*Binding variants within services*, *Parameters*, and *Strategy Pattern*) in a service-oriented product line project. The case study was accomplished and the technologies were compared among each other. The results of the case study pointed out that the differences among the technologies are very few, and some hypotheses could not be rejected.

However, it was important for defining an initial decision model based on some important points discussed in the case study analyses. Moreover, when we were building the decision model, some decisions were considered, and some technologies were pointed out with a priority level. In this way, some technologies may be suitable to be applied according to a particular context or customer needs. Nevertheless, it is clear that replications of this study must be performed taking into account a subject's team and other domains, which could be used to identify the application of the variability mechanisms to implement specific feature types, technologies, and the use of other mechanisms to draw more concrete conclusions.

As future work, we intend to use more quality factors, such as *composability*, *scalability*, *portability* and *performance* in our model. Moreover, we will consider more variabilities such as *Service Contract Variability*, which is important to understand about services, priorities, responsibilities and

warranties through the Service Level Agreement (SLA). Consequently, the use of other variability mechanisms decisions for service-oriented product lines will be applied to refine the model. In addition, we also intend to replicate the case study using more subjects and a more complex domain in order to compare it with the results previously reported.

ACKNOWLEDGEMENTS

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, and Brazilian Agency (CNPq process number 475743/2007-5).

REFERENCES

- [1] D. Smith and G. Lewis, "Service-oriented Architecture (soa) and software product lines: pre-implementation decisions", in Proceedings of the 3rd International Workshop on Service Oriented Architectures and Product Lines (SOAPL '09), San Francisco, California, USA, September 2009, pp. 166-171.
- [2] S. Cohen and R. Krut, "1st internacional workshop on service-oriented architectures and software product lines (soapl '07)", in Proceedings of the 11th International Software Product Line Conference (SPLC '07), Kyoto, Japan, September 2007.
- [3] M. Anastasopoulos and C. Gacek, "Implementing product line variabilities", in SIGSOFT Software Engineering Notes, ACM, volume 26, New York, NY, USA, 2001, pp. 109-117.
- [4] S. Segura, D. Benavides, A. Ruiz-Cortés, and P. Trinidad, "A taxonomy of variability in web service flows", in Proceedings of the 1st International Workshop on Service Oriented Architectures and Product Lines (SOAPL '07), Kyoto, Japan, September 2007.
- [5] S. Günther and T. Berger, "Service-oriented product lines: Towards a development process and feature management model for web services", in Proceedings of the 2nd International Workshop on Service Oriented Architectures and Product Lines (SOAPL '08), Washington, DC, USA, October 2008, pp. 131-136.
- [6] M. Fowler, "Inversion of control containers and the dependency injection pattern", Technical Report, <http://martinfowler.com/articles/injection.html>, 2004.
- [7] T. Santos and A. Santos, "Parameterizing java software (in portuguese)" in Proceedings of the 9th Brazilian Symposium on Programming Languages (SBLP '05), Recife, PE, Brazil, 2005, pp. 257-270.
- [8] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design patterns: elements of reusable object-oriented software", Addison-Wesley, 1995, pp. 315-324.
- [9] T. Erl, "Service-oriented architecture: concepts, technology, and design", Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
- [10] OSGi Alliance, "Osgi service platform core specification", Release 4, Version 4.1, 2007.
- [11] E. S. Almeida, E. C. R. Santos, A. Alvaro, V. C. Garcia, S. R. L. Meira, D. Lucrédio and R. P. de Mattos Fortes, "Domain implementation in software product lines using osgi", in Proceedings of the 7th International Conference on Composition-Based Software Systems (ICCBSS '08), Madrid, Spain, February 2008, pp. 72-81.
- [12] SCA, "Service component architecture", <http://www.osoa.org/display/Main/Service+Component+Architecture+Home>, 2009, Last access March/2009.
- [13] JAX-WS, "Java api for xml web services", <https://jax-ws.dev.java.net/>, 2009, Last access on April/2009.
- [14] L. Briand, E. Emam, and S. Morasca, "Theoretical and empirical validation of software product measures", Technical Report ISERN-95-03, Fraunhofer Institute for Experimental Software Engineering, 1995.
- [15] C. Sant'anna, A. Garcia, C. Chavez, C. Lucena and A. von Staa, "On the reuse and maintenance of aspect-oriented software: An assessment framework", in Proceedings of the 7th Brazilian Symposium on Software Engineering, 2003.
- [16] T. J. McCabe, "A complexity measure", IEEE Transactions on Software Engineering, SE-2(4), 1976, pp. 308-320.
- [17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design", IEEE Transactions on Software Engineering, 20(6), 1994, pp. 476-493.
- [18] R. Martin, "Oo design quality metrics: An analysis of dependencies", <http://www.objectmentor.com/resources/articles/oodmetric.pdf>, 1994, Last access on January/2009.
- [19] V. R. Basili, L. C. Briand and W. L. Melo, "How reuse influences productivity in object-oriented systems", Communications of the ACM, 39(10), 1996, pp. 104-116.
- [20] F. Castor, N. Cacho, E. Figueiredo, R. Maranhao, A. Garcia and C. M. F. Rubira, "Exceptions and aspects: the devil is in the details", in Proceedings of the 14th International Symposium on Foundations of Software Engineering (SIGSOFT '06), ACM, New York, NY, USA, 2006, pp. 152-162.
- [21] M. Eaddy, A. Aho and G. C. Murphy, "Identifying, assigning, and quantifying crosscutting concerns", Assessment of Contemporary Modularization Techniques, International Workshop, 0, 2, 2007.
- [22] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor and F. Dantas, "Evolving software product lines with aspects: an empirical study on design stability", in Proceedings of the 30th international conference on Software engineering (ICSE '08), ACM, New York, NY, USA, 2008, pp. 261-270.
- [23] P. T. Quynh and H. Q. Thang, "Dynamic coupling metrics for service-oriented software", International Journal of Computer Science and Engineering, ed. 3, vol. 1, 2009.
- [24] B. Kitchenham, L. Pickard and S. L. Pfleeger, "Case studies for method and tool evaluation", IEEE Transactions on Software Engineering, 12(4), 1995, pp. 52-62.
- [25] B. Kitchenham and L. Pickard, "Evaluating software engineering methods and tools: part 9: quantitative case study methodology" SIGSOFT Software Engineering Notes, ACM, 23(1), 1998, pp. 24-26.
- [26] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell and A. Wesslén, "Experimentation in Software Engineering: an Introduction", Kluwer Academic Publishers, 2000.
- [27] V. R. Basili, G. Caldiera and H. D. Rombach, "The goal question metric approach", in Encyclopedia of Software Engineering, Wiley, 1994.
- [28] M. Perepletchikov, C. Ryan, K. Frampton and Z. Tari, "Coupling metrics for predicting maintainability in service-oriented designs", in Proceedings of the 2007 Australian Software Engineering Conference (ASWEC '07), IEEE Computer Society Press, Washington, DC, USA, 2007, pp. 329-340.
- [29] M. Ribeiro, P. Matos and P. Borba, "A decision model for implementing product lines variabilities", in Proceedings of the 2008 ACM Symposium on Applied Computing, ACM, New York, NY, 2008, pp. 276-277.
- [30] F. M. Medeiros, E. S. Almeida and S. R. L. Meira, "Towards an approach for service-oriented product line architectures", in Proceedings of the 3rd International Workshop on Service Oriented Architectures and Product Lines (SOAPL '09), San Francisco, California, USA, September 2009, pp. 151-157.
- [31] D. Muthig and T. Patzke, "Product line implementation technologies: programming language view", Technical Report 057.02/e, Fraunhofer Institute for Experimental Software Engineering, 2002.