Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

MODULAR REASONING FOR SOFTWARE PRODUCT LINES WITH EMERGENT FEATURE INTERFACES

Jean Carlos de Carvalho Melo

DISSERTAÇÃO DE MESTRADO

Recife - PE
07 de Março de 2014

Universidade Federal de Pernambuco
Centro de Informática

Jean Carlos de Carvalho Melo

MODULAR REASONING FOR SOFTWARE PRODUCT LINES WITH EMERGENT FEATURE INTERFACES

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Paulo Henrique Monteiro Borba*

Recife - PE
07 de Março de 2014

ACKNOWLEDGEMENTS

Em primeiro lugar, louvo a Deus por essa importante conquista. Ele tem me auxiliado em todos os desafios da minha vida. Toda honra e glória sejam dadas ao SENHOR, Criador e Mantenedor da nossa vida.

Segundo, a minha linda família que me apoia desde o príncipio e continuam batalhando para me oferecer boas condições de estudo. Agradeço também a minha amada, Edinez, pelo ombro amigo e companheiro que me ajudou nos momentos em que precisei, e pela compreensão.

Terceiro, ao meu orientador Paulo Borba pela excelente orientação (incentivos, contribuições e sugestões) que proporcionaram a conclusão deste trabalho.

Quarto, aos professores Márcio Ribeiro e Kiev Gama pelos valiosos comentários.

Quinto, agradeço aos membros do SPG e do LabES pelas contribuições que ajudaram na conclusão deste trabalho e, em especial a: Henrique Rebêlo, Jefferson Almeida, Leopoldo Teixeira, Márcio Ribeiro, Paola Accioly, Rodrigo Andrade e Társis Toledo.

Sexto, agradeço aos amigos que fiz durante o mestrado. São eles: Elvio, Rapha, Super Jeff, Francisco (meu patrão), Evandro (xuxu), Pedro, Chico e Cartaxo. Gostaria também de agradecer aos meus colegas (e ex) de apto: Douglas, sr. Jailson, Alessandro e Mailson. Obrigado galera pelas pizzas, happy hours, etc.

Finalmente, agradeço ao INES — Instituto Nacional de Ciência e Tecnologia para Engenharia de Software — e ao CNPq por financiar este trabalho.

RESUMO

Uma Linha de Produto de Software (LPS) consiste em uma família de sistemas que compartilham um conjunto gerenciado de funcionalidades e são desenvolvidos a partir de um núcleo comum de artefatos. Esses artefatos correspondem a componentes, classes, arquivos de propriedade, e outros tipos de arquivos que são compostos de diversas formas para especificar ou construir produtos específicos. As *features* detêm os pontos variáveis da LPS. Elas são frequentemente implementadas usando pré-processadores. Embora os pré-processadores sejam largamente utilizados, eles poluem o código prejudicando a compreensão do mesmo, tornando a manutenção propensa a erros e, consequentemente, mais cara.

Para minimizar esse problema, pesquisadores propuseram Interfaces Emergentes (*Emergent Interfaces*) que estabelece contratos entre os elementos de código que compõem as *features* com a finalidade de capturar suas dependências. Porém, eles não oferecem uma interface de *feature* global que considere todos os fragmentos de uma determinada *feature*. Como resultado, o desenvolvedor pode introduzir erros na LPS visto que o mesmo não pode entender e raciocinar sobre uma dada *feature* por completo.

Para solucionar isto, nós propomos o conceito de Interfaces de *Features* Emergentes (*Emergent Feature Interfaces*) que infere as dependências entre *features* considerando todos os fragmentos de cada *feature*. Desta forma, nossa abordagem ajuda o desenvolvedor a compreender uma *feature* de forma independente visto que ele está ciente das dependências da *feature* como o todo, evitando assim a quebra dessas dependências. Para implementar a nossa abordagem, nós adaptamos o Emergo. Posteriormente, avaliamos nossa proposta usando cinco LPSs comparando-a com Interfaces Emergentes. Os resultados preliminares sugerem que Interfaces de *Features* Emergentes são viáveis e úteis para manter LPSs.

Além disso, nós observamos que as LPSs atuais são multilinguagens e complexas. Isto significa que capturar dependências entre *features* torna-se ainda mais complicado. Com isso em mente, propomos uma análise automática que computa dependências entre *features* a partir de uma gama de artefatos escritos em diferentes linguagens de programação.

Assim sendo, usamos a ideia de Interfaces de *Features* Emergentes para melhorar a mantenabilidade de LPSs multilinguagens. Nós também desenvolvemos uma ferramenta protótipo e avaliamos a nossa análise através de um estudo de caso. As descobertas iniciais mostram que existem dependências entre *features* que estão espalhadas nos diferentes artefatos e que as mesmas podem ser facilmente quebradas caso um desenvolvedor altere alguma extremidade da dependência.

Palavras-chave: Linhas de Produtos de Software, Pré-processadores, Dependências de feature, Sistemas de software multilinguagens, Dependências entre linguagens

ABSTRACT

A Software Product Line (SPL) represents a family of software systems developed from reusable artifacts, which contain variation points. Artifacts correspond to components, classes, property files, among others that are composed in different manners to specify or build specific products. Features in turn hold the variability of a SPL. One widespread technique to implement features of a SPL is preprocessors, but it obfuscates the source code and reduces comprehensibility, making maintenance error-prone and costly.

To minimize this problem, researchers propose Emergent Interfaces (EI) to capture dependencies between part of a feature that a developer is maintaining and the others. The authors develop a tool called Emergo for improving the maintainability of preprocessor-based software systems. Yet, they do not provide an overall feature interface considering all parts in an integrated way. As a consequence, a developer still might introduce bugs in the software product line since she cannot safely understand and reason about one complete feature before changing the code.

To address that, we propose the concept of Emergent Feature Interfaces (EFI), an evolution of Emergent Interfaces, that consists of inferring feature dependencies by looking at a feature as a whole. EFI provide to the developer to see the dependencies of a given feature through a global interface, which considers all parts of a feature in an integrated way. That way, EFI help the developer to achieve independent feature comprehensibility and, consequently, she can change a feature code aware of its dependencies, avoiding breaking other features. We adapted Emergo to implement our proposal and we evaluate our proposal in terms of size and precision comparing with EI by using five preprocessor-based systems. The results of our study suggest the feasibility and usefulness of the proposed approach.

Besides, we observe that contemporary software product lines are large and multi-language. This means that capturing feature dependencies for these types of systems can be even harder. With this in mind, we propose a cross-language automated analysis for improving the maintainability of multi-language software product lines. Our approach

uses the idea of emergent feature interfaces to capture feature dependencies out of a multitude of artifacts written in different languages. We developed an open-source tool called GSPAnalyzer to implement our technique. To evaluate the proposed approach, we ran a case study with a multi-language product line named RGMS and the results brought preliminary evidence that exists feature dependencies between heterogeneous artifacts and these dependencies can be easily broken if a developer changes either dependence end.

Keywords: Software Product Lines, Preprocessors, Feature Dependencies, Multi-language Software Systems, Cross-Language Dependencies

LIST OF FIGURES

LIST OF TABLES

CONTENTS

CHAPTER 1

INTRODUCTION

A Software Product Line (SPL) represents a family of software systems developed from reusable artifacts [20, 56]. Artifacts correspond to components, classes, property files, among others that are composed in different manners to specify or build specific products. The idea of SPL is the systematic and efficient creation of products based on strategic software reuse. By reusing artifacts, we can build products through features defined in accordance with customers' requirements [56]. In this context, features are the semantic units by which we can distinguish product line variants [70]. Feature models represent the commonalities and variabilities and define the legal combinations of features of a product line [30].

To implement features in an SPL, developers often use preprocessors [35, 58, 21, 40], which is a well-known technique, mainly in industry, to deal with variability. Preprocessor directives like `#ifdef` and `#endif` encompass feature code. Despite their widespread usage [40, 59], preprocessors obfuscate the source code, reducing comprehensibility and making maintenance error-prone and costly [65, 38]. Besides that, preprocessors do not provide support for separation of concerns. In the literature, `#ifdef` directives are even referred as "ifdef hell" [24, 43]. This way, the developer might introduce errors in an SPL when changing a feature code, because one cannot safely reason about a feature without looking at the code of other features.

Virtual Separation of Concerns (VSoC) [35] has been used to address some of these preprocessor problems by allowing developers to hide feature code not relevant to the current maintenance task. The main idea of VSoC is to allow developers to focus on a feature without being distracted by the other features. However, different features eventually share variables, so VSoC does not modularize features since developers do not know anything about these variables in hidden features. Features eventually share code elements like variables. We refer to *feature dependency* whenever we have such sharing like when a feature assigns a value to a variable that is subsequently used by another feature. As a result, the developer might introduce some bugs in the SPL when changing

a variable of a determined feature. Thus, one change in one feature might lead to errors in others. Moreover, these errors can cause behavioral problems in the SPL [59]. In many cases, bugs are only detected by customers running a specific product with the affected feature combination [33].

To minimize this problem, researchers propose Emergent Interfaces (EI) [58, 57] to capture dependencies between part of a feature that a programmer is maintaining and the others. In terms of implementation, the authors develop a tool called Emergo [61, 60], an Eclipse plug-in, for improving the maintainability of preprocessor-based software systems. Although this approach considers `#ifdef` blocks (parts of a feature) to compute EI and still have the VSoC benefits, they do not provide an overall feature interface considering all parts in an integrated way. A feature is likely scattered across the source code and tangled with code of other features (through preprocessor directives) [36]. This way, each `#ifdef` block represents one part of the feature. Thus, there is no global understanding of a given feature. As a consequence, a developer still might introduce bugs when maintaining features since she cannot safely understand and reason about one complete feature before changing the code.

To address that, we propose the concept of Emergent Feature Interfaces (EFI) [47], an evolution of Emergent Interfaces, that consists of inferring feature dependencies by looking at an entire feature. Instead of knowing about parts of a feature, EFI look for feature dependencies considering a feature as a "component" which has provided and required interfaces in order to improve modular reasoning for software product lines. EFI provide to the developer to see the dependencies of a given feature through a global interface, which considers all parts of a feature in an integrated way. This way, EFI help the developer to achieve independent feature comprehensibility. Consequently, she can change a feature code aware of its dependencies, avoiding breaking the relations among features. We adapted Emergo to implement our proposal and we evaluate our proposal in terms of size and precision comparing with EI by using five preprocessor-based systems. The results of our study suggest the feasibility and usefulness of the proposed approach.

Besides, we observe that modern web applications are multi-language. Under the developer's perspective, these systems are complex to maintain since they contain a collection of heterogeneous artifacts (i.e., distinct artifacts written in different languages), holding relations among them. Shaw [63] points out that better forms of modularization and composition are necessary, since web development, in particular, still retains an ad hoc character with many opportunities for improvement. This can be even worse if these web

applications are software product lines because they contain variabilities in their artifacts. In this sense, capturing feature dependencies between different kinds of artifacts is difficult. That is, we still have the feature modularization problem, but now in a multi-language context. Therefore, we propose an automated technique to support the maintenance of SPLs based on the Grails framework, which infers feature dependencies between heterogeneous artifacts, analyzes each dependency, and detects unsatisfied dependencies. For example, an unmatched code element between a page (.gsp) and a controller (.groovy). Our analysis uses the idea of emergent feature interfaces to capture feature dependencies out of a multitude of artifacts written in different languages. We developed an open-source prototype tool called GSPAnalyzer to implement our technique. To evaluate the proposed approach, we ran a case study with a multi-language product line named RGMS. The results brought preliminary evidence that feature dependencies between heterogeneous artifacts occur in practice and these dependencies can be easily broken if a developer changes either dependence end.

1.1 CONTRIBUTIONS

This work makes the following contributions:

- The concept of Emergent Feature Interfaces to help developers when maintaining preprocessor-based software systems, allowing them reason about a feature modularly (Section 3.2);
- Extension of Emergo to support our approach. It computes and shows EFI after developers select a given maintenance point, which might be a feature. Emergent Feature Interfaces provide global feature interfaces containing provided and required information and a simplified view of the existing dependencies (Section 3.3);
- Comparison between Emergent Feature Interfaces and Emergent Interfaces in terms of size and precision (Section 3.4);
- A technique to capture feature dependencies between heterogeneous artifacts (Section 4.2);
- Implementation of our cross-language automated analysis: GSPAnalyzer (Section 4.3);

- A case study that brought preliminary evidence concerning the feasibility of our approach to support modular reasoning for web-based multi-language software product lines (Section 4.4).

1.2 OUTLINE

The remainder of this dissertation is organized as follows:

- Chapter 2 reviews the main concepts used to understand this dissertation;
- Chapter 3 presents the concept of emergent feature interfaces and our empirical evaluation;
- Chapter 4 describes our cross-language automated analysis and our case study; and
- Chapter 5 draws our conclusions, summarizes the contributions of this research, and discusses related and future work.

CHAPTER 2

BACKGROUND

This chapter presents the key concepts we use in this dissertation. First, we present Software Product Lines (SPLs) and their benefits in Section 2.1. We also discuss how to implement features in an SPL using a widespread mechanism called preprocessors (or *conditional compilation*) in Section 2.2, and show several problems with this technique. We then present the Virtual Separation of Concerns (VSoC) approach in Section 2.3, which can address some of these preprocessor problems. Finally, we present the concept of Emergent Interfaces (EI) in Section 2.4 that minimizes the feature modularization problem. To do so, it proposes the use of sensitive-feature data-flow analysis [17], avoiding to explicitly generate and analyze all products using brute-force. Our work in turn uses sensitive-feature data-flow analysis as service. In other words, we do not implement any analysis, instead, we choose one data-flow analysis with high performance, provided by Brabrand *et al.* [17], and then put it in Emergo. To see the implementation details, Toledo work describes different ways of implementing data-flow analysis for SPLs [69].

2.1 SOFTWARE PRODUCT LINES

A Software Product Line (SPL) is *a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* [20]. Core assets are artifacts that can be used to instantiate products [29]. SPL is also *a paradigm to develop software-intensive systems using platforms and mass customization* [56]. Platform represents a set of artifacts that can be combined to derive the products of an SPL, whereas mass customization is related to the flexibility of individual customization, i.e., differentiating a product for a specific customer. The idea of SPL is the systematic and efficient creation of products based on strategic software reuse. By reusing assets, we can build products through features defined in accordance with customers' requirements [56]. A feature can be seen as a prominent or distinctive user-visible aspect, quality, or characteristic of a software system [30]. In this context, features are commonly used to

specify and distinguish product line variants [70]. Figure 2.1 illustrates the commonalities and variabilities of a car.

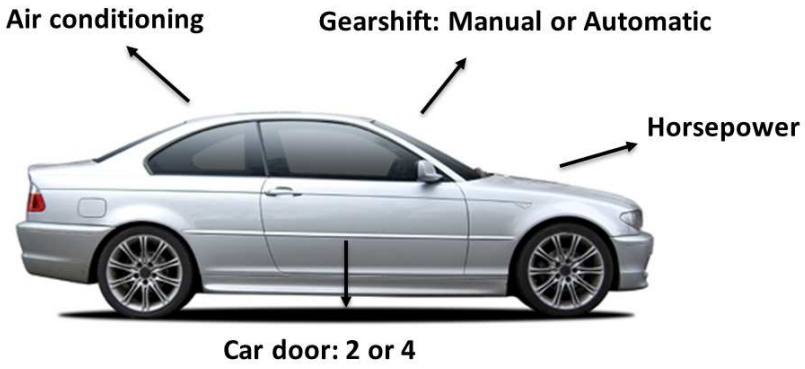


Figure 2.1: Characteristics of a car.

Feature-oriented domain analysis (FODA) method [30] is one of the first contributions to represent and manage the variability of a software system in terms of features. In short, this notation allows us to reason on the variability of an SPL. Features are likely represented in a feature model, typically in the form of a feature diagram with its constraints. Feature model defines the legal combinations of features of a product line [30]. A feature diagram in turn is a visual notation of a feature model. To better illustrate this approach, Figure 2.2 depicts the feature diagram of the car example.

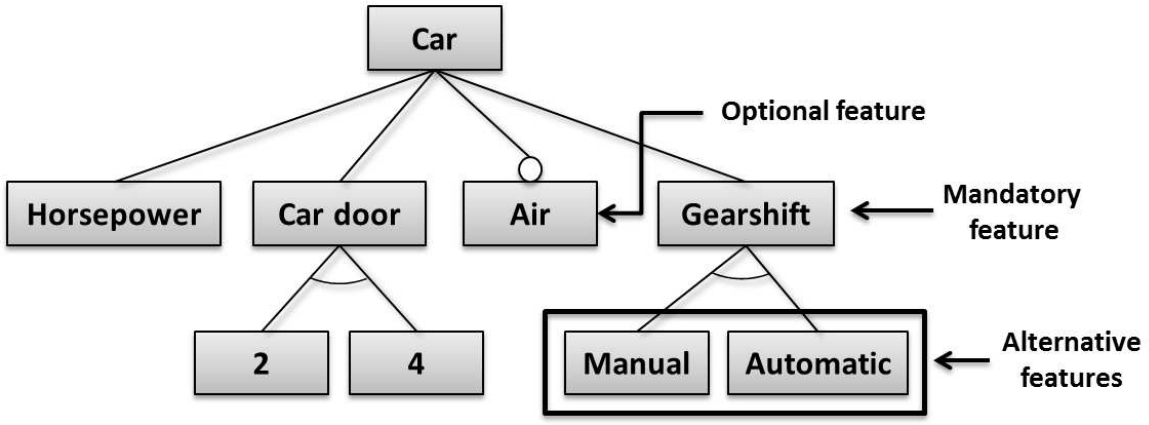


Figure 2.2: Car feature diagram.

As shown in Figure 2.2, a car has diverse types of features. For example, every car contains a transmission that can be automatic or manual as well as a motor. However,

not all cars have air conditioning. As matter of fact, Kang *et al.* [30] define three types of relationships in a hierarchical decomposition of features:

- **Mandatory.** Feature must be selected in all variants, unless that its parent is optional;
- **Optional.** Feature may be present or not in a product (empty circle);
- **Alternative.** Features are mutually exclusive. Thus, only one feature can be selected for a given product (empty arc).

In addition to the parental relationships between features, the FODA notation allows us to set feature constraints: (i) *requires* - the selection of a feature in a product implies the selection of another, and (ii) *excludes* - two distinct features cannot be part of the same product.

2.1.1 Benefits

Many companies are adopting the SPL approach because of these following benefits [20, 56]:

- **Reduction of development costs.** SPLs are meant to reuse artifacts so that individual products are not developed from scratch. This implies in cost reduction. Nevertheless, we need to design and implement the core assets in the first place so that we can reuse them afterwards. In other words, in the beginning the SPL design has a high cost. However, empirical studies reveal that this initial investment pays off when having three products [20];
- **Enhancement of quality.** We have more opportunities to find bugs and fix them since the core assets are reviewed and tested in each product, increasing the quality of the products;
- **Reduction of time-to-market.** In initial stages, the time-to-market is high since we first need to design and develop the core assets. But, it is shortened after the core assets are built because we can reuse these artifacts for building new products.

There are many techniques to support the development of SPL: preprocessors [35, 58, 21, 40], aspect-oriented programming [11, 34], design patterns [12], programming transformations [52, 10], among others. In this work, we focus on preprocessors since they

are common in industrial practice. Thus, the next section presents preprocessors in more detail.

2.2 PREPROCESSORS

Preprocessor [35, 58, 21, 40] is a well-known technique, mainly in industry, to implement variability of a software system. Preprocessors are also known as *conditional compilation*. As its own name suggests, after preprocessing the source code, the compiler decides which code blocks should be compiled based on directive tags (or preprocessor variables). Listing 2.1 depicts part of the code of a feature from the Lampiro product line, called BT_PLAIN_SOCKET. Notice that we encompass the feature code by using an `#ifdef` preprocessor directive. To build a product with the BT_PLAIN_SOCKET feature, we define the BT_PLAIN_SOCKET tag and let the compiler consider the code for compilation. Otherwise, the compiler ignores the code, which means we are building a product without the BT_PLAIN_SOCKET feature.

```
1 xmlStream = new SocketStream();
2 ...
3 // #ifdef BT_PLAIN_SOCKET
4 Channel connection = new SocketChannel("socket://" + cfg.getProperty(Config
    .CONNECTING_SERVER), xmlStream);
5 // #endif
6 ...
7 ((SocketChannel) connection).KEEP_ALIVE = Long.parseLong(cfg.getProperty(
    Config.KEEP_ALIVE));
```

Listing 2.1: Code snippet from the Lampiro product line. Lampiro uses preprocessors to implement features.

Besides Lampiro, real software systems use preprocessors, e.g., Apache web server, Linux and FreeBSD operating systems, VIM text editor, GIMP graphics editor, and gcc compiler [40]. However, a lot of developers face several problems using preprocessors [65, 23, 39, 33]. Despite their widespread usage [40, 59], preprocessors obfuscate the source code reducing comprehensibility, making maintenance error-prone and costly [65, 38]. Besides that, preprocessors do not provide support for separation of concerns. In the literature, `#ifdef` directives are even referred as "ifdef hell" [24, 43]. This way, the

developer might introduce errors in an SPL when changing a feature code, because one cannot safely reason about a feature without looking at the code of other features. For example, the variable that is defined in line 4 of Listing 2.1 and used outside the `#ifdef` block. Detecting these errors is difficult because we need to eventually compile and run a product with the problematic feature combination.

2.3 VIRTUAL SEPARATION OF CONCERNS

Virtual Separation of Concerns (VSoC) [35] has been used to address some of these preprocessor problems by allowing developers to hide feature code not relevant to the current maintenance task. The main idea behind VSoC is to allow developers to focus on a feature without being distracted by the other features. This approach is called "virtual" because there is no physical separation of the feature code. The feature code are still there, scattered and tangled.

In terms of implementation, Kästner *et al.* developed a tool named CIDE (Colored IDE) [33], an Eclipse plugin, that associates background colors with the statements that belong a feature. Instead of preprocessor directives, such as `#ifdef` and `#endif`, visual colors are used as annotation. The tool simply collapses the feature code, hiding it from the user, even though it is still there in the original place. Figure 2.3 shows a screenshot of CIDE. Notice that CIDE mixes the respective background colors to handle overlapping features and exhibits the visual representations directly in the editor.

To avoid the syntax errors that can occur when using preprocessors, CIDE only allows *disciplined annotations*. Essentially, developers can annotate coarse-grained structures, such as complete assignments and declarations, because it is not possible to mark every token of the source code. Additionally, CIDE provides a product-line-aware type system that guarantees that a well-typed SPL produces only well-typed variants [32].

Although VSoC avoids code pollution, it does not modularize features since developers do not know anything about hidden features. There is no information about how a given feature interacts with the others. As matter of fact, different features eventually share the same variables. Thus, maintenance in one feature might break another ones.

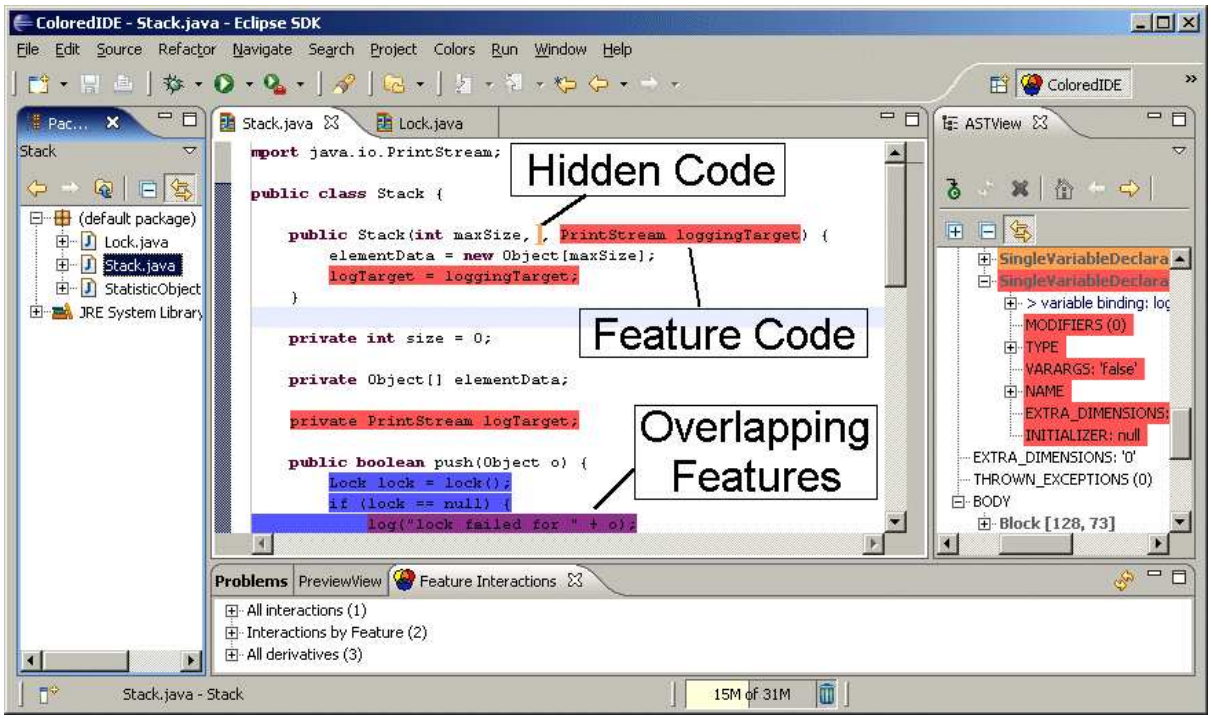


Figure 2.3: CIDE screenshot (extracted from [2]).

2.4 EMERGENT INTERFACES

To minimize the feature modularization problem, researchers propose Emergent Interfaces (EI) [58] that capture dependencies between part of a feature that a developer is maintaining and the remaining ones. This approach is called "emergent" because the interfaces emerge on demand to inform the developers about the possible impacts during a maintenance task. EI have a floppy structure, i.e., not predefined. The idea of EI aims to establish contracts between parts of features to prevent developers from introducing errors into the other features. In addition, this approach still has the benefits of VSoC in which allows the developers to focus on a feature without the distraction caused by the code of other features [57].

The authors developed a prototype tool named Emergo to implement the concept of emergent interfaces [61]. Emergo is an Eclipse plug-in that computes dependencies between parts of features in preprocessor-based systems. Figure 2.4 depicts the Emergo in execution.

As we can see, Emergo provides two views based on table and graph to show emergent interfaces. Besides that, it is aware of the feature model to capture only actual dependencies.

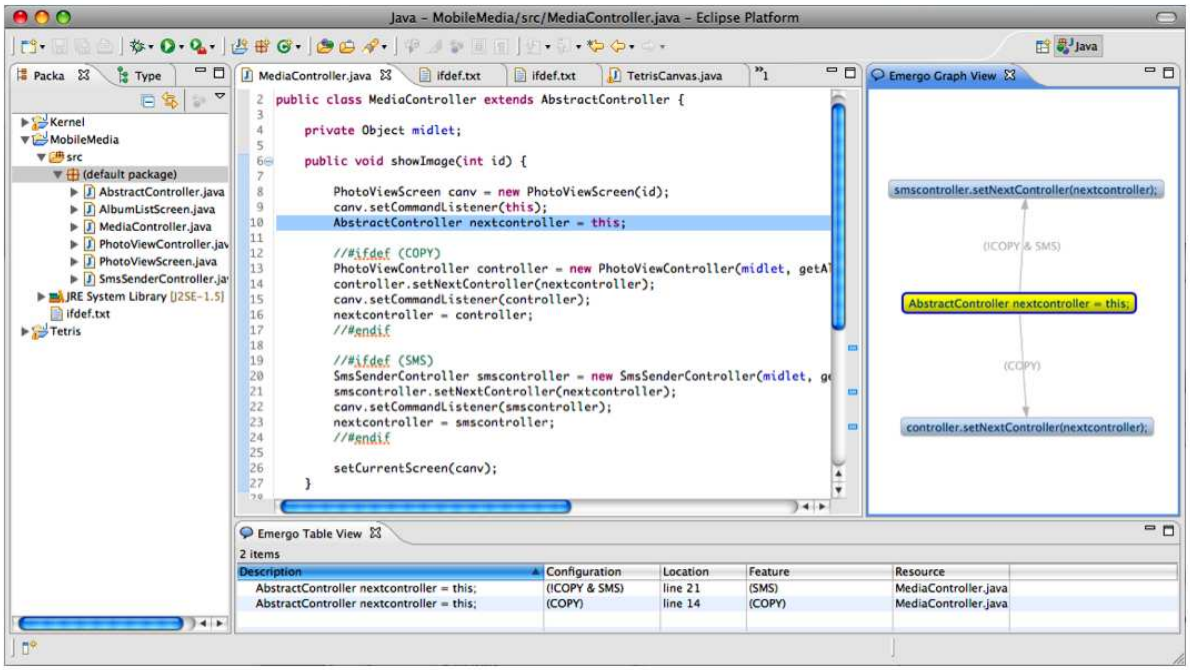


Figure 2.4: Emergo screenshot (extracted from [4]).

Emergo can compute EI between methods or within a single method since it provides inter-procedural and intra-procedural data-flow analyses [16, 69]. Generally speaking, inter-procedural analysis catches data dependencies from a method to the others. Intra-procedural analysis in turn computes dependencies exclusively within a given method.

We present the EI concept in more detail, comparing it with our approach, in Chapter 3.

CHAPTER 3

EMERGENT FEATURE INTERFACES

As we discussed in Chapter 1, Virtual Separation of Concerns (VSoC) [35] reduces some of the preprocessor drawbacks by allowing developers to hide feature code not relevant to the current maintenance task. VSoC provides to the developer a way of focusing on a feature, which is important for her task at the moment [33]. However, this approach is not enough to provide feature modularization, which aims at achieving independent feature comprehensibility, changeability, and development [51]. In other words, there is no proper modular support from a notion of interface between features since a developer does not know anything about hidden features. Consequently, she might introduce errors in the software product line when changing a feature code, because one cannot safely reason about a feature without looking at the code of other features. To minimize this problem, researchers propose Emergent Interfaces (EI) [58, 57] to capture dependencies between part of a feature that a developer is maintaining and the other features. Yet, they do not provide an overall feature interface considering all parts in an integrated way. This approach only considers parts of a feature at a time. Thus, there is no global understanding of a given feature. As a consequence, a developer could easily miss feature dependencies since she cannot understand and reason about one complete feature before changing the code.

To solve the briefly discussed problem, in this chapter we present our proposal that complements Emergent Interfaces. Instead of considering only parts of a feature, we capture dependencies from an entire feature for improving modular reasoning for software product lines.

3.1 MOTIVATION

Emergent Interfaces reduce the feature modularization problem by capturing data dependencies between a maintenance point and parts of other feature implementation of a software product line [58]. Using this approach, the developer can maintain part of a feature being aware of the potential impacts in others [59]. Nevertheless, we show that EI

do not suffice to provide modular reasoning for software product lines, since she cannot understand and reason about one complete feature before changing the code.

According to Cataldo *et al.* [19], companies have adopted feature-driven development approach, which consists of implementing software systems based on features, to enhance development flexibility, to facilitate formal modeling of systems and even to lead to higher levels of quality. Thus, requirements are driven by features. A maintenance task might require a new feature implementation or only change an existing one (e.g., for fixing bugs). But, to accomplish a maintenance task, the developer needs to understand the feature assigned for her. Features tend to crosscut the code base [40]. So, developers could easily miss feature dependencies when maintaining feature without proper modular support from a notion of interface between features. In fact, developers usually do not have enough knowledge for changing the code in order to fix bugs [72]. For instance, in a study on incorrect bug-fixes from large operating systems,[1] 27% of the incorrect fixes are made by developers who have never touched the source code files associated with the fix [72]. With this lack of knowledge, developers can introduce errors when changing a feature code, making maintenance even more expensive. Maintenance and enhancement of software systems is expensive and time consuming [42]. Alone understanding of the software system stands for 50% to 90% percent of the maintenance cost [66]. Therefore, providing feature interfaces would contribute to improving the modular reasoning and reducing the high cost of software maintenance and evolution.

In this context, we present two maintenance scenarios in order to illustrate the problems mentioned above and addressed in this work. First, consider a JCalc[2]-based product line of a standard and a scientific calculator written in Java. The code of the product line is available in the online appendix. We transform the JCalc project into an SPL to utilize it only as a running example having mandatory, optional and alternative features; we do not consider this product line in our evaluation. The class *JCalc* contains the *main* method responsible for executing the calculator (cf. Listing 3.1).

```
1 public static void main(String args[]){
2     //...
3     //#ifdef PT_BR
4     title = "JCalc - Calculadora Padrão e Científica";
5     //#endif
```

[1] including Linux, OpenSolaris, FreeBSD and also a mature commercial OS

[2] http://jcalculator.sourceforge.net/

```
6       //...
7       //#ifdef GUI
8       JFrame frame = new JFrame(title);
9       initResolution(frame);
10      //#endif
11
12      //#ifdef LOOKANDFEEL
13      initLookAndFeel(frame);
14      //#endif
15
16      //#ifdef GUI
17      frame.addWindowListener(new WindowAdapter(){
18          public void windowClosing(WindowEvent e){
19              System.exit(0);
20          }
21      });
22      JCalcStandardFrame myFrame = new JCalcStandardFrame();
23      JPanel myPane = myFrame.getPane();
24
25      frame.getContentPane().add(myPane, ...);
26      frame.pack();
27      frame.setVisible(true);
28      //#endif
29  }
```

Listing 3.1: JCalc.java

As we can see, this method has three features: `PT_BR`, `GUI`, and `LOOKANDFEEL`. Notice that the features are tangled along the *main* method. Also, the feature `GUI` is scattered across the method twice.

Now, suppose that a developer needs to maintain the feature `GUI`. First of all, she should look at the current feature to understand its entire role, achieving independent feature comprehensibility. To do so, she must get the emergent interfaces for each code encompassed with `#ifdef GUI`. The emergent interfaces related to the first and second `#ifdef GUI` statements, generated from intra-procedural analysis, are shown in Figure 3.1, respectively.

From now on, we only refer to emergent interfaces built out of intra-procedural analysis to better understand the problems addressed in this work. We explain how EI compute

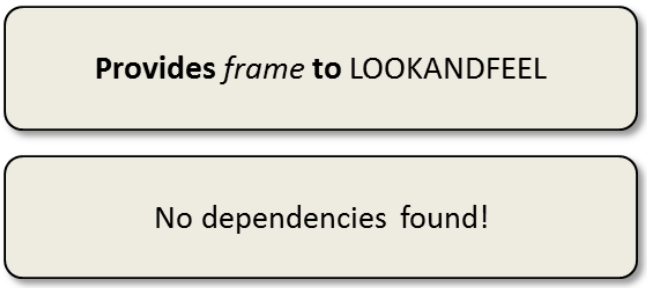


Figure 3.1: EI for `#ifdefs GUI`

dependencies in Section 2.4. The first emerged information alerts the developer that she should also analyze the feature `LOOKANDFEEL`. Under the VSoC perspective, the developer does not know that this dependency exists since VSoC hides feature code not relevant to the current maintenance task, in this case, the maintenance in the feature `GUI`. So, when using EI, the developer is aware of the dependencies that include the other feature parts. It is important to stress that `GUI` is scattered in other classes as well, but we omitted them for simplicity.

Besides that, the emergent interface informs the developer that there is no feature dependencies involving the feature `GUI` and the remaining ones. However, this information misses that `GUI` requires the title variable, which is defined in `PT_BR`, as can be seen in Listing 3.1. In other words, EI do not take into consideration required interfaces.

In addition to the problem of required interfaces, EI have another limitation regarding the amount of preprocessor directives per feature. We illustrate that in a second scenario extracted from the text editor Vim.[3] Vim is a highly configurable text editor built to enable efficient text editing. The text editor Vim has approximately 385 800 LOC and contains 778 features [37].

Listing 3.2 depicts the source code for syntax highlighting in Vim. The *highlight_changed* function translates the 'highlight' option into attributes and sets up the user highlights. According to Listing 3.2, the *highlight_changed* function has many preprocessor directives (`#ifdefs`), which represent the features `USER_HIGHLIGHT` and `FEAT_STL_OPT`. Notice that these preprocessor directives do not follow follow a comment ("//") because C and C++ compilers come with C preprocessor (cpp) that natively recognizes preprocessor macros and statements.

[3] http://www.vim.org/

```
1  int highlight_changed(){
2      //...
3      #ifdef USER_HIGHLIGHT
4          char_u userhl[10];
5      #ifdef FEAT_STL_OPT
6          int id_SNC = -1;
7          int hlcnt;
8      #endif
9      #endif
10     //...
11     #ifdef USER_HIGHLIGHT
12         //...
13     #ifdef FEAT_STL_OPT
14         //...
15     #endif
16     //...
17     #ifdef FEAT_STL_OPT
18         highlight_stlnc[i] = 0;
19     #endif
20     //...
21     #ifdef FEAT_STL_OPT
22         struct hl_group *hlt = HL_TABLE();
23     #endif
24     //...
25     #ifdef FEAT_STL_OPT
26         //...
27     #endif
28     //...
29     #ifdef FEAT_STL_OPT
30         highlight_ga.ga_len = hlcnt;
31     #endif
32     #endif /* USER_HIGHLIGHT */
33     return OK;
34 }
```

Listing 3.2: The *highlight_changed* function from the text editor Vim

In this context, suppose that a developer should study the feature `FEAT_STL_OPT` in order to implement a user requirement. Thus, she has to make use of the emergent interfaces generated for each code block encompassed with `#ifdef FEAT_STL_OPT`. After

that, she needs to join all interfaces gotten previously. In this function, she would have to observe six interfaces (see Listing 3.2). This becomes worse as the scattering increases. Recently, researchers [40] analyzed 40 software product lines implemented in C and they claim that a significant number of features incur a high scattering degree and the respective implementation scatters possibly across the entire system. It would be often hard to the developers to join all emergent interfaces from every part of a feature. Thus, this process is time consuming and error-prone, leading to lower productivity.

Because of the potentially hard work to get all emergent interfaces, the developer might forget some relevant information for maintaining the feature under her responsibility. For instance, in our scenario, the developer might overlook the information that the feature `FEAT_STL_OPT` provides *hlt* pointer to another feature, as can be seen in Listing 3.2. As a consequence, she might introduce bugs in some SPL variant leading to late error detection [33], since we can only detect errors when we eventually happen to build and execute a product with the problematic feature combination. This means that the overall maintenance effort increases. All things considered, we have other problem related to emergent interfaces we only have access to partial, and possibly redundant, emergent interfaces at a time.

In addition to EI of feature parts, there is an information overload since the interfaces are computed one by one and, then, the developer has to join them. This joining process might be expensive and further some of these interfaces might have duplicate information. In face of that, the developer is susceptible to consider code unnecessarily, wasting time. For instance, consider Listing 3.3 and suppose we would like to maintain FEATURE_A using EI. To do so, first we need to select the line 3 and then the line 7 to get the emergent interfaces before indeed changing the code. However, the respective emergent interfaces present duplicate information, as it can be seen in Figure 3.2. The following section describes how we address these problems.

```
1 void method(){
2     #ifdef FEATURE_A
3         int a = 0;
4     #endif
5
6     #ifdef FEATURE_A && FEATURE_B
7         int b = a;
8     #endif
```

```
9
10      #ifdef FEATURE_C
11          m(b);
12      #endif
13  }
```

Listing 3.3: Code with overlapping features

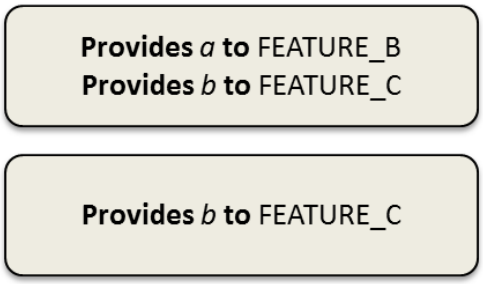


Figure 3.2: EI with duplicate information

3.2 THE CONCEPT OF EMERGENT FEATURE INTERFACES

To solve the aforementioned problems, we propose the idea of Emergent Feature Interfaces (EFI) [47], an evolution of the Emergent Interfaces approach, which consists of inferring contracts among features and capturing feature dependencies by looking at a feature completely. It is important to stress that our approach still has the benefits of EI. In addition to the notion of interface between parts of features, we provide developers to understand and reason about one complete feature. So, depending on maintenance task, the developer can use EI or EFI. This means that the developer should use EI if she knows the exact maintenance point, showing that she understands the feature code. Otherwise, she should use EFI to firstly understand the feature and reason about it before changing the code. From a notion of interface between features, we can detect feature dependencies by using sensitive-feature data-flow analysis [17]. So far, we do not detect all types of dependencies since we focus on capturing data dependencies, more precisely, def-use chains. But, our proposal can be extended to compute other kinds of dependencies, including exceptions, control flows, and approximations of pre and post conditions. That way, we use a broader term for contracts than "Design by contract" proposed by Meyer [49] since we infer the contracts by analyzing the code, and do not consider invariants, pre and post

conditions.

We establish contracts between the feature being maintained and the remaining ones through the interfaces. The concept of interfaces allows us to know what a given feature provides and requires from others. Considering the first interface of Figure 3.1, EI do not inform us about the required interfaces, but the feature `GUI` requires the *title* variable from the feature `PT_BR`. Therefore, we improve EI by adding required interfaces for computing the emergent feature interfaces.

In addition to establishing contracts, EFI obtain the existing dependencies between the feature we are maintaining and the remaining ones, taking into account the location of them. These dependencies occur when a feature shares code elements, such as variables, with others. In general, this happens when a feature declares a variable that is used in another feature. For example, in our first motivating example (see Listing 3.1), the variable *title* is initialized in `PT_BR` (alternative feature) and, subsequently, used in `GUI` (mandatory feature). We use feature-sensitive data-flow analysis [16] to perform the same analysis on all possible products without explicitly having to generate them. In doing so, we capture feature dependencies and, then, show them to the developers. In short, we keep data-flow information for each possible feature combination.

To clarify, we present how emergent feature interfaces work. Consider the example with regard to the JCalc product line of Section 3.1, where a developer is supposed to maintain the feature `GUI`. As our proposal derives from EI, the developer still needs to select the maintenance point but with a slight difference since she can also ask about a determined feature. In other words, the developer can select both a code block and a feature declaration (i.e. `#ifdef FeatureName`). The developer is responsible by the selection as illustrated by the dashed rectangle in Figure 3.3, in this case, `#ifdef GUI`. Then, we perform code analysis based on data-flow analysis to capture the dependencies between the selected feature and the other ones. Finally, the feature interface emerges.

The EFI in Figure 3.3 states that maintenance in the feature `GUI` may impact products containing `LOOKANDFEEL`. This means that `GUI` provides the actual *frame* value to the feature `LOOKANDFEEL` and requires the *title* value from `PT_BR`. Reading this information, the developer is now aware of the existing relations between the `GUI` and `LOOKANDFEEL` features and also between `GUI` and `PT_BR`. The emerged information has been simplified because the developers only need to know the dependencies inter-feature, not intra-feature. That is, EFI only show dependencies between the feature he is maintaining and the remaining ones. We can check this in Listing 3.1 where the variable *frame* is being

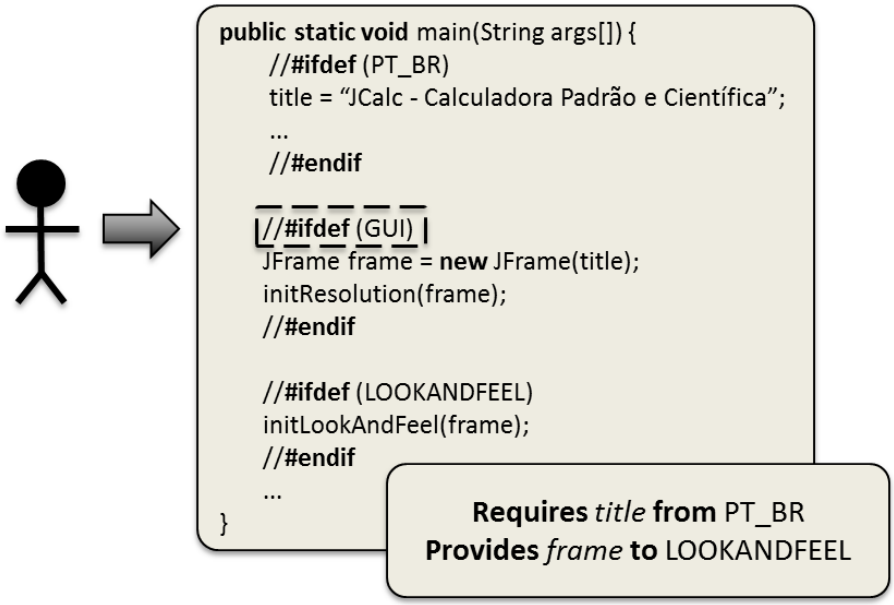


Figure 3.3: Emergent feature interface for the feature `GUI`

utilized in other places but the emergent feature interface (see Figure 3.3) only exhibits *frame* dependency concerning the feature `LOOKANDFEEL`. Also, we address the problem regarding required interfaces. Figures 3.1 and 3.3 show the difference between EI and EFI. Therefore, she focuses on information that should be analyzed avoiding considering code unnecessarily, wasting time.

It is important to highlight that EFI compute both *direct* and *indirect* dependencies among features. For example, consider a variable A is set in FEATURE_1 and used in FEATURE_2 to assign its value in other variable B (B = A). Finally, suppose that FEATURE_3 uses B. In this case, EFI notify the developer that FEATURE_1 has a direct dependency to FEATURE_2 (through A) as well as FEATURE_2 to FEATURE_3 (through B). Besides that, EFI provide the *indirect* dependency between FEATURE_1 and FEATURE_3 since FEATURE_3 depends on FEATURE_1 indirectly (transitivity property).

Note that the code might have many other `#ifdefs` making the interface's construction more complex. According to the results presented by the authors of the concept of EI, every SPL has methods with directives, but the percentage of methods with preprocessor directives vary across the software systems [59].

Assuming that a developer is unaware of a feature, and taking into account the features tend to crosscut the SPL code, it is better to look at the dependencies of an entire feature

instead of seeing them by part. In doing so, it is easier to the developers to understand a given feature through a macro vision than to get all interfaces one by one and, then, join them. For instance, in our second scenario (cf. Section 3.1) there are several `#ifdefs` and, in special, the feature `FEAT_STL_OPT` is scattered across the *highlight_changed* function (cf. Listing 3.2). Instead of the developer having to repeat six times the `#ifdef FEAT_STL_OPT` selection to get all interfaces, we provide an integrated way to avoid this information overload. This way, the developer only needs to select `#ifdef FEAT_STL_OPT` one time, then the data-flow analysis is performed and, finally, the feature interface is emerged (as shown in Figure 3.4). As we can see, no dependencies were found between `FEAT_STL_OPT` and the remaining features. Again, when reading this information, the developer already knows that the feature `FEAT_STL_OPT` neither requires any variable nor provides to other features.

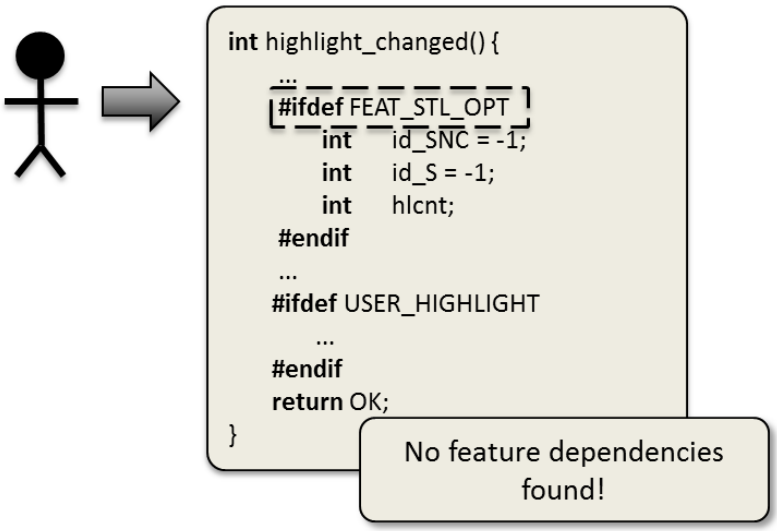


Figure 3.4: Emergent feature interface for the feature `FEAT_STL_OPT`

Therefore, our idea complements EI in the sense that we evolve this approach taking into account a feature as a module. In that sense, interfaces enable modular reasoning [36] since we can understand a feature in isolation without looking at other features. In addition to the notion of interface between parts of features, EFI provide a global interface that presents the existing dependencies of a given feature, preventing developers to miss any feature dependency and, consequently, to introduce errors into the SPL. This way, EFI help the developer to achieve independent feature comprehensibility. As a result, she can change a feature code aware of its dependencies, avoiding breaking the relations among

features [51]. This improvement is feasible and useful to improve modular reasoning for software product lines, with the potential of improving productivity.

We present how EFI work in terms of implementation in the next section.

3.3 IMPLEMENTATION

To implement our approach, we adapted a tool called Emergo, an Eclipse plugin, originally proposed by Ribeiro *et al.* [61] for improving maintainability of preprocessor-based product lines. It is available online at: `https://github.com/jccmelo/emergo`.

Figure 3.5 depicts both the extended Emergo's architecture and the data-flow from developer's selection up to the interface visualization. The architecture follows a heterogeneous architectural style based on the layered (GUI, Core, and Analysis) style and independent components. To show the process for getting EFI, we explain step by step the activity diagram-like (as seen in Figure 3.5).

First of all, the developer selects the maintenance point that indicates what code block or feature she is interested at the moment. Then, the component **GenerateEIHandler** sets up the classpath from the accessible information at the project. Besides that, it gets the compilation unit in order to know whether it is a Java or a Groovy project, treating each type of project in accordance with its peculiarities.[4] Meanwhile, we associate the maintenance point selection with a feature, or a combination of features, which we denote as a feature expression. Then, we compute the Abstract Syntax Tree (AST) from the Eclipse infrastructure. After that, we mark each tree node that represents the selected code by the developer. This marking of the AST nodes from text selection is important to bind AST node on Soot's `Unit` later. In Soot, the interface `Unit` represents a generic statement or command in any of the available intermediate representations, including Jimple.

Incidentally, Soot [71] is a framework for analysis and optimization of Java code. It accepts Java bytecode as input, converts it to one of the intermediate representations, applies analyses and transformations, and converts the results back to bytecode. Soot uses intermediate representations of programs, with the most prominent being Jimple, a typed 3-address representation designed for optimizations.

[4] `http://groovy.codehaus.org/Differences+from+Java`

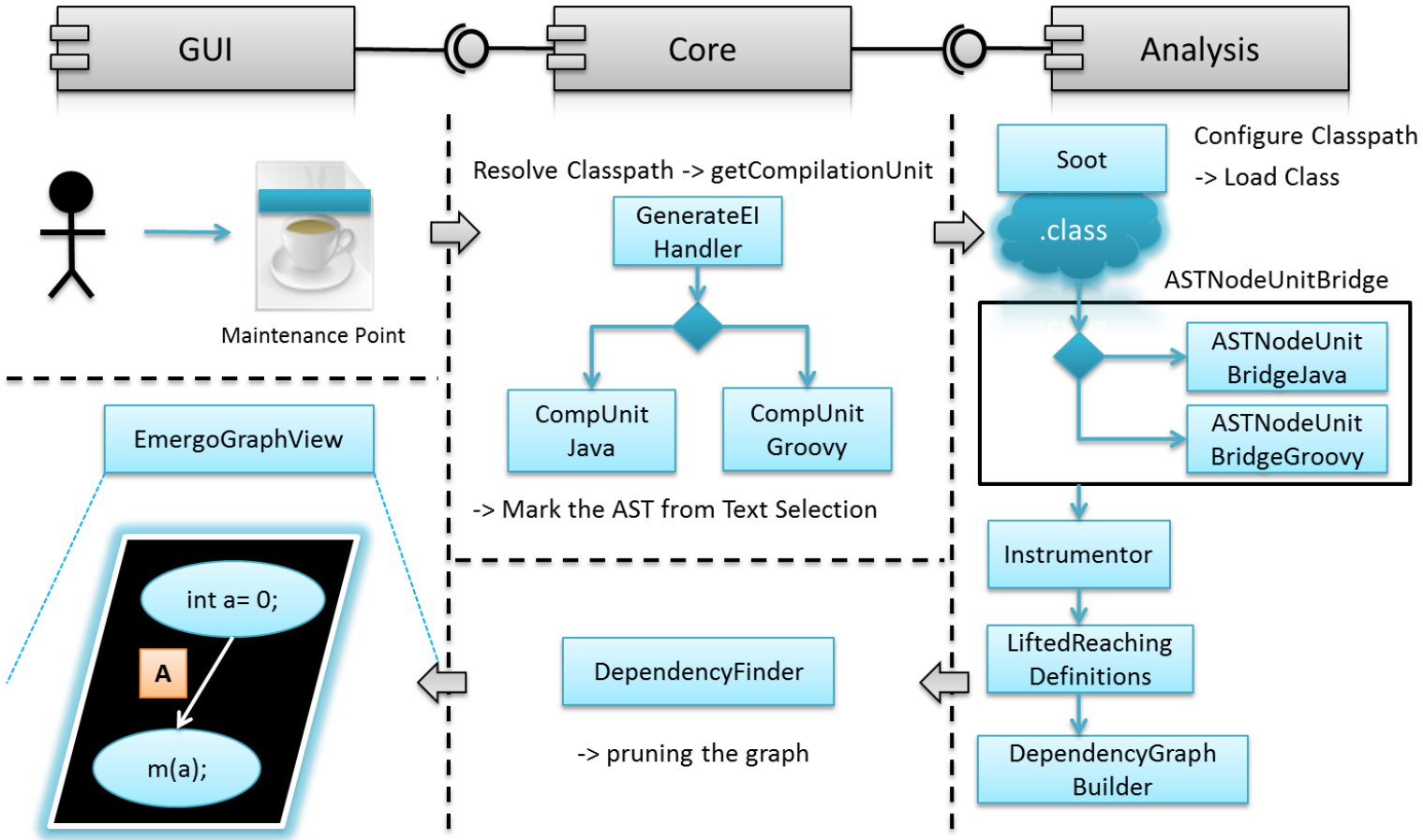


Figure 3.5: Emergo's architecture and activity diagram-like

Soot also provides a tagging mechanism that allows one to attach arbitrary information to `Units`. We exploit this to attach feature information to individual Jimple statements.

As we consider SPLs, we use the Soot framework to execute feature-sensitive data-flow analysis [17] and then capture feature dependencies. This data-flow analysis is feature-sensitive because it avoids to explicitly generate all products from an SPL, i.e., without using brute-force. That quickly becomes prohibitive as the number of possible products increases due to the combinatorial nature of SPLs.

Before we apply data-flow analysis, the component **Soot** gets the classpath provided by the **GenerateEIHandler** and configures it by putting all bytecode in a specific place. Then, **Soot** loads the class that the developer is maintaining. In other words, **Soot** gets the class bytecode and converts it into Jimple, which is the main intermediate representation of the Soot.

In addition to load the class, we use the bridge design pattern to deal the difference between Java and Groovy independently. This way, we can bind AST nodes on Soot Units, which correspond to statements. After this step, we have a mapping between AST nodes and statements and, hence, we are able to get the units in selection.

This mapping is passed to the **Instrumentor** that iterates over all units, looking up for their feature expressions, adding a new Soot `Tag` to each of them, and also computing all the feature expressions found in the whole body. `Units` with no feature expression receive an empty Soot `Tag`. The idea is to tag information onto relevant bits of code so that we can then use these tags to perform some optimization in the dependency graph at the end.

To clarify, from now on, consider the sample code snippet in Listing 3.4 that has two features (`A` and `B`). `A` holds a variable definition (`int a = 0;`), `B` in turn uses this variable.

```
1 public static void main(String[] args){
2     //#ifdef A
3     int a = 0;
4     //#endif
5
6     //#ifdef B
7     m(a);
8     //#endif
9 }
```

Listing 3.4: Sample code.

After compiling the source code to Jimple and instrumenting it using Soot, we get the tagged code as shown in Figure 3.6. Note that the Jimple statement (`b0 = 0;`) represents the assignment `int a = 0;` in the source code (cf. line 3). In addition, we can observe that the statement (`b0 = 0;`) belongs to a feature expression, in this case, `A`.

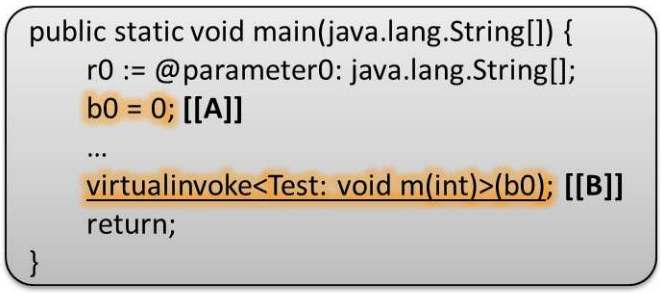


Figure 3.6: Jimple annotated.

The instrumentation process consists of annotating the statements that are encompassed by an `#ifdef F`, where F could be any feature or expression of features, with the boolean expression [[F]]. Generally speaking, an expression of features represents the set of configurations that a statement is bound to. Whenever a statement is not encompassed with preprocessor directives, i.e., it does not belong to any feature, so it should be present in all possible configurations.

After that, we build the Control Flow Graph (CFG) and, then, run reaching definitions analysis through the component **LiftedReachingDefinitions** that uses the Soot data-flow framework. The Soot data-flow framework is designed to handle any form of CFG implementing the interface `soot.toolkits.graph.DirectedGraph`. It is important to stress that our reaching definitions analyses are feature-sensitive data-flow analysis [17]. This way, we keep data-flow information for each possible feature combination. Figure 3.7 illustrates the result of the reaching definitions analysis for the code snippet in Listing 3.4. As we can see, our example has four possible product configurations, which represent the power set of the features that are inside the method. So, 0 represents a product without `A` and `B`, 1 for `A` selected, 2 for `B` selected and 3 for both features selected, i.e., `A` and `B`. Observe also that the value of `b0 (= 0)` reaches the method call to configuration 3, which means that the `A` and `B` features are defined to the product.

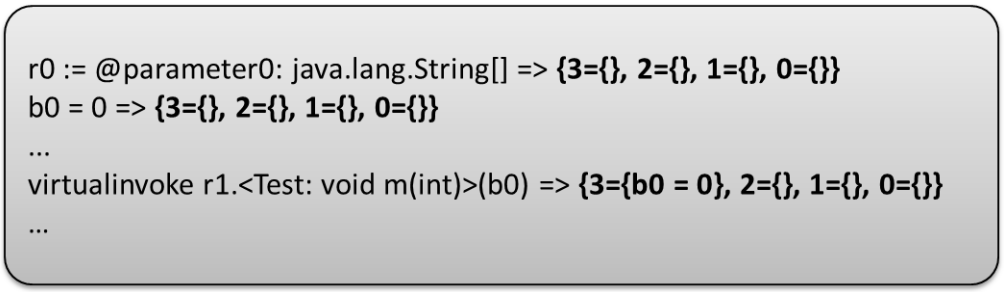


Figure 3.7: Reaching definitions analysis result.

Then, the component **DependencyGraphBuilder** accepts the mapping between AST nodes to statements, units in selection, CFG, and all possible feature combinations as input, iterates over the CFG for creating the nodes from units in selection, which represent an use or a definition. If the node is a definition we get all uses and for each use found we create one directed edge on the dependency graph which represents the EFI. Otherwise, we just get its definition and connect them. Recalling that both paths support transitivity property.

After the dependency graph is populated, we prune it to avoid duplicate edges and having more than one edge between two given nodes.

Finally, the component **EmergoGraphView** shows the dependency graph in a visual way where the developer becomes aware of the feature dependencies, preventing her to introduce errors in the SPL. Figure 3.8 depicts the dependency graph for our running example. The value of `a (= 0)` reaches `m(a)` only if the features `A` and `B` are selected. Like Emergo, besides this graph view, we also provide a table view. These information alert the developer about what interfaces might be broken if she changes the code in specific places.

Thus, selecting the statement `int a = 0;` as a maintenance point, the emergent feature interface for this selection is presented in Figure 3.9. In this case, the interface would be the same when selecting the feature declaration (`#ifdef A`). This interface informs the developer that the `B` feature can be impacted if she changes the *a* variable belonging to the `A` feature.

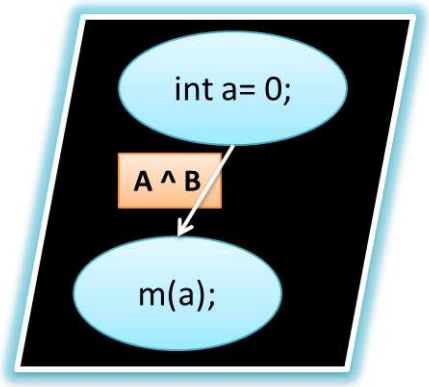


Figure 3.8: Dependency graph.

To sum up our contributions on Emergo, we remove the Johnni Winther's compiler,[5] a variability-aware Java compiler, and put the Soot framework on its place, because Soot is being maintained by the community at Sable Research Group.[6] Then, as Soot is able to analyze Java bytecode, we enhance Emergo to capture feature dependencies both in Java and in Groovy code, since they compile to Java bytecode. Besides that, we deal with different aspects between Java and Groovy, making variability points on code level, as shown in Figure 3.5 (through the decision nodes).

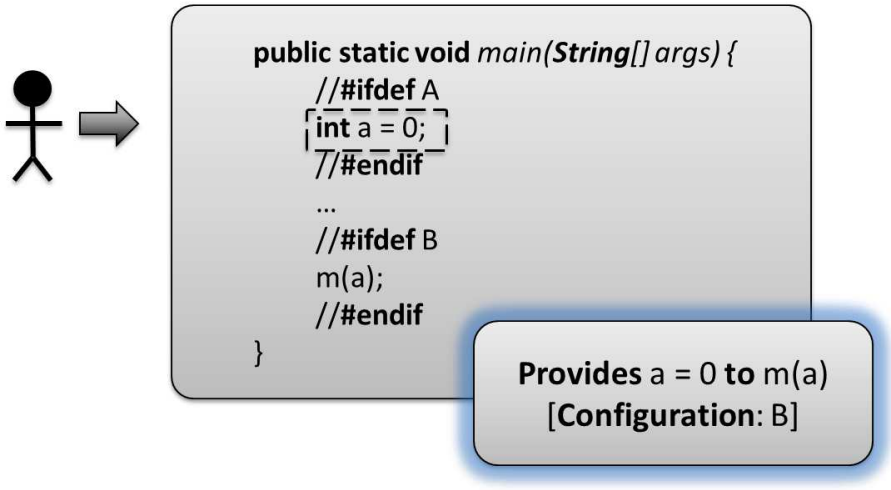


Figure 3.9: EFI for our running example.

[5] https://github.com/jccmelo/emergo/blob/master/trunk/Emergo%20-%20EI/lib/jw-compiler.jar

[6] http://www.sable.mcgill.ca/soot/

Figure 3.10 shows the adapted Emergo in execution. It is important to highlight that under the intra-procedural perspective all functionalities of the former Emergo are preserved. In other words, the developer can select a maintenance point (be it an assignment or not) as well as a feature, the latter can be seen in the screenshot.

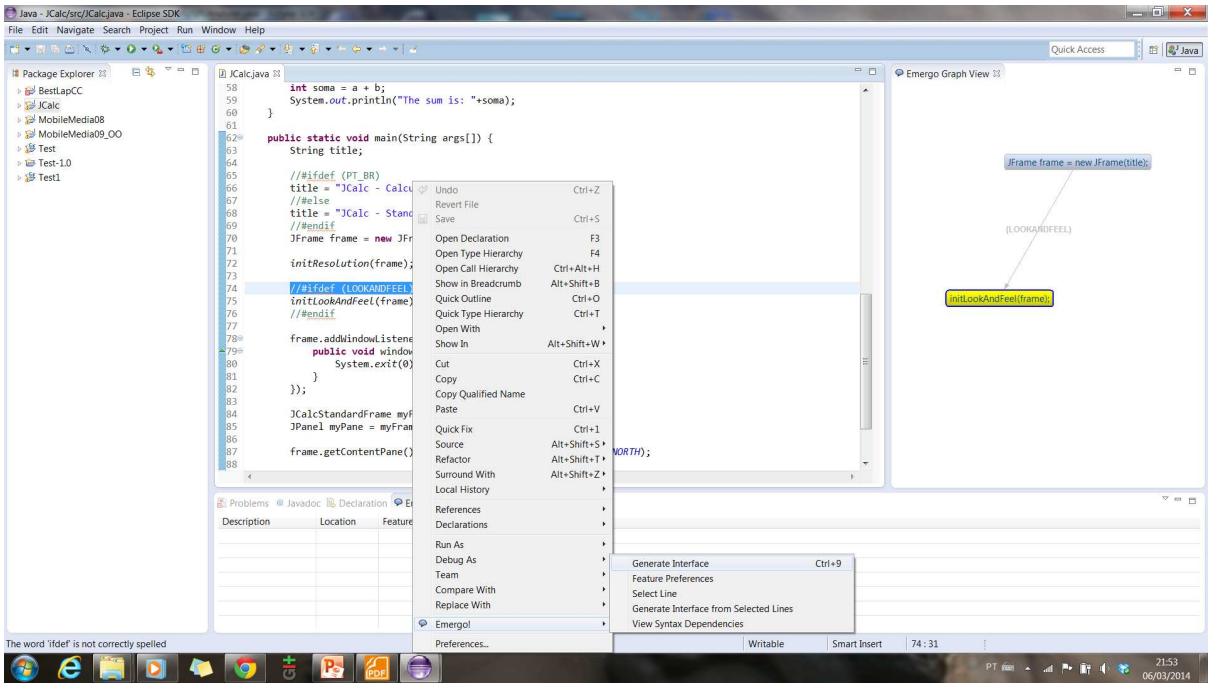


Figure 3.10: Extended Emergo's screenshot.

3.3.1 Limitations and Ongoing work

Our tool currently implements emergent feature interfaces, returning one integrated interface per feature. However, we so far consider simple preprocessor directives. That is, our preprocessor only recognizes feature expressions (e.g., `#if` and `#ifdef`) with one feature associated. Improving our preprocessor is an ongoing work.

Another limitation is related to capture emergent feature interfaces for Groovy software systems. Although Groovy uses a similar syntax to Java, Groovy has its particularities. In doing so, our tool knows about some Groovy statements like an assignment with the def keyword, commands without semicolons. But, for example, we have not supported lambda expressions yet.

Besides that, we capture only data dependencies among features, but our proposal can be extended to compute other kinds of interfaces, including dependencies related to

exceptions, control flows, and approximations of pre and post conditions. We also might use the AST to capture feature dependencies between a method declaration and its calls, for example.

We use Soot to execute feature-sensitive data-flow analysis [17] and then capture feature dependencies, but this analysis is intra-procedural. Thus, we need to plug in our tool to the framework Heros,[7] which is a general framework for solving inter-procedural problems in a flow-sensitive manner. This way, another ongoing work is to provide inter-procedural analysis to capture feature dependencies among classes, packages, and components since a feature can be scattered in different places.

Moreover, we should work in our algorithm to provide simplified views to the developers, dealing with possible huge interfaces.

3.4 EVALUATION

To assess the effectiveness and feasibility of our approach, we conducted a simple empirical study following guidelines from Runeson and Host [62]. Our evaluation addresses these research questions:

- **RQ1:** Is there any difference between Emergent Interfaces and Emergent Feature Interfaces in terms of size and precision?
- **RQ2:** How do Emergent Feature Interfaces' dependency detection compare to Emergent Interfaces?

3.4.1 Study settings

Our study includes five preprocessor-based systems in total. All of these software product lines are written in Java and contain their features implemented using conditional compilation directives. These software product lines contain several features. For instance, the *Lampiro* product line has 11 features, *MobileMedia* in turn contains 14 features [22]. *Lampiro* is a product line of instant messaging for mobile devices. *MobileMedia* is a product line that provides mobile devices manipulate photos, musics, and videos [25]. Among these systems, *Best lap* and *Juggling* product lines are commercial products. Hence, their source code is not available. At last but not least, *Mobile-rss* is a software product line related to mobile feed application domain.

[7] http://sable.github.io/heros/

We reuse the data produced[8] by other research [59], whose authors proposed the EI concept. Table 3.1 shows the product lines, including their characteristics such as the amount of preprocessor directives utilized in the entire product line and number of methods. We count preprocessor directives like `#ifdef`, `#ifndef`, `#else`, `#if`, and so on. *MDi* stands for *number of methods with preprocessor directives*, for example, *Mobile-rss* has 244 methods out of 902 that contain preprocessor directives (27.05%). *MDe* in turn stands for *number of methods with feature dependencies*. In particular, we use *MDe* metric to select the methods with feature dependencies to answer our research questions. According to the presented data in Table 3.1, these metrics vary across the product lines.

Table 3.1: Characteristics of the experimental objects [59].

System	Version	MDi	MDe	# methods	# cpp directives
Best lap	1.0	20.7%	11.95%	343	291
Juggling	1.0	16.71%	11.14%	413	247
Lampiro	10.4.1	2.6%	0.33%	1538	61
MobileMedia	0.9	7.97%	5.8%	276	82
Mobile-rss	1.11.1	27.05%	23.84%	902	819

Given a maintenance point in some of these product lines, we evaluate what is the difference between EI and EFI. Our aim consists of understanding to what extent the latter complement the former.

To answer our research questions, we randomly select a subset of methods with feature dependencies [59] and then compare the results produced by EFI to the results generated by EI. Note that the same set of selected methods is used to conduct the comparisons between EFI and EI. From these five experimental objects, we have 446 methods with preprocessor directives. We randomly select ten methods that contain feature dependencies. Firstly, we decide to pick two methods per product line because our study includes five software product lines and we try to balance the selection among them. We also select the maintenance points in a random way. In doing so, we identify some valid maintenance points dismissing comment, whitespace and method/class declaration since our data-flow analysis is intra-procedural. Then, we compute EI and EFI for each maintenance point chosen. To select these methods and maintenance points, we use RANDOM.ORG.[9]

[8] The results are available at: http://www.cin.ufpe.br/~mmr3/gpce2011

[9] http://www.random.org/

3.4.2 Results and Discussion

After discussing the study settings, we present the results of our evaluation for each method with feature dependencies as shown in Table 3.2. For each method selected, the table shows EI and EFI produced from the maintenance points. It is important to quote that depending on maintenance point selection the method might have no dependency among features. Although these ten methods contain feature dependencies, there are two cases where no dependencies were found, that is, these variables in selection are not neither used nor defined in another feature.

As can be seen, EI always return 'No dependencies found!' in every case that the maintenance point is not an assignment. Thus, we can confirm that EI miss required interfaces. On the contrary, EFI take into consideration both provided and required interfaces. Yet, in a worst-case scenario when a method does not have feature dependencies the two approaches return the same result (empty interface). For instance, there is no difference between EI and EFI for the *Resource.save* method (see Table 3.2). Although this method has no dependencies, EFI look for the definition of the *playerID* variable across the method in order to alert the developer about the required interface (backward contract) between the feature she is maintaining and the other. In this case, EFI returned 'No dependencies found!' because *playerID* variable is defined at the same feature that contains its use. Otherwise, EFI would return 'Requires *playerID* variable from feature X' where X represents the feature name. This type of case happened, for example, at the constructor of the class *ResourceManager* where EI did not find any dependencies whereas EFI did.

Besides that, EI do not provide support for feature selection as a maintenance point. This is bad since the developer might want to understand a feature as a whole before applying any change in the code. For example, consider the Best lap product line's *MainScreen.paintRankingScreen* method, if the developer wanted to know what feature dependencies exist between the feature `device_screen_128x128` and the remaining ones, she should select all parts of the feature (one-by-one). This is a potential hard work depending on the amount of fragments (`#ifdefs`) of the feature. In this context, our approach is useful and feasible since EFI provide macro information per feature, improving modular reasoning for software product lines (see the first line of the Table 3.2).

Another important aspect is the simplified view that EI do not offer to the developers. For instance, the method *PhotoViewController.handleCommand* has an *imgByte* declara-

tion encompassed with `#ifdef sms || capturePhoto`. This variable is used in different places (`sms || capturePhoto` and `copyPhoto`). EI show both use places in their message whereas EFI only alert the developer about dependencies outside the current feature configuration. This way, the developer only needs to worry about `copyPhoto` since she is aware of the feature she is maintaining. Thus, we think that a global interface may help developers, improving the modular reasoning for software product lines.

In summary, we believe that when the number of feature dependencies increases, our approach is better than EI because the probability of finding at least one required interface increases as well. Moreover, whenever the developer needs to understand a specific feature (i.e., to see its interfaces), the EFI approach is a good option. Thus, the answer to the first research question, concerning about the difference between EI and EFI, is yes in cases where the maintenance point is not an assignment, including a particular occasion when the developer selects a feature such as `#ifdef device_screen_128x128`. On the other hand, in some cases, our approach loses in terms of size (cf. Table 3.2) because we compute a global interface for each selected feature. However, EFI have more precise information than EI. The second question has already been responded along the previous paragraphs.

Table 3.2: Evaluation results.

System	**Method**	**Maintenance Point**	**EI**	**EFI**
Best lap	MainScreen.paintRankingScreen	#ifdef device_screen_128x128	Do not provide support for this selection!	Provides *rectBackgroundPosX, rectBackgroundPosY, positionPosX, loginScorePosX, etc* values to root feature.
Best lap	Resources.save	dos.writeUTF(playerID);	No dependencies found!	No dependencies found!
Juggling	TiledLayer.paint	firstColumn = (clipX - this.x)/this.cellWidth;	Provides *firstColumn* value to game_tiledlayer_optimize _backbuffer feature.	Provides *firstColumn* value to game_tiledlayer_optimize _backbuffer feature.
Juggling	Resouces.load	playerLogin = dis.readUTF();	No dependencies found!	Requires *dis* variable from root feature.
Lampiro	ChatScreen.paintEntries	int h = g.getClipHeight();	Provides h value to root feature.	Provides *h* value to root feature and requires *g* variable from root feature.
Lampiro	ResourceManager.ResourceManager	while ((b = is.read()) != -1) {...}	No dependencies found!	Requires *is* variable from GLIDER feature.
MobileMedia	PhotoViewController.handleCommand	byte[] imgByte = this.getCapturedMedia();	Provides *imgByte* value to configurations: [sms || capturePhoto], and [copyPhoto && (sms || capturePhoto)].	Provides *imgByte* value to copyPhoto feature.
MobileMedia	SelectMediaController.handleCommand	List down = Display.getDisplay(...);	Provides *down* value to Photo, MMAPI and Video features.	Provides *down* value to Photo, MMAPI and Video features.
Mobile-rss	UiUtil.commandAction	m_urlRrnItem = null;	No dependencies found!	No dependencies found!
Mobile-rss	RssFormatParser.parseRssDate	logger.finest("date=" + date);	No dependencies found!	Requires *date* variable from root feature.

3.4.3 Additional analysis

We conduct another empirical study to get more evidence in order to assess the effectiveness and feasibility of our approach. To do so, we follow the process we describe previously.

We use the same five preprocessor-based systems. We randomly select other 24 methods with feature dependencies and then compare the results from EI and EFI. But, in this study, we select these methods according to the number of methods with dependencies (using the MDe metric) for each software product line. In other words, the selection of the methods to be analyzed is directly proportional to the amount of methods with feature dependencies. For example, we pick two methods of the *Best lap* product line that contains approximately 40 methods with dependencies, whereas we choose only one method of *MobileMedia* since it has 16 methods with dependencies. In the same way, we randomly select the respective maintenance points.

Table 3.3 presents the results of our additional evaluation. For each maintenance point, this table shows the EI and EFI outcomes, respectively. Again, we reinforce our claim from previous evaluation that EI accept only assignment as maintenance point (see the method *EncodingUtil.getEncoding* in Table 3.3). Developers cannot request interfaces for method calls, for example. EFI in turn support both assignments and method calls. Moreover, when using our approach, developers can request interfaces for a given feature to understand and reason about it completly. For example, the *Resource.load* method from *Best lap* contains a feature named *arena* that requires the *dis* variable (see the first line of the Table 3.3).

Generally speaking, we identify that a maintenance point can be a definition (e.g., `perfectKickCounter = 0;`), an use (e.g., `System.out.println(selectedIndex);`) or a feature declaration (e.g., `#ifdef ITUNES`). It is important to highlight that an assignment might provide a variable as well as require another one. For instance, the `lastWidth = width;` assignment provides `lastWidth` and requires `width`, ignoring feature information. We can check that in Table 3.3.

To sum up, we get more evidence to claim that our approach improves EI. Remembering that EFI still have the benefits of EI. This way, the developer can select a maintenance point directly if she knows the specific place that needs to modify (EI). In addition, she can request interfaces to firstly understand and reason about a determined feature before changing the code (EFI).

Table 3.3: Additional results.

System	Method	Maintenance Point	EI	EFI
Best lap	Resources.load	#if arena	Do not provide support for this selection!	Requires *dis* variable from root feature.
Best lap	GameScreen.gc_loadBVGResources	this.anim_backupTrackEdge Straight = new BVGAnimator(..);	No dependencies found!	No dependencies found!
Juggling	TiledLayer.setStaticTileSet	tileImages[index] = tile;	No dependencies found!	Requires *tile* variable from nokiaui feature.
Juggling	GameController.processGameUpdate	perfectKickCounter = 0;	No dependencies found!	No dependencies found!
Juggling	Ball.animateFlipBall	int totalFlipAnimation = 1;	Provides *totalFlipAnimation* value to root feature.	Provides *totalFlipAnimation* value to root feature.
MobileMedia	BaseController.goToPreviousScreen	if (currentScreenName == null || ..)	No dependencies found!	Requires *currentScreenName* variable from root feature.
Mobile-rss	HTMLParser.getTextStream	#ifdef DLOGGING	Do not provide support for this selection!	Requires *m_fileEncoding, m_docEncoding* variables from root feature.
Mobile-rss	EncodingStreamReader.Encoding StreamReader	m_fileEncoding = "UTF-8";	Provides *m_fileEncoding* value to DLOGGING feature.	Provides *m_fileEncoding* value to DLOGGING feature.
Mobile-rss	EncodingUtil.getEncoding	logger.severe(ce.getMessage(), ..);	No dependencies found!	Requires *ce* variable from root feature.
Mobile-rss	HTMLParser.parseStream	String elementName = super.getName();	No dependencies found!	No dependencies found!
Mobile-rss	HtmlView.handleError	Logger logger = Logger.getLogger("View");	No dependencies found!	No dependencies found!
Mobile-rss	RssReaderMIDlet.getTextItem	#ifdef DMIDP20	Do not provide support for this selection!	Requires *textLabel, text, etc* variables from root feature.
Mobile-rss	PageMgr.sizeChanged	lastWidth = width;	No dependencies found!	No dependencies found!
Mobile-rss	PageMgr.commandAction	#ifdef DTEST	Do not provide support for this selection!	No dependencies found!

System	Method	Maintenance Point	EI	EFI
Mobile-rss	KFileSelectorImpl.resetRoots	System.out.println(defaultDir);	No dependencies found!	Requires *defaultDir* variable from root feature.
Mobile-rss	FeedListParser.parseHeaderRedirect	m_redirect = true;	No dependencies found!	No dependencies found!
Mobile-rss	RssFeed.init	int CATEGORY = 7;	No dependencies found!	No dependencies found!
Mobile-rss	KFileSelectorImpl.displayAllRoots	String root = roots.nextElement();	Provides *root* value to DTEST feature.	Provides *root* value to DTEST feature.
Mobile-rss	RssReaderMIDlet.appendCompatBmk	String prevStore = rss1.getStoreString(true);	No dependencies found!	Requires *rss1* variable from DCOMPATIBILITY1 feature.
Mobile-rss	XmlParser.getAttributeValue	String value = currentElementData.substring(..);	Provides *value* to DLOGGING feature.	Provides *value* to DLOGGING feature.
Mobile-rss	Settings.load	int numRecs = 0;	Provides *numRecs* value to the DLOGGING and DTEST features.	Provides *numRecs* value to the DLOGGING and DTEST features.
Mobile-rss	UiUtil.getAddChoiceGroup	choiceGroup.setLayout(..);	No dependencies found!	Requires *choiceGroup* variable from root feature.
Mobile-rss	RssItunesItem.unencodedSerialize	#ifdef DITUNES	Do not provide support for this selection!	Provides *author, subtitle, and summary* values to root feature.
Mobile-rss	KFileSelectorImpl.openSelected	System.out.println(selectedIndex);	No dependencies found!	Requires *selectedIndex* variable from root feature.

3.4.4 Threats to validity

In this section we present the threats to validity of our simple empirical study.

3.4.4.1 Conclusion validity The metrics used in our evaluation do not suffice to measure directly to what extent our approach reduces the maintenance cost and improves the productivity. Although we cannot respond questions like "Is the SPL maintenance easier when using our approach than using EI?" and, "How feature dependencies impact on maintenance effort when using EI and EFI?" precisely, we believe that our study is an approximation to answer these questions because we provide more abstract and accurate interfaces than EI.

3.4.4.2 External validity Regarding external validity, we use only five software product lines and analyze 34 methods in total. In spite of that, we select the methods according to the number of methods with feature dependencies, using the MDe metric. Thus, the chosen methods are directly proportional to the amount of methods with dependencies. The results bring preliminary evidence about the feasibility and usefulness of the proposed approach. We hope that EFI improve EI for other software product lines. In this sense, we believe that the results might be the same for other methods with dependencies. However, we acknowledge that more studies are required to draw more general conclusions.

3.4.4.3 Internal validity To minimize selection bias, we randomly choose 34 methods and the maintenance points. Yet, we get a subset of the product lines presented by Ribeiro *et. al* [59] in order to test our tool. For this, all five product lines selected are written in Java. Another threat is that we do not have access to the feature model of all SPLs, so the results can change due to feature model constraints, but we test both approaches (EI and EFI) of equal manner. Besides that, all interfaces were built from a intra-procedural perspective. This way, the results would be different with inter-procedural analysis, or with other types of dependencies like method declarations/method calls, exceptions, and control flows. Moreover, we manually compute EI and EFI, as shown in Tables 3.2 and 3.3. This can contain some error, but we are familiar with these approaches and we still revise each generated interface twice.

CHAPTER 4

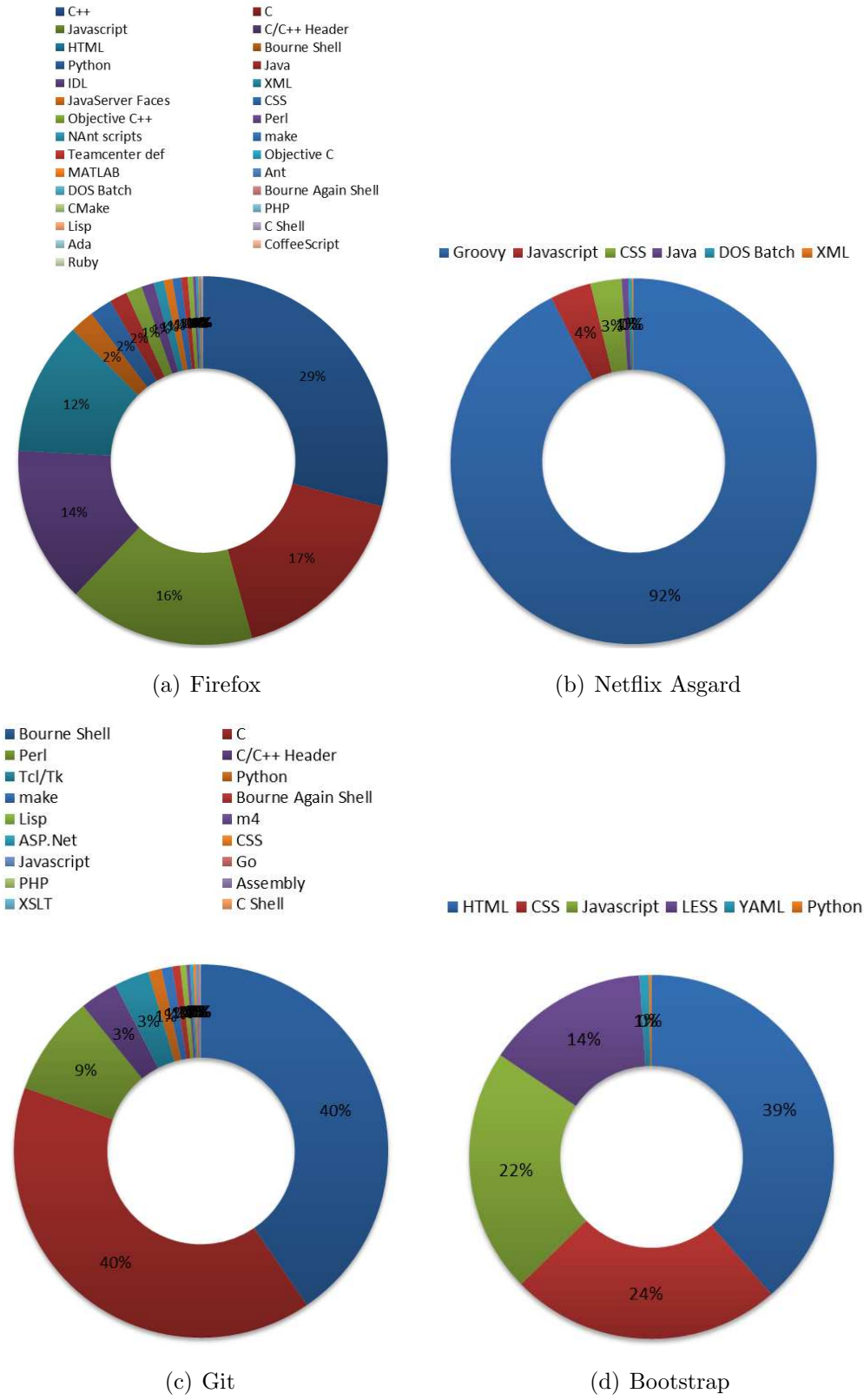
MULTI-LANGUAGE SOFTWARE SYSTEM ANALYSIS

The number of web applications has increased quickly over the last years. But, up to now, techniques to support the maintenance of these kinds of systems mostly focus on single language settings, whereas they are often constructed out of a multitude of artifacts written in different languages [50]. These systems need better forms of modularization and composition. Web development, in particular, retains an ad hoc character with many opportunities for improvement [63]. This can be even worse if these web applications are software product lines (SPLs) because they contain variabilities in the artifact level. In this sense, capturing feature dependencies between different kinds of artifacts is difficult. Thus, we still have the feature modularization problem (cf. Chapter 3), but now in a multi-language context. With this in mind, we propose a cross-language automated analysis for improving the maintainability of multi-language software product lines. To implement our proposal, we develop a prototype tool that uses the idea of emergent feature interfaces (EFI), described in Chapter 3, to improve the modular reasoning for multi-language software product lines based on the Grails framework.

In this chapter we discuss how to provide better support to maintain multi-language software product lines. We focus on researching web application characteristics to build technological foundations to allow enhanced maintenance support.

4.1 MOTIVATION

Software systems are becoming more decentralized, heterogeneous, and as a consequence, larger and more complex. They are built out of an amount of artifacts written in different languages. These artifacts represent all files that are modified in development time of a given software system such as files holding source code in diverse languages. In this sense, modern software systems are multi-language. For instance, Eclipse developers tend to use more than one language for their development work in a project, since approximately one third of them work with Java, C/C++, JavaScript, and PHP [7]. The survey further indicates that developers are using JavaScript as a secondary language.



(a) Firefox (b) Netflix Asgard

(c) Git (d) Bootstrap

Figure 4.1: Multi-language software systems and their language composition.

In addition, PHP developers report that they are using one to two additional languages, and again JavaScript is the favorite after PHP [8].

Figure 4.1 shows the language composition of four projects hosted by GitHub.[1] These projects have different domains, sizes, and languages. As we can see, both Bootstrap, a front-end framework for faster and easier web development, and Netflix Asgard, a web-based tool for managing cloud-based applications and infrastructure, contains six distinct languages. Mozilla Firefox in turn has more than 30 languages (see Figure 4.1(a)). We use the cloc[2] tool to count the number of languages for each project. According to this figure, all of these projects comprise various languages.

Other prominent example of a large multi-language system is the Linux kernel, containing around 25 languages.[3] Additionally, it has over 10,000 configurable features, providing a lot of variability with regard to hardware platforms and application domains [67].

Besides, complex enterprise software systems are built from several languages. For example, Dolibarr ERP & CRM,[4] an open-source software to manage a professional or foundation activity, uses more than ten languages, including PHP, JavaScript, Ruby, among others. OpenERP[5] in turn, an open-source enterprise resource planning (ERP) software, contains 19 languages including configuration files, build scripts, etc. The most recent versions of OpenERP are mostly implemented as a web application.

Over the last years, we are facing a rapidly growth of the importance and pervasiveness of web applications. According to the Eclipse community survey report [9], Web Applications and Rich Internet Applications are the primary types of software that the Eclipse developers are involved in developing. Nowadays, they are vital assets of modern enterprise software [13]. As concrete examples where their importance is visibly confirmed we have social networking, online banking, e-commerce, and so on.

However, these complex web applications contain development artifacts that have diverse types of dependencies among fragments of an artifact or entire artifacts. Breaking these dependencies will generate the same problems that we have already discussed in Chapter 3. That is, maintenance in a artifact/feature can affect another feature/artifact. However, these dependencies involving diverse artifacts are not captured by our previous solution. Thus, we need to extend our proposal to take into consideration other types of

[1] https://github.com/
[2] http://cloc.sourceforge.net
[3] Cloc count of http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.12.6.tar.bz2
[4] http://www.dolibarr.org/
[5] https://www.openerp.com/

dependencies such as cross-language dependencies.

In this work, we assume two possible kinds of dependencies. First, we call intra-artifact dependency when a statement (code fragment) refers to another statement in the same artifact. Second, whenever code fragments placed in distinct artifacts refer to each other, we say that this dependency is inter-artifact. In addition, dependencies might happen even with artifacts of different programming languages. Figure 4.2 depicts the relationship between two code fragments in the same programming language. Both Member and Publication classes belong to the model layer, and a member can have many publications. Although this reference is in the same programming language, it is an inter-artifact dependency because it involves different files.

```
1 package rgms.member
2
3 import rgms.publication.Publication
4 import rgms.publication.ResearchLine
5
6 class Member {
7   static hasMany = [publications: Publication]
8   //...
9 }
```

Listing 4.1: Member.groovy

```
1 package rgms.publication
2 // imports omitted
3
4 abstract class Publication {
5     def title
6     Date publicationDate
7     //...
8 }
```

Listing 4.2: Publication.groovy

Figure 4.2: Relationship between two Groovy classes.

Besides that, as web applications become more widespread, sophisticated, and complex, these fragments might hold dependencies among them in different languages such as HTML-Java or Java-Groovy and vice-versa as well. For instance, an excerpt of the Netflix Asgard project's index page (Listing 4.3) refers to a fragment that belongs to a controller class (Listing 4.4). Figure 4.3 illustrates this inter-artifact dependency between GSP[6] and Groovy[7] code from Asgard.[8] The link tag establishes a relation with the *list* action, allowing the users to see all managed applications (cf. Figure 4.3).

In fact, this kind of dependency between different languages can be easily broken if a developer changes either dependence end. For example, imagine a developer renaming *list* on line 8 in index.gsp to something other than *list*. Assuming that the tests do not get this broken relation, when an end user clicks on this link the software system will throw an error saying the property '*list*' is not found in the ApplicationController class. This error message is only visible at runtime. This means that the dependency between GSP and Groovy code is now broken. Observe that a simple change in a string constant (e.g., renaming *list*) breaks a part of the system, making the registered applications inaccessible and the *list* definition unused (cf. line 10). In general, today's integrated development environments (IDEs) still do not support multi-language software maintenance, since they do not provide neither static nor dynamic analysis to verify broken dependencies. In our example, the developer needs to discover by herself that ApplicationController must be modified as well to accomplish the task. Thus, developers have to look at heterogeneous artifacts and reason about such dependencies without supporting tools. Most of the IDEs do not know about dependencies between different languages [53]. Nevertheless, IntelliJ IDEA provides some cross-language support mechanisms [28]. It provides refactorings, code completion and error highlighting for specific language combinations, e.g., HTML and CSS.[9] However, these mechanisms are only offered for some exclusive language combinations since refactorings are completely tied to a particular language for example [28].

[6]Groovy Server Pages (GSP) is a presentation language for web applications, similar to JSP.
[7]Groovy is an object-oriented programming language for the Java platform.
[8]https://github.com/Netflix/asgard
[9]http://www.jetbrains.com/editors/html_xhtml_css_editor.jsp?ide=idea

```
1  <html>
2  <head>
3    <meta name="layout" content="main"/>
4  </head>
5  <body>
6      <!-- ... -->
7      <td>
8          Manage <g:link controller="application" action="list" title="...">
               Applications</g:link>
9          <!-- ... -->
10     </td>
11     <!-- ... -->
12 </body>
13 </html>
```

Listing 4.3: index.gsp

```
1  package com.netflix.asgard
2
3  // imports omitted
4
5  class ApplicationController {
6
7      def applicationService
8      //...
9
10     def list = {
11         UserContext userContext = UserContext.of(request)
12         List<AppRegistration> apps = applicationService.
               getRegisteredApplications(userContext)
13         //...
14     }
15     //...
16 }
```

Listing 4.4: ApplicationController.groovy

Figure 4.3: Dependency between different languages.

To support multi-language web-based software maintenance, we can offer interfaces to the developers so that they can see the relationships involving interrelated artifacts in a global view of the system. For instance, suppose that in a web application there exists two development artifacts in different languages. In order to perform a determined task, a developer needs to modify one of them. Under today's tooling, she might not know about the existing dependencies between the artifacts due to the lack of interfaces informing about such relations that each artifact holds. Hence, the developer needs to analyze each artifact to be sure that the maintenance she performed does not break other artifacts. This manual work is error-prone, and she might forget to look at an indispensable artifact. This way, she might break some dependencies among different artifacts since she is not aware of the artifacts impacted. Most of the broken dependencies are only revealed at runtime, especially in web applications [53, 44]. Figure 4.4 presents a concrete example, but it is only for illustrating that there is a feature dependency between different programming languages. In this scenario, a developer changed the declaration of the method fillMemberDetails by adding one parameter as can be seen in Listing 4.5. But, she did not know that TestRecord fragment refers to the fillMemberDetails definition (see Listing 4.6). She could test before committing the code, but due to the combinatorial nature of software product lines, the problematic feature combination was not tested. Consequently, she committed the code and sent a pull request. Few days later, the integration team, responsible by the merges, discovered that the code submitted breaks one existing dependency between TestRecord and MemberCreatePage (Figure 4.4). The problematic pull request is available at: `https://github.com/spgroup/rgms/pull/199`. Although the IntelliJ IDEA IDE is able to detect this kind of issue, this problem was found in a web application software product line called RGMS, which aims to manage members, publications and research lines from a research group. We present the RGMS in more detail in Section 4.1.1.

This kind of problem, which involves diverse languages, challenges the developers because they need to deal with a lot of dependencies among heterogeneous artifacts. First of all, it is necessary to understand the existing cross-language dependencies before changing the source code. Note that it is not trivial to detect broken relations between heterogeneous artifacts without a supporting tool for maintaining multi-language software systems. To better maintain multi-language software systems, developers must at least know which constructs of each language relate to each other, reason about such dependencies and navigate along them in a heterogeneous context. With this in mind, appropriate tools for

improving the maintainability of multi-language software systems are desirable. However, it is complicated to implement this kind of tool because each multi-language software system has domain-specific dependencies, depending on the framework that is being used. It is difficult to design one tool for supporting software system maintenance and evolution in a generic way.

```
1  package pages.member
2
3  import geb.Page
4  import pages.GetPageTitle
5
6  class MemberCreatePage extends Page {
7      //...
8  -   def fillMemberDetails(String name,..) {
9  +   def fillMemberDetails(String name,..,String additionalInfo) {
10         //...
11     }
12     //...
13 }
```

Listing 4.5: MemberCreatePage.groovy

```
1  // imports omitted
2  //...
3
4  //#if($Record)
5      page.fillMemberDetails(name, username, email, university);
6  //#end
7
8  //...
```

Listing 4.6: TestRecord.java

Figure 4.4: Relationship between Java and Groovy code.

This can be even worse if these software systems are software product lines since they

contain variation points in their artifacts. Putting together multi-language characteristic with preprocessor-based artifacts, we have multi-language software product lines. It is more difficult to maintain them because the developers should be aware of the cross-language and feature dependencies. In the SPL context they need to handle with parts of the artifacts should be compiled or not based on preprocessor directives. Thus, aside from the multi-language problem, we have the feature modularization problem as described in Chapter 3. For instance, Figure 4.4 shows a scenario where a developer modified a method, belonging to a mandatory feature, to fix some bugs of the system but the optional feature `Record` uses this method (cf. line 5). Thus, some products work as expected and some do not, since the feature `Record` has a broken dependency. In other words, all products with `Record` will throw an error at compilation time. Therefore, a change in one feature might lead to errors in others. Moreover, these errors might cause both compilation and behavioral problems in the system [59]. In many cases, bugs are only detected by customers running a specific product with the affected feature [33].

The following sections discuss the problem of multi-language software product lines in more detail. We show the feature dependencies between heterogeneous artifacts through one example of a web application software product line.

4.1.1 RGMS

To better illustrate the feature modularization problem in a heterogenous collection of artifacts, we present two maintenance scenarios from a preprocessor-based multi-language software product line called RGMS.[10] RGMS stands for Research Group Management System, which was developed in a graduate course. The purpose of the advanced software engineering course from the Federal University of Pernambuco aims to teach graduate students how to build software product lines from scratch as well as from an individual system or a set of them. To achieve this goal, the students were requested to develop the RGMS product line. RGMS is a Grails product line that aims to manage members, publications and research lines from a research group. RGMS contains several features and its features are implemented using preprocessor directives. Figure 4.5 shows the languages used in the development of the RGMS, which has almost 40 KLOC.

[10] https://github.com/spgroup/rgms/

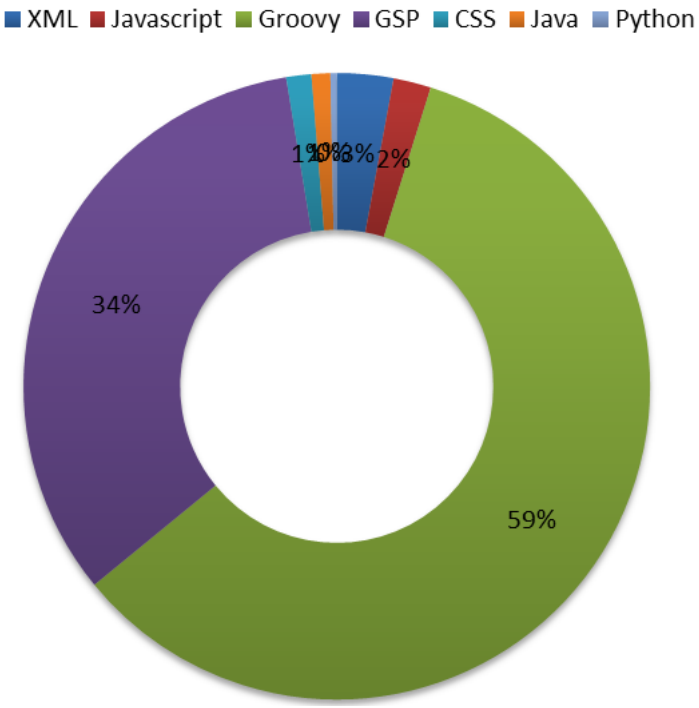


Figure 4.5: RGMS's composition of languages.

To clarify the understanding of the software product line, we present the RGMS feature model, based on extended FODA notation [30], in Figure 4.6.

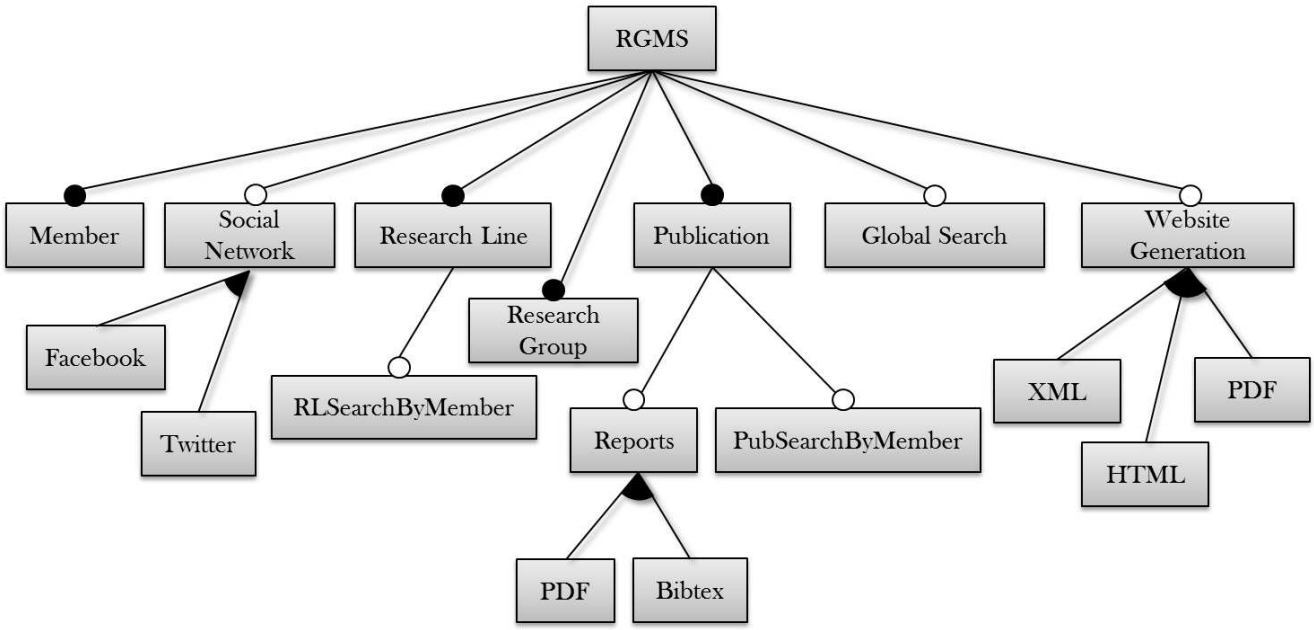


Figure 4.6: RGMS feature model.

The `Member, Publication, Research Line, and Research Group` features are responsible for managing these entities insertion, search and deletion from the system. Meanwhile, the `Reports` feature is responsible for publication reports in two different formats: `PDF or Bibtex`. The `Social Network` feature is responsible for publishing on `Facebook or Twitter` the activities of a research group. The `Website Generation` feature in turn allows the members to generate a website for their research group in three different formats: `XML, HTML and PDF`. Also the `RLSearchbyMember` feature is responsible for searching research lines related to a member registered on the system. Similarly, `PubSearchbyMember` retrieves the publications associated with the members of the research group. Lastly, `Global Search` retrieves members, publications, research lines and research groups.

RGMS is also a software product line of web applications. In fact, each RGMS product is a web application, since RGMS is based on the Grails framework. Web applications integrate different technologies such as scripting languages, and databases. Moreover, these technologies written in diverse languages are linked together such as a page holds a link to an action, which belongs to a controller class, and the latter in turn contains a `redirect`[11] statement that refers to other page. For example, consider Figure 4.7 to understand these types of dependencies between distinct artifacts. It shows that a RGMS page (*login.gsp*) has many relations with diverse artifacts like controller and domain classes. We highlight only two dependencies. First, a statement of the RGMS's login page (cf. line 3) refers to the username field of the class User. Note that this dependency is between a page and a domain class written in GSP and Groovy, respectively.

Suppose that a front-end developer needs to change the login page to perform a given task. After changing the name attribute (cf. line 3), she needs to talk to her fellow worker (i.e., a back-end developer assuming that exists this division) to reflect the change in User or she should modify the class User by herself. Otherwise, the tag form will break since end users cannot sign in to RGMS. To sum up, a simple change in a page fragment might affect parts of a software system as well as a system as a whole. In this case, login operation is not available until the change to be done in the second part of the dependence end (class User).

[11] This is a client-side redirection and tells the browser to request another page.

```groovy
package rgms.authentication

import rgms.member.Member

class User {
    String username
    //...
}
```

Listing 4.7: User.groovy

```
<g:form id="log" controller="auth" action="signIn">
    <!-- ... -->
    <g:textField name="username" required="true" />
    <!-- ... -->
    <g:submitButton name="signIn" value="Sign In"/>
    <g:link action="lostPassword">I have lost my password</g:link><br/>
    <g:link action="register">Create an account</g:link>
</form>
```

Listing 4.8: login.gsp

```groovy
package rgms.authentication
// imports omitted

class AuthController {

    def signIn = {
        //...
        if (user.passwordChangeRequiredOnNextLogon) {
            render(view:'resetPassword')
        } else {
            render(view: 'initial')
        }
    }
    //...
}
```

Listing 4.9: AuthController.groovy

Figure 4.7: Dependencies between GSP and Groovy code.

The second example is similar to the first one, but it involves a page and a controller class. The attribute action of form refers to other fragment that belongs to AuthController class (cf. line 6). Notice that into signIn action exists two render statements that establish more relations with other pages. That is, if a developer renames the attribute action of the RGMS's login page (cf. line 1) she might introduce an unmatched error between login.gsp and AuthController.groovy. Observe that this dependency exists for all products, since it occurs among mandatory features. Consequently, the page resetPassword becomes a dead artifact because the unique reference to resetPassword lies on AuthController (cf. line 6). Thus, a maintenance in one page fragment might cause unsatisfied dependencies too, by violating the existing contracts among a variety of artifacts. These contracts tend to be undocumented and scattered throughout the code base [27]. Each contract has at least two artifact fragments involved. These fragments can be in the same file such as dependencies between HTML and JavaScript code, or in distinct files like relationships between HTML and CSS code. In this chapter, we focus on inter-artifact dependencies because it is harder to see them, since the developers might not be aware of the dependence ends when the relation involves separate artifacts. In some cases, developers still need to know a specific programming language to perform a determined task without causing problems in the system.

Besides, these artifacts can hold variability of a software product line, making its maintenance even more complicated. Figure 4.8 presents a real maintenance scenario extracted from the RGMS product line. In this scenario, a development team, responsible for fixing some bugs found in the social networking context, added an actionSubmit tag to allow the end users to share periodical information on Facebook. Notice that the attribute action of show.gsp (cf. line 4) has a reference to the share action definition in PeriodicoController (cf. line 7). In addition, this share method belongs to the `Facebook` feature. But, for some reason they forgot to encompass the actionSubmit tag with the `Facebook` feature. Thus, the button named 'Share on Facebook' will not work for products without `Facebook`.

In summary, preprocessor-based and multi-language characteristics together can make software maintenance even more complicated. The following section discusses our proposal to infer feature dependencies in a heterogenous context for better maintaining preprocessor-based multi-language software product lines. More precisely, we focus on capturing feature dependencies between heterogeneous artifacts in web-based software product lines that use the Grails framework.

```
1 <!-- ... -->
2 <fieldset class="buttons">
3     <!-- ... -->
4     <g:actionSubmit name="share" action="share" value="Share on Facebook" />

5 </fieldset>
```

Listing 4.10: Periodico/show.gsp

```
1  package rgms.publication
2  // imports omitted
3
4  class PeriodicoController {
5      //...
6      //#if($Facebook)
7      def share(){
8          def periodicoInstance = Periodico.get(params.id)
9          def user = User.findByUsername(SecurityUtils.subject?.principal)
10         //...
11         redirect(action: "show", id: params.id)
12     }
13     //#end
14     //...
15 }
```

Listing 4.11: PeriodicoController.groovy

Figure 4.8: Dependencies between preprocessor-based artifacts.

4.2 CROSS-LANGUAGE AUTOMATED ANALYSIS

In this section we present our solution to improve the maintainability of preprocessor-based multi-language software product lines. We focus on capturing feature dependencies between heterogeneous artifacts in web-based software product lines that use the Grails framework.

As described previously, contemporary web applications are multi-language. Under

the developer's perspective, these systems are complex to maintain since they contain a collection of heterogeneous artifacts, holding relations among them. Shaw [63] points out that better forms of modularization and composition are necessary, since web development, in particular, still retains an ad hoc character with many opportunities for improvement. This can be even worse if these web applications are software product lines because they contain variabilities in their artifacts. In this sense, capturing feature dependencies between different kinds of artifacts is difficult. That is, we still have the feature modularization problem (as described in Chapter 3), but now in a multi-language context. Therefore, we propose an automated technique to support Grails software product line maintenance, which infers the relationship between heterogeneous development artifacts, analyzes each dependency taking into account feature information, and detects unsatisfied dependencies. For example, an unmatched code element between a page (.gsp) and a controller (.groovy).

From now on, we present how our cross-language automated technique works considering the maintenance scenarios shown previously. Consider the example extracted from Netflix Asgard project of Section 4.1, where an excerpt of the index page refers to other belonging to a controller class. We classify this type of dependency as inter-artifact because it involves distinct artifacts that, in their turn, were developed using different programming languages (see Figure 4.3). We said that if a developer changes either dependence end by renaming the attribute `action` in index.gsp (cf. line 8) to something else it might throw an error informing that the property '*list*' is not found in the ApplicationController class. Note that a simple change in a string constant (e.g., renaming *list*) might break the behavior of part of the system, in this case, making a link inaccessible. In addition, these unsatisfied dependencies are often discovered via error messages and in many cases at runtime, leading to lower productivity.

Our automated analysis uses the idea of emergent feature interfaces (cf. Chapter 3) to inform the cross-language feature dependencies to the developers, improving the modular reasoning for multi-language software product lines based on Grails. In doing so, the developer should select a maintenance point or a complete artifact (.gsp). After selecting the code, we run our cross-language analysis to capture the existing dependencies between the maintenance point and other artifacts taking into consideration feature information. As a result, the feature interface emerges.

Figure 4.9 illustrates the emergent feature interface for the Netflix Asgard scenario. The EFI states that a maintenance in the link tag might break the existing relation between the attribute `action` from the page and a fragment that belongs to a controller.

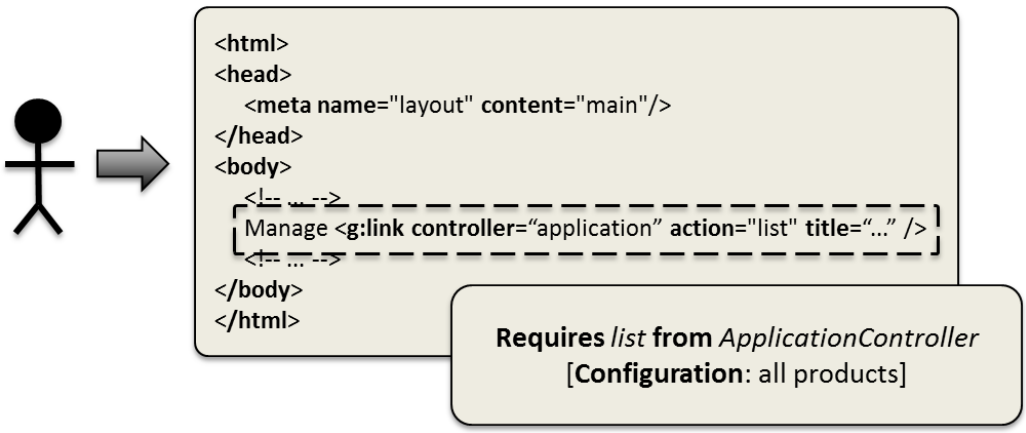


Figure 4.9: Emergent feature interface for the Groovy link tag.

Observe that this dependency exists for all products, since it occurs among mandatory features. After reading this interface, the developer is now aware of the existing contract between the *list* in index.gsp and the *list* method in ApplicationController.groovy. Thus, she knows that if it is necessary to alter the *list* method that lies on ApplicationController, she needs to take a look at the `action` attribute in index.gsp and vice-versa. Our technique provides information about the artifact a developer is maintaining and the remaining ones. In addition, the developer can decide to see the dependencies found or only the unsatisfied ones. This means that a developer of the maintenance team might only wish to see the unsatisfied dependencies to fix a given bug. This way, she can focus on relevant dependencies for her task at the moment.

Notice that aside from the multi-language characteristic, artifacts can hold variability, making the interface's construction even more interesting. For instance, we present an example that happened in practice (cf. Figure 4.4). A developer changed the declaration of the method fillMemberDetails by adding one parameter. After changing the code, she committed it. However, this maintenance caused an error in another feature, since she did not know that a fragment of the class TestRecord (cf. Listing 4.6) refers to the fillMemberDetails declaration in MemberCreatePage. Figure 4.10 depicts the result of our technique. The interface indicates that fillMemberDetails is used in the class TestRecord when the feature `Record` is defined. Seeing this information, the developer would know that TestRecord requires fillMemberDetails from MemberCreatePage class. That is, she could understand the existing cross-language feature dependency before changing the code. Also, this dependency involves mandatory and optional features. Thus, using our approach, she might detect potential broken relations between heterogeneous artifacts with variation

points. Developers need to understand the existing dependencies and, then, reason about that in order to maintain multi-language software product lines.

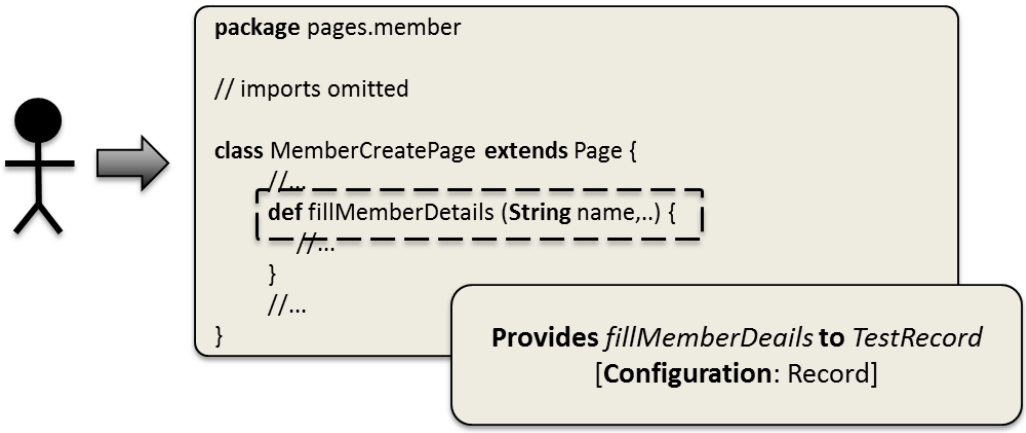


Figure 4.10: Emergent feature interface for an action definition.

To better illustrate, Figure 4.8 exhibits that the attribute `action` of the page show refers to the method share in PeriodicoController class. Besides, the share declaration belongs to the feature `Facebook`. However, a developer added an actionSubmit tag in a mandatory feature, introducing an unsatisfied feature dependency in the system because she did not pay attention that the method share is encompassed with preprocessor directives (`#if` and `#end`). Using our approach, she would have known that her maintenance is incomplete. Figure 4.11 shows the EFI for this case saying that the developer should encompass the action share in show.gsp with `#if($Facebook)`. In other words, she is now aware of that the button named 'Share on Facebook' do not work for products without `Facebook`.

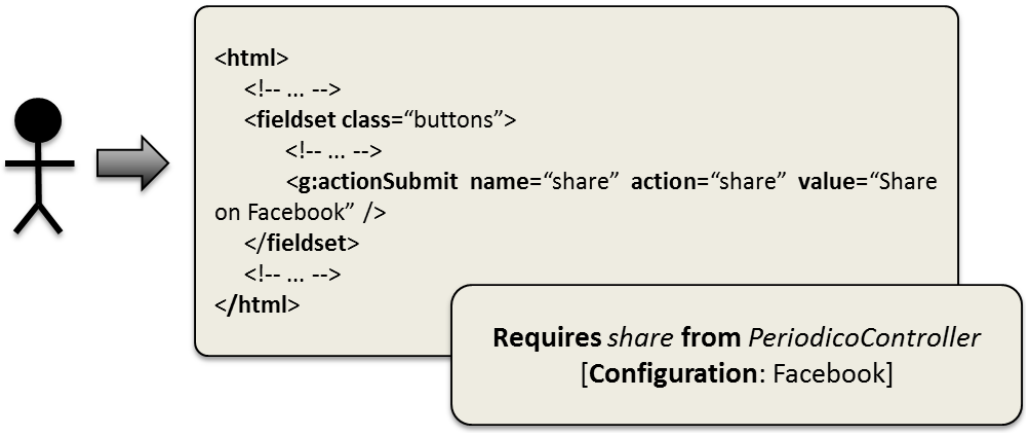


Figure 4.11: Emergent feature interface for the Groovy actionSubmit tag.

In general, for each tag selected we identify its respective page, controller and domain through Grails packaging and naming conventions. Afterwards, we preprocess them to know which statements belong to a feature and which do not. Then, we get the tag attribute values (e.g., *share*) and match these page attributes with the actual actions and fields of the classes (controller and domain) taking into consideration the feature information. Thus, if the set of actions and fields contains the current page attribute we classify this dependency as satisfied. Otherwise, this dependency is unsatisfied. Finally, we compute the EFI for the tag selected. The next section we explain more about it.

It is important to stress that an artifact can hold a variety of dependencies involving a collection of distinct artifacts. These artifacts in turn can have diverse references to other fragments and so forth. This case can be seen in Figure 4.7. We have worked to take into account this kind of transitivity in our analysis.

Therefore, our approach uses the idea of EFI taking into account feature information in an amount of heterogenous artifacts. The EFI alert about the existing cross-language feature dependencies to the developers understand and reason about the impact of a possible change in the source code during multi-language software product line maintenance. Thus, our technique is useful to achieve independent feature and artifact comprehensibility. As a result, the developer can change a feature or an artifact aware of the cross-language feature dependencies, avoiding breaking the contracts among features [51]. Our cross-language automated analysis also helps the developer to reason about feature dependencies in a heterogenous context on multi-language software product lines, with the potential of improving productivity.

The following section presents how we compute EFI to catch cross-language dependencies in terms of implementation.

4.3 IMPLEMENTATION

We have implemented our cross-language analysis in an open-source prototype tool called GSPAnalyzer,[12] that computes feature dependencies between a given web page and controller/domain classes. Our tool currently is not incorporated into the Emergo because this latter works only in a single language settings, but we should integrate it in the future. We use the Cobra Toolkit [3] to analyze web pages. Cobra is an HTML renderer and parser written purely in Java, which provides support for CSS and JavaScript. In Cobra,

[12]Available at: https://github.com/jccmelo/GSPAnalyzer

the class that performs parsing is `HtmlParser`, which is able to generate HTML DOM trees. Thus, in this work, we choose to use Cobra Toolkit because of its ability to parse HTML code.

After parsing a web page we get a generated tree. In doing so, we walk in the tree looking for Groovy tags to set the feature representation, which means that every tag is represented with an empty feature representation or with an existing one. To know what feature encompasses a given tag, we implement a variability-aware preprocessor[13] capable of preprocessing software product lines with directives such as `#if`, `#ifdef`, and `#endif`. Our preprocessor is able to preprocess both web pages and Java/Groovy classes on condition that these artifacts are following the standard of velocity [6] or antenna [1] annotation. In fact, all directives follow a comment ("//" for Groovy and Java, "<!−−" for HTML) that starts at the beginning of a line (whitespace is allowed left of them, but no code). This way, they do not interfere with normal compilation. We only preprocess the source files, but we do not compile them. So, we still have to run the Java compiler for example on the preprocessed files afterwards. Then, we get the relations for each tag of the page. For each dependency found, we classify whether it is satisfied or not (i.e., broken). For this, we identify the provider and requirer code elements and with a GroovyLoader object we obtain all actions and fields from a determined class. The code snippet in Listing 4.12 shows a method responsible for getting all actions from a controller class.

```
1  package br.ufpe.cin.grails.pages.analysis;
2
3  // imports omitted
4
5  public class GroovyLoader {
6      //...
7
8      public List<String> getActionsFromControllerClass(String classpath){
9          List<String> actions = new ArrayList<String>();
10
11         GroovyClassLoader gcl = new GroovyClassLoader();
12         Class clazz = gcl.parseClass(new File(classpath));
13         // exception handling omitted
14
```

[13] Available at: https://github.com/jccmelo/Preprocessor4SPL

```
15          Method[] methods = clazz.getDeclaredMethods();
16          for (int i = 0; i < methods.length; i++){
17              //...
18              actions.add(methods[i].getName());
19          }
20
21          return actions;
22      }
23  }
```

Listing 4.12: GroovyLoader.groovy

Then, we use an asset named configuration knowledge (CK) generated from the preprocessing phase. Listing 4.13 shows an example of configuration knowledge. We look for statements encompassed with preprocessor directives. After that, for each statement found we get the feature expression and the line number, respectively. In doing so, we generate the CK. Note that our CK holds a mapping of a feature, or a combination of features, which we denote as a feature expression to source code line. Thus, after preprocessing an artifact (be it a page or a class), we know which statements belong to a feature and which do not. This way, for each Groovy tag we get its attribute values (e.g., *share* as shown in Figure 4.11) and, then, we match these page attributes with the actual actions and fields of the classes (controller and domain) taking into consideration the feature information. Therefore, if the set of actions and fields contains the current page attribute we classify this dependency as satisfied. Otherwise, this dependency is unsatisfied.

```
1 <ck>
2     <config expression="A" line="[6, 14, 15]"/>
3     <config expression="B" line="[10, 21]"/>
4 </ck>
```

Listing 4.13: ConfigurationKnowledge.xml

Algorithm 1 presents the process to capture feature dependencies in a multi-language context, concerning what we described above. At the end, the algorithm returns the set of unsatisfied cross-language dependencies involving diverse features. We follow this algorithm to do our case study. It is important to quote that we consider the feature

model only with mandatory and optional features without any constraints.

Algorithm 1 Capture cross-language feature dependencies in a grails project

Input: a grails project
Output: feature dependencies

```
projectPath ← a grails pathname string
gspFiles ← GETGSPs(projectPath)
for gsp in gspFiles do
  PREPROCESS(gsp)
  dependencies ← GETDEPENDENCIES(gsp)
  unsatisfiedDependencies ← ANALYZE(dependencies)
  if brokenDependencies == true then
    view.SHOW(unsatisfiedDependencies)
  end if
end for
```

Figure 4.12 depicts a high level view of our tool, enabling us to abstract the irrelevant details and focus on the "big picture". First of all, our tool requires that the developer enters the project path to infer feature dependencies and then generate them as emergent feature interfaces.

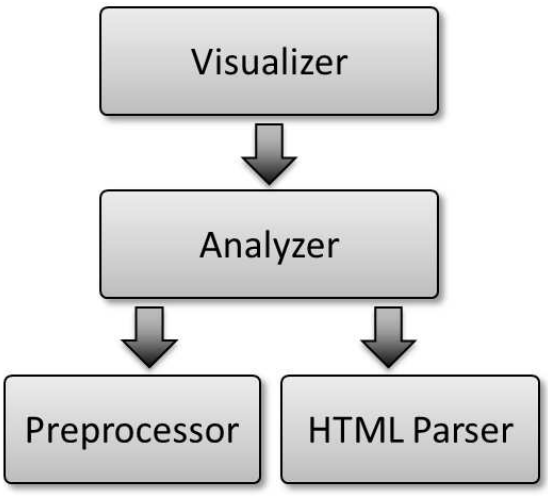


Figure 4.12: GSPAnalyzer architecture.

The **Visualizer** component receives the project path to be analyzed, and passes the source code location to the **Analyzer** component. However, to analyze the source

code and capture the feature dependencies we need to identify which code elements are encompassed by which features. For this, the **Analyzer** component depends on the **Preprocessor** component to preprocess each artifact instrumenting the nodes of the tree with feature information by saving it internally into a kind of configuration knowledge. Such tree is generated by the **HTML Parser** component. This information from the generated tree and the configuration knowledge is then retrieved to feed the **Analyzer** component to execute our cross-language analysis.

After executing cross-language analysis for each web page, we build the emergent feature interfaces. The **Analyzer** component is also responsible for this task. To do so, it takes the result of the analysis and crosses the obtained information with the controller and domain classes. In other words, for each page we join all dependencies belonging to the same feature and check whether a given dependency is broken or not. At last, the **Visualizer** component displays to the developer the dependencies found through emergent feature interfaces.

4.3.1 Limitations and Ongoing work

The idea behind our approach is to allow the developers to submit both an artifact and a specific maintenance point so that before changing the code they can know the feature dependencies among diverse languages. This way, they can reason about a determined artifact or feature of the system by looking at the emerged interface. However, in terms of implementation our tool only recognizes complete artifacts, but not fragments (i.e., maintenance points). Providing the ability for the developers to select a maintenance point is an ongoing work.

Another limitation is that our tool is so far available to capture cross-language feature dependencies of Grails software product lines. In other words, our tool currently only supports projects that use the Grails framework, i.e., it works for GSP, Groovy and Java code. But, we believe that our solution can be harnessed to deal with other frameworks such as JSF and Spring MVC. Up to now, GSPAnalyzer captures only dependencies from page to controller and domain, but does not in reverse order. In addition, we so far capture cross-language dependencies involving the Groovy `actionSubmit, fieldValue, form, if, and link` tags. Expanding to the remaining tags is another ongoing work. We observe that implementing such tools is challenging due to heterogeneous artifacts, and even more so due to dependencies between them, which are often domain-specific.

Moreover, we need to provide more types of analyses for capturing feature dependencies in a heterogeneous context. We do not capture dynamic feature dependencies (e.g., Javascript dependencies on demand) because our analysis is static. Our tool currently executes only syntactic analysis.

4.4 EVALUATION

In this section we describe a case study that evaluates the feasibility of our approach. We run our cross-language automated analysis on RGMS.[14]

We address the research questions below following the guidelines from Runeson and Host [62].

- **RQ1:** Are there cross-language dependencies in the RGMS?
- **RQ2:** What types of dependencies exist from GSP to Groovy/Java code?

Next, we present our study settings and then discuss the results.

4.4.1 Study settings

Our experimental object is a multi-language software system since RGMS contains seven languages. We only deal with three (GSP, Groovy, and Java) of the seven languages. However, GSP, Groovy, and Java cover 94% of the RGMS's composition of languages, as can be seen in Figure 4.5. We presented the RGMS feature model in Section 4.1.1, and according to the feature model (cf. Figure 4.6) and the configuration knowledge that we omitted for simplicity, we can understand that RGMS is a product line aside from multi-language system. RGMS has gradually been upgraded since it is utilized on the advanced software engineering course from the Federal University of Pernambuco for at least 2 years. For instance, in Figure 4.8 we presented a motivating scenario that a developer team was requested to fix some bugs involving the `Facebook` feature. The `Facebook` feature is a new one, actually it was developed at the end of 2013. Table 4.1 summarizes the statistics of the RGMS.

We consider all artifacts, except pictures and pdf files. Besides, we count LOC metric without including whitespaces and comments. Considered altogether RGMS has more than 46 KLOC.

[14] http://rgms.rcaa.cloudbees.net/

Table 4.1: Statistics of the RGMS.

Description	Value
no. of artifacts	371
no. of languages	7
no. of features	17
no. of lines of code	40026

Our research questions aim to understand whether there are feature dependencies between different languages in practice, and how we can classify these dependencies taking into account their characteristics. To gather such knowledge, we chose to pick up all web pages and then execute our tool to find feature dependencies between diverse languages. More precisely, we submitted 76 pages (.gsp) to the GSPAnalyzer in order to capture feature dependencies between GSP and Groovy/Java code, since our tool only enables to analyze relations between those languages. Up to now, GSPAnalyzer captures only dependencies from page to controller and domain, but does not in reverse order. The next section analyzes the results from our study and discusses some lessons learned.

4.4.2 Results and Discussion

We automatically compute cross-language dependencies for each .gsp file. All experimental data and results are available at the online appendix, including the RGMS version that we use. Figure 4.13 illustrates a Boxplot graphic concerning the amount of dependencies found and their distribution. We plot Boxplot to observe data dispersion based on the number of dependencies. Analyzing the results of our preliminary evaluation we find that two pages have no dependencies, whereas other pages contain more than ten dependencies. The maximum number of dependencies is 18 in periodico/show.gsp. This means that the number of cross-language dependencies vary across the artifacts (.gsp). We believe that this number of dependencies can be higher since we consider only five from dozens of Groovy tags. In addition, the quantity of dependencies that occurs most frequently is two, i.e., most of the pages have two dependencies. In summary, most of the pages contain cross-language dependencies.

Table 4.2 shows only the unsatisfied cross-language dependencies between diverse features. To identify these broken relations, we run our automated analysis.

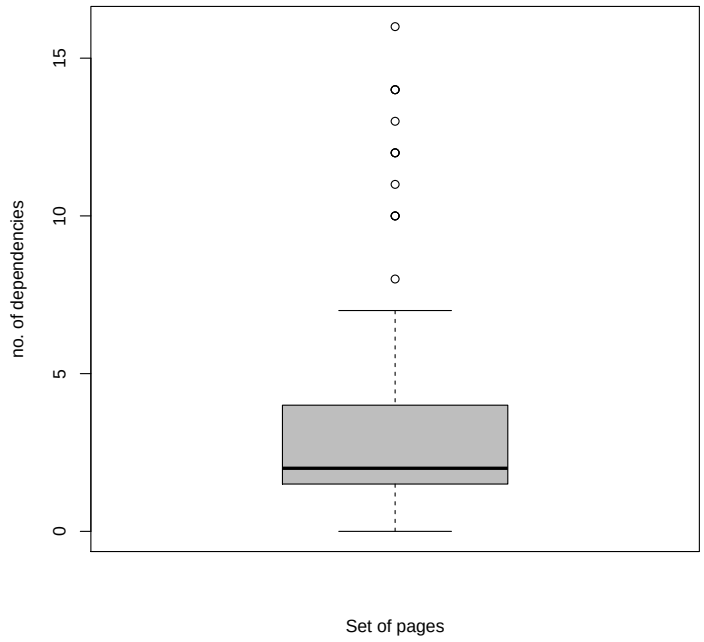


Figure 4.13: Boxplot graphic.

As we can see, the first artifact (bibtexGenerateFile/list.gsp) contains two unsatisfied dependencies. But, they do not involve two different features because the both page attributes (create and show) are neither present in the controller nor in the domain class. Thus, RGMS will throw an error when exercising these page fragments for all product configurations. Unlike those dependencies, the (periodico/show.gsp) file has one unsatisfied dependency involving a mandatory feature and other optional (`Facebook`).

Table 4.2: Unsatisfied dependencies found.

Artifact	**Unsatisfied Dependencies**
bibtexGenerateFile/list.gsp	Requires create from BibtexGenerateFileController Requires show from BibtexGenerateFileController [Configuration: all products]
periodico/show.gsp	Requires share from PeriodicoController [Configuration: Facebook]
researchLine/show.gsp	Requires publications from ResearchLine Requires members from ResearchLine Requires show from PublicationController [Configuration: all products]

```
<!-- ... -->
<li class="fieldcontain">
    <!-- ... -->
    <g:each in="${researchLineInstance.publications}" var="p">
        <g:link controller="publication" action="show" id="${p.id}" />
  </g:each>
</li>
```

Listing 4.14: researchLine/show.gsp

```
package rgms.publication
// imports omitted

class PublicationController {
    //...
    def index() { ... }

    //#if($Bibtex)
    def generateBib() { ... }
    //#end

    //#if($contextualInformation)
    def static Member addAuthor(Publication publication) { ... }

    def static Set membersOrderByUsually() { ... }

    def static Member getLoggedMember() { ... }
    //#end

    def upload(Publication publicationInstance) { ... }

    def static newUpload(Publication publicationInstance, ...) { ... }

    //#if($facebook)
    def static sendPostFacebook(Member user, String title){ ... }
    //#end
}
```

Listing 4.15: PublicationController.groovy

Figure 4.14: Unsatisfied feature dependencies

As shown in Table 4.2, we found three unsatisfied dependencies in show.gsp of the researchLine package. This means that the current version of the RGMS has problems. In Figure 4.14 we illustrate one of these problems. The Groovy link tag (g:link) holds a cross-language dependence, but taking a closer look at PublicationController we can check that there is no action named show. This specific case involves only a mandatory feature, since in the referred class there is no feature that encompasses show action. If it does, we could only have problems with some variants of the product line, depending on the feature selection.

Another example of unsatisfied dependencies lies on Figure 4.8 that we have already discussed. This example shows a cross-language dependence between a mandatory feature and an optional one. In other words, this relation is between different languages and features. The Groovy actionSubmit tag refers to the share action definition in PeriodicoController, but the former belongs to a mandatory feature whereas the latter is encompassed with `Facebook`. This means that this dependency is unsatisfied when `Facebook` is not selected. Therefore, the button named 'Share on Facebook' will not work properly for every product that does not have `Facebook` feature.

Table 4.3 summarizes how many dependencies we identified with our tool on RGMS. We answer the first research question saying that heterogeneous artifacts tend to hold feature dependencies. As can be seen, the number of automatically identified cross-language dependencies is 304. From 76 pages we found 304 cross-language dependencies. We have four dependencies per page on average. The number of relations vary across the artifacts. For instance, in lostPassword.gsp we found just one dependency whereas we found 14 in show.gsp from bookChapter package. Therefore, feature dependencies between different languages occur in practice, at least in this case. In addition, we believe that this number can be larger, since our tool only supports some Groovy tags.

Table 4.3: Evaluation summary.

Type of dependencies	**No. of occurrences**
Satisfied dependencies	304
Unsatisfied dependencies	6

Regarding the second research question, we classify the dependencies into two categories. In the first place, all dependencies are between different languages, i.e., GSP and Groovy/Java. The two basic types are: satisfied dependency and unsatisfied dependency.

The first one belongs to the class of potential problem for the software product line. Since, at a given moment, a developer might introduce errors in the system by breaking these satisfied relations. For this, we think that before maintaining the code, the developer should be aware of the satisfied dependencies in order to does not break any relationship. In other words, code maintenance can easily break a software product line or some specific products when she does not know about the existing relations. Unsatisfied dependencies in turn are actual problems for the system, so the developer need to fix these errors as soon as possible to that works properly. In fact, as RGMS is a product line of web applications, web applications that contain errors like pages that display incorrectly result in loss of revenue and credibility [18]. To detect these errors, RGMS uses the Cucumber[15] tool for running automated acceptance tests, which determine if the requirements of a contract are met. However, the results reveal that satisfied dependency is the most common type for the RGMS. Several unsatisfied dependencies might have been detected and fixed during the development phase. This fixing implies in additional cost of running and testing the product line again. In summary, we present the number of dependencies for each type in Table 4.3.

4.4.3 Threats to validity

4.4.3.1 Conclusion validity So far we cannot answer questions like "How helpful is our approach to improve the modular reasoning for multi-language software product lines?" and, "What is the possible impact that our approach causes in practice?" precisely, but we believe that our study is an approximation to answering these questions because we offer interfaces to the developers so that they can see the existing relations of a software product line. In doing so, they can know which artifacts have dependencies, reason about such dependencies and navigate along them, improving the modular reasoning for software product lines.

4.4.3.2 External validity We acknowledge that we need of other case studies with different sizes, purposes, architectures, granularity, and complexity to draw more general conclusions. However, the results bring preliminary evidence about the feasibility of the our cross-language automated approach.

[15] http://cukes.info/

4.4.3.3 Internal validity This threat concerns to the product line selection. Although RGMS is an academic software product line, during each semester it is implemented and improved by different developers, which are MSc or PhD students and some of them have industrial experience. Besides that, RGMS is a product line that fits into the characteristics we want to study in this work. It is a multi-language software system and its artifacts are implemented using conditional compilation. Thus, we believe that RGMS is a good system to begin with.

Besides, we automatically capture both satisfied and unsatisfied dependencies for each artifact. But, the unsatisfied dependencies can contain some error since our tool so far does not recognize the Groovy constructs `hasMany` and `belongsTo`. In other words, we do not get all fields from a given domain class if it has one of those constructs. We observe that this case happened in researchLine/show.gsp as shown in Table 4.2. From three unsatisfied dependencies found, two of them are not actual broken dependencies. But, this only happened in this isolated situation. Thus, we should fix this problem as soon as possible.

CHAPTER 5

CONCLUDING REMARKS

This dissertation presents how our approach to capture feature dependencies might be applied to maintain features in software product lines (SPLs), achieving independent feature comprehensibility. First, we present an approach that provides an overall feature interface considering all parts of a feature in an integrated way what we call Emergent Feature Interfaces (EFI), which complement Emergent Interfaces (EI). We also discuss our progress over EI by adding required and global feature interfaces, implemented in a tool called Emergo. After a selection, Emergo shows an EFI to the developer, keeping her informed about the dependencies between the selected feature and the other ones. We evaluate our proposal in terms of size and precision comparing with EI by using five SPLs. The results of our study suggest the feasibility and usefulness of the proposed approach.

Second, we use EFI for supporting the maintenance of web-based multi-language SPLs, since the number of web applications have increased quickly over the last years. Besides that, web applications retain an ad hoc character with many opportunities for improvement. Thus, we propose an automated technique that infers the relationship between heterogeneous development artifacts, analyzes each dependency taking into account feature information, and detects unsatisfied dependencies. To implement our technique, we developed a prototype tool called GSPAnalyzer that computes feature dependencies between a given web page and controller/domain classes. At last, we evaluate our technique with a multi-language product line named RGMS. The results bring preliminary evidence that exists feature dependencies between heterogeneous artifacts and these dependencies can be easily broken if a developer changes either dependence end.

Therefore, our work, which consists of emergent feature interfaces and cross-language analysis, may help the developers to understand a feature independently and to reason about it, with the potential of improving productivity.

5.1 SUMMARY OF CONTRIBUTIONS

In this dissertation we presented the following main contributions based on Chapter 3:

- The concept of Emergent Feature Interfaces to help developers when maintaining preprocessor-based software systems, allowing them reason about a feature modularly;
- Extension of Emergo to support our approach. It computes and shows EFI after developers select a given maintenance point, which might be a feature. Emergent Feature Interfaces provide global feature interfaces containing provided and required information and a simplified view of the existing dependencies;
- Comparison between Emergent Feature Interfaces and Emergent Interfaces in terms of size and precision.

Chapter 4 presented the following contributions:

- A technique to capture feature dependencies between heterogeneous artifacts;
- Implementation of our cross-language automated analysis: GSPAnalyzer;
- A case study that brought preliminary evidence concerning the feasibility of our approach to support modular reasoning for web-based multi-language SPLs.

5.2 LIMITATIONS

Features tend to crosscut the SPL code, and thus providing, or computing, a complete interface for them is difficult, if not impossible. The idea is to provide a global view of a feature abstracting irrelevant details. We focus on capturing data dependencies, but our proposal can be extended to compute other kinds of interfaces, including dependencies related to exceptions, control flows, and approximations of pre and post conditions. The feature modularization problem can be seen in any SPL, since features can be explicitly annotated on the code base or not (implicit). This way, our solution is over techniques for implementing features in an SPL. But, for the time being, our tool only runs on features implemented using conditional compilation.

We use an intra-procedural feature-sensitive data-flow analysis that is based on a single data-flow analysis: reaching definitions. For this, we cannot generalize our findings to other kinds of analysis. Specifically, we see in Section 3.3.1 that the results of our evaluation could be different if we use an inter-procedural analysis for example. In fact, the EFI would be huge with inter-procedural analysis depending on the product line

size. Thus, we intend to compute EFI on scope. For example, we could capture feature dependencies within a class, or a package, or even a component, instead of considering the entire product line. Regarding the performance on intra-procedural, we do not have problems since we choose one reaching definitions analysis with high performance, provided by Brabrand *et al.* [17].

Concerning our cross-language automated analysis, we so far provide to the developers to know feature dependencies of Grails software product lines (cf. Section 4.3.1). Besides that, we currently check only syntactic dependencies. Another limitation regarding web applications is that the static analysis is likely to give only an approximate picture, and dynamic analysis allows a proper understanding of complex and dynamic application behavior. With dynamic analysis, we can track other information such as the session and cookie data, and the type of link actually exercised (e.g., hyperlinks). Thus, we need to provide more types of analysis for capturing feature dependencies in a heterogeneous context.

5.3 RELATED WORK

We divide our related work according to the Chapters 3 and 4.

5.3.1 Feature Modularity

Many work investigate incorrect maintenance [26, 72, 64]. Sliwerski *et al.* [64] propose a way to find evidence of bugs using repositories mining. They found that developers usually perform incorrect changes on Friday. Anvik *et al.* [14] applied machine learning in order to find ideal programmers to correct each defect. On the other hand, Gu *et al.* [26] studied the rate of incorrect maintenance in Apache projects. The proposed work helps in the sense of preventing errors during SPL maintenance, since EFI would show the dependencies between the feature we are maintaining and the remaining ones.

Some researchers [41] studied 30 million code lines (written in C) of systems that use preprocessor directives. They found that directives, such as `#ifdef` and `#endif`, are important to write the code. But, the developers can easily introduce errors in the program. For example, open a parenthesis without closing it or even write a variable and then use it globally. This type of error (i.e., syntax error) is not common in practice [45]. But, beyond syntax error a developer might introduce semantic errors [31], which means behavior problems. In this work, we focus on interfaces for annotation techniques, more

precisely, conditional compilation to prevent these types of errors during SPL maintenance. We show some problems that can arise when maintaining features since a feature is likely scattered and tangled across the code. We propose the Emergent Feature Interfaces concept to improve the modular reasoning for SPLs and, then, we analyze five SPLs implemented with preprocessors. Additionally, we adapt the Emergo, an Eclipse plug-in, for helping the developers to avoid breaking feature contracts.

One widespread technique to implement features of an SPL is preprocessors, but it obfuscates the source code and reduces comprehensibility, making maintenance error-prone and costly. Virtual Separation of Concerns (VSoC) [35] has been used to address some of these preprocessor drawbacks by allowing developers to hide feature code not relevant to the current maintenance task. However, different features eventually share the same variables, so VSoC does not modularize features, since developers do not know about hidden features. Thus, the maintenance of one feature might break another. To minimize this problem, researchers propose the idea of Emergent Interfaces [58, 57, 61] to capture dependencies between part of a feature that a developer is maintaining and the others. Yet, they do not provide an overall feature interface considering all parts in an integrated way. As a consequence, the developer cannot safely understand and reason about one complete feature before changing the code. Our proposal complements this one by capturing dependencies among entire features and providing a global feature interface taking into consideration all parts of a feature. Thus, EFI prevent developers to miss feature dependencies and, consequently, introduce errors in the SPL, improving modular reasoning.

Another approach to capture dependencies between modules is Conceptual Module [15]. This approach allows the developers to define conceptual modules, set of lines of code as logic units, and to perform queries over them to capture other lines that should be part of a given module and to compute dependencies among other conceptual modules. Also, it uses flow analysis to compute the relations among conceptual modules. Generally speaking, we do the same with a slight difference, since we capture dependencies among entire features. That is, we compute EFI to inform the developer about the feature dependencies involving the feature that she is maintaining and the other features, allowing modular reasoning for SPLs.

Other researchers [46] propose analysis of exception flows in the SPL context. For instance, an optional feature signaled an exception and other feature handled it. When exception signalers and handlers are added to an SPL in an unplanned way, one of the

possible consequences is the generation of faulty products. Our approach has a different manner for improving the maintainability of SPLs. We detect the feature dependencies by executing feature-sensitive data-flow analysis in order to improve modular reasoning for SPLs when evolving them. We do not consider implicit feature dependency that occurs in an exceptional control flow, since we focus only on dependencies among annotated features (with preprocessor directives).

Finally, using the feature model, it is known that not all feature combinations produce a correct product. Depending on the problem domain, selecting a feature may require or prevent the selection of others (e.g., alternative features). Feature model is a mechanism for modeling common and variable parts of an SPL. Safe composition is used to ensure that all products of a product line are generated correctly [68]. Thaker *et al.* [68] determined composition restrictions for feature modules and they used these restrictions to ensure safe composition. However, safe composition only catch type errors, e.g., undefined class/method/variable. Our approach differs from safe composition because we use EFI to provide to the developers to see feature dependencies, preventing both type and semantic errors during SPL maintenance. Nonetheless, safe composition is complementary because the developer may ignore a feature dependency showed by our approach and, then, introduce a type error. So, safe composition approaches catch it after the maintenance task.

5.3.2 Cross-Language Analysis

Pfeiffer and Wasowski [55] introduced a taxonomy of design choices for multi-language development environments (MDLEs) by looking at how the current IDEs and programming editors provide development support in a single language settings. They observed that *visualization, navigation, static checking* and *refactoring* are implemented by all IDEs. To implement these characteristics in an MDLE, they developed TexMo, which is an editor that allows to interrelate source code in multiple languages. Further, the authors ran a controlled experiment with 22 participants in which they perform evolution tasks on a web application using TexMo [54]. Furthermore, they developed a set of tools to provide cross-language support for multi-language software systems [53]. However, they do not provide support to feature maintenance in an SPL context. Our approach differs from that because we take into consideration feature information. We propose a cross-language automated analysis to compute dependencies between diverse features in a

multi-language context. Then, we evaluate our approach using a web-based multi-language SPL called RGMS. Additionally, we develop GSPAnalyzer, a standalone open-source tool, to implement our technique, helping the developers to avoid breaking cross-language relations.

The QWickie tool [5] provides navigation and renaming support between interrelated HTML and Java artifacts. However, it only works with Wicket code. In other words, QWickie cannot be used to develop with other frameworks that accept HTML and Java code. Similarly, the IntelliJ IDEA IDE implements some multi-language development support mechanisms. It only provides multi-language refactorings for exclusive languages, e.g., HTML and CSS. Our tool is similar to these because it is able to capture dependencies between GSP, Groovy and Java code. In addition, we take feature into consideration in a heterogeneous context.

For web applications, Mesbah *et al.* [48] propose an automated technique to support styling code maintenance that analyzes the relationship between the CSS rules and DOM elements and, then, detects unmatched elements. This technique aims to assist CSS-based development by eliminating redundant CSS rules. They implemented their technique in a tool called CILLA, which detects unused CSS code. We differently develop a tool named GSPAnalyzer to capture cross-language dependencies in web-based multi-language SPLs. In addition, we consider feature relationships. But, our tool currently detect only cross-language dependencies in Grails SPLs, i.e., between GSP and Groovy/Java code.

To the best of our knowledge, capturing cross-language dependencies between different features from an SPL maintenance perspective has not been addressed in the literature.

5.4 FUTURE WORK

Our work can be enhanced and extended in several ways. In the first place, we intend to improve Emergo with more robust emergent feature interfaces. Also, we should provide inter-procedural analysis to our tool captures feature dependencies among classes, packages and components. Besides, we intend to compute EFI on scope due to the size of them when using inter-procedural analysis. For example, we could capture feature dependencies within a class, or a package, or even a component, instead of considering the entire product line. We also should conduct more studies, including a controlled experiment with developers, to draw more general conclusions. Although we do not conduct a controlled experiment involving developers in order to claim more precisely whether using EFI is

better than EI in terms of maintenance effort, we can claim (through our study) that EFI have potential benefits to prevent developers of introducing errors in the SPL, since EFI present more global and accurate information. The results of our study, on five SPLs, suggest the feasibility and usefulness of the proposed approach where the minimum result is equals to EI.

We evaluated our cross-language automated analysis using only the RGMS product line. We intend to evaluate our technique with more multi-language SPLs. To do so, we need firstly to improve our tool and integrate it into Emergo and, then, to conduct larger case studies to obtain more empirical data. Further, we will investigate how our analysis can be used for improving the maintainability of multi-language SPLs in practice.

Finally, we would like to register that there exists an open research question concerning distributed heterogeneous artifacts. In distributed development, a multi-language SPL is composed of artifacts that are distributed over diverse repositories on many different computers. So, the question is about how to capture cross-language dependencies in a distributed environment. Thus, we think that researches in this field is promising. Besides, the research community still has not researched how to perform static checking when parts of the information required for the checking is not available.

BIBLIOGRAPHY

[1] Antenna preprocessor. http://antenna.sourceforge.net/.

[2] CIDE. http://www.fosd.de/cide/, seen: Feb. 2014.

[3] Cobra: Java html renderer & parser. http://www.lobobrowser.org/cobra.jsp.

[4] Emergo. http://twiki.cin.ufpe.br/twiki/bin/view/SPG/Emergo, seen: Feb. 2014.

[5] Qwickie: Eclipse plugin for wicket. https://code.google.com/p/qwickie/.

[6] Velocity preprocessor. http://velocity.apache.org/.

[7] The open source developer report - eclipse community survey, 2011. http://www.eclipse.org/org/community_survey/Eclipse_Survey_2011_Report.pdf, seen: Jan. 2014.

[8] Zend technologies ltd.: Taking the pulse of the developer community, 2011. http://static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf, seen: Jan. 2014.

[9] The open source developer report - eclipse community survey, 2013. http://eclipse.org/org/press-release/20130612_eclipsesurvey2013.php, seen: Jan. 2014.

[10] Vander Alves. *Implementing Software Product Line Adoption Strategies*. PhD thesis, Federal University of Pernambuco, Recife, Brazil, 2007.

[11] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC)*, volume 3714 of *LNCS*, pages 70–81. Springer-Verlag, 2005.

[12] Michalis Anastasopoulos and Cristina Gacek. Implementing Product Line Variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR)*, pages 109–117. ACM Press, 2001.

[13] G. Antoniol, M. Di Penta, and M. Zazzara. Understanding web applications through dynamic analysis. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 120–129, 2004.

[14] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE, pages 361–370. ACM, 2006.

[15] Elisa L. A. Baniassad and Gail C. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the 20th International Conference on Software Engineering (ICSE)*, pages 64–73. IEEE Computer Society, 1998.

[16] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 13–24. ACM, 2012.

[17] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development X*, 2013.

[18] Margaret Burnett. What is end-user software engineering and why does it matter? In *Proceedings of the 2Nd International Symposium on End-User Development (IS-EUD)*, pages 15–28. Springer-Verlag, 2009.

[19] Marcelo Cataldo and James D. Herbsleb. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE, pages 161–170. ACM, 2011.

[20] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[21] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Proceedings of*

the 15th European Conference on Software Maintenance and Reengineering (CSMR), pages 191–200. IEEE Computer Society, 2011.

[22] Márcio de Medeiros Ribeiro. *Emergent Feature Modularization*. PhD thesis, Federal University of Pernambuco, Recife, Brazil, 2012.

[23] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170, 2002.

[24] J. M. Favre. Understanding-in-the-large. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC)*, pages 29–38. IEEE Computer Society, 1997.

[25] Eduardo Figueiredo, Nélio Cacho, Claudio Sant'Anna, Mário Monteiro, Uirá Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 261–270. ACM, 2008.

[26] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE)*, pages 55–64. ACM, 2010.

[27] Ahmed E. Hassan and Richard C. Holt. A visual architectural approach to maintaining web applications. In Kang Zhang, editor, *Software Visualization*, volume 734 of *The Springer International Series in Engineering and Computer Science*, pages 219–242. Springer US, 2003.

[28] Dmitry Jemerov. Implementing refactorings in intellij idea. In *Proceedings of the 2Nd Workshop on Refactoring Tools (WRT)*, pages 13:1–13:2. ACM, 2008.

[29] Pedro Matos Jr. Analyzing techniques for implementing product line variabilities. Master's thesis, Federal University of Pernambuco, Recife, Brazil, 2008.

[30] Kyo-Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[31] Christian Kästner. *Virtual Separation of Concerns*. PhD thesis, Universität Magdeburg, 2010.

[32] Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE Computer Society, 2008.

[33] Christian Kästner and Sven Apel. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[34] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *Proceedings of the 11th International Software Product Line Conference (SPLC)*, pages 223–232. IEEE Computer Society, 2007.

[35] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, 2008.

[36] Christian Kästner, Sven Apel, and Klaus Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC, pages 5:1–5:8. ACM, 2011.

[37] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. Partial preprocessing c code for variability analysis. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 127–136. ACM, 2011.

[38] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 49–57. IEEE Computer Society Press, 1994.

[39] Duc Le, Eric Walkingshaw, and Martin Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *IEEE International Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150, 2011.

[40] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

[41] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceeding of the 10th International Conference on Aspect Oriented Software Development (AOSD)*, pages 191–202. ACM, 2011.

[42] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[43] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys, pages 191–204. ACM, 2006.

[44] P. Mayer and A. Schroeder. Cross-language code analysis and refactoring. In *Source Code Analysis and Manipulation (SCAM), IEEE 12th International Working Conference on*, pages 94–103, 2012.

[45] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, pages 75–84. ACM, 2013.

[46] H. Melo, R. Coelho, and U. Kulesza. On a feature-oriented characterization of exception flows in software product lines. In *Software Engineering (SBES), 26th Brazilian Symposium on*, pages 121–130, 2012.

[47] Jean Melo and Paulo Borba. Improving modular reasoning on preprocessor-based systems. In *Software Components Architectures and Reuse (SBCARS), Seventh Brazilian Symposium on*, pages 11–19, 2013.

[48] Ali Mesbah and Shabnam Mirshokraie. Automated analysis of css rules to support style maintenance. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE Press, 2012.

[49] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[50] Jeff Offutt. Quality attributes of web software applications. *IEEE Softw.*, 19(2):25–32, 2002.

[51] David L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972.

[52] Thomas Patzke and Dirk Muthig. Product Line Implementation Technologies. Technical report, Fraunhofer Institut Experimentelles Software Engineering, 2002.

[53] Rolf-Helge Pfeiffer. *Multi-language Development Environments - Design Space, Models, Prototypes, Experiences*. PhD thesis, IT University of Copenhagen, Denmark, 2013.

[54] Rolf-Helge Pfeiffer and Andrzej Wasowski. Cross-language support mechanisms significantly aid software development. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 168–184. Springer-Verlag, 2012.

[55] Rolf-Helge Pfeiffer and Andrzej Wasowski. Texmo: A multi-language development environment. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA)*, pages 178–193. Springer-Verlag, 2012.

[56] Klaus Pohl, Gunter Bockle, and Frank J. van der Linden. *Software Product Line Engineering*. Springer, 2005.

[57] Márcio Ribeiro, Paulo Borba, and Christian Kästner. Feature maintenance with emergent interfaces. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014. To appear.

[58] Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent Feature Modularization. In *Onward!, affiliated with ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, pages 11–18. ACM, 2010.

[59] Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–32. ACM, 2011.

[60] Márcio Ribeiro, Társis Toledo, Paulo Borba, and Claus Brabrand. A tool for improving maintainabiliy of preprocessor-based product lines. In *Tools Session of the 2nd Brazilian Congress on Software (CBSoft)*, 2011.

[61] Márcio Ribeiro, Társis Toledo, Johnni Winther, Claus Brabrand, and Paulo Borba. Emergo: A tool for improving maintainability of preprocessor-based product lines. In *Proceedings of the 11th International ACM Conference on Aspect-Oriented Software Development (AOSD), Companion, Demo Track*, pages 23–26. ACM, 2012.

[62] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, 2009.

[63] Mary Shaw. Modularity for the modern world: Summary of invited keynote. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD, pages 1–6. ACM, 2011.

[64] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30:1–5, 2005.

[65] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer Technical Conference*, pages 185–198. Usenix Association, 1992.

[66] Thomas A. Standish. An essay on software reuse. *IEEE Trans. Softw. Eng.*, 10(5):494–497, 1984.

[67] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys, pages 47–60. ACM, 2011.

[68] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007.

[69] Társis Wanderley Tolêdo. Dataflow Analysis for Software Product Lines. Master's thesis, Federal University of Pernambuco, Recife, Brazil, 2013.

[70] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 191–200. ACM, 2006.

[71] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON*, pages 125–135, 1999.

[72] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 26–36. ACM, 2011.

APPENDIX A

ONLINE APPENDIX

In this appendix we put the following links to source code, repository, and data mentioned in this work:

- The JCalc product line code is available at: http://twiki.cin.ufpe.br/twiki/bin/viewfile/SPG/Emergo?rev=1;filename=JCalc.zip
- Emergo repository: https://github.com/jccmelo/emergo
- GSPAnalyzer repository: https://github.com/jccmelo/GSPAnalyzer
- The results of the multi-language experiment are available at: https://github.com/jccmelo/GSPAnalyzer/tree/master/experiment