Metrics for Feature-Oriented Programming

Ramon Abilio¹, Gustavo Vale², Eduardo Figueiredo², Heitor Costa³

¹ IT Department, Federal University of Lavras – UFLA, Lavras, MG, Brazil
² Computer Science Department, Federal University of Minas Gerais – UFMG, Belo Horizonte, MG, Brazil
³ Computer Science Department, Federal University of Lavras – UFLA, Lavras, MG, Brazil

ramon.abilio@dgli.ufla.br, {figueiredo,gustavoval}@dcc.ufmg.br, heitor@dcc.ufla.br

ABSTRACT

Feature-oriented programming (FOP) is a programming technique to implement software product lines based on composition mechanisms called refinements. A software product line is a set of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment. The literature reports various software metrics for software product lines developed using object-oriented and aspect-oriented programming. However, after a literature review, we observed that we lack the definition of FOP-specific metrics. Based on this observation, this paper proposes a set of eight novel metrics for feature-oriented programming. These metrics were derived both from our experience in FOP and from existing software metrics. We demonstrate the applicability of the proposed metrics by applying them to a software product line.

CCS Concepts

• General and reference ~ Metrics • Software and its engineering ~ Software product lines.

Keywords

Software Product Lines; Feature-Oriented Programming; Software Quality; Software Metrics.

1. INTRODUCTION

The growing need for developing highly configurable software systems, such as software product lines, demands better support for managing software variability [18]. Software variability is the ability of a software system or artifact to be changed, customized, or configured for use in a particular context [8]. Software variability enables the generation of customized software products. However, it also creates new challenges in software development and maintenance tasks. For instance, interactions caused by configurations, also named feature interaction, may create serious threats and make generated products more susceptible to defects. Several programming techniques, such as Aspect-Oriented Programming (AOP) [19], Delta-Oriented Programming [26], and Feature-Oriented Programming [8], have been proposed aiming to improve the modularization of features.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
WETSoM’16, May 16 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4177-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897695.2897701

Software Quality; Software Metrics.

A feature is an increment in a program functionality [34]. Feature-oriented programming (FOP) is a programming technique to implement software product lines based on composition mechanisms called refinements. A software product line is a set of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment [24]. Software product lines aim to promote large-scale and systematic reuse of software components [10]. This reuse is possible due to the product line architecture: common features of a domain compose the core application and other features define variation points [16].

Software metrics are the pragmatic means for assessing different quality attributes, such as maintainability and changeability, of software systems [9][23]. Certain metric values can help to reveal specific components of a software system that should be closely monitored [30]. For instance, software metrics can be used to indicate whether a critical design flaw is affecting a component or part of the system. In this case, developers may suspect that the component implementation should be improved. Typical examples of design flaws include cases of large components realizing a non-cohesive set of features [30].

New metrics have been proposed to assess quality attributes of software product lines [11][25][32][33]. For instance, Couto et al. [11] proposed a set of 13 metrics to assess preprocessor-based software product lines. Revelle et al. [25] proposed metrics to quantify coupling dependencies between features. Moreover, after a literature review, Montagud et al. [23] catalogued 165 metrics used in the context of software product lines. Although some metrics have been proposed and used for assessing software product lines, we still lack the definition of specific metrics for feature-oriented programming.

After performing a literature review, we identified and classified 33 metrics used to assess software product lines (Section 3). The existing metrics were classified in three categories according to the techniques they aim to quantify: (i) Annotative; (ii) Compositional; and (iii) Independent. Metrics in the Annotative group were explicitly proposed to assess software product lines developed by using Annotative techniques, such as #IFDEF directives (preprocessors). Similarly, Compositional metrics focus on techniques like AOP and FOP. Finally, we classified in the Independent group those metrics whose definitions allow their application in both annotative and compositional techniques for software product lines.

After reviewing these 33 metrics from the literature, we observed that very few metrics in the Compositional group focus on FOP-specific mechanisms, such as constants and refinements. Therefore, this paper proposes a preliminary set of eight metrics for feature-oriented programming (Section 4). These metrics have
have been defined based on the studied metrics and on our previous experience using FOP [14] [30]. In addition, we rely on the Goal-Question-Metric (GQM) method [6] to define metrics derived from research questions. We demonstrate the applicability of the proposed metrics (Section 5) by applying them to a software product line developed using a FOP language named AHEAD [7]. The proposed metrics allowed us to quantify the size and complexity of the target system. To conclude this paper, Section 6 discusses related work and Section 7 presents the final remarks and directions for future work.

2. FEATURE-ORIENTED PROGRAMMING

Software variability [31] indicates the possible combinations of features to build a product and it can be represented by means of a feature model [17]. Feature model is a formalism to capture and to represent the commonalities and variabilities among the products generated by a software product line [5]. Figure 1 depicts a partial feature model of a software product line named TankWar [28]. In this figure, some features are specific for different platforms, e.g. PC and Handy. Furthermore, there are alternative features, such as, the size of the Map feature (M_240, M_600, and M_780 features).

Figure 1. Partial Feature Model of TankWar.

In order to develop a software product line, we can use different approaches, such as, annotative [20] and compositional [26]. Compositional approaches, such as FOP, implement features as distinct (physically separated) code units [18]. FOP is based on the concept of stepwise refinements [8]. Stepwise refinement is a paradigm to develop a complex program from a simple program by incrementally adding details [8]. The program increments and original fragments are called refinements and constants, respectively [8]. Classes implement basic functions of a system (constants) and extensions in these functions constitute the features (refinements).

In this paper, we used AHEAD as a representative for FOP. The AHEAD Tool Suite was developed to support FOP in the Eclipse IDE and it has tools for realization and composition of features [7]. AHEAD relies on the Jakarta (Jak) programming language (superset of Java) [7]. Constants and refinements are defined in Jak files, but constants are pure Java-code and refinements are identified by the keyword refines.

Figure 2 depicts a code example in AHEAD. The a_i class implements a stack with two methods: push and pop. The refines keyword in the a_j class indicates that a_j refines a_i. In this example, the a_j class adds new methods (backup() and restore()) to a_i. The refinement also extends the behavior of the push() method (in the a_i class) by adding the calling of the backup() method. The calling of the push() method in the refinement chain is performed using the Super keyword. We used AHEAD to compose base code and different feature modules. Different products are generated according to inclusion of features in the composition process [18].

Using AHEAD, the source code of a software product line is organized in the feature folders and each concrete feature has a subfolder with its name. For example, TankWar has 31 feature folders. This organization supports the feature modularization because we have the constants and the refinements of a specific feature in a separated folder. AHEAD allows other types of artifacts in a feature folder. For example, the Soundeffekt feature is in a folder with audio files.

Figure 2. Example of code (adapted from [18]).

3. METRICS FOR PRODUCT LINES

We performed an ad hoc literature review to identify source code metrics defined and used in the context of software product lines. This literature review involved four researchers that followed a pre-defined protocol. In the protocol, we described, for example, the aims of the review, the search string, the steps followed to select the papers, and the selected papers. We selected 11 relevant papers among 672 references.

Studying the selected papers, we identified 33 metrics defined or used to evaluate software product lines, regardless of the programming technique or language. Some metrics, such as Lines of Code (LOC), Number of Classes (NOC), and Number of Packages (NOP), are used in more the one paper [11] [30]. The 33 metrics are detailed in the project website [12] by showing their Names, Properties, Approaches, Domains, Descriptions, and Value Interpretations.

Table 1 classifies the 33 metrics according to the approach mentioned in their definitions in three groups: Annotative, Compositional, and Independent. Metrics in the Annotative group were explicitly proposed to assess software product lines developed by using Annotative techniques, such as #IFDEF directives (preprocessors). Similarly, metrics in the Compositional group focus on techniques like AOP and FOP. The Independent
group includes metrics whose definitions allow their application in both annotative and compositional techniques. We can observe that Annotative and Independent approaches have 18 and 14 metrics, respectively. There is only one metric, namely Feature Crosscutting Degree (FCD), defined exclusively for the Compositional approach (excluding the Independent group).

We studied these metrics in the context of the papers that cited or proposed them. Some authors made their own interpretation and, in some cases, summarized the definition of the same metric in different ways. For example, the LOF metric were defined in the paper [11] as "number of lines of code (without comments and blank lines) responsible for the implementation of a given feature F", but the original definition [20] is "number of lines between two #ifdefs in source code files and sum them per project". In the first definition, it is not clear to which approach the metric was designed and, in the original one, it is clear that the metric was proposed to the Annotative approach.

Table 1: Metrics Grouped by Approach

<table>
<thead>
<tr>
<th>Annotative Approach</th>
<th>Independent Approach</th>
<th>Compositional Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>MethodBody</td>
<td>Feature Crosscutting Degree (FCD)</td>
</tr>
<tr>
<td>BeforeReturn</td>
<td>NestedStatement</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Number of Classes (NOC)</td>
<td></td>
</tr>
<tr>
<td>ClassSignature</td>
<td>Number of Packages (NOP)</td>
<td></td>
</tr>
<tr>
<td>EndMethod</td>
<td>Packages</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Scattering Degree (SD)</td>
<td></td>
</tr>
<tr>
<td>InterfaceMethod</td>
<td>StartMethod</td>
<td></td>
</tr>
<tr>
<td>Lines of Feature Code (LOF)</td>
<td>Statement</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Tangling Degree (TD)</td>
<td></td>
</tr>
<tr>
<td>Concentration</td>
<td>Hybrid Feature Coupling (HFC)</td>
<td></td>
</tr>
<tr>
<td>Dedication</td>
<td>Lines of Code (LOC)</td>
<td></td>
</tr>
<tr>
<td>Disparity</td>
<td>Maximum Textual Feature Coupling (TFCmax)</td>
<td></td>
</tr>
<tr>
<td>Distance Between Features (DIST)</td>
<td>Number of Features (NOF)</td>
<td></td>
</tr>
<tr>
<td>Feature Scattering</td>
<td>Structural Feature Coupling (SFC)</td>
<td></td>
</tr>
<tr>
<td>Dynamic Concern Diffusion over Components (dCDC)</td>
<td>Structural Feature Coupling Prime (SFC&quot;)</td>
<td></td>
</tr>
<tr>
<td>Feature Tangling</td>
<td>Textual Feature Coupling (TFC)</td>
<td></td>
</tr>
</tbody>
</table>

Concentration, Dedication, Disparity, and DIST metrics were used to assess closeness between features and the modular structure. These metrics were proposed and cited in several papers [13][25][32][33]. We then associated these metrics to the Independent approach, because their definition depends on context, i.e., a component may be, for example, file, class, or method. Therefore, they may be used to assess properties of object-, aspect-, or feature-oriented software systems, for instance. The DIST metric was proposed to assess closeness [32], but it has also been used to assess coupling [25].

Five metrics for coupling were proposed in the same paper [25]: (i) Structural Feature Coupling (SFC), (ii) Structural Feature Coupling Prime (SFC'), (iii) Textual Feature Coupling (TFC), (iv) Maximum Textual Feature Coupling (TFCmax), and (v) Hybrid Feature Coupling (HFC). In some cases, one metric is a variation of another, such as the SFC and SFC' metrics. In other cases, one metric is a composition of others, such as HFC. These variations were made to capture different aspects of the coupling between features. That is, the HFC metric aims to capture both the structural and textual coupling between features [25].

We identified in this literature review five metrics to quantify size: (i) Lines of Code (LOC), (ii) Lines of Feature Code (LOF), (iii) Number of Classes (NOC), (iv) Number of Features (NOF), and (v) Number of Packages (NOP). The NOF metric was proposed to count how many features a program possesses [21]. Although it was originally used for aspect-oriented programming, its definition is generic and can be applied to other techniques. In addition to size, the NOF metric was associated to Tangling in the another paper [13] and the authors did not make clear why they changed the original property [21].

The Tangling Degree (TD) metric was defined in two different papers [11][20]. The first definition [20] was as follows: "number of different feature constants that occur in a feature expression". Another work [11] defined TD as "for each pair of features F1 and F2, this metric counts the number of #ifdef expressions where F1 and F2 are combined by AND or OR operators". In the first, each constant that defines a feature is counted. However, in the second definition, the whole annotation (#ifdef) is counted. The authors of the second paper [11] cited the first one, but they did not mention that there is another definition for the same metric. Actually, they adapted the original definition with more details including an example of counting.

4. FOP METRICS

After the literature review of metrics for software product line, we observed the lack of metrics focusing on FOP-specific mechanisms, such as refinements and constants. However, by observing a feature model (e.g., the one presented in Figure 1) and inspecting the source code (e.g., a code in AHEAD), it is hard to conclude how complex the feature-oriented software system is. For instance, we are not able to grasp the number of constants and refinements or to say how many of them have relationships. To address this limitation in the literature, this section proposes a set of eight metrics for feature-oriented programming. Section 4.1 presents a lightweight formalism used in the metrics’ definitions. Section 4.2 proposes the metrics and Section 4.3 presents a tool to support the proposed metrics.

4.1 FOP Definitions

Formally, a feature-oriented software system s (or system for short) consists of a set of features, denoted by Fea(s), and each feature f consists of a set of components (Comp(f)) or other artifacts (audio, text files, models, images). The components of f can be constants (Con(f)) or refinements (Ref(f)) and they consist of a set of attributes and methods. A constant c can have a set of refinements (Ref(c)) and a refinement r can have a set of methods refinements (Ref(r)). Therefore, in general, a system s consists of a set of: (i) features Fea(s), (ii) constants Con(s), (iii) constant refinements Ref(s), (iv) methods Met(s), (v) method refinements MetRef(s), (vi) attributes Att(s), (vii) refined constants RefCon(s), and (viii) refined methods RefMet(s).

Considering these sets, the set of components of a feature f can be expressed by:

\[ \text{Comp}(f) = \{ \text{Con}(f) \cup \text{Ref}(f) \} \]

Similarly, the components of a system s can be expressed by:

\[ \text{Comp}(s) = \{ \text{Con}(s) \cup \text{Ref}(s) \} \]
4.2 The Proposed Metrics

After studying metrics available in the literature (Section 3), we observed that they do not address specific mechanisms of compositional approaches, more specifically, the FOP mechanisms (constants and refinements). Therefore, we proposed eight metrics to address such mechanisms. In spite of different goals, we noted that we can compare FOP to OOP in some ways. For instance, the constant-refinement relationship in FOP can be compared to the superclass-subclass relationship in OOP. That is, a constant may have a number of refinements as a super-class may have a number of subclasses.

Based on that observation and inspired by existing OOP and AOP metrics, we defined eight metrics for FOP. For example, the Number of Operation Overrides (NOO) metric was based on the Number of Operation Overrides (NOO) metric [22], which indicates the number of overrides that an operation (method) has. We used the formalist presented in Section 4.1 to express each proposed metric considering a system s, a feature f, and a method m. In addition, we used two pipes (“|..|”) to represent the number of elements of a set. As the proposed metrics counts components, methods, and their refinements, the permissible values are integers (discrete). The proposed metrics are defined as follows.

**Number of Constants (NOC).** This metric is number of constants (classes, interfaces) used to realize a feature. A system must have at least one constant (a Java class without the `refines` keyword in its class signature) and the value of NOC metric suggests the number of points that can be refined. Therefore, this value indicates the maintenance tasks complexity. Similarly to NOC metric, this metric can be expressed by:

\[ NOC_f = | \{con \mid con \in Con(f)\} | \]
\[ NOC_s = | \{con \mid con \in Con(s)\} | \]

where \( NOC_f \) is number of constants of a feature and \( NOC_s \) is number of constants of a system.

**Number of Refinements (NOR).** This metric is number of refinements used to realize a feature. If we sum the obtained value from each feature, we have an indication of how many refinements exist in the system. Therefore, the metric’s value suggests the interaction among features. The expressions of this metric are:

\[ NOR_f = | \{ref \mid ref \in Ref(f)\} | \]
\[ NOR_s = | \{ref \mid ref \in Ref(s)\} | \]

where \( NOR_f \) is number of refinements of a feature and \( NOR_s \) is number of refinements of a system.

**Number of Components (NOC).** This metric counts how many components (constants/refinements) were necessary to implement a feature. If we sum the obtained value from the measurement of each feature, we will have an overview of the system implementation. The metric’s value suggests how complex the maintenance tasks can be. This metric can be expressed by:

\[ NOC_f = | \{com \mid com \in Comp(f)\} | \]
\[ NOC_s = | \{com \mid com \in Comp(s)\} | \]

where \( NOC_f \) is number of components of a feature and \( NOC_s \) is number of components of a system.

**Number of Constant Refinements (NCR).** This metric is number of refinements that a constant has. NCR was inspired in the Number of Children metric [9], which indicates the number of immediate subclasses subordinated to a class in the class hierarchy. The metric’s value suggests how complex the maintenance can be. This metric can be expressed by:

\[ NCR = | \{r \mid r \in Ref(c) \land c \in Con(s)\} | \]

**Number of Refined Constants (NRC).** This metric is number of constants that have refinements. The metric’s value indicates how many constants were refined in the system. Therefore, this value suggests how complex the maintenance can be. This metric can be expressed by:

\[ NRC = | \{c \mid \exists r \in RefCon(c) \land c \in Con(s)\} | \]

**Number of Method Refinements (NMR).** This metric is number of refinements that a method has. That is, given a method, how many refinements it has. The metric’s value suggests how complex the maintenance can be. This metric can be expressed by:

\[ NMR = | \{mr \mid mr \in MetRef(m) \land m \in Met(s)\} | \]

**Number of Refined Methods (NRM).** This metric is number of methods that were refined. The metric’s value indicates how complex the maintenance can be due to the interaction among features. This metric can be expressed by:

\[ NRM = | \{m \mid \exists r \in RefMet(m) \land m \in Met(s)\} | \]

**Number of Features with Code (NFC).** This metric is based on NOF metric [21], which indicates the number of features in a system. However, we are interested only in features that have code artifacts. Therefore, we adapted the original definition for this purpose and the NFC metric is number of features that have code artifacts. This metric can be expressed by:

\[ NFC = | \{f \mid \exists com \in Comp(f) \land f \in Feat(s)\} | \]

4.3 Tool Support

We developed a measurement tool as an Eclipse plug-in to measure FOP systems developed in AHEAD. Figure 3 depicts tool Measures view that presents the measurement results of 18 metrics in a tree view. The tool presets the value per measure; e.g., the total of Method’s Lines of Code is 3,910. If the user selects the “Save as CSV” button (on the top-right position), the tool saves the results in a Comma-Separated Values file. Figure 3 presents the values obtained from the measurement of TankWar. We can observe in Figure 3 that the tool implements more than the eight metrics. In addition to the eight proposed metrics, the tool measures an AHEAD code with respect to the Number of Operation Overrides metric (NOO) [22] and the Lines of Code (LOC), for instance. Users can also expand each measure to check the value for a method or component (Figure 4). For instance, the `malen()` method from the `ExplodierenEffekt.jak` component and the `explodieren` feature has the Cyclomatic Complexity value equals 5.
5. PRELIMINARY EVALUATION

This section briefly presents a preliminary evaluation of the metrics. The tool presented in the previous section was used to apply the proposed metrics. We measured the TankWar source code to have an overview on its overall structure and complexity. TankWar is a game developed in AHEAD by students of the University of Magdeburg and it runs on PC and mobile devices. TankWar has been used in other previous studies [1][2][7][28].

In order to evaluate TankWar, we used the Goal-Question-Metric (GQM) method [6]. Following, we present the goal, questions, and related metrics according to this method.

**Goal:** Analyze TankWar from the purpose of evaluating its code with respect to maintainability from the point of view of the developers in the context of feature-oriented programming.

**Questions:**

Q1) How many features, components, constants, and refinements the software system has?

Q2) If we have to remove a specific feature, how many features would be affected by this change?

Q3) If we have to change a method behavior, how many methods and refinements would be affected?

**Metrics:**

Q1): NFC, NOC, NOCt, NOR, NRC, NMR, and NRM;

Q2): NOC, NOR, NOCt, and NCR

Q3): NMR

Figure 5 shows the relationships among the goal and the questions, and among the questions and the metrics. We can observe, for instance, that the metrics: NOC, NOCt, NOR and NCR are used to support the answers of questions Q1 and Q2.

**Answering Q1.** We used the measurement tool to measure TankWar and we observed that it has 88 components (NOC) developed to realize 30 features (NFC), where 30 components are constants (NOCt) and 58 are refinements (NOR). From those constants, 9 were refined (NRC). We observed that exist constants with only one refinement, but also exist constants with 14 refinements (NCR). The latter may indicate a design flaw in the TankWar source code that should be further investigated. In total, 29 different methods were refined (NRM) and these methods have 85 refinements (NMR).

**Answering Q2.** To realize the PC and Tools features, 6 and 3 components (NOC) were developed, respectively. The Tools feature has 1 constant (NOCt) - Tool.jak and 2 refinements (NOR) - Tank.jak and TankManager.jak. The Tool.jak constant has 7 refinements (NCR) localized in the IMG_tool, Beschleunigung, einfrieren, Bombe, Energie, Feuerkraft, and Mars features. Therefore, if we remove the Tools feature, 7 features would be affected because they have refinements of the Tool.jak constant.

We can also observe in the feature model (Figure 1) that the IMG_tool feature is in a different sub-tree in relation of the other features. In other cases, such as, the Maler.jak constant (PC feature), all the 14 refinements (NCR metric) are in different sub-trees, more precisely, in the Beschleunigung, Bombe, China_Type99, einfrieren, Energie, Feuerkraft, fuer_PC, Germany_Leopard, M_240, M_600, M_780, Mars, Re_fuer_PC, and USA_M1Abrams features. Therefore, if we remove the PC feature, or at least the Maler.jak constant, 14 features would be affected because they have refinements of Maler.jak constant.

**Answering Q3.** The toolBehandeln() method has 9 refinements. This method is in the Tank.jak, which is a

<table>
<thead>
<tr>
<th>Measure</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method's Lines of Code</td>
<td>3910.0</td>
</tr>
<tr>
<td>Cyclomatic Complexity</td>
<td>1176.0</td>
</tr>
<tr>
<td>Explodieren</td>
<td>22.0</td>
</tr>
<tr>
<td>ExplodierenEffekt.jak</td>
<td>1.0</td>
</tr>
<tr>
<td>ExplodierenEffekt(TankManager)</td>
<td>1.0</td>
</tr>
<tr>
<td>fillTriangle(int, int, int, int, int, int)</td>
<td>10.0</td>
</tr>
<tr>
<td>explodieren()</td>
<td>1.0</td>
</tr>
<tr>
<td>TankManager.jak</td>
<td>1.0</td>
</tr>
<tr>
<td>Tank.jak</td>
<td>1.0</td>
</tr>
<tr>
<td>Missile.jak</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Figure 3. Measures View.**

**Figure 4. Measures View Expanded.**
constant in the TankWar feature. Observing the results of the measurement, we noticed that the toolBehandeln() method appears in the Tank.jak constant (TankWar feature) and in a Tank.jak refinement (Tools feature) as a non-refinement method. However, Tank.jak in Tools feature is a refinement. Thus, we expected a call of Super in toolBehandeln(). As it does not have a call of Super, the method in the Tank.jak refinement overrides the original behavior without refining it. Therefore, the original behavior it is not propagated in the refinement chain.

Using the metric value, a software developer can notice that changing the original behavior, nine refinements have to be revisited or tested. In the case of toolBehandeln(), the developer may notice an unexpected situation in which a method appears as a non-refinement method both in a constant and in a refinement with eight refinements.

In spite of our study on the existing metrics, the following threats to validity exist: (i) we may have missed some feature-oriented programming characteristics; and (ii) we have few AHEAD-based systems available to define thresholds values. To minimize those threats: (i) we carefully studied the AHEAD mechanisms analyzing the literature and the code of eight software product lines, which were available on the Web; (ii) the number of features of the analyzed product lines ranges from 11 to 106 and the number of lines of code ranges from 98 to 16,719. Therefore, we are not able to define thresholds values, but we could evaluate the proposed metrics.

6. RELATED WORK

Several software metrics have been proposed or adapted due to the new modularization techniques for software development. In the context of software product lines, these techniques have different mechanisms for feature modularization, such as constants and refinements, in feature-oriented programming.

Wong and Gokhale [32] presented two metrics to determine the distance between features (DIST) based on blocks of code. Such a measurement can elucidate how features of the software are close to each other dynamically (DISTD) and statically (DISTS). To illustrate the use of the metrics, they performed a case study. As result, the authors suggested that the metrics provide a quantitative way to measure the distance between two features and they can be useful to understand how a modification made to one feature is likely to affect other features.

Revelle et al. [25] questioned if the measurement of coupling at the feature-level is useful. They proposed feature coupling metrics based on structural and textual information and performed a case study. The study concludes that those metrics are useful to assess coupling at a higher level of abstraction than classes and can be useful to find bugs, guide testing effort and assess change impact.

Couto et al. [11] performed a Software Product Line extraction from ArgoUML - an open source tool used for designing systems in UML - using Annotative approach (conditional compilation). The authors proposed 13 metrics to evaluate granularity and localization. As result, a framework for evaluation and characterization of preprocessor-based product lines was proposed and the authors believed that using the results of this characterization, it was possible to shed light on the major challenges involved in extracting features from real-world systems.

Different from Wong and Gokhale [32] and Couto et al. [11], that proposed metrics for annotative and independent approaches, we proposed metrics to assess compositional-based source code. Revelle et al. [25] focused on coupling using higher level of information present in source code and in text documents. This study proposes a metric suite to assess the source code of feature-oriented programming.

7. FINAL CONSIDERATION

Software metrics play an important role in the evaluation of software quality as well as to make the knowledge on software more precise. Along the years, several metrics were proposed to evaluate software developed in different technologies (Object-, Aspect-, and Feature-Oriented Programming). For Feature-Oriented Programming, there are measures for Annotative approach, but it lacks metrics for Compositional approach.

In this paper, we revisited 33 metrics from the literature that have been used to assess software product lines. We also grouped these metrics by their approaches and proposed eight novel metrics for feature-oriented programming. A tool that allows the measurement of the source code of a software system developed using feature-oriented programming supports the proposed metrics suite. One can argue that a constant with a large number of refinements is not a problem because just a few refinements will be present in a product due to the selected features. However, the maintenance costs may be high to manage such a high number of variation points. Therefore, it is important to assess the system code to obtain quantitative data about the components and methods.

Using the proposed metrics, we can have a notion of the system’s size and complexity. With the NFC, NOC, NOCt, and NOR metrics, we know the system’s size in terms of number of features, artifacts, its types and relationships. With the NRC, NCR, NRM, and NMR metrics, we know how the relationship among the features is, expressing the complexity in maintaining the features. The metrics suite can express the degree of dependency among features, i.e., how many features are realized based on the code of other features. Therefore, the decision on the inclusion or exclusion of a feature can be based on the feature model and on the source code measurement.

As future work, we suggest (i) evaluating the metrics suite using other compositional approaches, such as, aspect and delta; (ii) performing empirical study with Software Engineers to confirm the usefulness of our metrics, and (iii) adding other metrics in the measurement tool.

8. ACKNOWLEDGMENTS

This work was partially supported by CNPq (grant 485907/2013-5) and FAPEMIG (grant PPM-00382-14).

9. REFERENCES


