4

Composition Measurement Framework

"Do one task after another and do not stop until the end."

Camila Nunes
PhD classmate - class 2009 - PUC-Rio

There is growing evidence that composition properties impact on program stability (Chapter 3). This impact is such that overcomes the impact of measurable module properties typically used in OO measurement. Unfortunately, the effects of composition properties on evolving programs are not well understood. This misunderstanding is mainly due to the lack of measurement frameworks to quantify composition properties. Existing frameworks (BARTOLOMEI *et al.,* 2006, SANTANNA *et al.,* 2003, BRIAND *et al.,* 1999) and metrics suite (ZHAO, 2004) are focused on quantifying properties of programs and their modules only. Some researchers have recently proposed metrics for programs structured with composition mechanisms. However, they are only intended to measure the properties of program modules, such as their coupling and cohesion (GARCIA *et al.,* 2005, GREENWOOD *et al.,* 2007, KUMAR *et al.,* 2009). 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 properties. It is not possible either to study their impact on program evolvability. In other words, the composition properties analysis is still in its infancy and it lacks the definition of measurement framework.

This chapter presents a framework aimed at characterizing and computing composition properties (Section 4.2). It complements the answer given to RQ1 in Chapter 3 on how to objectively analyze the impact of using composition mechanisms on program stability. In addition, it answers the third research question of this thesis (RQ3 in Section 1.3), which states: *What is the impact of composition properties on program stability?* The requirements of the

measurement framework are presented in Section 4.2. The proposed framework consists of terminology (Section 4.2) and a metrics suite (Section 4.3) for composition code, which can be used in programs structured with diverse sets of composition mechanisms. 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. Some known limitations of the proposed framework are presented in Section 4.8. We studied the role of the measurement framework to support stability analysis of 34 releases pertaining to six software projects (Section 4.6). The programs were structured with two different advanced programming techniques: AOP and FOP. These techniques were chosen as they support a wide variety of composition mechanisms. They enabled us to understand the impact of composition code on program evolvability in different contexts. The framework evaluation is presented in Section 4.6. Related work and threats to validity associated with our framework are presented in Sections 4.7 and 4.9. Finally, Section 4.10 summarizes this chapter.

4.1 Measurement Framework Requirements

The purpose of the measurement framework is to improve the efficiency of quantifying composition properties regardless of the advanced programming technique. By means of this measurement framework developers will be able to better understand code composition properties. The characterization is the starting point for promoting the quantification of composition properties by defining a suite of composition metrics (Section 4.2.2). Our aim is providing developers with support for analyzing composition properties when they are implementing and changing the software.

To reach this goal it is necessary to guarantee that two important requirements are being addressed: (i) given the complexity of the composition code the framework should be simple in order to facilitate its use, and (ii) the framework also needs to be generic enough to be used with different advanced programming techniques and their composition mechanisms. In order to make it easy-to-use we deployed effort in representing all the framework elements trough few intuitive concepts, which are relevant to manage composition properties. These names are associated with a basic terminology, which allows developers to easily distinguish the quality attributes related with composition properties and thus, compare different metrics using it. Due to its generality,

the framework can be instantiated for different advanced programming techniques. The measurement framework proposed in this thesis is evaluated in Section 4.6.

4.2

Terminology and Composition Properties

The proposed framework consists of basic concepts and a metrics suite for composition code. First, we present the basic terminology (Section 4.2.1) to describe key properties of composition code (Section 4.2.2) in a consistent manner. The examples provided in this section are based on the CaesarJ programming language (ARACIC *et al.,* 2006) notation.

4.2.1

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 it has been largely used in other works for the definition of metrics (BRIAND *et al.,* 1999, BARTOLOMEI *et al.,* 2006, FIGUEIREDO *et al.,* 2009). Moreover, set theory 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 4.1. It defines the relationships between different components of a program, such as module and program elements, and the composition properties.

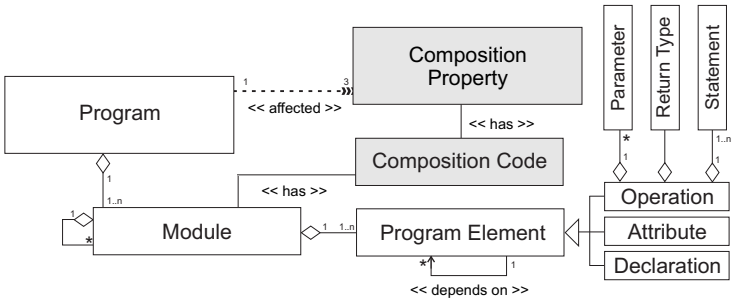

Figure 4.1: Composition Code Measurement: Basic Terminology

A Running Example. For illustration purposes, Figure 4.2 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 B1 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 advices are classified as operations in CaesarJ; the same applies to AOP-specific languages, such as AspectJ (KICZALES *et al.,* 2001). Pointcut expressions, intertype declarations and mixin composition expressions are classified as declarations.

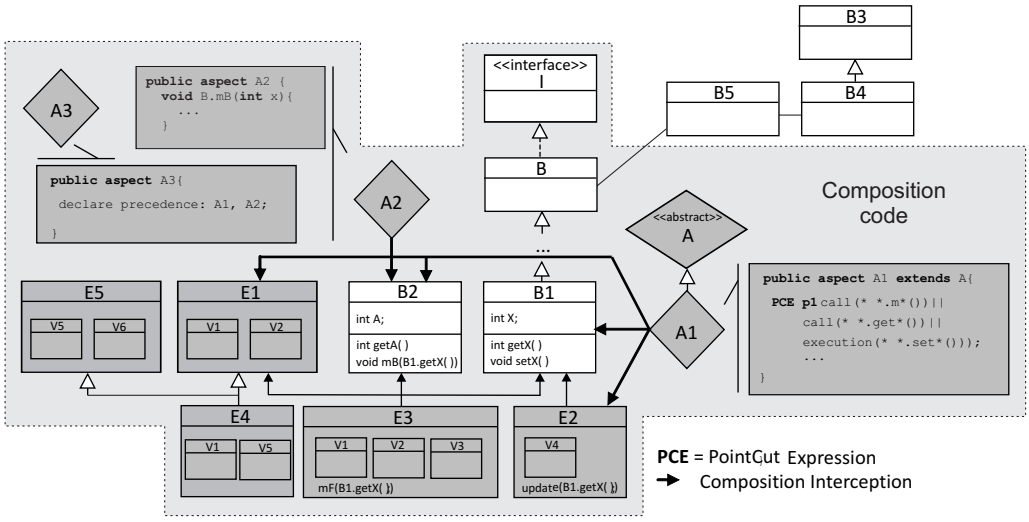

Figure 4.2: Program in CaesarJ

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 intertype declarations operate statically, at compile-time, affecting a program's modules hierarchy. Inter-type declarations may declare members or change the inheritance relationship between classes. In Figure 4.2 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 4.2, 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 4.1) are formalized through the definitions 1 to 4 and illustrated using the example presented in Figure 4.2.

Definition 4.1 *(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 a 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 into consideration the example illustrated in Figure 4.2, we can highlight the following composition mechanisms: pointcut in `A1`, intertype-declaration in `A2`, declare precedence in `A3` and virtual classes in `E1`, `E2`, `E3`, `E4` and `E5`.

Definition 4.2 *(Composition Mechanism)* Given two languages, L_1 and L_2, a composition mechanism is a mean 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 can be either different or the same languages 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, AspectJ and its dialects. As far as AspectJ is concerned, L_1 is considered to be Java, used to define the classes; whereas L_2 comprises the set of constructs to define aspects. In subject-oriented programming languages, such as Hyper/J (HASSOUN and CONSTANTINIDES, 2003), L_1 and L_2 are the same language, i.e. Java; the only difference is the set of composition mechanisms. The code generated by the use of composition mechanisms is called composition code. In Figure 4.2, the composition code is made up by the modules in the light grey area.

Definition 4.3 *(Program and Composition Code)* A program P consists of a set of modules, M. There exist, two subsets of modules: M_b, the set of modules of L_1 and M_e, the set of modules of L_2 where $M := M_b \cup M_e$. A composition, Com, is a set of modules $M_{Com} := M_{cb} \cup M_{ce}$, where $M_{cb} \subset M_b$, $M_{ce} \subset M_e$, $M_{cb} \neq \emptyset \wedge M_{ce} \neq \emptyset$.

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 4.4 *(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_i, m_j), where $e_i \in E_{mi} \wedge e_j \in E_{mj}$. 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 4.2 the composition dependencies are illustrated by the composition interceptions. 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 update(`B1.getX()`) 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`.

4.2.2

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 4.6). 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 4.2. In order to make our discussion more concrete, we used CaesarJ as example; i.e. L_1 and L_2, 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 4.2 illustrates how diverse the composition code can be. In this example, there are different modules supported by L_2 (*e.g.* virtual classes and aspects) and L_1 (*e.g.*, Java classes and interfaces) used to realize the composition.

Taking into consideration the example illustrated in Figure 4.2, 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 4.2, a precedence mechanism (aspect `A3`) 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 analyze 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, L_1 and L_2 respectively. In this context, the term scope refers to the extent of the enclosing context where the program elements of L_2 are associated with. In the example illustrated in Figure 4.2, the composition scope is defined from the modules supported by L_2

In order to understand the scope of the composition in Figure 4.2, 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 L_1 and L_2, 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 analyze the composition volatility in Figure 4.2, 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.

4.3 The Measurement Suite

This section presents a metrics suite that relies on the terminology presented in Section 4.2.1. The metrics are intended to quantify the composition properties (Section 4.2.2). 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 4.3). 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 4.1. It provides brief definitions of the metrics and their association with the composition properties (Section 4.2.2), which they are intended to measure. Each metric is described in terms of: (i) an informal definition (Table 4.1), (ii) a formal definition based on the terminology presented in Section 4.2.1, and (iii) an illustrative example.

The existing relationship between the composition properties (Section 4.2.2) and the composition metrics are illustrated in Table 4.1, 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.

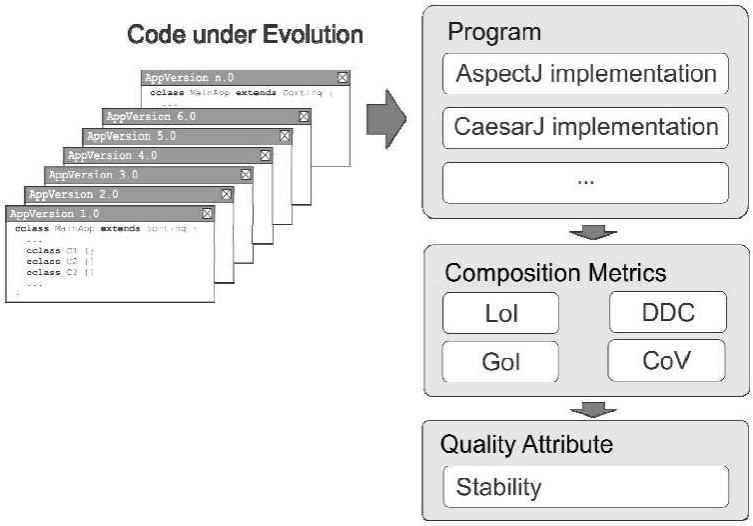

Figure 4.3: Measurement Framework Overview

Table 4.1: Composition Metrics

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

For the formal definition of the metrics, let's consider $m \in M$ such that (i) DD_m be the set of program elements that represent direct dependencies of `m` and (ii) ID_m 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, $m = |DD_m|/|E_M|$. As a result, LoI of a program P can be defined as $LoI_{P,M_e} = \sum_{m \in Me} LoI_{P,m}$.

Code 4.1 shows the main function pseudocode for LoI metric. The data representation is realized by the list `program_elements` (line 2), which is initialized by `InitValues()` (line 5). In this function the data representations are retrieved from the analyzer. The loop (lines 6 to 13) calculates the LoI metric. For each program element in the program, it is calculated the number of other elements it is depended on. The current element (line 8) is catched and all the references to it are obtained (line 9). References to itself are removed by the operation `remove`(..) (line 10). The total of reference is obtained by the size of the list `exp_refs` (line 11). Finally, the the percentage of LoI is calculated in line 14.

Code 4.1: LoI main function

```
01 float localImpact {
02  List program_elements;
03  int total_program_elements;
04  float loi_metric;
05  InitValues();
06  while (int i = 0; i < parser.getElements().size(); i++) {
07    int LoI = 0;
08    program_elements = parser.getElements().getElementAt(i);
09    List exp_refs = program_elements.getElementsReference();
10    exp_refs.remove(program_elements.getName());
11    LoI = exp_refs.size();
12    total = total + LoI + 1;
13  }
14  return loi_metric = total/total_program_elements;
15 }
```

Considering the example of Figure 4.2, we have four modules directly affected by the composition: `E1`, `E2`, `B2` and `B1` 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

$GoI_{P,m} = |DDm| + |IDm|/|EM|$. As a result, GoI of a program P can be defined as $GoI_{P,Me} = \sum_{m \in Me} GoI_{P,m}$.

The relative value of this metric considers the total number of program elements involved. For the example illustrated in Figure 4.2, the entire composition directly affects the modules `E1`, `E2`, `B2` and `B1`. The modules `E3` and `E4` are considered indirectly-affected as the values of parameter `X` used by it is modified by `E2`. Thus, the GoI value to the example illustrated in Figure 4.2 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.

Code 4.2 shows the main function pseudocode for a GoI metric. The idea behind the function is the same as that described in Listing 1 (LoI metric). The only difference is that the list of references retrieved for GoI metric encompasses now direct and indirect references (Section 4.2.1). The process of recovery is implemented by the function `getTotElementsReference()` (line 9).

Code 4.2: GoI main function

```
01 float globalImpact {
02  List program_elements;
03  int total_program_elements;
04  float goi_metric;
05  InitValues();
06  while (int i = 0; i < parser.getElements().size(); i++) {
07    int GoI = 0;
08    program_elements = parser.getElements().getElementAt(i);
09    List total_refs = program_elements.getTotElementsReference();
10    total_refs.remove(program_elements.getName());
11    GoI = total_refs.size();
12    total = total + GoI + 1;
13  }
14  return goi_metric = total/total_program_elements;
15 }
```

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 $CoV_{P,Me} = \sum_{m \in Me} CoV_{P,m}$.

For the example illustrated in Figure 4.2, in order to add the composition modules (`A`, `A1`, `A2`, `A3`, `E3` and `E2`) to the program, the module B1 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.

Code 4.3 shows the main function pseudocode for a CoV metric. The list of current and previous program elements as well as the number of manipulated program elements are represented in lines 2 to 6. The loop (lines 8 to 11) quantify the number of manipulated program elements. One program element is considered added when it did not exist in the previous list of elements (old_e). If an element has its behavior modified it is labeled is modified and thus added to the list mod_e. Program elements have their signature modified are considered new ones. The number of elements removed is calculated in line 13. Finally, the effort of composition is given by the equation presented in line 14.

Code 4.3: CoV main function

```
01 int compositionVolatility(){
02  List program_elements;
03  List old_e;
04  List mod_e;
05  List add_e;
06  List rem_e;
07  InitValues();
08  while (int i = 0; i < parser.getElements().size(); i++) {
09      if(program_elements(i).status()==mod) mod_e.Add(i);
10      else(program_elements(i).status()==add) add_e.Add(i);
11  }
12  if(legth(old_e)-(legth(old_e)-legth(add_e)))>=0)
13     rem_e.Add(i)
14     return lenght(mod_e + add_e + rem_e);
15 }
```

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

$$length(m) = \begin{cases} 0 & \text{if} \quad m \notin DD \vee m \notin ID \\ maxlength(m+1) & \text{otherwise} \end{cases}$$

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 from it 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 4.2, the DDC of module `E2` is the size of the path from it to `I`.

Code 4.4 shows the main function pseudocode for a DDC metric. The loop (line 3 - 6) implements the core of the function. The idea is to go down from

each program element to its leaf. Each new dependence is added in a the vector `longestPath`. After that, go to the next program element while exist dependencies. Finally, the length of the vector `longestPath` is retrieved and returned as the deep dependency chain of `element`.

Code 4.4: DDC main function

```
01 int deepDependencyChain(CodeElement element){
02  Array longestPath = new Array();
03  while(element.depedence()) {
04    longestPath.add(element);
05    element = element.dependence;
06  }
07  return lenght(longestPath);
08 }
```

4.4

The Composition Properties Measurement Tool

The Composition Properties Measurement Tool (CoMMes) supports the task of extracting composition program elements, distinguishing the composition code in a given program. After the extraction process, it is possible to apply composition metrics and thus quantify the composition properties impact on program stability. A simplified CoMMes architecture and its user interface are presented in Sections 4.4.1 and 4.4.2, respectively.

4.4.1

CoMMes Architecture

The CoMMes architecture is presented in terms of its components, which are illustrated in Figure 4.4. There are three main components: composition extractor, composition measurement model and composition metric collector. Each component is described in the following.

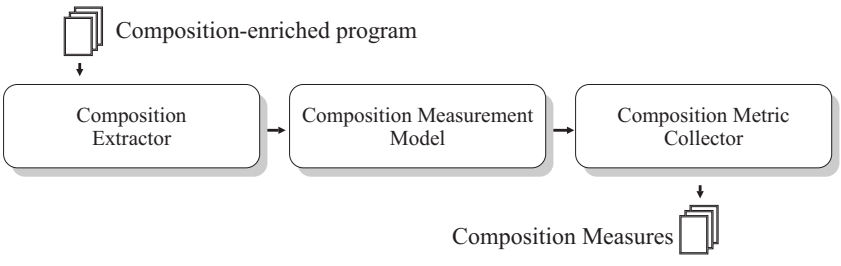

Figure 4.4: CoMMes Architecture

Composition Extractor. This module is in charge of generating the composition measurement model. It takes as input programs implemented in advanced programming language and detects their program elements. It processes the composition-enriched program and builds the model. The current version of the composition extractor was tested with programs implemented in AspectJ and CaesarJ programming languages.

Composition Measurement Model. This model encompasses the data structure defined for composition measurement purposes. It is a suitable representation of the architecture in order to make the measures collection easier. The data representation follows the generic conceptual meta-model for composition programs presented in Section 4.2.1. The current version of the composition measurement model accommodates two different advanced programming techniques: AOP and FOP. These techniques are represented by AspectJ and CaesarJ programming languages.

Composition Metric Collector. This module is responsible for computing the composition metrics (Section 4.3 - Table 4.1). It takes as input the composition measurement model and computes the metrics for each composition based on the algorithms presented in Section 4.3.The measure collection relies on the algorithms presented in Section 4.3.

4.4.2

User Interface

The current version of CoMMes is built based on a command-line interface. This means that the user interface with CoMMes is performed by issuing commands to it in the form of successive lines of text (command lines). In order to run CoMMes the following command is required:

```
CoMMes /C<ArqConst> [/B<DirBase>] [/L<ArqList>]
```

<**ArqConst**> it is a mandatory parameter. It refers to the file that contains guidelines to generate the input files to the composition measurement model. The default extension for the `ArqConst` file is .COMP. The current version requires to inform the target composition individually.

<**DirBase**> it is an optional parameter. It refers to the output directory name. When it is not informed `DirBase` is the current directory.

<**ArqList**> it is an optional field. It refers to name of the file that contains the output report. The default extension for `ArqList` is .LIST.

The command line parameters can be informed in any order. CoMMes validates them, read the files and validate them as well. The parameters `/C`, `/B` and `/L` can be informed in upper or lower case. Finally, to obtain a brief help file, enter the command: /? or /H as follows.

```
CoMMes /? or CoMMes ? or CoMMes h or CoMMes /H
```

The <ArqConst> file contains the description of the construct to be generated by applying the CoMMes. This file consists of comment lines and lines associated with the composition instructions. A line of comment is identified by a '#' character in the first position of the line. Lines of comments and blank lines can occur anywhere and always will be ignored. The lines are aggregated into a composition instruction section. The name of a section must follows the pattern "[<section name>]" and must be drawn from the first row position. Anything after [End] is treated as comment. Figure 4.6 exemplifies how a composition file can be structured.

```
# MobileMedia composition for feature music

[ProgramElements]

.../mobilemedia/alternative/music/AbstractMusicAspect.aj
.../mobilemedia/alternative/music/MusicAspect.aj

[End]
```

Figure 4.5: COM File Template

As a result, the CoMMes tool generates as output the composition metrics values as illustrated in Figure

4.5

Application of the Measurement Framework

This section presents the study goal and research hypotheses (Section 4.5.1), the target applications used to evaluate the proposed framework (see Appendix A), and the study procedures (Section 4.5.2).

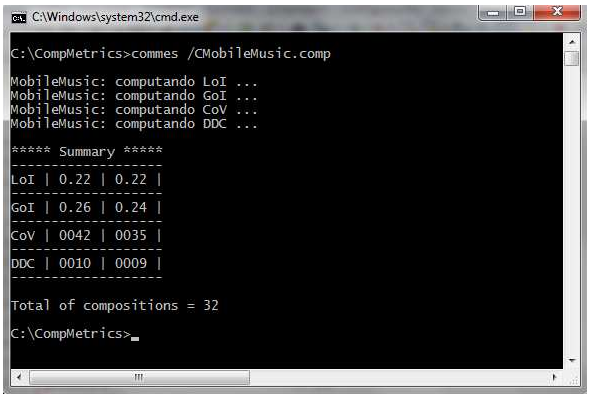

Figure 4.6: CoMMes output

4.5.1

Goal and Hypotheses

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 4.3). Our analysis was performed using the procedures described in Section 4.5.2. Our research aims were twofold. First, we aim at evaluating whether the composition properties (Section 4.2.2), as quantified by our metrics (Section 4.3), affect are related to the 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:

H0: *Composition properties are not related to the instability of evolving programs.*

H1: *Composition properties are related to the instability of evolving programs.*

4.5.2

Procedures

All target application releases (Appendix A) were analyzed 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 (Chapter 3). Change propagation metrics 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. 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 4.2 for AspectJ and CaesarJ, which are representative examples of contemporary programming languages.

Table 4.2: Conceptual Framework Instantiation

Framework Component	**AspectJ**	**CaesarJ**
Program	AspectJ Program	CaesarJ Program
Module	aspect, interface and class	aspect, interface, class and virtual classes
Program Element	Method, pointcut-advice declarations, intertype declarations expressions and advices	Method, pointcut-advice declarations, intertype declarations expressions, advices and mixin composition expressions
Property	Diversity, Scope and Volatility	Diversity, Scope and Volatility

Our third step consists in applying the composition metrics to the target applications (Appendix A). The goal is to gather insight about the usefulness of the composition metrics (Section 4.3). In particular, we analyzed whether the composition metrics are related to program stability (Chapter 2). At this step, we aim at verifying whether composition metrics are able to work as indicators of program instabilities.

4.6 Framework Evaluation

This section discusses the impact of composition properties on program stability using our measurement framework. The data of the composition metrics were automatically collected using our prototype tool. The proposed framework has been applied programs structured with both FOP (CaesarJ language) and AOP (AspectJ language) techniques. The use of both FOP and AOP implementations in the chosen applications (Appendix A) enables us to analyse the impact of composition properties on different scenarios. We extend this discussion is Section 4.6.2 by comparing our composition metrics and coupling measures.

Statistical Tests. For the statistical tests performed in Section 4.6.2, we used the R language and environment. We applied the Kolmogorov-Smirnov test to verify if our samples were normally distributed (DAVID, 2000). As our samples were normalized we applied the parametric Pearson's correlation coefficient (DAVID, 2000); to the goal is to obtain evidence about the correlation of the composition metrics with stability. We used a confidence level of 95% (α = 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 4.5.1.

4.6.1 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 (KELLY, 2006). We chose to focus our analysis on stability because it is a key quality attribute on program evolvability (KELLY, 2006). Table 4.3 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 release. The analysis of these results reveals that the composition metrics have a strong correlation with 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 AOP releases and from 0.83 to 0.86 for FOP releases. 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 4.3). 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 4.3, 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 H1 is accepted (Section 4.5.1).

Table 4.3: Correlation of composition properties with stability per system

Composition Metrics	MM – AOP Coefficient	iB – AOP Coefficient	GP – AOP Coefficient	MM – FOP Coefficient	iB – FOP Coefficient	GP – FOP Coefficient
LoI	0.71	0.97	0.68	0.61	0.86	0.71
GoI	0.78	0.99	0.72	0.83	0.86	0.81
CoV	0.85	0.99	0.74	0.90	0.50	0.79
DDC	0.76	0.89	0.75	0.60	0.86	0.72

MobileMedia = MM / iBatis = iB / GameUp = GP

Even though all the composition metrics (Section 4.5.2) were found to be related to stability, those quantifying composition scope tend to work as better indicators. According to the values in Table 4.3, 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 4.8 and 4.9 illustrate some key results for the interplay of composition 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 4.7). 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.

```
01 class C1 {
02   …
03   int m1( ) { … }
04   int m2( ) { … }
05   …
06 }
```

```
01 aspect A1{
02   pointcut save():call(int *.m*( ));
03   ...
04   after( ): save(){
05      System.out.println("…");
06   }
07   …
08 }
```

Figure 4.7: Slice of iBatis Code (AspectJ)

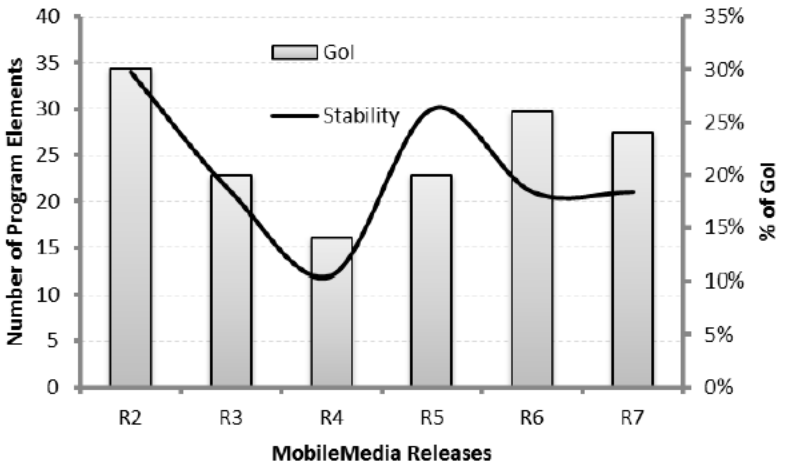

Figure 4.8: Stabiliy vs GoI for MobileMedia

We observed that composition code with higher GoI values than 20% are considered detrimental to stability. This information is useful as it works like a warning to programmers. Equipped with this knowledge, programmers are able to have in their mind that changes in both pointcut save and its dependent program elements can propagate others changes. This way, instead of changing

the pointcut save expression to avoid the interception of the method `m3()`, the programmer can start to consider the possibility of creating a new pointcut. The creation of a new pointcut would contribute to the scope from the existing pointcut save more constant and in turn minimize the possible changes associated with it. Alternatively, whenever possible, programmers can decide to realize refactoring operations before evolving the program as a strategy for decreasing the GoI percentage and thus decreasing its side effects as well.

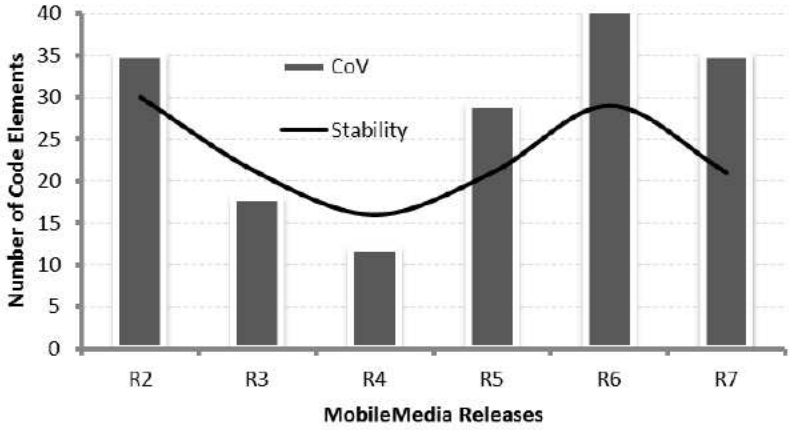

Figure 4.9: Stability vs. CoV for MobileMedia

The Consistency of Global Scope as Stability Indicator. Figure 4.8 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 release Ri indicate better stability in the next release (Ri+1). From R1 to R2 the composition scope declined in 31%. Analyzing 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 R3 to R4 both composition scope and stability continued together on a downward trajectory.

However, overwhelmingly they increase in R5. The explanation for this is that in R6, 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 4.9). Also in R5, 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 release R4, a number of refactoring operations in its program elements was required in order to prepare its code for R5 (Figure 4.9). 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 4.9). 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 4.8 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.

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.

4.6.2

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

(BRIAND *et al.,* 1999, ZHAO, 2004). The coupling metric counts the occurrence of dependencies between modules in two directions: afferent and efferent (ZHAO, 2004, BARTOLOMEI *et al.,* 2006). In addition, we have observed that the first quantitative evaluations for composition mechanisms are emerging (BURROWS *et al.,* 2010, BURROWS *et al.,* 2011). 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 analyst the existing correspondence between program stability and both coupling and composition properties, we will take into consideration each instance of composition specification in the code separately. Figure 4.10 illustrates a MobileMedia change scenario where the evolution behavior 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 releases because they encompass all changes in MobileMedia for both compositions (`C1` and `C5`). For each composition, we analyzed 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 4.11). 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 releases (*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.

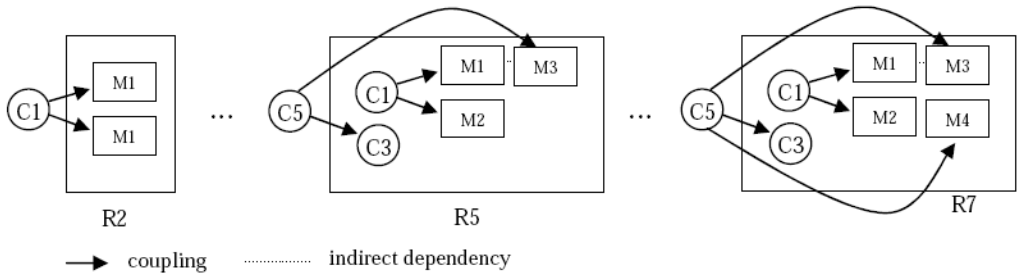

Figure 4.10: Composition Evolution

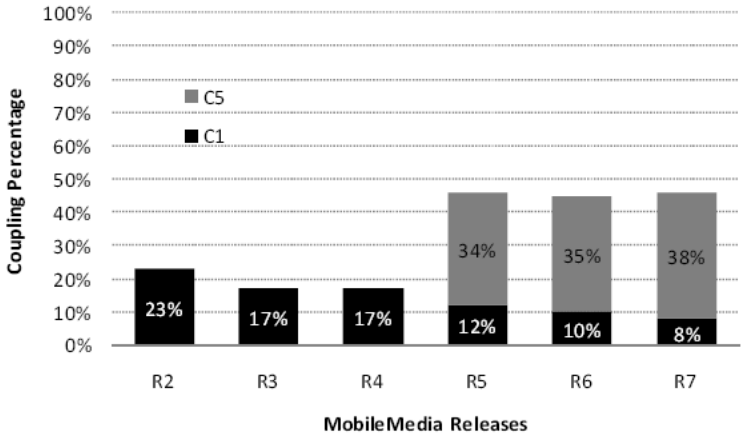

Figure 4.11: Coupling of the Composition Code (C1 and C5)

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 source on indirectly-related program elements joining a composition. The modules presented in Figure 4.12, which are associated with compositions `C1` and `C5`, are highlighted by circles (`M1`, `M2`, `M3` and `M4` - Figure 4.12. Taking into consideration the values for coupling illustrated in Figure 4.11, 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 4.11), which means that lower coupling may not mean better stability.

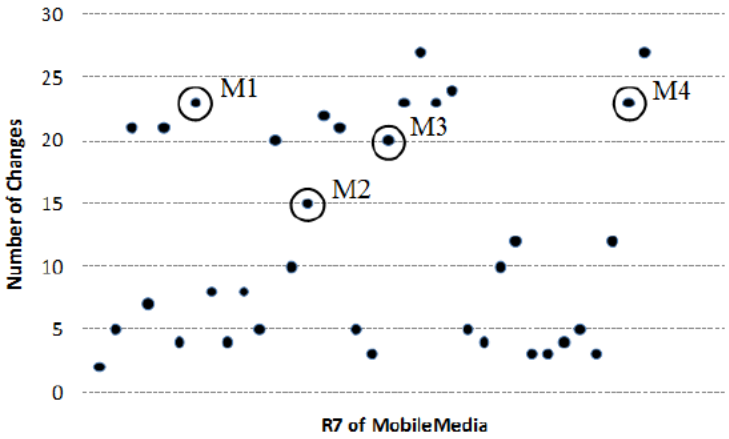

Figure 4.12: Stability of MobileMedia per modules

4.6.3

Other Framework Applications

The measurement framework was also evaluated in a different context: integration of different SPLs. The basic idea behind SPL integration is to foster the reuse of previously-implemented features across a family of independently-developed SPL with minimum change rate.

In this complementary study we analyzed whether stability of evolving SPLs are often related (or not) to composition properties. For this purpose, we have used the composition metrics presented in Section 4.3. This exploratory analysis is carried out involving the stepwise integration of independently-developed SPLs. Each feature integration was identified and implemented on demand, meaning that none of the target SPLs was designed with the required changes (integrations) in mind. The goal was to assess the degree of stability achieved with AOP and FOP and identify its potential association with composition properties (Section 4.2.2). To perform this analysis, CaesarJ was chosen as representative of FOP for two reasons: (i) its compiler proved to be robust during a pilot assessment, and (ii) there are public reports of their successful adoption in industrial projects. On the other hand, AspectJ is the most popular AOP language and many other AOP frameworks follow its programm(ing model. Moreover AspectJ and CaesarJ support programming mechanisms that are also part of other AOP and FOP languages.

Our findings provided evidence that composition properties, quantified by the composition metric suite (Section 4.3), are consistent indicators of stability. The confirmation was observed regardless of advanced programming techniques and composition mechanisms being used. In particular, we identified that the feature integrations cope property is the most strongly associated with stability. In this context, the metric global impact (GoI) indicates stability. Stability is also indicated by the feature integration volatility (CoV metric). Therefore, our findings can better inform programmers to tame possible side effects of feature integration structure.

The results obtained from this study were similar to those results presented in Sections 4.6.1 and 4.6.2. This similarity makes evident that composition properties are constant indicators of advanced program stability in different domains.

4.7 Related Work

Over the past few years, many measurement frameworks have been proposed (BRIAND *et al.,* 1999, SANTANNA *et al.,* 2003, ZHAO, 2004, BARTOLOMEI *et al.,* 2006). 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. For instance, Zhao (ZHAO, 2004) 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 (BARTSCH and HARRISON, 2006) extended the framework proposed by Briand *et al.* (BRIAND *et al.,* 1999) 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 properties. They also do not take into consideration different composition mechanisms supported by a wide range of advanced programming techniques. Also, they did not evaluate them in terms of stability of evolving programs.

Bartolomei *et al.* (BARTOLOMEI *et al.,* 2006) went one step further and proposed a generic framework that captures and takes into consideration the composition mechanisms supported by AspectJ and CaesarJ 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 explored different composition properties (Section 4.2), and (ii) 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 composition 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.

4.8 Known Limitations of the Framework

There are a number of attributes of the proposed framework, and there are also a few limitations. This framework characterizes composition properties and provide means to quantify the impact of these properties on program stability. We claim that it is generic enough to operates with a variate of advanced programming techniques. However, further evaluations in different context, such as Delta-Oriented Programming (SCHAEFER *et al.,* 2010), are still required to validate its generality.

Regarding the delta-oriented programming technique, a program is modularized in core module and a set of delta modules (SCHAEFER *et al.,* 2010). Core modules are the starting points for generating all other products by delta module application. Delta modules specify changes to the core module in order to implement other products. These modules are composed by means of propositional constraints. This scenario seems to be easily mapped in the proposed framework. Core and delta modules as well as the propositional constraints file can be conceptually represented as *modules* in our framework. The propositional constraints file also define the *composition code* of the program. Finally, propositional constraints can be treated as a *program element*, called *declarations*.

An additional limitation of the framework is the extent to which composition properties are covered. Additional studies using different advanced programming techniques will allow us to identify if there are other composition properties that are harmful to program stability.

4.9 Threats to Validity

With respect to the validity of our study, the conclusion validity threats are related to the data set. In other words, the analyzed data set might not be large enough to allow broader statistical analysis. 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.

4.10 Summary

Composition properties are harmful to program stability and thus dealing with then it a key factor to evolve programs in a more stable way. In this context, this chapter 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. The programs were structured with two composition techniques: aspect-oriented programming and feature-oriented programming. Our analysis revealed that composition properties, as supported by our metrics suite, were consistent indicators of program instabilities.

Composition properties exerted more influence on the stability superiority (or inferiority) of a program than a conventional stability indicator, (*i.e.*, coupling). 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 and if there was additional information about the composition properties available at the composition design. 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. We also believe that richer composition specification will bring benefits to developers in terms of stability when they are coding (Chapter 5).