Benchmarking Stability of Aspect-Oriented Product-Line Decompositions

Francisco Dantas, Eduardo Figueiredo, Alessandro Garcia, Claudio Sant’Anna, Uirá Kulesza, Nélio Cacho, Sérgio Soares, Thaís Batista, Roberta Coelho, Mauricio Alférez, Ana Moreira, Afonso Pimentel, João Araujo
1 Software Engineering Laboratory, Opus Research Group, PUC-Rio, Brazil
2 Computer Science Department, Federal University of Bahia, Brazil
3 Informatics Department, Federal University of Rio Grande do Norte, Brazil
4 School of Science and Technology, Federal University of Rio Grande do Norte, Brazil
5 Informatics Center, Federal University of Pernambuco, Brazil
6 CITI, Departamento de Informática, FCT, Universidade Nova de Lisboa, Portugal

Abstract—Product-line stability is essential to promote the longevity of its core assets and products. Even though modularity is a key principle to achieve stable product lines, many of their varying concerns are crosscutting. Aspect-oriented decompositions are aimed at fostering superior modularity of such crosscutting concerns. However, their ability to improve stability of product lines has been rarely tested and, as a consequence, their industrial adoption has been hindered. This paper presents a stability benchmark for aspect-oriented product lines. It allows characterizing, quantifying, and comparing product-line stability promoted by aspect-oriented and conventional variability mechanisms. The benchmark consists of an exemplar’s assets, from where instabilities can be concretely observed from requirements to source code. The usefulness of the benchmark has been assessed through a family of empirical studies.

Keywords—Benchmarks, Software Product Lines, Stability, Empirical Evaluation, Aspect-Oriented Software Development.

I. INTRODUCTION

Stable product-line designs are particularly difficult to achieve as they are often developed in an incremental fashion [1]. Product-line software is stable if, when observed over two or more versions, the differences between their quality measures are insignificant [2][3]. Instabilities can be observed in multiple product-line assets, and some examples are increasing coupling and decreasing cohesion. In general, the higher the scope of instability is (e.g., non-functional requirement or a mandatory feature), the greater its impact. Modularity is a pivotal principle to achieve stability in constantly-evolving product lines [4].

However, there is very limited empirical knowledge about the actual impact of aspect-oriented (AO) techniques on the stability of product line assets [4]. Empirical studies are rare and barely replicated by different research groups. Even worse, preliminary studies [4][5] have either strictly focused on the source code analysis, or on each product line asset in isolation. There is no investigation of how aspectual decompositions affect end-to-end stability of product-line requirements, design and implementation. Such narrow analysis limits the conclusions that can be drawn on the aspect-orientation effectiveness.

This paper reports the methodology derived from the design of the benchmark, which aims at supporting the stability assessment of aspect-oriented (AO) techniques for SPL development. It is expected that the benchmark development would allow academics and practitioners collaboratively assess the stability of AO decompositions throughout a SPL lifecycle. Benchmarking design was initially based on the characterization of a simple, albeit generic, terminology for SPL stability (Section II). Benchmark also consists of multi-phase assets that can be reused across empirical studies of AO product lines (Section III). The stability assessment of AO techniques is supported by (Section III): (i) a SPL exemplar – i.e. the benchmark application, (ii) the multiple releases of core and varying assets for such an exemplar, (iii) a set of metrics for multi-dimensional quantification of stability in requirements, architecture and implementation assets, and (iv) initial data generated through the initial application of AO and non-AO techniques to the benchmark application.

The usefulness of the benchmark was assessed in the context of diverse empirical studies, carried out by different authors. Our benchmark (Section III) offers an effective contribution to the SPL community. First, it provides a common terminology for quantifying stability of product lines and thus it defines a universal vocabulary for the characterization of a set of stability metrics, which can be applied regardless of evaluated technology and SPL development phase. Thus, this common suite of metrics will facilitate the communication among software engineering practitioners. Second, to better understand the effects of AO techniques throughout the SPL development cycle, a set of feature stability metrics were identified and adapted. Finally, the benchmark became an extensive and evolvable open
resource [6] to empower the progress of AOSD techniques for SPLs. In order to evolve the benchmark, the SPL community could enrich it, for instance, with other multi-release SPLs, and propositions or validations of other relevant stability metrics.

II. SOFTWARE PRODUCT-LINE STABILITY

This section presents the common terminology used to characterize stability of SPL assets. It also provides the basis that supported the generic definition of some benchmark elements, such as the stability metrics.

Basic Terminology for SPLs. The purpose of the benchmark (Section III) is also to support the comparison of non-AO and AO techniques for SPLs across multi-phase artifacts. Therefore, we seek to use terminology that is, as much as possible: (i) agnostic to (AO and non-AO) language and paradigm particularities, and (ii) agnostic to specificities of particular development phases, such as requirements and implementation. Figure 1 presents a partial view of our simplified meta-model, mainly based on [7], for characterizing a SPL. This was created for enabling our lifecycle-wide stability analysis of SPLs.

Element is a generic term used to express each concrete entity pertaining to assets (artifacts) of a SPL. On the other hand, concern is a generic term used to characterize anything a stakeholder wants to realize via concrete SPL elements, including domain-specific features, nonfunctional requirements, or architectural decisions. This explicit distinction between elements of SPL assets and stakeholders’ concerns are important when analyzing stability (Section III.D) as we need to observe how the underlying decomposition technique allows better modularity of features, architectural decisions and so forth. Some of them can be realized in a crosscutting fashion, i.e. affecting multiple elements of a SPL.

The variability of SPLs is mainly comprised of varying elements, variation points, and variants. It also comprises other elements, such as variability dependency, but we have omitted them for the sake of keeping our discussion focused. A variation point defines what is variable in an SPL. A variant is related to at least one variation point. Concerns can be realized by an arbitrary set of varying or mandatory elements. Each of the SPL elements is comprised of a set of modules. The nature of the modules varies according to target the programming or specification language. For instance, they can be either use cases or viewpoints in requirements descriptions, components in an architecture description language [8], or classes and aspects in programs.

Stability in SPL Artifacts. According to Kelly [2], a software element is stable if ripple effects (i.e. change propagation) are not observed in the presence of modifications, and its modularity properties remain constant throughout the software releases [2]. Therefore, examples of harmful instabilities in a SPL are the decrease of feature modularity, coupling increase between varying and mandatory elements, or changes traversing the implementation of multiple mandatory or varying elements. In the problem space (e.g., requirements modeling), stability is assessed in terms of changes or modularity problems occurring in releases of SPL requirements artifacts. In the solution space, stability is assessed in terms of changes or modularity problems occurring in releases of architecture, design, and implementation artifacts.

III. DEVELOPING THE BENCHMARK

The development of software engineering benchmarks is far from being trivial as there is no methodological framework to guide their construction [8]. According to Sim et al. [10], a benchmark is a test or set of tests used to compare the performance of alternative tools or techniques. As every benchmark, it comprises of three conceptual components [11]: motivating comparison, task sample and performance measure. The motivating comparison in benchmark is to enable the comparison of AO (and non-AO) techniques from the point of view of SPL stability. The comparison can be drawn in the context of a specific software engineering phase or across the entire SPL lifecycle (Section III). Benchmark also acts as a vehicle to facilitate collaborative identification of unknown stability effects of AOSD (Section III.D). The task sample in benchmark consists of a representative sample of maintenance tasks (change scenarios) applied to the artifacts of a SPL. The performance measure is a set of metrics with the purpose of quantifying different dimensions of stability.

A. Methodology

The benchmark methodology was conceived to achieve three important goals of a stability benchmark for SPL techniques, as follows: (i) it must provide results that are representative and indicative of recurring product-line changes, (ii) it must provide results that are comparable across SPL development techniques (if aspect-oriented or not), and (iii) it must provide means to allow SPL stability analysis at different levels of abstraction.

To achieve these goals, the benchmark methodology encompassed the derivation of a number of milestones. The selection of the target benchmark application (Section III.C) was of primary importance as it provided the basis for the following steps. This application was used first to define and select product-line change scenarios (goal 1). To begin with, seven releases of MobileMedia were developed (Section III.C). Meanwhile, several benchmark artifacts were created to enable the comparison of AO and non-AO techniques for

![Figure 1 - SPL MetaModel]
SPLs (goal 2). Such artifacts were generated based on an initial set of comparable AO and non-AO techniques for SPL development (Section III.C). They also consisted of several SPL models for MobileMedia requirements, multi-view architecture design, and implementation (goal 3). Some examples are feature models, use cases, and architectural diagrams. These artifacts were also produced for all the MobileMedia releases so that the benchmark’s metrics suite (Section III.D) could be applied to them. These metrics were defined to yield SPL stability analyses at different levels of abstraction (goal 3).

B. Benchmark Elements

Figure 2 presents these core benchmark elements, as follows: (I) the application which characterizes SPLs comprising a variety of crosscutting and non-crosscutting concerns, (II) some development approaches that are applied to this application to generate artifacts for stability analysis, (III) the artifacts produced in the development and evolution process, (IV) a suite of metrics associated with a variety of internal and external software attributes and (V) a repository of results generated by the metrics. It is important to highlight that our benchmark is not limited to the application of the techniques (II) described in Figure 2. The benchmark user can create many other artifacts using different AO and non-AO approaches.

MobileMedia was selected for several reasons. First, it is a representative application for the mobile devices domain, since it has several variation points related to the heterogeneous mobile platforms and many alternative and optional features. In this context, we can conclude that 14 out of 19 (or 74%) MobileMedia features are associated with variation points (alternative or optional features), since there are only 5 mandatory features. Second, according to the benchmarking framework in [15], MobileMedia encompasses all types of crosscutting concerns. That framework also points out the diversity of both implementation languages used and construction elements of each language used in its development.

Figure 2 presents these core benchmark elements, as follows: (I) the application which characterizes SPLs comprising a variety of crosscutting and non-crosscutting concerns, (II) some development approaches that are applied to this application to generate artifacts for stability analysis, (III) the artifacts produced in the development and evolution process, (IV) a suite of metrics associated with a variety of internal and external software attributes and (V) a repository of results generated by the metrics. It is important to highlight that our benchmark is not limited to the application of the techniques (II) described in Figure 2. The benchmark user can create many other artifacts using different AO and non-AO approaches.

![Figure 2. Benchmark Elements](image)

C. The Target Application

The benchmark application selected was MobileMedia, a SPL that manipulates photos, music, and videos on mobile devices. It was developed on top of MobilePhoto [11] and it supports different feature combinations for different devices, depending on its capabilities. The base application is responsible for selecting photos’ images from a list and viewing them on the device screen. MobileMedia evolves through additional variants (i.e., mandatory and optional features), which comprise functionalities that might only be supported by certain classes of devices. The components of the base application are implemented using the lowest common denominator of JME to ensure they can run on any JME compliant device. There are Java, AspectJ [12], CaesarJ [13] and Compose* [14] implementations for the target application (Section III.C).

Figure 3 presents a simplified view of the MobileMedia feature model. The alternative features are just the types of media supported: photo, music, and/or video. Examples of core features are: create/delete media, label media, and view/play media. Additionally, some optional features are: transfer photo via SMS, count and sort media, copy media and set favorites. The core features of MobileMedia are applicable to all the mobile devices that are JME enabled.

![Figure 3. Simplified feature model of MobileMedia](image)

In order to expose AO and non-AO techniques to a plethora of SPL change scenarios, a number of feature dependencies (2nd part of Table II) were changed, removed, or added in MobileMedia releases. We have used concern traces [16] to analyze those dependencies. Concern traces record how concerns at one higher level of abstraction manifest in more detailed models in a lower level of abstraction (e.g., how features in feature models manifested in architectural models). Based on such traces, we analyzed [15] two broad categories of dependencies between SPL features using the MobileMedia: feature overlap and intertwine. An overlap dependency exists when two or more concerns share the same elements of a
SPL element. The intertwine (or interlace) dependency, however, occurs when two or more concerns are realized by disjoint elements of the same component. Table II shows the number of intertwine and overlap occurrences in MobileMedia releases 4 to 7 and also classifies MobileMedia features through all existent releases.

From Table II it is important to note that the presence of feature dependencies, such as interwines and overlaps, helps to reveal potential instabilities in the AO and non-AO decompositions throughout the MobileMedia releases. Also, this evidences that the nature of the involved features (Mandatory (M), Optional (O), Alternative (A)) is related with the increment of dependencies between them. This is the case of the increment in the number of dependencies between features especially in R6-R7 that coincides with the continuous increment of optional and alternative features (e.g., from 1 in R2 to 9 optional features, and from 0 in R5 to 3 in R7). This situation can be seen, for example, during R6 where the introduction of the Music concern provokes many changes and potential assets’ instabilities. The number of changes and the growing number of dependencies between feature dependencies might appeal to the use of better modularization techniques such as AO techniques as well as ways to assess their efficiency.

**Alternative Decompositions.** Both AO and non-AO designs of the MobileMedia SPL are based on the Model-View-Controller (MVC) pattern. The key difference is that AO decompositions rely on aspects to modularize crosscutting concerns. A crosscutting concern can either affect mandatory or varying elements (or both). A number of AO (and non-AO) techniques were used to specify and implement MobileMedia and some examples were given in Figure 2. Due to space constraints, we discuss in more detail our choices governing the initial set of AO implementation techniques applied to the benchmark application.

At the implementation level, three AO programming languages, AspectJ [12], CaesarJ [14] and Compose* [14], were used for all seven releases. The use of these three heterogeneous languages allows researchers to draw broader conclusions in the analysis of variability mechanisms available in the family of AO languages. First, we chose AspectJ because it is the most consolidated AOP language and it has heavily influenced all industry-strength AOP frameworks, such as Spring and JBoss AOP. However, considering certain liabilities of conventional AspectJ-based models in SPL evolution scenarios, which were discovered in our initial benchmark-based assessments [4], we decided to evolve our benchmark. Therefore, MobileMedia was afterwards implemented in CaesarJ and Compose*.

CaesarJ was considered in our study because it is a hybrid language that supports both AOP and Feature-Oriented Programming (FOP) [15] mechanisms. Besides evaluating the claimed advantages of FOP for extensibility and reuse of SPL features, we also wanted to evaluate if FOP leads to a better modularization than AOP in general. Finally, Compose* was considered because it has emerged with the goal of enhancing conventional object- and aspect-oriented programming models. It offers a new flavor of software decomposition through the notion of composition filters [15]. It has been widely used in different application domains [17][18][19][20], but it has rarely been systematically assessed in the context of SPL development. It is important to highlight that the Java, AspectJ, CaesarJ and Compose* implementations were aligned to ensure a consistent use of code style, functionality and semantics. The quantitative assessment of the design and implementation was based on the application of metrics suites to all approaches’ versions of the target system (Section III.D).

**TABLE I. SUMMARY OF SCENARIOS IN MOBILEMEDIA**

<table>
<thead>
<tr>
<th>R#</th>
<th>Description</th>
<th>Type of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Better robustness support. In the AspectJ version, exception handling was implemented according to [21]</td>
<td>Inclusion of a non-functional requirement</td>
</tr>
<tr>
<td>R2</td>
<td>Better usability. New feature added to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature added to edit the photo’s label</td>
<td>Improvement of non-functional requirement; Inclusion of optional and mandatory features</td>
</tr>
<tr>
<td>R3</td>
<td>Better usability. New feature added to allow users to specify and view their favorite photos</td>
<td>Improvement of non-functional requirement; Optional feature</td>
</tr>
<tr>
<td>R4</td>
<td>New feature added to allow users to keep multiple copies of photos in different albums</td>
<td>Inclusion of optional feature</td>
</tr>
<tr>
<td>R5</td>
<td>New feature added to send and receive photos via SMS</td>
<td>Inclusion of optional feature</td>
</tr>
<tr>
<td>R6</td>
<td>New feature added to store, play, and organize music. The management of photo (e.g., create, delete and labeling) was turned into an alternative feature. All alternative features (e.g., sorting, favorites, and copy) were also provided for music.</td>
<td>Architectural decision changed; Changing one mandatory feature into two alternatives</td>
</tr>
<tr>
<td>R7</td>
<td>New feature added to manage videos.</td>
<td>Inclusion of alternative feature</td>
</tr>
</tbody>
</table>

**TABLE II. SUMMARY OF MOBILEMEDIA FEATURES**

<table>
<thead>
<tr>
<th>R#</th>
<th>MobileMedia Type of Features</th>
<th>Feature Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>R1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>R2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>R4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>R5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>R6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>R7</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

From the implementation perspective, one of the key stability goals is to ensure that the SPL variation points and variants can be easily added or removed from the base application. In other words, these changes should not cause ripple effects (Section II) across the SPL assets. The OO approach allows this only if the base application is not dependent on the optional code for compilation and execution. For this, the employed implementation strategy relies on inheritance, the chain of responsibility pattern [22] and a Java preprocessor to accomplish the de-coupling of modules. To inform the preprocessor to include relevant
lines of code, conditional compilation (if and statements) is required.

In AspectJ, to enable the operations of addition and removal of optional features we had to ensure one-way communication between optional and base modules. That is, we had to create rules defining that optional modules can call public methods of any class in the base application, but not the other way around. The modules that compose a feature are a set of standard Java classes plus one or more AspectJ aspects. In CaesarJ, the use of collaborating virtual classes enables modularization of different features and, therefore, promotes a better reuse of the application components. CaesarJ provides a feature-oriented development by treating inner classes as virtual ones with linear mixin layers. Thus, including a new feature into a SPL instance is done by means of mixin composition [13].

Compose* provides support to composition filters model, which is a modular extension to the conventional object-based model and thus it can be applied to object-based systems, where objects can send messages among them [20]. Compose* supports the modular definition of features which are generally spread throughout the system and tangled with core features. While AspectJ separates them using pointcuts, advice or inter-type declarations, Compose* uses the concept of composition filters and its mechanism of superimposition to management variability [25]. Filters are superimposed on all program elements affected by the feature implementation. The filter definitions of a filter module form an advice. The pointcuts are determined by the selectors and message matching in the filter definitions.

D. Stability Metrics

According to benchmark goals (Section III.A), a set of metrics should be applied in order to support stability analysis of the AO approaches adopted. The metrics presented here have the goal of measuring the three main dimensions of the stability definition (Section II): (i) stability of features (and concerns in general), (ii) stability of modules, and (iii) change propagation (in order to detect rippled effects). Table III summarizes the purpose of each metrics suite. These suites were defined in terms of the SPL meta-model, which was briefly described in Section II. Therefore, these metrics can be simultaneously applied to SPL assets, such as requirements models and source code, and the relative advantages and drawbacks of the AO technology can be discussed in a broader context. For each of the stability dimensions being considered, we have tried to maximize the selection of existing metrics validated in previous empirical studies (e.g. [4]). Even though existing stability metrics are general-purpose, we had to specialize and adapt some of them to enable the analyses of stability dimensions according to particularities of SPLs and AO techniques.

Quantifying Stability of Features and Modules. The first group of metrics allows SPL developers to quantify feature-specific properties and their stability across multiple releases. For instance, it supports quantifying the degree of tangling and scattering of a feature in a particular SPL release, across multiple SPL releases, or in specific products.

The modularity metrics, which characterize the second metrics suite, were used to enable us to analyze to what extent a certain modularity principle remained constant through the SPL evolution. For instance, the coupling metrics were used to support increase of dependencies between core and varying modules. The details about these metrics can be found elsewhere [4].

<table>
<thead>
<tr>
<th>Stability characteristics</th>
<th>Purpose of each metric suite</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Feature Stability</strong></td>
<td>Measure the feature stability in terms of how the scattering and tangling of features increase or decrease across SPL releases</td>
</tr>
<tr>
<td><strong>Module Stability</strong></td>
<td>Measure the variation of modularity properties, such as cohesion and coupling, across SPL releases</td>
</tr>
<tr>
<td><strong>Change Propagation</strong></td>
<td>Measure the number of changes in varying and mandatory elements</td>
</tr>
</tbody>
</table>

Figure 4 helps to illustrate how a feature tangling and scattering metrics were used to compute the instability of a feature implementation. The figure shows a slice of Mobile Media’s class code, which partially realizes the Persistence feature. As illustrated, the number of transition points, between the persistence-specific code and non-persistence code, is six. This means that the tangling of the feature Persistence is six. The scattering of this feature is (at least) 2, as two methods (lines 06-08 and 10-13) are contributing for what is being evaluated; other classes (not shown in the figure) can obviously contribute to persistence scattering in case they also realize this feature. An increase of feature scattering, for instance, is a sign that feature implementation is unstable and, therefore: either its implementation should be revisited, or the (non-)AO technique under analysis is not able to modularize this specific mandatory or varying feature.

![Figure 4. Feature shadowing of the MobileMedia’s AlbumData class](image)

Quantifying Ripple Effects. Finally, the third group of metrics encompasses a suite of typical change impact measures [2] such as number of components (classes and aspects) added or changed, number of added or modified lines of code, and so forth. The purpose of using these metrics is to detect propagation effects, when introducing or changing a specific feature, in terms of different module granularities: components, operations, and lines of code. The lower the change impact measures the more stable and resilient the design is to a certain change. Instabilities vary
from mild to moderate and very harmful. A very harmful instability occur when a SPL evolution requires modifications across multiple core, variation points, and variants.

IV. FINAL CONSIDERATIONS

One of the original goals for creating our stability benchmark was to allow rapid development of studies for stability of SPLs. Considering the wide range of work that has taken place using the benchmark since its establishment (e.g. [4][16][23][24][25][26]), we believe that this goal has been achieved. Additionally, another direction we believe could be improved is the use of experience reports. Empirical studies were performed to assess if AO techniques address SPL modularity needs. Since these studies are rarely replicated and independent validation is almost non-existent, benchmarks are required to be reused across the scientific and industry community.

By applying benchmark, we enabled the execution of several empirical studies. The execution of such initial studies also allowed researchers: (i) to characterize advantages and liabilities of existing AO techniques for variability management, and (ii) to propose new hybrid techniques (such as COSMOS-AO [26] – an aspect-oriented, component-based model) for fostering the creation of more stable SPLs. By performing evaluations on many approaches provided by the benchmarking, it was possible to demonstrate how well it can be used to measure stability in SPL. Thus, it is possible to indicate the most appropriate adoption of approaches to be used. Finally, the benchmarking enabled us to provide an early feedback on new technologies.

REFERENCES