On the Role of Composition Code Properties on Evolving Programs

Francisco Dantas¹, Alessandro Garcia¹ and Jon Whittle²
¹Opus Research Group – Software Engineering Lab, Informatics Department, PUC-Rio, Brazil
²School of Computing and Communications, Lancaster University –Lancaster, United Kingdom
{fneto,afgarcia}@inf.puc-rio.br, whittle@comp.lancs.ac.uk

ABSTRACT
Composition code defines the binding of two or more modules in a program. Post object-oriented programming techniques are increasingly providing expressive mechanisms to enable the flexible definition of composition code. Such composition mechanisms are intended to support programmers in factoring out the complexity of a program and facilitate its evolution. However, the properties of composition code might introduce new flavours of complexity, and in turn cause side effects on program evolvability. Unfortunately, the role of composition code properties in evolving software systems is not yet well understood. This gap is mostly due to the lack of a measurement framework to characterize and quantify composition code. Existing metrics suites are focused on quantifying properties of programs and their modules only. Therefore, programmers are not able to analyse and understand the impact of particular composition properties on program evolvability. This paper presents a framework aimed at understanding the impact of particular composition properties on evolving software systems.

Composition code properties. Existing frameworks are focused on quantifying properties of programs and their modules only. Some researchers have recently proposed metrics for programs structured with advanced composition mechanisms. However, they are only intended to measure the properties of program modules, such as their coupling and cohesion [9][17][18][31]. As a result, there is not even an understanding about basic characteristics comprising composition code. Without this knowledge, it is not possible to define a metrics suite intended to compute composition code properties. It is not possible either to study their impact on program evolvability.

Unfortunately, the effects of composition code structure on evolving programs are not well understood. This misunderstanding is mainly due to a lack of measurement frameworks to quantify composition code properties. Existing frameworks [12][14][15][20] and metrics suite [13] are focused on quantifying properties of programs and their modules only. Some researchers have recently proposed metrics for programs structured with advanced composition mechanisms. However, they are only intended to measure the properties of program modules, such as their coupling and cohesion [9][17][18][31]. As a result, there is not even an understanding about basic characteristics comprising composition code. Without this knowledge, it is not possible to define a metrics suite intended to compute composition code properties. It is not possible either to study their impact on program evolvability.

This paper presents a framework that encompasses basic terminology (Section 2) and a metrics suite (Section 3) for composition code. It is not the intention of our framework to capture all possible properties of composition code; instead, it focuses on three significantly-different dimensions of composition code complexity. We studied the role of the measurement framework to support stability analysis of 22 versions pertaining to 4 software projects (Section 4). The programs were structured with...
two different programming techniques: aspect-oriented programming and feature-oriented programming. These techniques were chosen as they support a wide variety of composition mechanisms. They enable us to understand the impact of composition code on program evolvability in different contexts.

Our study results revealed that certain composition properties are often detrimental to program stability (Section 5). In particular, our investigation has shown that specific composition metrics were consistent indicators of software instabilities along the analysed project histories. For instance, measures of composition scope were the most strongly correlated with program instability in all the cases. They even outperformed coupling measures, classical indicators of software evolvability in object-oriented programs [19][20]. We also compare our work with related work in Section 6. Threats to validity are discussed in Section 7 and finally, conclusions are presented in Section 8.

2. TERMINOLOGY AND COMPOSITION CODE PROPERTIES

The proposed framework consists of basic concepts, presented in this section, and a metrics suite (Section 3) for composition code. First, we present the basic terminology (Section 2.1) to describe key properties of composition code (Section 2.2) in a consistent manner. The examples provided in this section are based on the CaesarJ [21] (FOP) programming language notation. These languages were chosen because they support a variety of composition mechanisms.

2.1 Terminology

We seek to define a terminology that is, as much as possible, language independent and extensible. We have chosen set theory to formalize our terminology because both it has been largely used in other works for the definition of metrics [13][14] and it has an expressive power that allow us to capture the essence of each composition property. A basic set of concepts related to composition code is presented in Figure 1. It defines the relationships between different components of a program, such as module and programs, and the composition properties.

![Figure 1. Composition Code Measurement: Basic Terminology](image)

A Running Example. For illustration purposes, Figure 2 illustrates a program in the CaesarJ language. We selected this example as CaesarJ offers a rich set of composition mechanisms; it also supports both aspect-oriented programming (AOP) and feature-oriented programming (FOP). This program consists of a set of modules: classes (E1 to E5 and E1 to B4), aspects A, A1 to A3, interface (I) and virtual classes (V1 to V6). Each module contains a set of program elements. A program element is a sequence of statements. There exist three types of program elements: attributes, operations and declarations. For instance, methods and advice are classified as operations in CaesarJ; the same applies to AOP-specific languages, such as AspectJ [22]. Pointcut expressions, intertwype declarations and mixin composition expressions are classified as declarations.

A pointcut selects well-defined points (joinpoints) of the program flow that should be extended by pieces of code called advice. Aspect inheritance provides a simple mechanism of pointcut overriding and advice inheritance. To use inheritance between aspects it is required to define an abstract aspect, with one or more abstract pointcuts, and with advice on the pointcut. Pointcuts-advice dynamically affects program flow whereas intertwype declarations operate statically, at compile-time, affecting a program’s modules hierarchy. Intertypedclarations may declare members or change the inheritance relationship between classes. In Figure 2 the aspect A2 introduces the method mB( ) in the module B2. As the aspects A1 and A2 intercept the same point in B2 (method getA ()), the order of the interception needs to be specified. The aspect A3, using a mechanism of declare precedence, defines such an order: A1 has precedence over A2. Besides, there are also several other composition mechanisms to implement FOP concepts such as virtual classes, mixin composition and wrappers. Virtual classes are inner classes of another outer class. They behave like virtual methods and thus can be overridden in a subclass of the outer class. In Figure 2, the class E4 has an inner class V1, which overrides V1 in E1 by inheritance relationship. Thus, in CaesarJ, a module can be created by composing several CaesarJ classes using simple and multiple inheritance mechanisms.

The basic concepts of our framework (Figure 1) are formalized through the definitions 1 to 4 and illustrated using the example presented in Figure 2.

**Definition 1 (Module and Program Element).** A module $M$ is a sequence of program elements, $E_M$. A program element can be an attribute, an operation or any other form of declaration. Let $Att_M$ be the set of attributes of $M$, $Op_M$ be the set of operations of $M$ and $Dec_M$ be the set of declarations of $M$, $E_M := Att_M \cup Op_M \cup Dec_M$.

By means of composition mechanisms, program elements supported by distinct programming languages can be combined, so that they can work together. Taking in consideration the example illustrated in Figure 2, we can highlight the following composition mechanisms: pointcut in A1, intertwype-declaration in A2, declare precedence in A3 and virtual classes in E1, E2, E3, E4 and E5.

**Definition 2 (Composition Mechanism).** Given two languages, $L_1$ and $L_2$, a composition mechanism is a means to combine one or more program elements implemented in either $L_1$ or $L_2$.

A program can be made up of a set of modules from $L_1$ and $L_2$, which are combined by means of composition mechanisms. $L_1$ and $L_2$ are sets of constructs that are either the same or different depending on the programming language at hand. For instance, $L_1$ and $L_2$ are different in most of the AOP languages, such as CaesarJ,

---

1 CaesarJ supports both FOP and AOP composition mechanisms. However, in this evaluation we are taking into consideration only its feature-oriented mechanisms.
AspectJ and its dialects. As far as AspectJ is concerned, $L_i$ is considered to be formed by Java constructs, used to define the classes; whereas $L_j$ comprises the set of constructs to define aspects. In subject-oriented programming languages, such as Hyper/J [23], $L_i$ and $L_j$ are the same language, i.e. Java; the difference from Java resides on the additional set of composition mechanisms supported by Hyper/J. The code implemented by the use of composition mechanisms is called composition code. In Figure 2, the composition code is made up by the modules in the light grey area.

**DEFINITION 3 (PROGRAM AND COMPOSITION CODE).** A program $P$ consists of a set of modules, $M$. There exist, two subsets of modules: $M_p$, the set of modules of $L_i$ and $M_a$ the set of modules of $L_j$ where $M = M_p \cup M_a$. A composition, $Com$, is a set of modules $M_{com} = M_p \cup M_a$, where $M_p \subset M_a$, $M_a \subset M$, $M_p \neq \emptyset$ and $M_a \neq \emptyset$.

Program elements, which belong to the composition code, depend between them. A composition dependency is an ordered pair of program elements which defines either a direct or indirect relationship between the elements.

**DEFINITION 4 (COMPOSITION DEPENDENCY).** A composition dependency, $D_{ij}$, between two program elements $e_i$ and $e_j$ is defined as a 4-tuple $(e_i, e_j, m_p, m_a)$, where $e_i \in E_{com}$, $e_j \in E_{com}$, $D_{ij}$ is considered indirect when there is a relationship between $e_i$ and $e_j$ characterizing a transitive closure of $D_{ij}$. On the other hand, $D_{ij}$ is considered direct when there is direct relationship between $e_i$ and $e_j$.

In Figure 2 the composition dependencies are illustrated by the composition interections. The dependency between the pointcut $p1$ (aspect $A1$) and the method $setX()$ (class $B1$) is an example of direct dependency as this relationship is declared in the implementation of $A1$ (* $*.set*()$). An example of indirect dependency is illustrated between $E2$ and $E3$. The method $updateB1(X())$ in $E2$ updates the value of the attribute $X$ in $B1$. There is no reference to $E3$ in $E2$. However, as $E3$ uses the attribute $X$, there is an indirect dependency between $E2$ and $E3$.

### 2.2 Composition Properties
Composition code entails new dimensions of complexity in a program. The essence of composition code relies on the understanding of its properties. Therefore, programs are built to have certain properties, which may exert an impact on the quality attributes, such as software stability (Section 5). Composition code is characterized by at least three basic properties: *diversity*, *scope*, and *volatility*. The realization of such properties on the source code is discussed using the example provided in Figure 2. In order to make our discussion more concrete, we used CaesarJ as example; i.e. $L_i$ and $L_j$, are respectively instantiated by Java and the set of CaesarJ constructs to support both AOP and FOP.

**Composition Diversity.** Composition code encompasses a significant diversity of modules. The term *diversity* refers to the amount and type of different modules that comprise the composition code. The example of Figure 2 illustrates how diverse the composition code can be. In this example, there are different modules supported by $L_i$ (e.g. virtual classes and aspects) and $L_j$ (e.g. Java classes and interfaces) used to realize the composition.

Taking into consideration the example illustrated in Figure 2, the composition diversity is characterized by the dependency among different modules: three concrete aspects ($A1$, $A2$ and $A3$), one abstract aspect ($A$), six virtual classes ($V1$ to $V6$) and one interface ($I$). In order to compose different modules the programmers need to have in their mind the different forms of modules dependency. For instance, direct dependencies are explicit in the code of Java classes and thus their execution order is pre-defined. On the other hand, the aspects can be dependent among them with no explicit reference. This form of dependency among different modules requires a special treatment. Considering the example in Figure 2, a precedence mechanism (aspect $A$) needs to be defined as the aspects $A1$ and $A2$ share a same declaration (*joinpoints*). The aspect $A3$ is in charge of defining the correct interception order of $A1$ and $A2$ in $B2$. An extensive reasoning about the composition code is inevitable in these cases in order to manage the composition diversity. For instance, the pointcut declaration (PCE...
call (* .m*()) intercepts all the calls to methods of Java classes whose name begins with m (*.m*()). These calls are scattered through many modules of the program (e.g. B2 and E3). Thus, programmers need to analyse the names of all the methods in order to confirm that the composition was correctly implemented and no wrong method has been picked out. In other words, this means to avoid that implicit rules of the composition (e.g. the set of modules that belong to the composition) are broken.

**Composition Scope.** Composition code is a set of modules implemented by two programming languages, L1 and L2 respectively. In this context, the term scope refers to the extent of the enclosing context where the program elements of L2 are associated with. In the example illustrated in Figure 2, the composition scope is defined from the modules supported by L2.

In order to understand the scope of the composition in Figure 2, it is essential to understand that the operation update(B1.getX()) in E2 updates the value of the attribute x declared in module B1. However, the original value of x is used by E3 (operation mF(*. .*)). Thus, the update of x by D cannot be ignored by E3 as these two modules depend on the manipulation of the correct value of x. Then, E2 explicitly impacts on B1 and also implicitly impacts on E3. For this reason, we can say that the global scope of E2 is B1 and E3. In addition, there can exist a long dependency chain of some modules connected with the composition code. For instance, the dependency of E2 with B1 is an example of long dependency chain and such a dependency generates a scenario that may affect the quality of the composition code. Changes on the top of the chain tend to be propagated in the other modules.

**Composition Volatility.** Composition dependencies are established in order to prepare the existing program code; otherwise, the composition of modules from L1 and L2 cannot work properly. The term volatility refers to the extent that these dependencies are broken when a single change is made in the composition code.

In order to analyse the composition volatility in Figure 2, it is important to take into consideration the existing composition dependencies. For instance, the use of wildcards (star notation) in A1 creates dependencies among A1 and the Java classes that implement methods get and set. The aspect A1 uses wildcards aiming at intercepting all the methods that begin with m (*.m*()) and get (*. .get*()). The PCE is based on the syntax of the source code and during the evolution process the syntax can be changed. In other words, names of methods can change and new methods that begin with m or get can be added. As a consequence, when the application tends to evolve, the PCE needs to be modified.

**3. THE MEASUREMENT SUITE**

This section presents a metrics suite that relies on the terminology presented in Section 2.1. The metrics are intended to quantify the composition code properties (Section 2.2). The goal is also to provide support for studying and assessing the impact of composition measures on quality attributes of evolving programs, such as software stability (Figure 3). The composition code is the input to the measurement process, which is in turn quantified through the set of composition metrics.

We defined four metrics for composition code, namely: Local Impact (LoI), Global Impact (GoI), Composition Volatility (CoV), and Depth of Dependency Chain (DDC). An overview of these metrics is presented in Table 1. It provides brief definitions of the metrics and their association with the composition properties (Section 2.2), which they are intended to measure. Each metric is described in terms of: (i) an informal definition (Table 1), (ii) a formal definition based on the terminology presented in Section 2.1, and (iii) a simple example.

**Figure 3. Measurement Framework Overview**

The existing relationship between the composition properties (Section 2.2) and the composition metrics are illustrated in Table 1, column 2 respectively. The metrics LoI and GoI are directly associated with the extension of the composition scope in a program. Therefore, they are used to quantify the scope property of the composition code. The broken dependencies in program elements are quantified by the CoV metric. In addition, as the DDC metric quantifies the length of dependency chain it operates as another indicator of volatility. Quantification of breakings in the source code is also a reflection of the diversity of modules involved in the composition. As the number of modules increases, thus the number of dependencies and breakings are expected to increase as well.

**Table 1. Composition Metrics**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Composition Property</th>
<th>Metric Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoI</td>
<td>Scope</td>
<td>The ratio between the numbers of program elements affected by the composition divided by the total of program elements.</td>
</tr>
<tr>
<td>GoI</td>
<td>Scope</td>
<td>Quantifies the composition scope by counting all the program elements affected through the use of composition mechanisms.</td>
</tr>
<tr>
<td>CoV</td>
<td>Volatility and Diversity</td>
<td>Quantifies the dependencies broken in the composition code while preparing it so that composition mechanisms can be properly applied.</td>
</tr>
<tr>
<td>DDC</td>
<td>Volatility</td>
<td>Quantifies the depth of dependency chain for each program element.</td>
</tr>
</tbody>
</table>

For the formal definition of the metrics, let’s consider m ∈ M such that (i) DDm be the set of program elements that represent direct dependencies of m and (ii) IDm be the set of program elements that represent indirect dependencies of m. In addition, let us also
consider that for the set of program element of $M$, there are sets of added program elements, $E_{add,M}$, removed program elements, $E_{rem,M}$, and modified program elements, $E_{mod,M}$.

**Local Impact (LoI) Metric.** Given a program $P$, for each module $m \in M$ the LoI impact of $m$ can be defined $LoI_{P,m}=|DD_{m}|+|E_{rel}|$. As a result, LoI of a program $P$ can be defined as $LoI_{P,Me}=\Sigma_{m \in M} LoI_{P,m}$.

Considering the example of Figure 2, we have four modules directly affected by the composition: $E_{1}$, $E_{2}$, $E_{3}$ and $E_{4}$, and $E_{1}$ as there are explicit references to them. Let’s consider program elements as modules affected by the invocation of just one program element. As the whole example has sixteen modules, the ratio between directly affected modules and number total of modules is $0.25$, which is equivalent to $25\%$ of the code. Lower values for this metric mean that the code affected by this composition is located in a few modules. For instance, this measure might be useful to indicate that changes in the composition code are likely to impact less modules and, therefore, better sustain the code stability.

**Global Impact (GoI) Metric.** This metric is a generalization of LoI. Given a program $P$, for each module $m \in M$ the GoI impact of $m$ can be defined $GoI_{P,m}=|DD_{m}|+|ID_{m}|+|E_{mod}|$. As a result, GoI of a program $P$ can be defined as $GoI_{P,Me}=\Sigma_{m \in M} GoI_{P,m}$.

The relative value of this metric considers the total number of program elements involved. For the example illustrated in Figure 2, the entire composition directly affects the modules $E_{1}$, $E_{2}$, $E_{3}$, $E_{4}$ and $E_{5}$. The modules $E_{3}$ and $E_{4}$ are considered indirectly-affected as the values of parameter $X$ used by it is modified by $E_{2}$. Thus, the GoI value to the example illustrated in Figure 2 is the sum of the number of affected elements divided by total of elements. This relation is equal to $0.38$ ($38\%$). This means that the composition code is impacting $38\%$ of the modules of the program.

**Composition Volatility (CoV) Metric.** Given a program $P$, for each module $m \in M$, the CoV of $m$ can be defined as $CoV_{P,m}=|E_{add,m}|+|E_{mod,m}|+|E_{rem,m}|$. As a result, the CoV of an entire program $P$ can be defined as $CoV_{P,Me}=\Sigma_{m \in M} CoV_{P,m}$.

For the example illustrated in Figure 2, in order to add the composition modules ($A$, $A_{1}$, $A_{2}$, $A_{3}$, $E_{3}$ and $E_{2}$) to the program, the module $B_{1}$ was modified. Thus, the value for this metric is $7$, which was calculated from the sum of manipulated program elements. Lower values for this metric is better because this means that less code was necessary to implement the composition which can imply in less modification.

**Depth of Dependency Chain (DDC) Metric.** Given a program $P$, for each module $m \in M$, the $DDC_{P,m}$ of $m$ can be defined $length(m)$, where

$$length(m) = \begin{cases} 0, & \text{if } m \not\in DD \text{ or } m \not\in ID \\ \max\{length(m)+1\}, & \text{otherwise} \end{cases}$$

This metric takes into consideration modules directly and indirectly affected. Each dependence chain has a root and leaves. The depth of dependency of a leaf is always greater than the root. In other words, DDC of a program element is the distance it from to its root, which is the module itself. If multiple dependencies exist, then the DDC is the longest path for the distance. For the example illustrated in Figure 2, the DDC of module $E_{2}$ is size of the path from it to $1$.

### 4. STUDY SETUP

This section presents the study goal and research hypotheses (Section 4.1), the target applications used to evaluate the proposed framework (Section 4.2), and the study procedures (Section 4.3).

#### 4.1 Goal and Research Hypothesis

The goal of this study was to evaluate to what extent composition properties are correlated with stability of evolving programs. In order to achieve this goal, we performed a comparative analysis of how changes are correlated with composition measures (Section 3). Our analysis was performed using the procedures described in Section 4.3. Our research aims were twofold. First, we aim at evaluating whether the composition properties (Section 2.2), as quantified by our metrics (Section 3), are related to stability of evolving programs. Second, we also aim at discussing some implementation factors that were detrimental to program stability. In this sense, this investigation relies on the analysis of one hypothesis (H), whose null (0) and alternative (1) definitions are as follows:

- $H_{0}$: Composition properties are not related to the instability of evolving programs.
- $H_{1}$: Composition properties are related to the instability of evolving programs.

#### 4.2 Target Applications

In order to evaluate the proposed measurement framework, four evolving composition programs were generated to iBatis [24] and MobileMedia [10]: one AspectJ and CaesarJ implementation for each one. Together, these evolving programs encompass 22 versions. The programs have medium-size and are representative from different domains. We selected these systems because they have been evaluated in previous research work with different purposes [9][10][16]. In addition, these programs contain many types of compositions with different complexity degrees. This way, they enabled us to observe the impact of composition properties on their stability along their versions.

**iBatis.** It is a Java-based open source framework for data mapping. It is composed by more than 60 versions incrementally implemented, where four of them are implemented using AspectJ and CaesarJ programming languages. These versions implement the following functionalities: type mapping, error context, and design patterns. These versions were chosen mainly due to the diversity of their composition code. In addition, they have undergone more frequent and heterogeneous changes through the iBatis history, and their own crosscutting nature and properties (e.g., as occurs with type mapping and error context) call for the composition mechanisms being assessed in our study.

**MobileMedia.** It is an application that provides support to manage (create, delete, visualize, play, send) different types of media (photo, music and video) on mobile devices. During its development and evolution, the initial core architecture was systematically enriched with mandatory, optional and alternative features. Seven versions of the MobileMedia were analysed. The core features are: create/delete media (photo, music or video), label media, and view/play media. Some varying features, amongst
others, are: transfer photo via SMS, count and sort media, copy media and set favourites. Unlike iBatis, which implements mainly new non-functional requirements, the MobileMedia evolution consists of implementing new functional requirements. This distinction allows us to analyse the full range of composition code structures and evaluate the impact of their properties on different evolution contexts.

4.3 Study Procedures
All target application versions (Section 4.2) were analysed according to a number of programming alignment rules. This procedure was applied to assure equal compliance to coding styles and included functionalities. As a second step, we also quantified the degree of stability of the target applications implementation. Program stability was quantified in terms of the program elements changes as proposed by Dantas and Garcia [9]. Change propagation metrics [24] were used with the purpose of quantifying the degree of stability of each program element. This means that the degree of stability is quantified by the number of program elements manipulated (i.e. added, removed and modified) along each program evolution (Section 4.2). Program elements are manipulated in either (i) to improve the program elements while preserving the existing code semantics (e.g. refactoring operations or bug fixes) or (ii) to increment the program in terms of new functionality. The conceptual framework is instantiated in Table 2 for AspectJ and CaesarJ, which are representative examples of contemporary programming languages used in this study.

<table>
<thead>
<tr>
<th>Framework component</th>
<th>AspectJ</th>
<th>CaesarJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>AspectJ Program</td>
<td>CaesarJ Program</td>
</tr>
<tr>
<td>Module</td>
<td>aspect, interface and class</td>
<td>aspect, interface, class and virtual classes</td>
</tr>
<tr>
<td>Program Element</td>
<td>Method, pointcut-advice declarations, intertype declarations expressions and advices</td>
<td>Method, pointcut-advice declarations, intertype declarations expressions, advices and mixin composition expressions</td>
</tr>
<tr>
<td>Property</td>
<td>Diversity, Scope and Volatility</td>
<td>Diversity, Scope and Volatility</td>
</tr>
</tbody>
</table>

Our third step consists in applying the composition metrics to the target applications (Section 4.2). The goal is to gather insight about the usefulness of the composition metrics (Section 3). In particular, we analysed whether the composition metrics are related to program stability (Section 5). At this step, we aim at verifying whether composition metrics are able to work as indicators of program instabilities.

5. DATA ANALYSIS AND DISCUSSION
This section discusses the impact of composition properties on program stability (Section 5.1) using our measurement framework. The data of the composition metrics were automatically collected using our prototype tool [25]. The proposed framework has been applied to programs structured with both FOP (CaesarJ language) and AOP (AspectJ language) techniques. The use of both FOP and AOP in the chosen applications (Section 4.2) enables us to analyse the impact of composition properties on different implementation scenarios. We extend this discussion in Section 5.2 by comparing our composition metrics and conventional coupling metrics. We focus our discussion on the most significant results. A complete data analysis can be found at the website of this study [25]. There, the reader can also find additional results of composition metrics as stability indicators.

Statistical Tests. For the statistical tests performed in Section 5.2, we used the R language and environment\(^2\). We applied the Kolmogorov-Smirnov test to verify if our samples were normally distributed [30]. As our samples were normalized we applied the parametric Pearson’s correlation coefficient [30]; the goal is to obtain evidence about the correlation of the composition metrics with stability. We used a confidence level of 95% (\(\alpha = 0.05\)). The Pearson correlation indicates three cases: values close to +1.0 indicate a strong positive (increasing) linear relationship; values close to -1 indicate a strong negative linear relationship; and finally, values between -1 and 1 indicate the degree of linear dependence between the variables. When the values are close to zero, this means that there is little relationship. The statistical tests were used to accept or reject the hypotheses listed in Section 4.1.

5.1 Composition Properties vs. Stability
The more code changes are required to realize a new program change, the more unstable its design is likely to become [26]. We chose to focus our analysis on stability because it is a key quality attribute on program evolvability [26]. Table 3 shows the correlation results between the composition metrics and the program stability. The Pearson’s correlation computation tests the pair (composition metric value, stability value) for each composition metrics per program version. The analysis of these results reveals that the composition metrics have a strong correlation with the observed instabilities. The high correlation is inferred as, while the maximum correlation value is 1, the correlation values obtained from our set of metrics vary from 0.61 (LoI metric) to 0.90 (CoV metric). While the minimum correlation between the metric LoI and stability is 0.61, GoI correlation values vary from 0.78 to 0.99 for AspectJ versions and from 0.83 to 0.86 for CaesarJ versions. The correlation between stability and GoI is more expressive as GoI captures indirect dependencies among programs elements, which are not captured by the LoI metric (Section 3). The metric CoV is also another strong indicator of stability. Its lowest correlation value is 0.70. However, similar to GoI, its highest correlation value is 0.99. As illustrated in Table 3, the correlation values for AOP are closer to 1 than the same values for FOP. This occurs because the AOP composition scope (AspectJ language) has a greater impact on the program when compared with the FOP scope (CaesarJ language). As the correlation values are very close to 1 (maximum), we conclude the composition properties have presented good indicators of stability. Therefore, we can state that the hypothesis \(H_1\) is accepted (Section 4.1).

\(^2\) http://www.r-project.org/ (11/03/2012)
Even though all the composition metrics (Section 3) were found to be related to stability, those quantifying composition scope tend to work as better indicators. According to the values in Table 3, we could state that the program stability is more strongly related to metrics in the following order (from the most to the least correlated): GoI, LoI, CoV and DDC. This ranking reinforces that the composition scope consistently emerges as the most significant property to explain program instabilities. We observed that this happens because the propagation of changes from a composition program element, for instance, is delimited by its composition code, which is quantified by the composition scope metrics. Figures 5 and 6 illustrate some key results for the interplay of composition code properties and program stability. They are used to support the discussion below.

The Role of “Wide” Composition Scopes on Stability. In order to illustrate a concrete example of modification associated with composition properties, we present a simplified slice of code extracted from iBatis (Figure 4). Considering this example, when a method m3() is added to the class C1 using AspectJ’s mechanisms the programmer needs to change the aspect A1 (pointcut save) in a way that m3() is not intercepted by the pointcut save. However, the scope of the composition implemented by the pointcut save embraces 80% of the iBatis’ source code. Without knowing the impact of composition scope generated from the pointcut save, programmers would tend to change it inadvertently. Fortunately, the GoI metric, when applied to the pointcut save provides insights about its composition scope impact on the program, which is 80%. The use of wildcards leads to high GoI values, which explain why these forms of composition are detrimental to program stability.

<table>
<thead>
<tr>
<th>Composition Metrics</th>
<th>MobileMedia – AOP</th>
<th>iBatis – AOP</th>
<th>MobileMedia – FOP</th>
<th>iBatis - FOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoI</td>
<td>0.71</td>
<td>0.97</td>
<td>0.61</td>
<td>0.86</td>
</tr>
<tr>
<td>GoI</td>
<td>0.78</td>
<td>0.99</td>
<td>0.83</td>
<td>0.86</td>
</tr>
<tr>
<td>CoV</td>
<td>0.85</td>
<td>0.99</td>
<td>0.90</td>
<td>0.50</td>
</tr>
<tr>
<td>DDC</td>
<td>0.76</td>
<td>0.89</td>
<td>0.60</td>
<td>0.86</td>
</tr>
</tbody>
</table>

Table 3. Correlation of Composition Properties with Stability per System

The Consistency of Global Scope as Stability Indicator. Figure 5 illustrates the variation in instability promoted by a single composition in a representative CaesarJ scenario. It is possible to observe that the variation in the stability degree is reflected by the values of GoI. The GoI metric was slightly better than the LoI metric due to the type of dependencies is dominated for direct dependencies. It is also possible to observe that low GoI values in one given version Ri indicate better stability in the next version (Ri+1). From R1 to R3, the composition scope declined in 31%. Analysing the stability of MobileMedia modules, those affected by the composition code, from R1 to R2, we also identified a decrease of 39% in the number of changes. From R2 to R3 both composition scope and stability continued together on a downward trajectory. However, overwhelmingly they increase in R4. The explanation for this is that in R4, new types of media (audio and video) were added in MobileMedia. As a consequence, the name of its modules, operations and declarations were prepared through the rename operations, which were reflected by the composition volatility, quantified by the CoV metric (Figure 6). Also in R4, the depth of the composition dependency chain, DDC metric, reached the total of 9 program elements. Changes in this dependency chain were propagated by all the program elements which make it up. Based on indicators like these provided by the GoI, CoV and DDC metrics, programmers are able to know the risk they are taking when they need to change the program elements that belong to the scope of the composition.

Composition Volatility vs. Composition Scope. We can highlight that to evolve the version R4, a number of refactoring operations in its program elements was required in order to prepare its code for R5 (Figure 6). In addition to the modification of existing modules and programs elements, new ones were added to R5. This variation is captured by the CoV metric (see Figure 6). The CoV metric quantifies the manipulation of elements that occurs within the composition code and it also operates as a consistent stability indicator. On the other hand, GoI goes further since it can be used to predict the stability of a Ri+1 based on the scope of Ri, when Ri+1 evolves over the composition code. The composition scope provides insights about the stability variation. We can observe in Figure 6 that the percentage of the composition scope is always aligned with the program instability variation. The propagation of
changes from the modification of a program element occurs through its dependencies, which are captured by the composition scope metric.

5.2 Coupling vs. Composition Measures

We selected coupling metrics to compare with our composition metrics as stability indicator. The reason is that coupling has been widely used as an internal quality attribute to indicate or predict the stability of programs [13][20]. The coupling metric counts the occurrence of dependencies between modules in two directions: afferent and efferent [12][14]. In addition, we have observed that the first quantitative evaluations for advanced composition mechanisms are emerging [27][28]. These studies often rely on metrics that quantify the level of coupling (and other module-driven properties) as the better indicators of program stability.

In order to analyse the existing correspondence between program stability and both coupling and composition properties, we will take into consideration each instance of composition declaration in the code separately. Figure 7 illustrates a MobileMedia change scenario where the evolution behaviour of two different compositions, called C1 and C5, can be observed. C1 was included in R2 while C5 was included in R5. We chose these versions because they encompass all changes in MobileMedia for both compositions (C1 and C5). For each composition, we analysed the coupling of modules that are part of this composition. The compositions C1 and C5 were implemented in CaesarJ. For each composition, it is presented its percentage of coupling related to the total coupling in the code (Figure 8). As illustrated the coupling of C1 is almost the same along the evolution. This occurs because the composition C1 does not share code with other compositions. There is only a decrease in its coupling percentage in the last versions (e.g. R6 and R7) due to the number of modules that were added to the program. On the other hand, the coupling of the composition C5 presents variation because C5 is coupled with other compositions that need it to work along the evolution.

Coupling Metrics are Agnostic to Indirect Dependencies. However, the key deficiency of coupling metrics is the following: they do not capture most of the indirect dependencies. As a result, they fail to indicate (or predict) the program instabilities sourced on indirectly-related program elements joining a composition. The modules presented in Figure 9, which are associated with compositions C1 and C5, are highlighted by circles (M1, M2, M3 and M4 – Figure 9. Taking into consideration the values for coupling illustrated in Figure 8, we can observe that the compositions C1 and C5 are coupled with less stable modules. However, they do not present a high percentage in terms of coupling (Figure 8), which means that lower coupling may not mean better stability.

Coupling vs. Composition Measures

Figure 5. Stability vs GoI for MobileMedia

Figure 6. Stability vs. CoV for MobileMedia

Progressive Increase of Composition Diversity over Time. We also observed an interesting phenomenon in all the systems with both AOP and FOP: the composition diversity consistently increases through the history of all the programs. In other words, the number of modules joining the composition code always increases as the programs evolve. This means that the composition code consistently embraces additional modules that were not planned to be in the original version of the composition code. This also means that the number of dependencies between programs involved in the composition code tends to increase. This dependency growth can be translated into: (i) more impact with regard to the composition scope, or (ii) more preparation of the source code to work properly with diverse program elements and modules. This explains why the composition code was often the source of instabilities in both AOP- and FOP-based systems.
extended the framework proposed by Briand of dependencies between aspects and classes. Bartsch and Harrison assessing the coupling in AO programs in terms of different types programs. This framework comprises a metrics suite for only For instance, Zhao [13] proposed a framework to describe new program properties, such as coupling, cohesion, size [9][10][31]. were intended to measure specific module-centric or general system stability metrics. The designs and implementations of MobileMedia and iBatis have been evaluated and continuously improved through the last years. Different maturity levels of the investigated systems may impact differently on their stability.

6. RELATED WORK

Over the past few years, many measurement frameworks have been proposed [13][14][15][20]. These existing frameworks supported the evaluation of maintainability of AO and OO programs; they were intended to measure specific module-centric or general program properties, such as coupling, cohesion, size [9][10][31]. For instance, Zhao [13] proposed a framework to describe new forms of dependencies between modules in aspect-oriented programs. This framework comprises a metrics suite for only assessing the coupling in AO programs in terms of different types of dependencies between aspects and classes. Bartsch and Harrison [29] extended the framework proposed by Briand et al. [20] for AspectJ. They described new types of specific coupling connections in AOP, such as coupling on advice execution, coupling on method call, coupling on field access. Unfortunately, none of these related works focused on composition code properties. They also do not take into consideration different composition mechanisms supported by a wide range of programming techniques. Also, they did not evaluate them in terms of stability of evolving programs.

Bartolomei et al. [14] went one step further and proposed a generic framework that captures and takes into consideration the composition mechanisms supported by AspectJ and CaeserJ languages. However, their analysis relies on the coupling created by the use of these mechanisms and how to account for them. Our framework is a further development of their work and, hence, delivers complementary contributions because: (i) we assessed and discussed the impact of these properties on stability of evolving programs. Our work is different and present novel ideas when compared to related work. This occurs because they do not provide means for quantifying the impact of composition properties, supported by post-OO programming techniques, on program stability. Finally, existing work did not discuss and gather evidence of how the evolving program stability is related to code composition properties.

7. THREATS TO VALIDITY

With respect to the validity of our study (Section 5), the conclusion validity threats are related to the data set. In other words, the analysed data set might not be large enough to allow broader statistical analyses. However, we tried to overcome this threat by using systems that were structured with very different techniques and underwent several software evolution scenarios. Threats to internal validity reside on the software history and maturation of the target application designs. The designs and implementations of MobileMedia and iBatis have been evaluated and continuously improved through the last years. Different maturity levels of the investigated systems may impact differently on their stability.

Threats to external validity reside on the limited size and complexity of the target applications, which may restrict the extrapolation of our results. However, while the results may not be directly generalized to professional programmers and real-world systems, the chosen projects allowed us to make useful initial assessments whether the composition metrics would be worth studying further. In spite of its limitations, the presented research constitutes an important initial empirical work on the composition metrics.

8. CONCLUSIONS AND FUTURE WORK

Composition code properties lead to the introduction of new flavours of complexity and may cause side effects on the program stability. In this context, this paper has presented a measurement framework for quantifying key properties of composition code. The framework was instantiated and evaluated in the context of a stability study involving four evolving programs, totalling 22 versions being analysed. The programs were structured with two advanced programming techniques: aspect-oriented programming and feature-oriented programming. Our analysis revealed that composition code properties, as supported by our metrics suite, were consistent indicators of program instabilities. Therefore, the results confirmed that the composition code properties are strongly related to program instabilities (hypothesis H1). The confirmation was observed regardless of programming techniques and composition mechanisms being used.

Composition properties exerted more influence on the stability superiority (or inferiority) of a program than a conventional stability indicator, i.e. coupling (Section 5.2). In particular, we identified that the composition scope property is the most strongly associated with stability. Moreover, in many cases, we observed that program instabilities could be avoided if the scope of certain composition declarations (e.g. pointcuts) was decomposed in narrower scopes (Section 5.1.). Therefore, we believe that the use of our composition measurement framework can also better inform
programmers to tame possible side effects of composition code structure.

As future work, we believe that our work can be further extended in many ways, including: (i) the evaluation of our framework in the context of analyses involving other post-OO programming techniques, such as Compose* [7] and HyperJ [23], and (ii) the use and evaluation of our framework in the context of other quality attributes, such as error-proneness. Those evaluations would enable us to reveal any extension needed for our framework. By now, we assessed the usefulness of the framework to study program stability, as reported in this paper, and program reusability (available at our supplementary website [25]).

9. REFERENCES