Stability of Product Lines with Composition Filters: An Exploratory Study

Francisco Dantas¹,², Alessandro Garcia¹
¹Opus Research Group - Software Engineering Lab, Informatics Department, PUC-Rio, Brazil
²State University of Rio Grande do Norte – UERN, Brazil
{fneto,afgarcia}@inf.puc-rio.br

ABSTRACT
With system development becoming increasingly incremental, design stability stands out as one of the most desirable software quality attributes. Development of stable software systems is particularly challenging in the domain of software product lines (SPLs), where their core architecture, set of features, and multiple products are continuously evolving. Contemporary programming languages, such as Compose*, are promising solutions to support enhanced SPL stability through the notion of composition filters. Aspect-oriented and feature-oriented programming models have often been used and assessed in the context of SPL development. However, there is little empirical knowledge on which situations such composition filters mechanisms, in fact, can be used to build SPLs with superior stability. This paper presents an exploratory study that compares the stability of Compose* and AspectJ implementations through multiple releases of two product lines. These implementations are evaluated by means of independently-validated stability metrics. Our results show that the use of Compose* to implement non-functional features and interacting code-sharing features often foster more stable SPL designs than AspectJ ones.

Keywords
Product lines, software decomposition mechanisms, software stability, metrics.

1. INTRODUCTION
Software stability is one of the most desirable quality attributes and probably the most important criterion for determining the longevity of a software project. A software design is stable [2] if its modularity properties are constant, and ripple effects [2, 3] do not manifest in the presence of changes. However, development of stable aspect-oriented (AO) SPLs is particularly challenging [13] as the modules of their core architecture and multiple products are continuously evolving. In addition, harmful changes in software are directly related to the notion of design instability. Thus, the stability of both the core and varying features of SPLs need to be prolonged in order to avoid that their architecture succumbs to typical day-to-day modifications [4].

Traditional Aspect-Oriented Programming (AOP) models, as fostered by AspectJ, have shown certain liabilities in SPL evolution scenarios [13]. Following this statement and based on the difficulty of providing stable code with conventional decomposition mechanisms [7, 8], many approaches such as Compose* [6] have emerged in order to enhance conventional object- and aspect-oriented programming models. Compose* offers a new flavour of software decomposition through the notion of composition filters. It has been widely used in different application domains [9, 10], but it has rarely been systematically assessed in the context of SPL development.

Corroborating our research interest, there is not much knowledge about the impact of composition filters on the stability of software product lines. Even though there are some recent studies on the stability assessment of aspect-oriented programs, they mostly focus on conventional programming models, such as AspectJ and their derivatives [13, 14, 15]. Sven Apel et. al [16, 17] proposed hybrid programming models that combine AOP and Feature-Oriented Programming (FOP) in different ways. Even though they performed some initial assessments of such models, they did not embrace composition filters in their comparisons with AspectJ. More importantly, they have not focused on design stability assessment.

This paper presents an exploratory (non-controlled) study that compares the stability of Compose* and AspectJ implementations through multiple releases of two product lines. In addition to the aforementioned motivation, Compose* was chosen for three reasons: (i) it is significantly different from the AspectJ model; Compose* is designed to improve the composability of object-based programs, (ii) its compiler proved to be robust enough during our pilot assessments, and (iii) there are public reports of their successful adoption in industrial software development projects [6, 18]. Our comparative analysis involves two SPLs, namely MobileMedia and iBatis (Section 2). Our findings are based on previously-defined stability metrics [2] in order to help us to quantitatively test some relevant hypotheses (Section 3). We present how the mechanisms of Compose* we used to implement releases of the target SPLs, derived from exiting AspectJ versions (Section 4). The discussion about the stability analysis is provided in Section 5. The results that we found are compared to previous studies (Section 6). Finally, the threats to validity and our final considerations are presented in Section 7 and 8, respectively.

2. TARGET PRODUCT LINES AND COMPOSITION FILTERS
This section briefly presents the SPLs used in this study (Section 2.1) and describes the notion of composition filters (Section 2.2).
### 2.1 Target Product Lines

In order to promote a systematic evaluation of Compose*, the selected cases were: (i) a large open-source software, called iBatis (110 KLOC), and (ii) an embedded mobile software, called MobileMedia (5 KLOC). They are from significantly different application domains, and they have been evolved over a long period of time. Also, they are interesting because they contain different categories of crosscutting features, such as domain-specific features and non-functional features. Their multiple releases and respective documentations were also available for analyses. In addition, they underwent various forms of changes (Section 4). These characteristics completely match the purpose of Compose*, which has a high power of reuse using composition filters – essential to minimize the changes of existing modules during incremental SPL development. We hope that the assessment of more than one SPL, from significantly-different domains, provides us with a fair comparison of Compose* and AspectJ. Besides, Java and AspectJ well-designed solutions of both SPLs were already available, facilitating the analysis of the new decomposition mechanisms analyzed in this study.

**MobileMedia.** It is a software product line (SPL) that provides support to manage (create, delete, visualize, play, send) different medias (photo, music and video) on mobile devices. During the SPL development and evolution, the initial core architecture was systematically enriched with mandatory, optional and alternative features. The core features are: create/delete media (photo, music or video), label media, and view/play media. Some optional features, amongst others, are: transfer photo via SMS, count and sort media, copy media and set favorites. Seven releases of the MobileMedia were analyzed.

**iBatis.** It is a Java-based open source framework for data mapping and it uses two main APIs: (i) SQL Maps for reducing JDBC code; and (ii) Data Access Objects (DAO) for abstracting the persistence implementation details. It is composed by more than 60 releases incrementally developed and its development is characterized as a reactive approach. Initially, four releases were chosen and implemented using the AspectJ language in its essence. The features that were aspectized were: concurrency, type mapping, design patterns, error context, exception handling, connection, session, and transaction. First of all, we analyzed if the iBatis aspectization process followed good implementation practices. We decided afterwards to refactor, using Compose*, the following features of iBatis: type mapping and error context. Besides, other non-functional features were also modularized, such as design patterns. We chose these features because they undergone more frequent and heterogeneous changes through the iBatis history.

### 2.2 Composition Filters Model

Composition Filters Model (CFM) 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. The idea is to filter these messages through a set of filters. Figure 1 gives us an overview of the CFM. Basically, filters are grouped (superimposed filtermodules) as illustrated in Figure 2. Each filter type has a filter type (e.g. Dispatch, Meta, After, Before and Error), which defines the behavior that should be executed when the message is accepted or rejected. Each filter should be applied (superimposed) to one or more objects using superimposition selectors. These selectors select a set of classes using a Prolog-based selector language.

![CFM Overview](image_url)

The CFM is implemented by Compose* tool, which can be defined as a language-independent aspect compiler for composition filters. According to [6], Compose* has been developed with the goal of providing a framework to experiment with new language concepts and features and also providing the ability for researchers and practitioners to apply the composition filters language. In order to illustrate how Compose* works, a simple example is shown in Figure 2. In this figure, the concept of concern is illustrated and discussed in the following.

A concern defines the filter modules (lines 2 to 11) and the superimposition block (lines 12 to 17). The filter module of our example contains an internal declaration and two filters definition named \texttt{sub2} (line 06) and \texttt{sub3} (line 09). An internal declaration creates a composition relation between the object on which this filter module is superimposed and an instance of the type declared in the internal. The internal can be used in the filter definitions as a destination for the message. The filter definition on lines 9 to 11 creates a dispatch filter that forwards a message to a new destination. In this case, messages matching \texttt{BaseController.handleCommand} and \texttt{ImageUtil.createImageData} are forwarded to the internal \texttt{sort} (line 04). In others words, the dispatch filter will forward all messages sent to the object to the \texttt{SortingFeature} instance, when the message matches a signature in the class \texttt{SortingFeature}.

A message consists of a target and selector: target.selector. The target is the object that receives the message and the selector is the called method. The second filter in this filter module (line 09) defines an after filter. An after filter sends a new message after a given message has returned. In this case, the message \texttt{sort.showImageUtil} will be dispatched when a call to the method \texttt{BaseController.showImage} have returned. The superimposition block (lines 12 to 18) determines which filter modules will be superimposed to selected classes. The selector definition (line 14) selects a collection of program elements; in this example, it selects all classes with the name \texttt{BaseController} and \texttt{ImageUtil}. The filter module \texttt{Sorting} is superimposed on all program elements selected by the selector sorts on line 16. The filter definitions of a filter module form an advice. The pointcuts are determined by the selectors and message matching in the filter definitions.
The \textit{concern} definition in this example creates a composition between the classes `BaseController` and `ImageUtil` and the class `SortingFeature` (line 94). The `SortingFeature` class implements the logic for a Sorting type, it manages the inclusion of a new feature of sorting photos in MobileMedia. Each `BaseController` and `ImageUtil` is associated with a `SortingFeature` instance. The `BaseController` and `ImageUtil` classes are extended to contain the methods defined in the filters. The execution of these methods is delegated to the `SortingFeature` class. Thus the interface of the class `SortingFeature` has been extended in a way similar to AspectJ’s inter-type declarations.

3. \textbf{HYPOTHESES AND METRICS}

The objective of our study is to understand if composition filters provide better stability than conventional AOP models for evolving software product lines. Our research was motivated by the fact that: (i) stability is a frequent goal through the development of software systems that are required to have long life, such as product lines, and (ii) composition filters are increasingly attracting attention to develop real software systems [19-23]. We decided to develop an exploratory study rather than a controlled one, because the use of CFM is not as popular as AspectJ yet. Our hypotheses were based on some initial evidences, which will be useful to controlled assessments in the future.

\textbf{Hypotheses}. Even though exploratory studies do not require explicit hypotheses definition, we decided to make our working hypotheses clear in order to expose our research questions in a way that they are easier to be understood. We have defined some hypotheses based on common claims found in the literature [2-4]. Table 1 presents our null and alternative hypotheses. The null hypotheses (H0) states that Compose* and AspectJ produce equally stable code. The hypotheses H1 argues that the decomposition mechanisms supported by Compose* are more appropriate than AspectJ ones to promote stable SPLs. The discussion on the hypotheses test is presented in Section 5.2.

\textbf{Table 1: Hypotheses}

<table>
<thead>
<tr>
<th>Hypid</th>
<th>Hypothesis Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>The use of Compose* and AspectJ decomposition mechanisms produces equally stable code of an SPL.</td>
</tr>
<tr>
<td>H1</td>
<td>The use of Compose* decomposition mechanisms produces SPL modules that are more stable than AspectJ ones in a SPL context.</td>
</tr>
</tbody>
</table>

\textbf{Multi-Dimensional Stability Metrics}. In order to support the test of our hypotheses (Table 1), some previously-validated stability metrics were used. The stability analysis is supported by a set of well-known metrics that quantify relevant and complementary stability dimensions. Two groups of metrics were used: (i) change propagation metrics, and (ii) modularity metrics. These metrics were applied to multiple SPL releases with the intention of respectively computing the (undesirable) SPL changes and the constancy of pivotal modularity properties through those releases. In other words, the latter was used to quantify the stability of the modularity properties in a system, such as separation of concerns, cohesion, coupling, and code conciseness metrics.

Change propagation metrics were used with the purpose of measuring the degree of modifications on code implementation. For instance, they enable us to check if a change, originally targeted on adding a new feature, also affected the core modules of a SPL. The used metrics were defined to quantify two complementary forms of modification: (i) refactorings – when the change is aimed at improving the system structure while preserving the existing code semantics, and (ii) alterations – when functionalities are added, removed, or modified through the system modules. For the first case, we used a metric, called Refactoring of Modules (RoM). This metric is used to quantify structural changes in classes, aspects, filters, and/or in their respective internal elements. For the second case, a metric, named Alterations in Code Elements (ACE), was used to compute the number of increments, deletions, and actual modifications in code elements. Examples of these elements can be a class, a method, an aspect, an advice, a pointcut, or a filter.

The modularity metrics were used to enable us to analyze to what extent a certain modularity principle remained constant through the SPL evolution. For instance, the concern metrics were used to support quantitative analyses of feature’s properties across multiple SPL releases. They enabled to capture whether the localization degree of each feature was stable or not across the modular decomposition of the SPL. The following concern metrics were used in this study: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), and Concern Diffusion over Lines of Code (CDLOC). As we are focused here on using these metrics to quantify feature locality, they are from herein called feature metrics. The attributes cohesion, coupling, and code size were measured with the following metrics: Coupling between Modules (CBC), Lack of Cohesion in Operations (LCO), and Lines of Code (LOC).

We chose all these stability metrics, presented above, for several other reasons. First, they had already been used and considered relevant in many other stability studies [13, 26]. Second, these metrics capture a number of internal program attributes, which have been empirically found to exert a direct impact on stability in software artifacts. Third, these internal attributes are more neutral and not specific to certain decomposition mechanisms (e.g. DIT – depth of inheritance – is a metric that should be avoided because it measures coupling in specific context of inheritance usage), thereby allowing us to achieve a more equitable comparison of AspectJ and Compose*. Finally, they can be collected in an automatic and reliable manner [25].

4. \textbf{EVOLVING SPLs}

This section discusses the evolution process of the target SPLs with composition filters (Section 4.1), and presents how variabilities were realized with Compose* (Section 4.2).

4.1 \textbf{Evolution Process}

We aim at evaluating whether and when the use of Compose* promotes a superior stability of aspect-oriented SPLs. Our evaluation was focused on the degree of existing modifications between the SPL releases. However, even considering our previous knowledge about the original AspectJ versions [13], we developed all the Compose* implementations without taking the future SPLs changes into consideration. In addition, we tried to maximize the stability of the modules following some Compose*-
based design guidelines [6, 21] that specifies how to implement a filter reuse strategy and promote more concise Compose* code.

Our second step was to review the entire modifications along the evolution of the AspectJ and Compose* versions. The goal was to identify and eliminate possible remaining anomalies, such as code replication. In other words, we intended to maximize the modularity of both AspectJ and Compose* and also enable a fair comparison.

4.2 Variability Management

Two advanced modularity techniques were used to implement variabilities of SPLs used in this study. Their power to manage variability was analyzed under multiple stability perspectives (Section 3). The strategy for implementing variability used in the AspectJ’s releases were the same used in a previous study where best modularity practices have already being applied and systematically assessed [13, 26]. As far as the Compose* versions are concerned, the variability implementation strategy is discussed in terms of examples in the following.

In order to add the optional FAVORITE feature to the MobileMedia, a filter is defined. Figure 3 illustrates this filter. The new methods defined in the Java class FavoriteFeature (line 05) are superimposed to ImageData. The superimposition is carried out through the selector function named isClassWithName (line 12). In other words, the methods dispatched by the filter fav will be part of the class ImageData as usually implemented with AspectJ’s inter-type declarations.

Compose* does not cope with the introduction of mandatory features in this study, since it is not targeted at modularizing them. However, this does not happen with alternative features, such as MUSIC and VIDEO. Considering that these features reuse some features such as SORTING, the superimposition mechanism provided by Compose* is used in order to reuse methods already defined in features like SORTING.

![Filter's code](image)

Figure 2. Compose* composition for MobileMedia

![Figure 3. Variability with Compose* Mechanisms](image)

Figure 3. Variability with Compose* Mechanisms

Figure 4 illustrates a scenario where the method setNumberOfViews is reused through the use of filters. The internals declaration sort (line 04) creates a composition relation between MultiMediaData (line 15) and SortingFeature (line 04) in order to execute the method setNumberOfViews after a message received from the object MultiMediaData. In AspectJ, it was defined a new aspect defined in order to provide the same functionality provided by the filter (Figure 4).
5. STABILITY ANALYSIS

This section discusses the stability of SPLs from complementary points of view: change density (section 5.1), concern metrics (section 5.2), and modularity metrics (section 5.3).

5.1 Changes on Code Elements

The more changes are required to realize a new SPL evolution scenario, the more unstable the SPL design is likely to become. However, certain categories of changes tend to be more problematic than others. For instance, changes to interfaces or abstract modules tend to cause more harmful ripple effects because many modules depend on them. Therefore, we categorized the nature of changes that occurred more frequently in both Compose* and AspectJ implementations. Table 1 summarizes the change classification observed in both MobileMedia and iBatis. The measures represent the tally of modifications made across all the releases.

Table 1. Changes in SPLs

<table>
<thead>
<tr>
<th>Change Type</th>
<th>MobileMedia</th>
<th>iBatis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AspectJ</td>
<td>Compose*</td>
</tr>
<tr>
<td>ACE</td>
<td>424</td>
<td>312</td>
</tr>
<tr>
<td>RoM</td>
<td>278</td>
<td>249</td>
</tr>
<tr>
<td>Total</td>
<td>702</td>
<td>561</td>
</tr>
</tbody>
</table>

Taking the Table 1 into consideration, we can observe that the amount of changes was lower in the Compose* implementations when compared to AspectJ ones. This information was important to calculate the improved reuse percentage of code elements in MobileMedia and iBatis. Even though this is a simplified way of quantifying reuse, it enables us to have an initial understanding of the effort required to accommodate changes while reusing existing AspectJ and Compose* modules in a system. The percentage was computed by using the change measures of the AspectJ implementation as the reference point. The percentage values are a result of the total of changed elements in Compose* versions divided by the total of changed elements in AspectJ. We observed an improved code reuse of 21% and 44% for the Compose* considering the MobileMedia and iBatis, respectively. Regarding to the modules abstract and interfaces, AspectJ showed a greater number of changes compared with Compose*.

This superiority of Compose* takes place mainly due to the tight coupling between a feature and the base program existing in AspectJ implementations. This is extremely maleficient, for instance, in situations where refactoring of names of classes happens. This kind of refactoring entails changes in both implementations. However, the mechanism of \textit{intertype declarations} used by AspectJ requires more changes than superimposition mechanisms. Figure 6 shows an example of this situation. This is an example of \textit{intertype declaration} in AspectJ where the class \texttt{ImageData} was refactored to \texttt{MediaData} in order to support the manipulation of different kind of medias (photo, audio and video). Thus, all the occurrence of \texttt{ImageData} was replaced to \texttt{MediaData} as illustrated in Figure 6 (line 8). Considering that there were many code refactoring like this, the RoM measure tend to be bigger in AspectJ implementations.

In Compose* implementation, just one modification in the filter was required in order to support the same refactoring operation (Figure 7 –line 13). In this case, the class \texttt{ImageData} (Figure 3 – line 13) was refactored to \texttt{MediaData} (RoM Changes). Thus, the methods previous superimposed to \texttt{ImageData} are now part of the class \texttt{MediaData}. The insertion of new methods, attributes, pointcuts and filters are examples of ACE changes. Figure 5 illustrates this kind of situation where a new aspect was created (AspectJ) to provide part of MUSIC feature functionality. To provide the same functionality in Compose* just the filter \texttt{music1} was add to the concern \texttt{music} as illustrated in Figure 4 (lines 08-10).

5.2 Feature-Level Stability

Based on the AspectJ implementations, three distinguished groups of results emerged from the analysis of feature measures: (1) functional features with no shared code with others, (2) features with shared code with others, and (3) non-functional features.
Group 1: Features with no shared code with others. This group encompasses all the features that do not share any piece of code with other features. Belonging to this group the following features: SORTING, FAVORITE and EXCEPTION HANDLING. As no code is shared between two or more features, there is no code reuse and thus the values found are the same for all metrics. We observed that the effectiveness of AO mechanisms to localize this kind of feature is due to the ability to transfer the code in charge of realizing the optional feature from classes to a set of dedicated classes (Compose*) and one or more glue aspects (AspectJ). Therefore, given the independent nature of features, one aspect in AspectJ match with one extra class in Compose*. This class is responsible for implementing the functionality that is separated of the base code and superimposed by the filters. As a consequence, the key mechanism of reuse (superimposition) supported by Compose* does not present any advantage for features with no shared code. This is attested by the similar degree of: (i) feature scattering across modules (CDC) and operations (CDO), and (ii) feature tangling across lines of code (CDLOC). Both approaches present the same values for all the feature metrics. In addition, they present similar level of stability, i.e. there is almost no variation of feature modularity across all the 7 MobileMedia releases and all the 4 Ibatis releases.

Group 2: Features with shared code with others. Features that shared code with others, such as SMS, MUSIC and VIDEO, belong to this group. The MUSIC feature is a representative one for this group because it has code shared with many other features. In general, the aspectisation process of this kind of sharing consists of creating a separate aspect to handle this common code. As a consequence, the number of components implementing those features (CDC) is higher in AspectJ versions because each set of common code must be modularised in a separated aspect (Figure 8 (a)). Besides this happened mainly due to difficulty of using AspectJ for addressing different SPL configurations (specific combination of features). AspectJ’s solution required the coding of different aspects representing different combinations of features, such as, SortingAndMusic aspect (Figure 5).

In Compose*, even considering the filters as separate modules, less components are required to implement most of the features. For instance, the aspect defined in Figure 5 is not a separate module in Compose* implementation. Instead of defining a new component, the method setNumberOfViews (line 10), which is being used by the feature FAVORITES, is superimposed in MultiMediaData class (Figure 4). Thus, as the CDO metric counts the number of methods and advices whose main purpose is the implementation of a feature and the number of other methods those access them. Thus, the methods that are reused from other features (they were defined with a different purpose) are not considered and CDO exhibit less values (Figure 8 (b)). The degree of feature tangling (quantified by the CDLOC metric) is less negatively affected on Compose* solutions since changes tend to be more localised in the filters due to the superimposition of methods defined in the other features. Besides, the existence of more modules (aspects) in AspectJ versions entails more number of transition points (tangling) for each feature through the lines of code. Figure 8 (c) confirms this finding through the values for the CDLOC measure.

Group 3: Non-functional features. Features such as ERRORCONTEXT, DESIGNPATTERNS and TYPE_MAPPING belongs to this group. They are named non-functional because they directly contribute to achieve non-functional requirements of the target SPLs, such as reliability and maintainability. The Compose* implementation of this kind of feature tend to be implemented with higher degree of composability and reuse: methods defined in the first release are directly reused on the other releases through the application of superimposition mechanisms. Thus, the number of operations is reduced. This justifies the decrease of scattering (CDO) and tangling (CDLOC) measures. However, the value of CDC for the first release of iBatis is higher in one unit because the number of aspects in AspectJ match with the number of extra class (Compose*) and the Compose* implementation has an additional filter defined. For the other releases, the number of new components is bigger than the number of extra classes plus filters. Figure 9 illustrates the measures for the feature ERRORCONTEXT.

5.3 Modularity Evaluation

The data collected for the coupling, cohesion and size metrics have mostly favoured the Compose* implementations as well. In fact, it is true that AspectJ decomposition mechanisms show improvements in modularity, despite some shortcomings in expressiveness. Some of the shortcomings are caused by tight coupling between a feature and the base program. An example is defining a crosscutting concern by enumerating the join points by name or according to certain naming conventions. In Compose*, the values of CBC are minor because less components are defined and thus the coupling occurs among a less number of modules as illustrated in Figures 4 and 5. Considering that we have fewer aspects and thus fewer pieces of advice, the value related to LCO is reduced once it measures the amount of advice/method pairs that do not work on same module. Finally, LOC is reduced in Compose* implementations considering that we have fewer components defined and concise files defining the filters. Figures 10 illustrates the measures of all releases for iBatis.
Finally, based on the measures we conclude that Compose* presents more stable code than AspectJ. Thus H0 was Rejected and H1 Accepted.

6. RELATED WORK

Bartory et. al. [4] arises the discussion about the use of aspects or features in software development process. However they do not assess stability and they do not focus on different AO approaches. Roo et. al [6] proposed the Compose language * (Section 2.3). By means of this language, the concept of aspect-oriented programming was included through the composition filters model. Others studies assessing the composition filter model were carried out. However, none of them [19-23] focused on the evaluation of SPL stability.

Mohagheghi et. al [27] described the results of an empirical study, where some hypotheses about the impact of reuse on defect-density and stability, and about the impact of component size on defects and defect-density in the context of reuse are assessed. They used historical data on defects, modification rate, and software size of a large-scale telecom system developed by Ericsson. This system was partly implemented with an object-oriented programming language though. However, different from their study, our findings focus on the SPL stability evaluation of a new programming model, i.e. Compose* (use reuse percentage as one of the analysis indicators).

Figueiredo et al. [13] present an empirical study focusing on stability assessment of two SPLs. The work analyzes the evolution of SPLs in terms of metrics for modularity, change propagation and feature interaction considering two programming techniques: conditional compilation and AspectJ. Differently from that study, we are focusing on the comparison of AspectJ and Compose*.

7. THREATS TO VALIDITY

Some threats to validity are of relevance to our study and are made explicit in the following.

Conclusion Validity. In the study, these threats are related to the implementation treatments. The versions were implemented by experienced AOP developer with good knowledge on the programming languages involved in the study, namely Java, AspectJ and Compose*. Even though the subject has not developed the AspectJ original versions of the SPLs, he had a partial knowledge of the possible modifications in future releases.

Construct Validity. The threat to construct validity includes the suite of metrics used both for quantifying changes and modularity properties. We used the feature metrics that allowed us to evaluate the modularity properties from each feature point of view. Most metrics were gathered through the AOP metrics tool [34], such as coupling, cohesion, and size. We adopted these metrics because they were all empirically found to have correlation with design stability [2]. In addition, they enabled us to make a more objective comparison with outcomes of relevant previous studies [28,29].

Internal Validity. Threats to internal validity reside on alignment rules used to implement Compose* releases. To reduce this threat, we performed a detailed analysis of the AspectJ code of the SPLs in order to reduce the inconsistencies and not propagating problems from the original AO implementations. It was necessary to ensure the quality of design in all versions and to do the comparison fairer and more equitable. Also, we followed the implementation guidelines of the Compose* [6,21].

External Validity. Threats to external validity are conditions that allow results generalization. In order to minimize this threat, we chose a SPL that had already been used in another empirical study.
[13] and another well-known framework for development of distributed Java-based applications. These applications are representative of real-world projects and have a significant size. Besides, they contain a series of (non-)crosscutting mandatory, optional and alternative features. This way, they enabled us to observe the differences among the results. However, it is still necessary to conduct other evaluations with other SPLs to be able to provide more evidences regarding our conclusions. The techniques were developed using the AspectJ language as the basis.

8. CONCLUDING REMARKS AND FUTURE WORKS

This paper reported a stability comparative assessment of Compose* and AspectJ in the context of SPL development. Our analysis provided evidences that the use of Compose* appears as a promising solution to implement non-functional and code-sharing features. This happens mainly because the use of Compose* supports superimposition, which tend to increase the degree of reuse of methods. Besides, it also provides more concise specification of filters (than the AspectJ code counterpart).

It is also important to put in perspective our findings with respect to the results of a previous study comparing AspectJ and conditional compilation [13]. We can draw some observations: (i) in our study, Compose* presented similar or even better results than AspectJ for all categories where AspectJ (in the previous study) presented better results than conditional compilation, and (ii) Compose* presented better measures than AspectJ and conditional compilation for scattering (CDC) metrics in terms of code-sharing features. This took place because the use of filters allowed creating different instances of the same object through the superimposition mechanisms and thus it avoided the tangling code caused by conditional compilation (Java).

As future work, we believe that this initial work can be further improved in many directions, including (but not limited to): (i) the same investigation can be carried out, but also including a hybrid programming language, such as Caesar!, that supports both AOP and FOP mechanisms, (ii) it is important to evaluate the trade-offs involving stability and other equally quality attributes such as feature reuse and error-proneness and (iii) a higher number of software product lines should be considered in order to confirm (or refute) our results.

9. REFERENCES


