Towards Indicators of Instabilities in Software Product Lines: An Empirical Evaluation of Metrics

Bruno B. P. Cafeo[1], Francisco Dantas[1,2], Elder J. R. Cirilo[1], and Alessandro Garcia[1]
[1]Opus Research Group – Software Engineering Lab, Informatics Department – PUC-Rio – Brazil
[2]Department of Computer Science, State University of Rio Grande do Norte, Brazil
{bcafeo,fdantas,ecirilo,afgarcia}@inf.puc-rio.br

***Abstract*—A Software Product Line (SPL) is a set of software systems (products) that share common functionalities, so-called features. The success of a SPL design is largely dependent on its stability; otherwise, a single implementation change will cause ripple effects in several products. Therefore, there is a growing concern in identifying means to either indicate or predict design instabilities in the SPL source code. However, existing studies up to now rely on conventional metrics as indicators of SPL instability. These conventional metrics, typically used in standalone systems, are not able to capture the properties of SPL features in the source code, which in turn might neglect frequent causes of SPL instabilities. On the other hand, there is a small set of emerging software metrics that take into account specific properties of SPL features. The problem is that there is a lack of empirical validation of the effectiveness of metrics in indicating quality attributes in the context of SPLs. This paper presents an empirical investigation through two set of metrics regarding their power of indicating instabilities in evolving SPLs. A set of conventional metrics was confronted with a set of metrics we instantiated to capture important properties of SPLs. The software evolution history of two SPLs were analysed in our studies. These SPLs are implemented using two different programming techniques and all together they encompass 30 different versions under analysis. Our analysis confirmed that conventional metrics are not good indicators of instabilities in the context of evolving SPLs. The set of employed feature dependency metrics presented a high correlation with instabilities proving its value as indicator of SPL instabilities.**

***Index Terms*—Software Product Line, Metrics, Experimentation, Stability, Feature Dependency.**

I. Introduction

Software product lines (SPLs) emerged as a prominent technology that aims to generate tailored programs (products) from a set of reusable assets speeding up the software development process [1]. SPL-based development focuses on the software decomposition into modular units of functionality defined as *features*. Features are used to describe the commonalities and variabilities of the products of a SPL [2]. For example, in a mobile operating system, individual configurations share a common set of features (e.g. phone call and text message) but differ in other features (e.g. screen resolution or media management).

Nowadays, one can observe that several quantitative evaluations of SPL are starting to emerge [3][5][6][11]. Many of these studies analyse different quality attributes of the SPL source code, such as design stability [6] and cohesion [5]. Despite the differences of the aforementioned studies, their analyses often rely on conventional metrics that quantify properties of implementation modules, such as classes and their methods. For instance, some Chidamber & Kemerer (CK) metrics [7] and their variants [8] have often been used as indicators of SPL quality attributes.

In particular, the success of a SPL design is largely dependent on the availability of metrics to indicate or predict its design stability [6]. Design stability encompasses the sustenance of the product line's modularity properties in the presence of changes [6]. Classic coupling metrics have been broadly used for a long time in standalone programs and found to be effective stability indicators [6][9]. So, on the one hand, the use of such conventional coupling metrics is expected to be effective in SPLs as well. This expectation emerges as the implementation of SPLs is often performed by using the same programming techniques already analysed with these metrics. On the other hand, programming techniques often used to implement SPLs and the modularisation mechanisms of these techniques cannot often fully modularize SPL features. In other words, conventional coupling metrics might be missing important details of the structure of the SPL. This problem might occur, for instance, when the modular units of the programming technique and their dependencies do not match the boundaries of SPL features and their dependencies.

Therefore, it is questionable if even conventional coupling metrics are effective indicators of SPL instabilities. In fact, the use of conventional metrics for SPL evaluation has suffered criticism in recent studies [5][10]. Such metrics are criticised for not being sensitive to features, and thus not taking into account features and their properties. This problem is amplified by the lack of empirical validation into the effectiveness of metrics in indicating quality attributes in the context of SPLs, which in turn creates uncertainty for developers when deciding on measurement strategies. More specifically, the ability of metrics to indicate instabilities in evolving SPLs still lacks through evaluation.

Concerned with the aforementioned issues, this work analyses the effectiveness of using conventional coupling metrics as indicators of instability in SPLs against metrics that are considered the most similar of coupling in SPLs, feature dependency. To achieve our goal, we evaluate existing conventional coupling metrics in two different programming techniques and compare the results with a set of feature dependency metrics. The latter ones are agnostic to particular programming techniques and quantify properties of feature dependencies. Data was collected from several releases of two SPLs previously used in studies of stability [6][16]. Both SPLs were implemented in OO with conditional compilation [11][12] and aspect-oriented (AO) programming techniques [13]. The results reveal that metrics that take into account feature properties are better indicators of instabilities than conventional coupling metrics. Our results also show that considering feature dependency properties in the measurements can improve the power of the metrics in indicating instabilities. The remainder of this paper is organised as follows: Section II describes the main concepts for the paper understanding. Next, Section III describes the metrics analysed in our study. Section IV presents the study setup. The results are analysed and discussed in Section V. Section VI presents the most related work to our work. Finally, Section VII draws the main conclusions and future work.

II. Background

Features are a useful abstraction to express variability in a SPL [14]. According to Ye and Liu [14], to fulfil their tasks, features usually need to interact with other features. These relationships are so-called feature dependency [15]. In the source code, a feature dependency occurs when program elements inside the boundaries of a feature depend on elements outside the feature [5].

Drawing a parallel between the modular units of the programming techniques analysed in this paper (Section II.A) and the modular unit of the SPLs (features) we can notice that feature dependency (Section II.B) is very similar to the concept of coupling [15] in terms of connection between units. Moreover, like coupling, feature dependency can be considered a propagator of instabilities in evolving SPLs [3], and thus it is justifiable its analysis as an indicator of instabilities in evolving SPLs.

This section presents the main concepts associated with feature dependencies (Section II.A), their properties (Section II.B) and how these dependencies are realised in the source code using different programming techniques (Section II.C).

A. Feature Dependency

Feature dependency refers to the relationship between program elements inside the boundaries of a given feature and program elements outside the feature. The basic concepts associated with feature dependency are formalised through definitions 1 to 3 and illustrated using the example presented in Figure 1 with AOP as a representative programming technique.

Basically, we define three components useful to our further definitions: program, module and program element. A program is a set of modules, which consist of program elements. A program element can be an attribute, an operation and even a declaration. Methods and advices, for instance, are classified as operations in AO languages, such as AspectJ; whereas pointcut expressions and intertype declarations are classified as declarations.

Definition 1 (Program, Module and Program Element). *A Program P consists of a set of modules M. A module M is composed by a set E_M of program elements e located in the same physical file. A program element e 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. The set E_M of program elements of M is defined as $E_M := Att_M \cup Op_M \cup Dec_M$.*

Modular units and programming mechanisms of the most programming techniques used to implement a SPL were not designed with the idea of feature as a modular unit of abstraction. Therefore, it is common the misalignment between the boundaries of each feature implementation and the modular units of programming techniques. Therefore, program elements realizing a feature in the source code are often spread over several modules' boundaries. Furthermore, a single module may contain intertwined program elements from different features [3]. We are considering that each program element of *P* belongs at least to the implementation of one feature.

Definition 2 (Feature). *A feature F consists of a set of program elements, E_F, such $E_F \subset E_M$.*

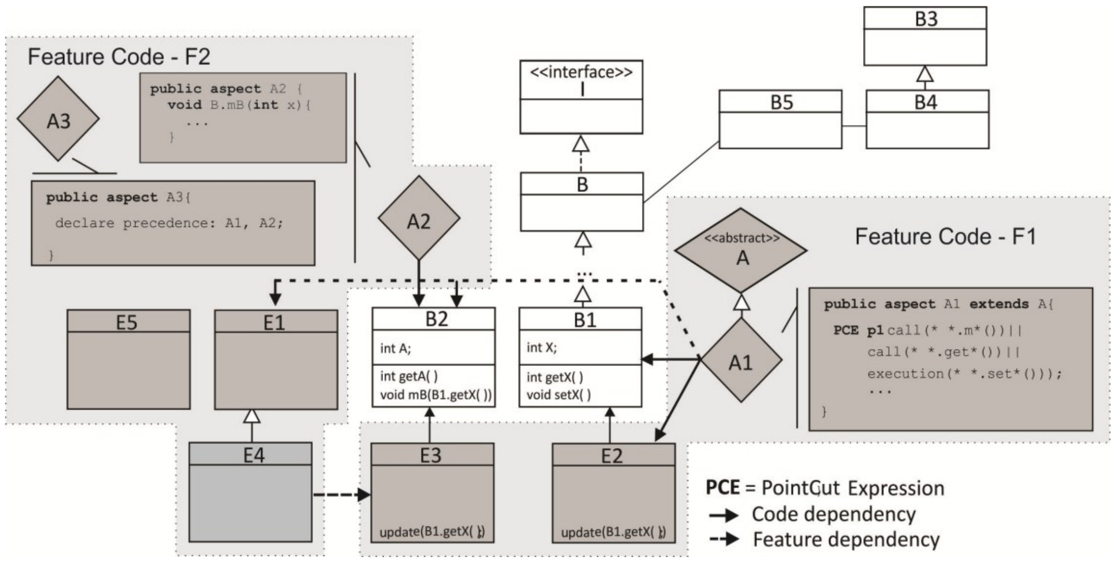
In order to fulfil their tasks, features usually need to interact with other features in a SPL, the so-called feature dependency. In the source code, this is materialised when program elements inside the boundaries of a feature depend on program elements outside the feature or when there are shared program elements among features such as shared attributes. In Figure 1, for instance there are feature dependencies between `A1` and `E1` and also `E3` and `E4`.

Definition 3 (Feature Dependency). *A feature dependency D_{F1F2} between two features can be defined as a set of at least one 4-tuple (e_i, e_j, F_1, F_2), where e_i and $e_j \in E \wedge e_i \in E_{F1} \wedge e_j \in E_{F2}$ and $F_1 \neq F_2$. D_{F1F2} is considered a dependency when there is direct relationship between e_i and e_j or when $e_i = e_j$. In other words, D_{F1F2} takes places when at least one program element $e_i \in E_{F1}$ refers to a program element $e_j \in E_{F2}$ or when there is a shared element between features ($e_i = e_j$).*

B. Feature Dependency Properties

Feature dependencies are realised to have certain properties, which may exert an impact on SPL stability. Corroborating with Dantas et al. [17], we observed that feature dependency code is characterised by at least two basic properties: scope and volatility.

The property scope refers to the extent of the enclosing context where the program elements involved in the dependency are associated with. In the example illustrated in Figure 1, the scope is defined by the set of modules, which belongs to the feature dependency code. For instance, the scope of the feature dependency encompasses all modules that are affected by the modules which make up their code, either by changing code or influencing behaviour. Finally, the property volatility refers to the extent that these dependencies are broken



```
public aspect A2 {
  void B.mB(int x){
    ...
  }
```

```
public aspect A3{
 declare precedence: A1, A2;
}
```

```
public aspect A1 extends A{
 PCE p1 call(* *.m*())||
     call(* *.get*())||
     execution(* *.set*()));
     ...
}
```

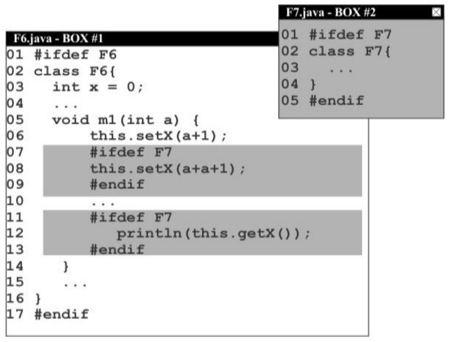
FIGURE 1. FEATURE DEPENDENCY IN AOP (ASPECTJ).

when a single change is performed in the code pertaining to the feature dependency code or the involved modules.

C. Programming Techniques

In order to have more generic results in our study (i.e. results as free as possible from the influence of programming technique), we consider two programming techniques in our study (Section IV): Conditional Compilation and AOP. We chose AspectJ [18] to implement feature dependency with AOP because it is the most consolidated AOP language and it supports a wide range of mechanisms to implement feature dependency. Conditional compilation, on the other hand, is a well-known and industry-strength technique for handling SPL evolution [11][12]. Using conditional compilation, pre-processor directives indicate pieces of code that should compile or not based on the value of pre-processor variables, creating dependencies among features. Such decision may be at the level of a single line of code or to a whole file. Figure 2 illustrates an example of feature dependency using conditional compilation in Java[1]. Blocks of *#ifdef* statements (BOX #1 – lines 07 and 11) associated with feature F7 (BOX #2) are included in code associated with feature F6. This means there is a dependency between F6 and F7 due to the use of the attribute *a* (BOX #1 – line 08), method *setX* (BOX #1 – line 08) and method *getX* (BOX #1 – line 12), all belonging to feature F6, in part of code enclosed by directives related to the feature F7.

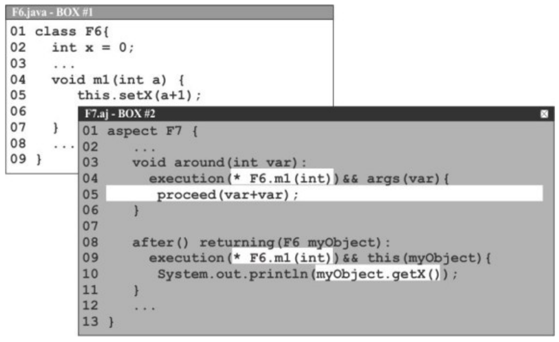
Alternatively, AOP languages support the implementation of features by a range of programming mechanisms, such as pointcuts, advices or inter-type declarations. Figure 3 illustrates a feature dependency scenario in AspectJ. Using the advice (BOX #2 – lines 03 and 08), the dependency between F6 and F7 is realised. In other words, the definition of the value for the attribute *x* is performed by an advice around (BOX #2 – line 03), which changes the value of the argument of the method *m1*, and after returns the control flow to *m1*. Moreover, the value of *x* is printed after the execution of *m1* through the advice after (BOX #2 – line 08), which calls the method *getX* (BOX #2 – line 10) from F6.



```
01 #ifdef F6
02 class F6{
03   int x = 0;
04   ...
05   void m1(int a) {
06       this.setX(a+1);
07       #ifdef F7
08       this.setX(a+a+1);
09       #endif
10       ...
11       #ifdef F7
12          println(this.getX());
13       #endif
14    }
15    ...
16 }
17 #endif
```

```
01 #ifdef F7
02 class F7{
03    ...
04 }
05 #endif
```

FIGURE 2. FEATURE DEPENDENCY IMPLEMENTATION USING CONDITIONAL COMPILATION (JAVA).



```
01 class F6{
02   int x = 0;
03   ...
04   void m1(int a) {
05      this.setX(a+1);
06
07   }
08   ...
09 }
```

```
01 aspect F7 {
02    ...
03    void around(int var):
04      execution(* F6.m1(int))&& args(var){
05       proceed(var+var);
06    }
07
08    after() returning(F6 myObject):
09      execution(* F6.m1(int))&& this(myObject){
10       System.out.println(myObject.getX());
11    }
12    ...
13 }
```

FIGURE 3. FEATURE DEPENDENCY IMPLEMENTATION USING AOP (ASPECTJ).

[1] The Java language itself does not support preprocessing directives. So, we use the Java language with a preprocessor called Antenna that supports conditional compilation - http://antenna.sourceforge.net/ - 28/01/2012

III. Target Metrics

We argue that conventional coupling metrics might not be good indicators of instability in the context of SPLs. For this reason, two different sets of metrics were used for analysing the effectiveness of indication of instability in evolving SPLs: conventional coupling metrics (Section III.A) and feature dependency metrics (Section III.B)

A. Conventional Coupling Metrics

We select a representative set of six metrics for the analysis. These metrics were proposed by Chidamber and Kemerer [7] for OOP and Ceccato and Tonella [8] adapted them in order to make them applicable to AO programs. Therefore, these metrics are useful benchmark for our study because for each conventional CK metric for OOP there is an equivalent extension for AOP, and thus allowing us to broaden our analysis and have results less dependent of programming techniques.

Some of these metrics have already been used in studies focusing stability in object- and aspect-oriented software, including SPLs [6][9]. Not only does this metric suite contains a variety of popular coupling metrics, but also contains other conventional measures such as cohesion and size. Nevertheless, to the best of our knowledge, the correlation of such metrics with SPL stability has neither been deeply investigated nor compared with feature dependency metrics, despite their use in SPL stability studies.

The conventional coupling metrics used in our study are presented in Table 1. In the first column is shown the conventional metrics and the second column presents a brief description of the metric based on the CK metric suite and the Ceccato and Tonella's equivalent metrics.

Table 1. Conventional coupling metrics

Metric	Description
WMC	It is the number of operations in a class or aspect. Methods, advice and intertype declarations are counted as an operation.
DIT	It is the length of the longest path from a given class or aspect to the root class or aspect in the hierarchy.
NOC	It is the number of immediate sub-classes and sub-aspects of a module.
CBO	It is the number of classes, aspects or interfaces declaring methods or attributes that are possibly called or accessed by other class or aspect.
RFC	It is the number of methods and advices potentially executed in response to a message received by a given class or aspect.
LCOM	It is the number of pairs of operations working on different class or aspect fields minus pairs of operations working on common attributes.

B. Feature Dependency Metrics

Despite the use of conventional coupling metrics in studies of SPL stability, we argue that they do not capture important feature dependency properties (Section II.B) that might affect the stability of SPLs [3][14[15]. Therefore, in order to try to overcome this limitation, we instantiate a suite of composition metrics proposed by Dantas et al. [17] in the context of feature dependency. The idea is to derive instability indicators from both convention coupling and feature dependency metrics in way that we can identify which group of metrics provide better indicators for SPL instability. The feature dependency metrics (Table 2) are presented based on the properties presented in Section II.B. For more details and formalisation about the feature dependency metrics the reader may refer to the respective work [17].

Table 2. Feature Dependency metrics Definition

Metric	Property	Metric Definition
GoS	Scope	Quantifies the feature dependency scope by counting all the program elements affected by the dependency.
LoS	Scope	The ratio between the numbers of program elements affected by the feature dependency divided by the total of program elements.
CoV	Volatility	Quantifies the dependencies broken in the feature dependency code in a change.
DDC	Volatility	Quantifies the depth of dependency chain for each program element involved in feature dependency implementation.

Taking into consideration the example illustrated in Figure 1 and AspectJ as representative language, to understand the *feature dependency scope,* 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, it might have a long dependency chain of some modules connected with the feature 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 feature code. Changes on the top of the chain tend to be propagated in other modules. Thus, the GoS value to the example illustrated in Figure 1 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 code is impacting 38% of the modules of the program.

To analyse the *feature code volatility* in Figure 1, it is important to take into consideration the existing feature 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. 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 dependency which can imply in less modification.

IV. Study Setup

This section describes our study configuration in terms of its goals (Section IV.A), the target systems (Section IV.B), and the data collection procedure (Section IV.C).

A. Goal Statement

The main goal of this study is to evaluate the effectiveness of two suites of metrics as instability indicators in evolving SPLs. For the purposes of the evaluation we compare the effectiveness of conventional metrics (see Table 1), commonly used in empirical studies of SPL stability, against feature dependency metrics (Table 2). The conventional metric suite adopted is the CK metric suite, which contains a variety of popular coupling metrics and other conventional measurements, such as cohesion and size. The feature dependency metrics were instantiated from a set of metrics proposed by Dantas et al. [17] trying to overcome limitations of conventional coupling metrics used in empirical studies of SPL stability (Section III).

B. The Target Systems

The two medium-sized SPL systems used in this study include two board games and one embedded mobile application. The first one is actually a family of two board games called *GameUP* and encompasses the games called *Shogi* and *Checkers* [16]. Each game is also a SPL in itself. The second SPL is an embedded mobile software called *MobileMedia* [6] which allows users to manipulate images, videos and music on different mobile devices. Following we describe the target SPL systems.

GameUP. It is a program family of two board games which are by themselves SPLs. In this work we only analysed Shogi and Checkers game releases. Shogi is a chess games whereas Checkers is an American checker game. Both of them provide features to manage various functionalities for customising the board (e.g. indicating moveable pieces) and the matches between players (e.g. indicating player turns). The evolution scenarios comprise the inclusion of optional and alternative features providing us a variety of feature dependencies, which are fundamental to conduct the investigation of this work.

MobileMedia. It is a SPL that provides support to manage photo, music, and video on mobile devices. The core feature represents basic media management actions such as create/delete media, label media and view/play media. The alternative features are the types of media supported such as photo, music and/or video. The optional features are transfer photo via SMS, count and sort media, copy media and set favourites. The evolution scenarios comprise different types of changes involving the inclusion of mandatory, optional and alternative features, as well as changing of one mandatory feature into two alternatives.

For more information about each of the two target SPL systems, the reader may refer to the respective work [6][16].

C. Data Collection

The study was divided into three major phases: (1) the mapping of elements of the source code to features, (2) the collection of metrics, and (3) the quantification of instabilities.

During the phase one the source code was mapped into features. To do so, a list of all target application features was created in the slices of code that implement them were mapped, i.e. identifying which segment of code contributes to which feature in the SPLs. The feature code mapping was facilitated using a SPL implementation tool [19].

The second phase was divided in two steps. The first step was the collection of conventional coupling metrics. Such metrics were collected using the CKJM tool [20] in the SPLs implemented with conditional compilation and the AOPMetrics tool [21] was used to collect the same equivalent CK metrics in the SPLs implemented in AspectJ. The second step involved the collection of the feature dependency metrics. These metrics were collected using the an automated tool for extracting metrics from composition code [17] which uses information of the structure of the feature (Phase 1) and also the dependencies between features to extract the measures.

The last phase quantified the instability in terms of the program elements (Section II.A) changes. Change propagation measurement [6][16] was used with the purpose of quantifying the degree of instability of each program element. This means that the degree of instability is quantified by the number of program elements changed along each program evolution.

It is important to notice that we are analysing SPL instabilities, and thus we are considering SPL evolutions in our study. So, we collected the analysed metrics and the instabilities per release of SPL. After that, we correlate each analysed metric with the corresponding release's instability considering all evolutions.

To analyse the effectiveness of each metric (conventional coupling and feature dependency metrics) as indicator of instability in evolving SPLs we conducted a Spearman's rank correlation test. This test is a non-parametric test that allows us to measure the correlation between our independent variables (each evaluated metric) and our dependent variable (instability). For this test we used a tool named R [22]. We assumed the commonly used confidence level of 95% (that is, p-value threshold = 0.05). For evaluating the results of the correlation tests, we adopted the Hopkins criteria to judge the goodness of a correlation coefficient [23]: < 0.1 means trivial correlation, 0.1-0.3 means minor correlation, 0.3-0.5 means moderate correlation, 0.5-0.7 means high correlation, 0.7-0.9 means very high correlation, and 0.9-1 means almost perfect correlation.

V. Data Analysis and Discussion

Tables 3 and 4 show the Spearman's rank correlation coefficients for MobileMedia and GameUP SPLs, respectively. It is worth to notice that the metric are sorted by crescent p-value. For each version of both applications, the target metrics (Section III) are applied in a way that we can analyse the correlation of these metrics with the SPL instabilities. The correlation analysis is carried out in terms of both programming technique: conditional compilation and AOP.

Conventional coupling metrics fail in indicating instability. Looking at the data of the statistical analysis, we can draw an interesting observation: the suite of feature dependency metrics better indicates instabilities than the conventional CK metric suite. We can notice in Tables 3 and 4 that both in MobileMedia and in GameUP the feature dependency metrics presented a high correlation according to the Hopkins criteria [23]. The only exception refers to the DDC for MobileMedia which presents a moderate correlation with instability. Despite that, the DDC metric presented a much superior correlation than DIT, the first ranked conventional coupling metric in MobileMedia, which was considered as having a minor correlation to instabilities. The observations herein lead us to the conclusion that metrics that consider feature dependency information substantially improve the capacity of indicating instabilities during the SPL evolution.

Feature dependency metrics succeeded in a fine-grained analysis. As mentioned in the study setup (Section IV) we are gathering the metrics per SPL release and not per modular unit (class, aspects or features). However, taking into consideration the instabilities of modules, we can confirm the superiority of the metric suite for feature dependency for indicating instability in comparison with conventional coupling metrics. For instance, analysing the conditional compilation version of MobileMedia, the feature Controller presents a high GoS, which according to our first conclusion, indicates a high probability of instabilities in that feature during the SPL evolution. In fact, confirming the metric indication, the feature Controller was the most instable feature during the SPL evolution. Unlike feature dependency metrics, conventional coupling metrics were not good indicators of instabilities in SPLs in a deeper analysis of the source code. The class with the highest DIT (best indicator of instabilities among conventional coupling metrics) in MobileMedia does not have the highest number of instabilities among the classes of the SPL.

This observation might suggest that conventional coupling metrics are not effective indicators of instabilities in SPLs. This finding holds even if one considers the modifications of the units (i.e., classes and aspects) they measure in the context of SPLs. On the other hand, feature dependency metrics proved to be good indicators of instabilities if we look from the feature perspective. As a consequence, these metrics can be used for driving efforts toward features with a high likelihood of suffer instabilities during the evolution of the SPL.

Considering only features as modular units in the measurements might not be enough. Based on the first conclusions presented in this section the reader may wonder whether adapting conventional coupling metrics to use feature as modular unit may be enough to have good indicators for instability. In other words, if one considers features as the modular unit in the measurements instead of programming techniques units (classes and aspects), and thus overlook other SPL properties such as feature dependencies, is it enough to improve the effectiveness of adapted conventional metrics? To answer this question, it is interesting to analyse the SPLs implemented in AOP. Taking MobileMedia as representative SPL, we can observe that it basically evolves by means of the addition of features with crosscutting behaviour. Crosscutting features are features that may affect many modules of the SPL [24]. This means that the modularisation in AOP is very close to the modular abstraction of a feature in this case. In this way, it is expected that conventional coupling metrics would indicate more precisely the instabilities in SPLs. Analysing the data separated by programming techniques, it is possible to say that the differences between correlations of feature dependency metrics and conventional coupling metrics are smaller when compared to conditional compilation, as expected. However, even with a modularisation matching the feature boundaries in AOP, the feature dependency metrics remains as better indicators of instabilities than conventional coupling metrics.

This observation suggests an important insight: only adapting conventional metrics to use feature as modular unit, such as in some studies where instability were not the focus [5] [10], might not be enough to have good instability indicators. In fact, Geipel and Schweitzer [26] argue that is recommended to take advantage of two properties in the context of software evolution: change characteristics and dependency structure. The feature dependency metric suite tries to use both properties during the measurement (Section II.B), and thus might be an explanation of the superiority of such metric suite even when the modular unit of the programming technique is similar to the abstraction of feature. It is worth to notice that further investigation is necessary to answer the question, since we are not considering conventional metrics that use feature as the modular abstraction in our study.

Feature dependency metrics leave a gap as instability indicator. Focusing on data presented in Tables 3 and 4 we can notice and interesting behaviour regarding the feature dependency properties (Section II.B). The correlation of feature

Table 3. Spearman's Rank for MobileMedia

Metrics	Coefficient	p-value
GoS	0,7327273	0,0067
LoS	0,6546208	0,0209
CoV	0,5600000	0,0582
DDC	0,4690416	0,1240
DIT	-0,1909091	0,5523
WMC	-0,0820122	0,8000
LCOM	-0,0535800	0,8686
RFC	0,0142630	0,9649
NOC	0,0109490	0,9731
CBO	-0,0017860	0,9956

Table 4. Spearman's Rank for GameUP

Metrics	Coefficient	p-value
CoV	0,8445441	0,00055
DDC	0,8095987	0,00142
GoS	0,6536422	0,0211
LoS	0,5354644	0,0728
NOC	-0,4928913	0,1035
CBO	-0,3986046	0,1993
DIT	-0,3982357	0,1998
WMC	-0,3668573	0,2408
LCOM	-0,3633298	0,2457
RFC	-0,3139452	0,3203

dependency metrics related to scope measurement were superior than feature dependency metrics related to the volatility property measurement in MobileMedia. However, in Games SPL we have the opposite: feature dependency metrics related to volatility property were better indicators of instabilities than feature dependency metrics related to the scope property. Such situation can be justified mainly because of the different evolution scenarios in these SPLs. The evolution scenarios of the MobileMedia comprise additions and changes of features. In this way, broad changes happen in the source code of other features in the scope of the dependencies of the feature which was added or changed (scope property). The evolution scenarios of the GameUP comprise basically integration of features from other SPLs. In other words, they require additions of source code to realise a new feature dependency or changes (i.e. break) existing feature dependencies (volatility property). Therefore, such behaviour is an indication that the combination of volatility and scope properties would create a metric that can be more closely correlated with instabilities in evolving SPLs.

VI. Related Work

Apel et. al. [5] adapted conventional metrics that were originally proposed for procedural and object-oriented systems to a set of cohesion metrics to provide a better understanding of the characteristics of cohesion in SPL. Dantas et al [17] developed a metrics suite intended to quantify the composition code properties and support assessing the impact of composition measures on quality attributes of evolving applications. Burrows et al. [25] proposed a novel metric to specific dependencies in aspect-oriented software systems that were not captured by conventional metrics. However, these metrics do not consider some important properties of SPLs, and thus the use of such metrics might not be appropriate to SPLs. Our work advances in the state of art showing that (1) SPL-specific metrics are better indicators of instability than the conventional ones, and (2) feature dependency properties can improve the effectiveness of metrics in indicating instabilities.

van der Hoek et al. [10] developed a class of variability-ware coupling metrics to evaluate the structure defined by SPL architectures. The suite of metrics is based on the concepts of services dependency. Nevertheless, inferring the volatility based only on dependency might result in a distorted view of changes prediction. In our work, in contrast, we followed the conclusions of Geipel and Schweitzer [26] and we took advantage of change characteristics and dependency structure.

VII. Conclusions

Our result show that in both evolving SPLs analysed, the instantiated suite of metrics that takes into account information of feature dependencies properties displayed the strongest correlation with instabilities. This suite outperformed a popular suite of metrics commonly used in empirical study of SPLs. Therefore, CK metrics have not proved to be good indicators of instabilities in evolving SPLs. Moreover, considering a fine-grained analysis, we could attest that conventional coupling metrics failed in indicate instabilities in the module where such metrics were looking at (i.e., classes and aspects). On the other hand, feature dependency metrics indicated more precisely where were the instabilities. It is worth to notice that, in spite of the relatively low number of target SPLs used in our study, they provided initial evidences that encourage further investigation. In our future research, we wish to deeper explore the properties of feature dependency to propose better indicators of instabilities. We also want to explore properties of feature dependency in terms of other quality attributes.

Acknowledgements

This work was funded by CAPES PhD Scholarship, CNPq (productivity grant 305526/2009-0 and Universal Project grant number 485348/2011-0) and FAPERJ (distinguished scientist grant E-26/102.211/2009, project grant number E-26/111.152/2011 and grant E-26/103.366/2012).

References

[1] P. Clements and L. Northrop. *Software Product Line: Practices and Patterns.* Addison-Wesley, 2002.

[2] K. Pohl, G. Bockle, and F. J. van der Linder. *Software Product Line Engineering.* Springer, 2005.

[3] M. Ribeiro et al. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In GPCE 2011, p. 23-32.

[5] S. Apel and D. Beyer. Feature cohesion in software product lines: an exploratory study. In ICSE 2011, p. 421-430.

[6] E. Figueiredo et al. Evolving software product lines with aspects: an empirical study on design stability. In ICSE 2008, p. 261-270.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Software Engineering*, 20(6): 476-493, 1994.

[8] M. Ceccato and P. Tonella. Measuring the effects of software aspectization. In *WARE'04 Workshop*, 2005

[9] L. P. Tizzei et al. Components meet aspects: Assessing design stability of a software product line, Information and Software Technology, 53(2):121-136, 2011.

[10] A. van der Hoek, E. Dincel, and N. Medvidovic. Using Service Utilization Metrics to Assess the Structure of Product Line Architectures. In METRICS 2003, p. 298-.

[11] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In ICSE 2008, p. 311–320.

[12] V. Alves et al. Extracting and Evolving Mobile Games Product Lines. In SPLC 2005, p. 70–81.

[13] G. Kiczales et al. Aspect-oriented prog. In ECOOP 1997, p. 220–242.

[14] H. Ye and H. Liu, Approach to Modelling Feature Variability and Dependencies in Software Product Lines, In *IEEE Software*, vol. 152, no. 3, 2005, p. 101–109.

[15] H. Cho, K. Lee and K. C. Kang. Feature Relation and Dependency Management: Aspect-Oriented Approach, In SPLC 2008, p. 3-11.

[16] A. Gurgel, F. Dantas and A. Garcia. On-Demand Integration of Product Lines: A Study of Reuse and Stability. In PLEASE 2011.

[17] F. Dantas, A. Garcia and Jon Whittle. On the role of composition code properties on evolving programs. In *ESEM* 2012, p. 291-300.

[18] G. Kiczales et al. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[19] E. Cirilo, et al. A Product Derivation Tool Based on Model-Driven Techniques and Annotations. *J. UCS*, 14(8):1344--1367, 2008.

[20] CKJM Metrics. http://www.spinellis.gr/sw/ckjm/ - 10/01/13.

[21] AOP Metrics. http://aopmetrics.tigris.org/ - 10/01/13.

[22] R Statistic Tool. http://www.r-project.org/ - 12/01/13.

[23] W. G. Hopkins. A New View of Statistics. Sports Science. http://www.sportsci.org/resource/stats - 18/01/13.

[24] J. M. Conejero and J. Hernández. Analysis of crosscutting features in software product lines. In EA 2008, p. 3-10.

[25] R. Burrows et al. An empirical evaluation of coupling metrics on aspect-oriented programs. In WETSoM 2010, p. 53-58.

[26] M. M. Geipel and F. Schweitzer, The Link between Dependency and Cochange: Empirical Evidence, IEEE Transactions on Software Engineering, vol. 38, no. 6, 2012, p. 1432-1444.