



Universidade Federal de Pernambuco  
Centro de Ciências Exatas e da Natureza  
Centro de Informática - CIn

Pós-graduação em Ciência da Computação

**ANALYSING FEATURE DEPENDENCIES IN  
PREPROCESSOR-BASED SYSTEMS**

Felipe Buarque de Queiroz

DISSERTAÇÃO DE MESTRADO

Recife  
24 de Agosto de 2012

Universidade Federal de Pernambuco  
Centro de Ciências Exatas e da Natureza  
Centro de Informática - CIn

Felipe Buarque de Queiroz

**ANALYSING FEATURE DEPENDENCIES IN  
PREPROCESSOR-BASED SYSTEMS**

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

Orientador: *Prof. Dr. Sérgio Castelo Branco Soares*

Recife  
24 de Agosto de 2012

**Catálogo na fonte**  
**Bibliotecária Jane Souto Maior, CRB4-571**

**Queiroz, Felipe Buarque de**

**Analysing feature dependencies in preprocessor-based systems. / Felipe Buarque de Queiroz. - Recife: O Autor, 2012.**

**xiv, 67 p.: il., fig., tab.**

**Orientador: Sérgio Castelo Branco Soares.**

**Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2012.**

**Inclui bibliografia e apêndice.**

**1. Engenharia de software. 2. Linhas de produto de software. I. Soares, Sérgio Castelo Branco (orientador). II. Título.**

**005.1**

**CDD (23. ed.)**

**MEI2012 – 173**

*To my grandfather.*

## AGRADECIMENTOS

Agradeço e dedico este trabalho às pessoas mais importantes de minha vida: minha mãe Rosa, por seu amor incondicional, pelo carinho, afeto e sábias palavras nos momentos difíceis de minha jornada; meu pai Alano, pelo constante incentivo e preocupação com o homem e profissional que tenho me tornado; e minha irmã Alana, pelos poucos mas cruciais momentos de conversa e palavras de apoio. Amo vocês!

À minha namorada Raquel, por conseguir aturar as minhas chatices e mudanças de humor durante a realização deste trabalho. Obrigado por estar sempre ao meu lado, mesmo estando a mais de 400 km de distância na maior parte do tempo.

Ao professor e orientador Sérgio Soares, não apenas pelos conselhos e dicas valiosas que me ajudaram a conduzir esta pesquisa, mas pelo companheirismo e compreensão nos momentos complicados no decorrer do desenvolvimento deste trabalho.

Aos professores Fernando Castor e Uirá Kulesza, pelos valiosos comentários, que em muito enriqueceram este trabalho. Obrigado pelas críticas, sugestões e questionamentos, os quais serão bastante úteis para a realização de futuras pesquisas.

Aos amigos feitos durante o mestrado, em especial Liliane, Salânio, Henrique e Márcio, este último pelas intensas discussões e pelo constante apoio e incentivo, sem os quais este trabalho não poderia ter sido concluído. Agradeço também aos demais membros do SPG, bem como a participação nas valiosas discussões semanais realizadas no GENTeS.

Aos amigos que dividiram apartamento, em especial Fernando Kenji Kamei, Leonardo Fernandes e Raphael Borborema, os quais pude compartilhar momentos de stress e descontração durante os 2 anos e meio de realização deste trabalho.

À BankSystem Software Builder pela compreensão e flexibilidade no cumprimento de minha jornada diária de trabalho, a qual tive de compartilhar para a conclusão desta pesquisa. Além disso, às amizades construídas na empresa, em especial Henrique Seabra e Filipe Andrade, pelas discussões e momentos de aprendizado e descontração em conjunto.

Finalmente, agradeço à FAPEAL por financiar minha pesquisa.

*Go ahead and burn your bridges*

*But only if you can swim*

*Deep in the sea of disbelief*

*This time your screams won't be heard.*

—DEATH TO TRAITORS (Beloved, 2003)

## RESUMO

O conceito de Linha de Produtos de Software (LPS) tem se consolidado ao longo dos últimos anos como sendo efetivo para lidar com o reuso de componentes de software em larga escala, quando existe uma forte necessidade de customizações em uma família de aplicações. Diversas abordagens, como a utilização de pré-processadores, são levadas em consideração para implementar *features* em uma LPS. Features são abstrações essenciais que tanto os clientes quanto os desenvolvedores conseguem entender. Elas representam aspectos, qualidades ou características distintas importantes de um sistema.

Features usualmente compartilham elementos como variáveis e métodos entre si, o que leva à ocorrência de dependências a nível de código. Tais dependências podem ser caracterizadas como quando uma feature declara uma variável utilizada por outra feature. Assim, quando se está mantendo uma LPS, desenvolvedores estão sujeitos a manter uma feature e causar problemas em outra(s).

Visto isso, este trabalho busca entender a ocorrência de dependências entre features de uma LPS que utilizam-se de pré-processadores para implementar variabilidade. Seguindo guias de mapeamentos sistemáticos, foi realizada uma revisão da literatura onde foram identificados os principais estudos sobre o uso de pré-processadores em LPS. Com isso, foi possível realizar três estudos sobre a ocorrência de dependências em LPS, onde 45 LPS reais foram avaliadas. Os dois primeiros estudos são relacionados à ocorrência de dependências intraprocedurais, ou seja, a ocorrência de dependências entre features dentro dos métodos de um projeto. No primeiro, o interesse era em saber a frequência da ocorrência de dependências entre features nos projetos. Verificou-se que  $7.61\% \pm 7.22\%$  dos métodos dos projetos utilizam diretivas de pré-processamento e que  $72.49\% \pm 17.69\%$  dos métodos com diretivas contém dependências. No segundo estudo, o interesse consiste em saber se existe correlação entre complexidade e dependências entre features, bem como onde tais dependências ocorrem. Através dos valores das correlações entre as métricas relacionadas a complexidade, pudemos concluir que quanto maior a complexidade, maior o número de dependências entre features nos projetos. Em relação a onde tais dependências ocorrem, verificou-se que, em média,  $48.8\%$  das dependências ocorrem quando se utiliza a diretiva `#if`. O terceiro estudo complementa os dois primeiros

através de uma análise interprocedural, onde é comparado o esforço quando se utiliza ou não a abordagem de interfaces emergentes para realizar tarefas de manutenção em LPS. Para isso, nós medimos o esforço através do número de fragmentos e do número de features que o desenvolvedor teria de analisar para realizar a manutenção de uma variável compartilhada ao longo de várias features envolvidas em dependências interprocedurais. Verificou-se que desenvolvedores tem um ganho de  $56.28\% \pm 21.70\%$  em relação ao número de fragmentos e  $46.29\% \pm 19.73\%$  em relação ao número de features que é preciso analisar quando se utiliza interfaces emergentes.

Com o resultado das análises, uma grande quantidade de dados foi disponibilizada, a qual pode ser utilizada como entrada por pesquisadores em seus estudos, como projetos de linguagens de programação e ferramentas de suporte a pré-processadores.

**Palavras-chave:** Linhas de Produto de Software, Pré-processador, Compilação Condicional.



## ABSTRACT

The concept of Software Product Lines (SPL) has been consolidated over the past year as being effective for dealing with the reuse of software components on a large scale, when there is a strong need for customization in a family of applications. Several approaches, such as the use of preprocessors, are taken into account to implement *features* in a SPL. Features are essential abstractions that both customers and developers can understand. They represent aspects, qualities or important characteristics for a system.

Features usually share variables and methods with each other, which leads to the occurrence of dependencies at code level. These dependencies can be characterized when one declares a variable in a feature and this variable is used by another feature. So when developers are maintaining a SPL, they are subject to modify one feature and cause problems on other(s).

Thus, this work aims to understand the occurrence of dependencies between features of SPL that use preprocessor mechanisms to implement variability. Following systematic mapping guidelines, we conducted a literature review when the main studies on the preprocessor usage in SPL were identified. It was then possible to conduct three studies on the occurrence of feature dependencies in SPL, where 45 real SPLs were evaluated. The first two studies are related to the occurrence of intraprocedural dependencies, in other words, the occurrence of dependencies between features into projects methods. In the first, we are concerned to know the frequency of occurrence of dependencies between projects features. We found that  $7.61\% \pm 7.22\%$  of projects methods used preprocessor directives and that  $72.49\% \pm 17.69\%$  of these methods with directives have dependencies. On the second one, we are concerned to know if there are correlation between complexity and feature dependencies, and where feature dependencies occur. Through the correlation values between the complexity metrics, we can conclude that the higher the complexity, the greater the number of feature dependencies in projects. Regarding where the dependencies occur, we found that, in average,  $48.8\%$  of dependencies occur through the use of `#if` directive. The third study complements the other two through an interprocedural analysis, where we compare the effort to perform SPL maintenance tasks when emergent interfaces approach is used or not. To do this, we measure the effort through

the number of fragments and number of features that the developer needs to analyze to maintain a variable shared across features involved in interprocedural dependencies. We found that developers have a gain about  $56.28\% \pm 21.70\%$  over the number of fragments and  $46.29\% \pm 19.73\%$  over the number of features that need to analyze when he is using emergent interfaces approach.

From the analysis results, we provide a large amount of data, which can be used as input by researchers in their studies regarding programming languages and preprocessors tools support.

**Keywords:** Software Product Lines, Preprocessor, Conditional Compilation.

# CONTENTS

<b>Chapter 1—Introduction</b>	1
1.1 Summary of Goals . . . . .	3
1.2 Outline . . . . .	3
<b>Chapter 2—Background</b>	5
2.1 Software Product Lines . . . . .	5
2.2 Conditional Compilation . . . . .	8
2.3 Software Metrics . . . . .	10
2.4 Software Maintenance and Evolution . . . . .	12
<b>Chapter 3—Literature Review</b>	15
3.1 Review Protocol . . . . .	16
3.1.1 Search Process . . . . .	16
3.1.2 Research Questions . . . . .	17
3.1.3 Keywords & Search Terms . . . . .	17
3.1.4 Inclusion & Exclusion Criteria . . . . .	18
3.1.5 Data Extraction . . . . .	19
3.2 Execution . . . . .	20
3.3 Results . . . . .	20
3.3.1 Overview . . . . .	21
3.3.2 RQ1 - Research Topics . . . . .	22
3.3.3 RQ2 - Kinds of Evaluation . . . . .	27
3.3.4 RQ3 - Main Results . . . . .	29
3.3.5 Other Results . . . . .	32
3.4 Limitations . . . . .	32
3.5 Summary . . . . .	33

<b>Chapter 4—Dependency Analysis</b>	<b>34</b>
4.1 Feature Dependencies . . . . .	34
4.2 Empirical Studies . . . . .	37
4.2.1 Sample Projects & Collecting Data . . . . .	38
4.2.2 Metrics . . . . .	39
4.2.3 First Study: Frequency of Feature Dependencies Occurrence . . . . .	42
4.2.4 Second Study: Complexity and Feature Dependencies . . . . .	43
4.2.5 Third Study: Interprocedural Analysis . . . . .	46
4.2.6 Threats to Validity . . . . .	48
4.3 Summary . . . . .	49
<b>Chapter 5—Concluding remarks</b>	<b>53</b>
5.1 Contributions . . . . .	54
5.2 Related Work . . . . .	54
5.3 Future Work . . . . .	57
<b>Appendices</b>	<b>59</b>
A.1 Primary Studies . . . . .	59

## LIST OF FIGURES

2.1	Different models of mobile phones . . . . .	6
3.1	Search process and its steps . . . . .	16
3.2	Distribution of relevant studies by search engine . . . . .	21
3.3	Number of relevant studies by year . . . . .	21
3.4	Distribution of relevant studies by country . . . . .	22
4.1	Overview for the process of projects data generation . . . . .	39
4.2	Correlations between NoM and NoDe, and SLoC and NoDe metrics . . . . .	44
4.3	Correlation between NoFE and NoDe metrics . . . . .	45
4.4	Correlation between NoKDi and NoDe metrics . . . . .	45

## LIST OF TABLES

3.1	Keywords and their derived words . . . . .	18
3.2	Summary of execution data . . . . .	20
3.3	Authors who have published more than one paper about conditional compilation use . . . . .	22
3.4	Summary of papers by research topic . . . . .	23
3.5	Summary of main tool developed/used on research papers . . . . .	32
4.1	Classification of correlation values. . . . .	42
4.2	Dependency occurrence on <i>cpp</i> 's variability mechanisms . . . . .	46
4.3	Data Analysis - Part I. . . . .	50
4.4	Data Analysis - Part II. . . . .	51
4.5	Data Analysis - Part III. . . . .	52

## LIST OF LISTINGS

2.1	Conditional compilation use on mobile game. . . . .	9
2.2	Conditional compilation use on Linux Kernel. . . . .	9
4.1	Feature dependency occurrence on <i>Vim</i> text editor. . . . .	35
4.2	Feature dependency occurrence on <i>Kernel Linux</i> . . . . .	36
4.3	Feature dependency occurrence on <i>Irssi</i> . . . . .	37
4.4	Interprocedural Feature dependency occurrence on <i>Clamav</i> . . . . .	47

## CHAPTER 1

# INTRODUCTION

The notion of Software Product Lines (SPL) has been established over the past years as being effective in dealing with the reuse of software components on a large scale, when there is a strong need for customization in a family of applications of the same organization. Several approaches to develop SPL have been proposed in recent years and recent studies exhibit a growing interest in their industrial use. A wide variety of companies already has substantially decreased their costs with software development, maintenance and time to market, and increased the quality of their software products [Bos02].

A product line may be considered a family of software systems developed from reusable assets underlying the same architecture (known as core assets). By reusing assets, it is possible to construct products through features defined according to customers' requirements. On the other hand, implementation activities become more complex because they also have to realize variabilities. The product line systems share a common set of features that satisfy the needs of a particular market segment [CN02]. In this context, features are the semantic units by which we can differentiate programs in an SPL [TBD06].

Reasoning about how to combine both core assets and product variabilities is a challenging task [GA01]. In other words, the challenge consists of understanding the available mechanisms for realizing variability and knowing which of them fits best for a given variability at hand [KLM<sup>+</sup>97]. One of these mechanisms is the use of preprocessors to implement features of an SPL. This technique consists of associating conditional compilation directives like `#ifdef` and `#endif` to encompass the feature code. The use of conditional compilation to implement variability on programs occurs, in part, due to C preprocessor (*cpp*) popularity. *cpp* is a tool developed to provide better support to metaprogramming capabilities of C language, including, among other features, the definition of conditional code fragments. Because it is a command line-based tool, *cpp* may be used with any text artifact, including other programming languages such as Java and C#.

In order to know about the relevant studies on preprocessor usage, we perform a literature review based on systematic mapping features. We defined a review protocol, specified the research questions and the methods that were used to undertake the mapping. Through the results, we could notice that despite their widespread use to



implement SPL features [AJC<sup>+</sup>05, KAK08, KMPY05], preprocessors have several drawbacks, including no support for separation of concerns [SC92]. This has a direct impact on software modularity, which can negatively influence aspects like comprehensibility and changeability [Par72], and consequently, maintainability and evolution.

Modularity problems caused by preprocessor use arise because of shared elements among features such as variables and methods. Sharing these elements may lead to occurrence of dependencies between features. Feature dependencies occur like when a feature assigns a value to a variable which is subsequently used by another feature, or when a variable is instantiated on a feature and used by another ones. Furthermore, functional dependencies among features imply that only certain combinations of variant features may coexist in any given system release [Jar07]. Insufficient information about such dependency rules might cause behavioral problems during SPL maintenance and evolution, since the programmer may not be aware of them.

In order to understand the occurrence of feature dependencies on SPLs, we define a set of software metrics regarding preprocessor usage and develop a tool to compute these metrics. The tool, named `pl-stats`, is able to compute the defined metrics on source code which uses preprocessor directives to implement variability. We collect data from the source code of 45 software projects of different domains, sizes and languages. In terms of complexity, these projects vary from simple and small, such as *MPSolve*, to complex and very large ones, such as the *Linux Kernel*.

Based on the collected data, we perform three studies regarding the occurrence of feature dependencies on SPL. The first two studies relate to the occurrence of intraprocedural dependencies, i.e. occurrence of feature dependencies within project methods. On the first one, we are interested on frequency of occurrence of feature dependencies on these projects. We found that  $7.61\% \pm 7.22\%$  of the projects methods use preprocessor directives and that  $72.49\% \pm 17.69\%$  of the methods with directives have dependencies. On the second one, we are interested in knowing if there is a correlation between software complexity and feature dependencies, and *where* these feature dependencies occur. Through the correlation values between the metrics regarding complexity, we might conclude the higher the complexity the greater the number of feature dependencies. Regarding *where* feature dependencies occur, we found that, on average, 48.8% of dependencies occur when using `#if` directive. The third study complements the first two through an interprocedural analysis, where we compare the effort to maintain dependent features using emergent interfaces and non emergent interfaces approaches in SPLs. We found that, when using emergent interfaces approach, developers has a gain of  $56.28\% \pm 21.70\%$  related to the

number of fragments and  $46.29\% \pm 19.73\%$  related to the number of features they would have to analyze, compared to not using emergent interfaces.

In summary, this work presents an initial assessment on feature dependencies occurrence in preprocessor-based systems, providing a large dataset regarding preprocessor usage and feature dependencies on software projects from different domains, sizes and languages. Furthermore, we develop a tool to compute a set of metrics regarding preprocessor usage on software projects, more specifically regarding the occurrence of *simple dependencies* on software projects. Finally, we present a mapping study regarding preprocessor usage on software projects. As the study was conducted following systematic mapping features, this allows other researchers to replicate it, adjusting some variables, like the terms of the search string, according to the goals of their researches.

In what follows, we present the summary of goals and the organization of this work.

## 1.1 SUMMARY OF GOALS

Based on the problems mentioned above, the main goal of this work is to show how feature dependencies occur on practice. We are interested in evaluating the use of conditional compilation mechanisms in the implementation of preprocessor-based product lines, more specifically, understanding the occurrence of code level feature dependencies regarding these mechanisms.

The specific goals of this work are:

- to perform a mapping study regarding the use of conditional compilation capabilities to implement variability;
- to develop a tool to sample data regarding feature dependencies on projects that use conditional compilation mechanisms; and
- to provide empirical data about feature dependencies on real software projects which use conditional compilation mechanisms to implement variability, performing an initial assessment regarding their occurrence.

## 1.2 OUTLINE

The remainder of this dissertation is organized as follows:

- Chapter 2 reviews the essential concepts used throughout the work: Software Product Lines, Conditional Compilation, Software Metrics and Software Maintenance and Evolution;

- Chapter 3 presents a mapping study about the usage of conditional compilation mechanisms, identifying the relevant studies about that, showing the main results and pointing some research gaps which can be explored by researchers on other works;
- Chapter 4 presents our studies about feature dependencies: we state our assumptions, describe the mechanism used to sample data about the projects and present the empirical studies, defining their research questions and discussing their results;
- Chapter 5 describes the concluding remarks, discussing our contributions, the related work and the future work.

## CHAPTER 2

# BACKGROUND

In this chapter we review essential concepts explored in this work. We discuss concepts related to Software Product Lines (SPL) in Section 2.1. In this context, in order to understand how variability is applied by preprocessors in SPL, we discuss the concepts and show examples of the use of conditional compilation mechanisms on Section 2.2. Next, we present the concepts of software metrics in Section 2.3, which were used to understand and define the metrics used in this work. Last, software maintenance and evolution are discussed in Section 2.4. This topic is important for this work since some drawbacks of preprocessor use during SPL maintenance and evolution have been discussed in the literature [KS94, SC92, Fav96].

### 2.1 SOFTWARE PRODUCT LINES

One increasing trend in software development is the need to develop multiple similar software products instead of just a single individual product. There are several reasons for this. For instance, products that are being developed for the international market must be adapted for different legal or cultural environments, as well as for different languages, and so must provide adapted user interfaces. Because of cost and time constraints it is not possible for software developers to develop a new product from scratch for each new customer, and so software reuse must be promoted.

Suppose that a company acquires a new project to start the development of a mobile game. To reach a large number of users, the game must work on different mobile phones. Even though it is a single game, there will be many different versions of it running on many different mobile phones. Due to particular restrictions and API specification of each phone, there are differences that must be taken into account. For example, consider the three mobile phones showed in Figure 2.1. The first phone on the left has a screen resolution of 160x128, while the phone on the right has 1920x1080.

A possible way to address this and other variations, such as amount of available memory, keyboard, sound API and others, is to develop each product separately. This way, we are able to target the specific needs and restrictions of each phone. However,

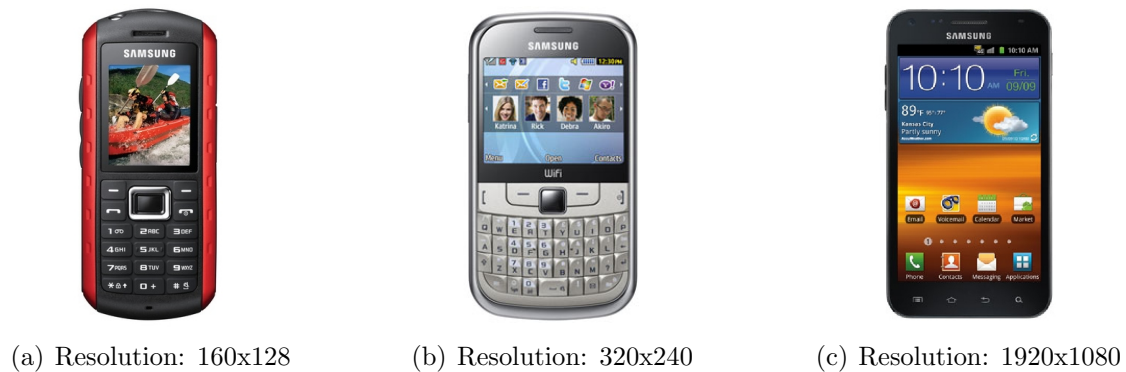


Figure 2.1: Different models of mobile phones. Source: [samsung.com](http://samsung.com)

we end up with duplicated code among these products, since in its essence, the game is the same across the different versions. Therefore, each version should be maintained separately too. In this context of mobile phones, we are talking in much more phone models than these three in the examples, we can refer hundred of different phones. This might lead to extra costs for developing a new version of the game, maintainability issues caused by resource shortages, among others [CN02, LSR07, PBL05, Kru02]. To avoid these problems, we can use concepts of software product lines engineering [CN02].

Software Product Line Engineering (SPLE) is a software engineering approach for developing families of software products using a common core asset base to satisfy the needs of a particular market segment [CN02]. This approach provides development and mass customization [PBvdL05], also representing the large production regarding product variabilities according to the market needs. The characteristic that distinguishes software product lines from other approaches is predictive versus opportunistic software reuse. Rather than put general software components into a library in the hopes that opportunities for reuse will arise, software product lines only call for software artifacts to be created when reuse is predicted in one or more products in a well-defined product line [Kru06], i.e., in the SPL reuse context is planned.

At the core of SPLE is the systematic reuse of software assets across the products in the product line. Such assets include plans, designs, code, test cases and any other artifact used to develop software, and cover the entire engineering process, from requirements to architecture, implementation, test, and maintenance. The core assets development include a process that establishes a reusable platform and consequently the commonalities and variabilities of the SPL, being known as *Domain Engineering*.

The Domain Engineering identifies common features of an area and handle their for reuse by applications of a specific domain. The focus is centered on the commonalities

presented by a set of applications. The Domain Engineering aims to transform these commonalities in tangible artifacts. The process defines the development of these artifacts through the evolutionary prototyping with short iterations supported by the spiral model proposed by Boehm [Boe86]. Thus, the artifacts from this stage would be stored in a repository, to be reused later.

In contrast to Domain Engineering, which focuses on commonalities in a set of applications, *Application Engineering* aims to develop applications with reuse of artifacts provided from Domain Engineering. Products or applications are instantiated using the artifacts to compose its functionalities. The higher the reuse of artifacts made by the products, usually means the more efficient is the product line. Application Engineering, as Domain Engineering, uses the spiral model with short iterations.

Although a full asset repository is available, every application has its specificity and needs some customizations. It is important to remember that, although more costly, new feature development should be considered to inclusion in the repository, for future reuse. This increases the reuse degree of the product line as is being used, enabling medium and long term feedback.

Some advantages of using software product line concepts:

- **Time-to-market reduction:** initially, the time-to-market of the SPL is high, because the core assets must be developed first. Afterwards, the time-to-market is reduced, because many components previously developed might be reused for new products;
- **Quality enhancement:** the core assets of an SPL are reused in many products. In this way, they are tested and reviewed many times, which means a higher chance of detecting faults and correcting them, improving product quality;
- **Development costs reduction:** when artifacts are reused in several different kinds of systems, this implies in cost reduction for each system, since there is no need to develop such components from scratch.

To guarantee these advantages, it is essential to manage the product variabilities of the SPL in a suitable way. Such variabilities play a key role in the SPL, since different applications of the SPL can be distinguished in terms of these variabilities [Alv07]. In addition, they enable the development of customized applications by reusing some predefined artifacts.

Several mechanisms may be used to implement variability in a product line. These mechanisms may act to a greater or lesser level of granularity. An example of these

mechanisms, is the use of inheritance, configuration files [Alv07], aspect-oriented programming [KLM<sup>+</sup>97], mixins [BC90] and conditional compilation, being the last the focus of this work.

## 2.2 CONDITIONAL COMPILATION

Between 1969 and 1973 Dennis Ritchie has developed a general-purpose computer programming language called “C”. Prior, it was developed for use with the Unix Operating System, but its use was expanded for developing portable application software and it became one of the most widely used programming languages of all time [tio12].

The C language is marked by being fast, portable and by its lightweight meta programming capabilities. These could be implemented and can be used by means of the C preprocessor features. The C preprocessor (*cpp*) is a tool that was intended initially to be used only with C, C++ and Objective-C source code. Given the simplicity of the language and state of the art in compiler technology in the mid-1970s, the decision to provide some language features in this extra-linguistic tool was justified, where the developer could not rely on it preserving characteristics of the input which was not significant to C-family languages. But, new versions of *cpp* allow it to be used with any text artifact, including other programming languages such as Java and C#.

The main features of *cpp* are file inclusion, macro definition and conditional inclusion, which are specified by preprocessor directives. Preprocessor directives are represented by lines included in the code of programs but that are not program statements. These lines are preceded by a hash sign (#), where the preprocessor is executed before the actual compilation of code begins. The preprocessor interprets all these directives before any code is generated by the statements.

The preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

The preprocessor recognizes a set of directives, such as `#define`, `#if`, `#ifdef`, `#elif`, `#else`, `#ifndef`, among others. They are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text (conditional compilation).

Preprocessor lines are recognized and carried out before macro expansion. Therefore, if a macro expands into something that looks like a preprocessor command, that command is not recognized by the preprocessor.

The conditional compilation mechanism can be used for many purposes. The Listings 2.1 and 2.2 show examples of conditional compilation use. The Java code snippet in the Listing 2.1 was extracted from the *BestLap*<sup>1</sup> product line. *BestLap* is a casual race game where the player tries to achieve the best time in one lap to qualify for the pole position. The game is highly variant due to portability constraints: it should run on numerous platforms. In fact, the game was deployed on 65 devices [Alv07].

---

Listing 2.1: Conditional compilation use on mobile game.

---

```

1 public void computeLevel() {
2     ...
3     totalScore = ...
4     ...
5
6     #ifdef ARENA
7         NetworkFacade.setScore(totalScore);
8     #endif
9 }
```

---

In this game there is a method responsible for computing the game score. The feature *ARENA*, represented in line 6, is an optional feature responsible for publishing the scores obtained by the players on the network. This way, players around the world are able to compare their results. The method also contains a variable called `totalScore`, responsible for storing the player's total score. Basically, the preprocessor will evaluate if the *ARENA* feature was defined previously (line 6). If it was just defined, the piece of code in line 7 is executed, in other words, the player's score is published on the network.

The C code snippet in Listing 2.2 was extracted from core of the *Linux Kernel*. *Linux Kernel* is the most important piece of the operating system used by the Linux family. It is a structure formed by million of lines of code and provides many features for use by an operating system, such as memory management and an Internet protocol suite.

---

Listing 2.2: Conditional compilation use on Linux Kernel.

---

```

1 ...
2 #ifdef CONFIG_WL1271
3     printk(KERN_INFO "%s: CONFIG_WL1271 detected", __func__);
4 #else
```

---

<sup>1</sup>BestLap is a commercial product developed by MeanTime Mobile Creations.



```
5     printk(KERN_INFO "%s: CONFIG_WL1271 not detected", __func__);
6     #endif
7     ...
```

---

During kernel loading, some tests are executed to verify if required modules are present on system. For a wifi module test, it is verified if the module is enabled (see Listing 2.2, line 3). If it is enabled, a message with the information is displayed on the console (line 4). Otherwise, a message with another information is displayed on the console, reporting the failure (see Listing 2.2, line 6).

Since conditional compilation provides a set of directives which are used to implement variable code, it may be used in the context of software product lines. Programmers can use the macro `#define` to set a feature constant on the code and provide a way to compose features. Feature constants can also be defined in makefiles, in configuration files, or during the compiler invocation. These feature constants can be combined with logical operators provided by *cpp*, composing complex feature expressions. A feature expression represents the condition that controls the inclusion or exclusion of feature code. That is, based on the evaluation of a feature expression, the lines of source code encompassed by `#ifdefs` are included or excluded, depending the results of expression. Based on the definition of feature constants, the programmer is able to influence the evaluation of a feature expression and, consequently, the presence or absence of feature code. We can see these concepts in Listing 2.1, where the feature `ARENA` is encompassed by `#ifdef` and `#endif` directives.

## 2.3 SOFTWARE METRICS

Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development. The objective is quantifying some characteristic or attribute of a computer software entity in a reproducible way. The quantifiable measurements may have numerous valuable applications, such as in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization and optimal personnel task assignments.

Despite the efforts to apply quantitative measurements, software measurement has become essential to good software engineering. Many of the best software developers measure characteristics of the software to get some sense of whether the requirements are consistent and complete, whether the design is of high quality, and whether the code is ready to be tested. Effective project managers measure attributes of process and product

to be able to tell when the software will be ready for delivery and whether the budget will be exceeded [FP98]. And so on. Each stakeholder of a project uses some kind of measurement to identify if the process, project or products are achieving the desired goal.

But measurement has been considered a luxury in software engineer. When measurements are made, they are often done infrequently, inconsistently and incompletely. These factors can be frustrating to those who want to make use of the results. For example, developers do not quantify or predict the quality of the products they produce. Thus, they cannot tell a potential user how reliable a product will be in terms of likelihood of failure in a given period of use, or how much work will be needed to port the product to a different machine environment [FP98]. Or yet, when developers are convinced to try another revolutionary new development technology. When this is done without doing a carefully controlled study, is not possible to determine if the technology is efficient and effective. Without measurement information it is not possible conduct an objective study to repeat the measurements in our own environment.

The reasons why a project derails are not always known. It is essential to measure and record characteristics of good projects as well as bad ones. It is necessary to document trends, corrective actions and resulting changes, aiming to control the projects, not just run them. To do that, the measurement objectives must be specific, tied to what the managers, developers and users need to know. Every measurement action must be motivated by a particular goal or need that is clearly defined and easily understandable. These objectives may differ according to the kind of personnel involved and at which level of software development and use they are generated. It is the goals that tell how the measurement information will be used once it is collected [FP98].

The measurements help in such activities of software engineering, as to understand what is happening during development and maintenance, where developers can assess the current situation to set goals for future behavior. Furthermore, measurements allow one to control what is happening on projects. Through the knowledge of the baselines, goals and understanding of relationships, it is possible to predict what is likely to happen and make changes to process and/or products aiming to achieve the intended goals [FP98]. This encourages to make improvements in it process and products, leading to achieve a better solution to meet the defined goals.

The usefulness of design and implementation practices can be evaluated through empirical studies. Software metrics are often used in empirical studies as indicators of the strengths and weaknesses of the studied approach. Metrics evaluate the use of abstractions during software development in terms of software attributes and are more effective

when they are associated with some assessment framework so that software engineers can understand and interpret the meanings of the collected data [SGC<sup>+</sup>03].

Software metrics is a term that embraces many activities on software development, including cost and effort estimation, and productivity measures and models. It is in the interest of managers to be able to predict project costs during the first phases in the software life-cycle. As a result, several models for software cost and estimation have been proposed and used. These models often share a common approach, where the effort is expressed as a function of one or more variables, such as size and level of reuse [FP98]. Furthermore, many managers make decisions based on productivity models, where they can take into consideration the rate at which lines of code are being written per person month of effort, for example. This is a simple measure and can be misleading, if not dangerous [Jon86]. More complex models are defined and used extensively, providing a significantly more comprehensive view of productivity.

Since this work aims at providing a set of data regarding preprocessor usage on SPLs, software metrics become an important tool to guide us in which set of metrics use to it. In this context, some metrics used in this work are concerning complexity. The complexity can be considered as a separate component of size, and it can be expressed in a more objective way. We consider define complexity in terms of some other metrics, such as source lines of code, number of methods, number of features and number of different kinds of preprocessor directives. This allow us to perform our studies using fine-grained metrics regarding complexity.

## 2.4 SOFTWARE MAINTENANCE AND EVOLUTION

Software maintenance is about change. It may be a small change to fix a bug, for instance, or to enhance software requirements to better satisfy a customer in particular or a set of customers in a specific market place. These changes lead companies to spend a good portion of time maintaining existing software code [Jar07], when they could be developing a new software and, consequently, achieving new goals earlier.

During many years, maintenance was understood just as fixing errors in released programs. Today, it is known that what happens to software after the first release is much more complicated than that. For any software system to be successful, functional improvements are inevitable and essential in its evolution: as the business rules change, users come up with new requirements. In some cases, it is necessary to reengineer existing software to take advantage of technological advances such as new platforms and architectures, to improve software structure, reach new clients, or win a new market share [Jar07].

And that is why software maintenance is characterized as the most expensive software activity. Various studies and surveys indicate that over 80% of the total maintenance effort is used for non-corrective actions [Pig97]. In addition, other studies indicate that software maintenance accounts for at least 50% of the total software production cost, and sometimes even exceeds 90% [SPL03]. Since software maintenance is present in all development contexts, this high maintenance cost can cause serious practical implications, such as to limit or prevent IT divisions from delivering new systems that might be of strategic importance to their companies.

Several factors in software development tasks make change hard, such as the software size, its complexity and the effects that modifications in it can induce. In the context of complexity, it can be determined by two factors: (i) the complexity of a problem and its solution at the conceptual level, and (ii) the fact that not only do programs express problem solutions, but also must address a range of issues related to solving a problem by a computer [F.P87]. These two factors cannot be cleanly separated from one another in program components, which can lead developers to use decomposition techniques—such as aspect-oriented programming [KLM<sup>+</sup>97] and XVCL [JBZZ03]—to combat complexity. Regarding the effects caused by modifications on software, a change in customer requirements may affect multiple software components, as well as their architecture. Any change that affects component interfaces or global system properties may unpredictably impact many software components [Jar07].

Besides the aforementioned problems, code structures with similarities often copied and duplicated in many places in a program, can make change hard and contribute significantly to high maintenance cost of the software. Part of this code redundancy is created intentionally and for a justified reason, such as to improve software performance or reliability, or are induced by an employed programming technology. Any changes on these redundancies expose programs to risks of malfunctioning, and the application of some techniques to overcome this problem, such as refactoring [FBB<sup>+</sup>99], may not be a viable option for business reasons.

Software can suffer from various errors, which may cause crashes, hangs or incorrect results, significantly threatening the reliability and the security of computer systems. Unfortunately, fixes to these bugs may still have problems, since they are written by humans. Some fixes either do not fix the problem completely or even introduce new problems, as reported by some big companies such as Microsoft [McM10] and Apple [McM06]. Mistakes in bug fixes may be caused by many possible reasons, as the tight deadlines, where fixers have much less time to think cautiously, especially about the potential side-effects

and the interaction with the rest of the system, or the focus on removing the bug comparing to general development, where testers may just focus on making an observed bug symptom disappear, but forgetting to test some other aspects, in particular how the fix interacts with other parts and whether it introduces new problems.

Maintenance problems aggravate during software evolution, when many changes get implemented over time. During the planning phase, program comprehension is essential to understand what parts of the software will be affected by a requested change. But it is common not document and delegate the problems from several sources of change on code to external tools, but this “approach” hasn’t proved successful so far. The low knowledge of changes that have affected software over time makes future change even harder. This indiscipline to deal with changes on software evolution adds to program complexity. Problems like these on software evolution did not receive much attention, since the main role of programming languages has been to provide abstractions for code structures to be executed on a computer, and the mechanism for change have been left on background [Jar07].

When the number of variant features in software increases, the problems get worse. Various combinations of these features might have been implemented into released systems, and may be needed in future releases. The existence of variability realization mechanisms has been identified as one the obstacles that impedes implementation of reuse strategies via the product line architecture approach [DSB05]. Furthermore, the occurrence of feature dependencies is a problem to be considered too. Functional dependencies among features imply that only certain combinations of features can coexist on the same software release. The lack of information about dependency rules may cause problems in software construction and deployment, resulting in an incomplete system or in problems with its build process.

Since new software release has been requested, the use of previous releases is a good strategy to save the time and effort of implementing a new system. However, features implemented in past releases may need to be adapted to the context of a new release [Jar07]. The facility to reuse features already implemented determines the evolution productivity, making the whole difference between ad hoc and systematic evolution and should be considered in maintenance and evolution activities.

## CHAPTER 3

# LITERATURE REVIEW

Aiming to identify the relevant studies regarding the use of conditional compilation, we conducted a literature review on which studies perform evaluation regarding the use of conditional compilation in software projects. We do this based on systematic mapping guidelines, simplifying some steps due to some restrictions, as we will explain as follows.

Systematic mapping studies are based on a well-defined research strategy, aiming to identify, evaluate and interpret all available research relevant to a particular research question [PFMM08]. These differ from systematic reviews in some points, as mapping studies generally have broader research questions, driving to ask multiple research questions, and the data extraction process of mapping studies is also much broader than the data extraction process for systematic reviews, due to these broader research questions. To document the mapping, we define the review protocol, that is a document which specifies the research questions and the methods that will be used to undertake the mapping. The document must include the strategy that will be used to search for primary studies including search terms and resources to be searched, as well as clearly define the selection criteria to evaluate each study in potential. Therefore, the definition of a review protocol enables readers and researchers to know the reviews rigour and completeness [dABMN<sup>+</sup>07]. A systematic mapping aims to present a fair evaluation of a research topic by using a trustworthy, rigorous, and auditable methodology [PFMM08].

Aware of systematic mapping guidelines, the literature review performed in this thesis considered some characteristics of them, like the definition of research questions, study selection criteria and data extraction strategy. However, some characteristics that give support to the rigour of Systematic Mapping were not considered. This was decided due to time constraints and the fact that this mapping is not the main contribution of this work. The main restrictions of this mapping study are described below:

- The review protocol, the search terms and inclusion/exclusion criteria were not evaluated by other researchers, other than the advisor of this research;
- The results of study selection were not revised by other researchers;

- The search process used only two search engines, although these index the most of papers from other sources.

The following sections define the literature review protocol and present the review execution data and results from selected review studies.

### 3.1 REVIEW PROTOCOL

As stated prior, the review protocol specifies the research questions and the methods that will be used to undertake the mapping. It should include all the elements of the review, as study selection criteria and data extraction strategy. The use of a review protocol aims to reduce the possibility of research bias as a malformed search process or a study selection guided by researcher intuition.

The following sections specify the research elements and give detailed information about each step in the mapping study as well as the results.

#### 3.1.1 Search Process

To try to assess the largest amount of studies possible in a short time period, we used the search engines Scopus<sup>1</sup> and EI Compendex<sup>2</sup>. We used only these two search engines because they index the most relevant databases in research area, covering the main computer science published journals as IEEE, ACM and IET.

The search process was defined in steps with specific goals, based on research terms, keywords and their synonyms. Figure 3.1 shows the search process and its steps.

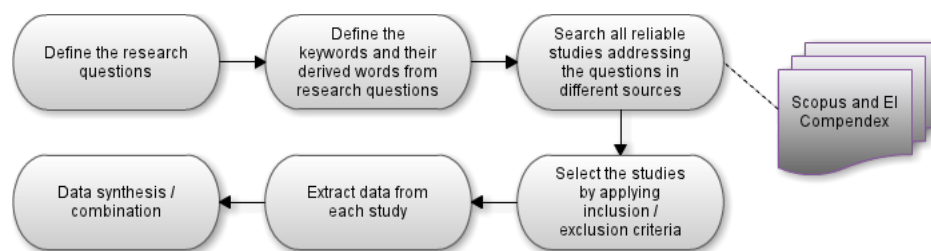


Figure 3.1: Search process and its steps.

The first step in the search process to conduct a systematic mapping is to define the research questions. Remaining search process steps will trust on these questions to achieve their goals. The next step is to define the keywords and their derived words.

<sup>1</sup>[www.scopus.com](http://www.scopus.com)

<sup>2</sup>[www.engineeringvillage2.org](http://www.engineeringvillage2.org)

Both are defined from research questions and will be used on search engines to retrieve all reliable studies addressing these questions. This will be done on the third step. Next, a set of relevant studies is selected by applying inclusion/exclusion criteria, where for each selected study, a set of data addressing the research questions is extracted. Finally, at the last step, the amount of extracted data is summarized and combined to provide an information that allows us answer the research questions.

### 3.1.2 Research Questions

The context of this review refers to evaluation of the use of conditional compilation in software projects. Thus, we defined the following research questions for our study:

**Research Question 1:** Which research topics were discussed in studies that evaluate conditional compilation use in software projects?

**Research Question 2:** What kinds of evaluation were performed?

**Research Question 3:** What were the main results?

### 3.1.3 Keywords & Search Terms

The search strings used in this research were constructed using the following strategy:

- Derive main terms based on the research questions and the topics being researched;
- Determine and include synonyms, related terms and alternative spelling for major terms;
- Check the keywords in initial searches on the relevant sources;
- Incorporate alternative spellings and synonyms using boolean “OR”;
- Link main terms using boolean “AND”;
- Pilot different combinations of research terms.

Following this strategy, the major terms and their derived words used to compose the search string are summarized in Table 3.1.

The search terms were formulated from the keywords and their derived words using the operator *OR*, as follow below:

- evaluation OR rating OR assessment



Table 3.1: Keywords and their derived words.

Keyword	Derived words
evaluation	rating, assessment
conditional compilation	preprocessor, pre-processor, preprocessing, pre-processing, macro, variability
software	program, application

- conditional compilation OR preprocessor OR pre-processor OR preprocessing OR pre-processing OR macro OR variability
- software OR program OR application

Thereafter, the search string was defined by using the search terms with the operator *AND*, as presented below:

*(evaluation OR rating OR assessment) AND (“conditional compilation” OR preprocessor OR pre-processor OR preprocessing OR pre-processing OR macro OR variability) AND (software OR program OR application)*

### 3.1.4 Inclusion & Exclusion Criteria

One important detail to consider is the inclusion and exclusion of papers. The researcher must be very careful when analyzing which papers will and will not be included, because some interesting papers can be erroneously excluded as a result of a misconducted analysis.

In the current study, all papers that were clearly out of scope were removed early in the process based on the analysis of the *title*, *abstract* and *keywords*. After the initial selection, the full versions of each paper were obtained and a more detailed analysis could be performed. The following criteria were applied in order to include or exclude the paper from the mapping:

- The papers should be available on the web;
- Papers that represent real papers, not a Power Point<sup>©</sup> presentation or extended abstract (tech reports, PhD Thesis and MSc Dissertations are considered as well);
- The papers are not duplicates;
- The papers should present any kind of evaluation of the use of conditional compilation, regardless scope or sample size;

- The paper is related to software engineering.

The quality analysis of the selected studies was not performed in this research. To include/exclude studies in/from list of select studies, only inclusion/exclusion criteria were considered.

### 3.1.5 Data Extraction

From the list of selected studies, some data was extracted aiming to answer the research questions defined previously. To help on this process, for each selected paper, two forms were used. They are shown below.

#### Form A

This form was used to collect general information from the selected primary studies. The following information was collected from each paper:

- **ID:** An identifier for the paper. It is useful to reference the selected paper in several parts of the mapping;
- **Source:** The publisher of the paper;
- **Year:** The year of publication;
- **Title:** The title of the paper;
- **Author:** The list of authors of the paper;
- **Institution:** The list of institutions that took part in the work;
- **Country:** The list of countries where the institutions are based.

#### Form B

In addition to Form A, Form B was used to extract the relevant information from the primary studies, mainly aiming answer the research questions stated in Section 3.1.2. For each primary study analyzed, the following information was collected:

- **Evaluation Date:** When the paper was evaluated;
- **Research Question 1:** Which research topics were discussed in the study?

- **Research Question 2:** What kinds of evaluation were performed in the study?
- **Research Question 3:** What were the main results of the study?
- **Notes:** Eventually some insight from the researcher can be documented in this field.

### 3.2 EXECUTION

The execution of research at search engines using the search string defined above does not take any time restrictions. A summary with execution data is presented in Table 3.2.

Table 3.2: Summary of execution data.

Source	Source Results	Potentially Relevant Studies	Not Relevant	Repeated	Incomplete	Relevant Studies
EI Compendex	4019	13	4006	0	0	2
Scopus	1448	15	1433	2	0	6
<b>Total</b>	<b>5467</b>	<b>28</b>	<b>5439</b>	<b>2</b>	<b>0</b>	<b>8</b>

As showed, the number of relevant studies (8) is smaller than the total number of studies (5467). This happened because of search string construction. The keywords and derived words used in search terms to construct the search string were not too specific, leading to find many articles of other research areas. On the other hand, using more specific keywords would put some articles not to be found, which would result in a threat to research results.

In addition to the relevant studies, 2 studies that not were identified by the search process were recommended by other researchers. They were included on our analysis and are listed in appendix of this work. The analysis of the all results took about 4 months and was conducted by only one researcher.

### 3.3 RESULTS

We identified 10 empirical studies on the use of conditional compilation in software projects. These studies cover a range of research topics and were performed in variable settings. We will describe characteristics of the studies and analyze how they answer the research questions defined in this work.

### 3.3.1 Overview

As we stated in Section 3.1.1, we use two search engines in our research. Figure 3.2 show the number of relevant studies found by the search engines and Figure 3.3 show the distribution of these studies across the years. We found two (20%) relevant studies in EI Compendex and six (60%) relevant studies in the Scopus search engine. Other two studies (20%) were not identified by the search process and were recommended by other researchers. As we can see, although EI Compendex source result is higher than Scopus (three times) in the first step, the number of relevant studies found on Scopus was greater than EI Compendex.

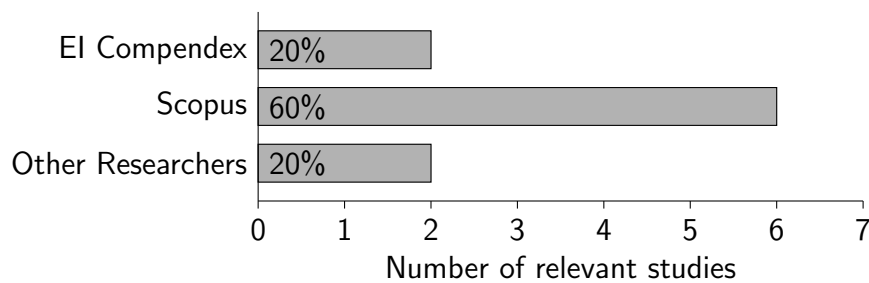


Figure 3.2: Distribution of relevant studies by search engine.

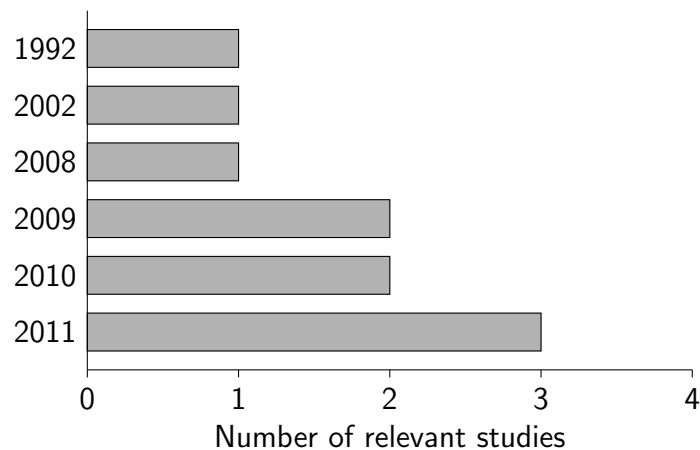


Figure 3.3: Number of relevant studies by year.

There were 43 different authors in the 10 selected studies. The ones who published more than one paper are presented on Table 3.3. The authors come from 15 institutions, based on 5 different countries. The Figure 3.4 shows the participation of each country in selected studies.

Table 3.3: Authors who have published more than one paper about conditional compilation use.

Author	#Studies
Christian Kästner	2
Jörg Liebig	2
Sven Apel	2
Eduardo Figueiredo	2

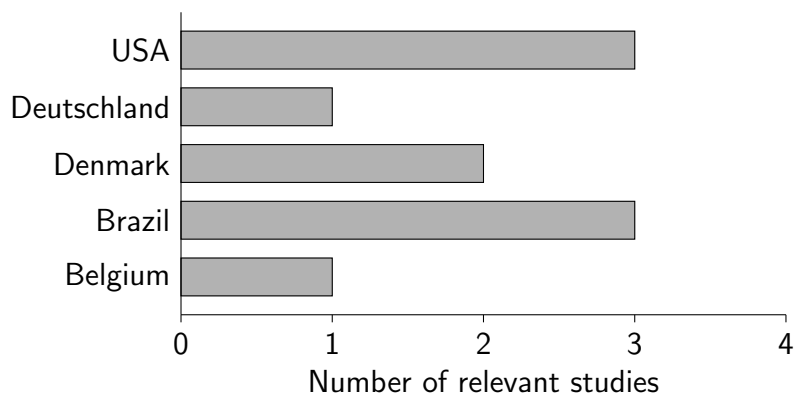


Figure 3.4: Distribution of relevant studies by country.

### 3.3.2 RQ1 - Research Topics

The objective of this question was identify the main software engineering research fields covered by research on conditional compilation use. In total, 11 different topics were explored by the primary studies. The topics were identified based on description given by the authors of the primary studies. The amount of investigated research topics is higher than the total amount of primary studies because some studies take more than one research topic. The most investigated topic was “SPL Maintenance and Evolution”.

The evidence extracted from the primary studies is summarized in Table 3.4 and described in the remaining of this section.

Each research topic is briefly described and the primary studies that are related to the topic are listed alongside with their correspondent evidence showing the relationship.

#### Variability Extraction

The C preprocessor is the tool of choice for the implementation of variability in many large-scale configurable software projects. However, this variability tends to be “hidden” in the code, which on the long term leads to variability defects, such as dead code or

Table 3.4: Summary of papers by research topic.

Research Topics	References - PS: Primary Studies	# of Studies (%)
Variability Extraction	PS01, PS10	2 (20%)
SPL Maintenance and Evolution	PS01, PS02, PS05	3 (30%)
Macro usage	PS02, PS08	2 (20%)
Program comprehension	PS03	1 (10%)
Refactoring	PS03, PS07	2 (20%)
Disciplined and undisciplined annotations	PS04	1 (10%)
Virtual separation of concerns	PS05	1 (10%)
Feature Dependencies	PS05	1 (10%)
Emergent Interfaces	PS05	1 (10%)
Portability	PS06	1 (10%)
AOP	PS07, PS09, PS10	3 (30%)

inconsistencies with respect to the intended software variability. This calls for variability extraction, which aims to provide variability data from implementation / source code for both developers during regular development phase and several kind of analysis, as allow the crosschecking between the variability model and the implementation or analysis by a SAT or BDD package.

The evidence which included the study in this topic is as follows:

- **PS01** — *“We suggest a novel approach to extract cpp-based variability.”*
- **PS10** — *“...this paper describes an experiment involving the extraction of a SPL from ArgoUML...”*

### SPL Maintenance and Evolution

Software maintenance is an activity in software engineering which comprehends the modification of a software after delivery, aims to correct faults or improve any functionality. On SPLs, this activity covers features across the code, which implies in accuracy with evolution concerns by developers. Some key issues could be the alignment with customer priorities, staffing, which organization does maintenance, estimating costs, limited understanding, testing and maintainability measurement.

Below we present the evidence that included the studies in this topic:

- **PS01** — *“We believe that our model can support maintenance and evolution of SPLs by automatically detecting several kinds of inconsistencies in different ways.”*

- **PS02** — *“Since CppChecker provides an accurate mapping from the pre-cpp and post-cpp token streams to ASTs, we believe it presents an opportunity for developing better refactoring tools for C/C++ programs. It is particularly interesting to develop tools for analyzing macro usage across programs for across-program understanding and maintenance.”*
- **PS05** — *“In this context, emergent interfaces can capture dependencies between the features we are maintaining and others, making developers aware of them.”*

### Macro Usage

Macro usage is the way that preprocessors can be used on software code to implement a set of features, including variability. Through the use of directives defined for each preprocessor, is possible to declare constants and encompass a code snippet that will be built depending on the result of the evaluation of a conditional expression.

The evidence which included the studies in this topic is as follows:

- **PS02** — *“...we introduce a novel, general characterization of inconsistent macro usage as a strong indicator of macro errors.”*
- **PS08** — *“We determine the incidence of C preprocessor usage—whether in macro definitions, macro uses, or dependencies upon macros—that is complex...”*

### Program Comprehension

Program Comprehension is a software engineering activity concerned with the ways software engineers maintain existing source code. It is known that 80% of the time in software development is spent with maintenance tasks. Of the maintenance time, 20% is spent changing the code while 80% of the time is spent just trying to understand the code [Pig97]. This lead to some drawbacks, such as rework by developers and, consequently, a spread of development costs. Program comprehension is necessary to facilitate reuse, inspection, maintenance, reverse engineering, reengineering, migration, and extension of existing software systems.

The evidence which included the study in this topic is as follows:

- **PS03** — *“These questions revive earlier discussion on program comprehension and refactoring in the context of the preprocessor.”*

## Refactoring

Refactoring is a disciplined technique for restructuring the internal structure of a software program aiming to make it easier to be understood and less costly to be modified, without changing its external behaviour. The idea is that internal changes improve the structure of code by small and constant changes. As changes tend to be small, the chances of introducing errors are reduced. Moreover, one avoids breaking the system while carrying out the restructuring, by means of gradual refactor over an extended period of time.

Below we present the evidence that supported the inclusion of the studies in this topic:

- **PS03** — *“These questions revive earlier discussion on program comprehension and refactoring in the context of the preprocessor.”*
- **PS07** — *“This paper applies preprocessor blueprints to find out whether or not the refactoring of conditional compilation patterns into advice is technically feasible, independent from the semantics or purpose of the conditional code.”*

## Disciplined and Undisciplined Annotations

As way to identify how developers are annotating their C code, disciplined annotations are defined by annotations on one or a sequence of entire functions and type definitions. Furthermore, annotations on one or a sequence of entire statements and annotations on elements inside type definitions are considered disciplined too. All other annotations are considered undisciplined.

Below there are the evidences that included the studies in this topic:

- **PS04** — *“We distinguish between disciplined annotations, which align with the underlying source-code structure, and undisciplined annotations, which do not align with the structure and hence complicate tool development.”*

## Virtual Separation of Concerns

Virtual Separation of Concerns (VSoC) has been proposed as a way to allow developers to hide feature code not relevant to the current task, reducing some of the preprocessors drawbacks. Developers do not physically extract the feature code, but just annotate code fragments inside the original code and use tool support for views and navigation. To annotate the code, background colors are used, so that code fragments belonging to a



feature are shown with a background color; hence the name. The main idea is to provide developers a way to focus on a feature implementation without being distracted by others.

Below there are the evidences that included the studies in this topic:

- **PS05** — *“Besides, we provide an empirical study to compare maintenance effort when using VSoC and emergent interfaces”*

### **Feature Dependencies**

Features of a software product line can share elements, such as methods and variables, with others. When this sharing occurs, there might be dependencies between the involved features. These dependencies can be characterized in many ways, such as when a feature assigns a value to a variable read by another one or when a feature instantiates a variable used by another one. Thus, due to these dependencies, in a maintenance task, a developer can maintain a specific feature and break another.

The evidence which included the study in this topic is as follows:

- **PS05** — *“Our analysis comprehends preprocessor usage and feature dependencies frequency.”*

### **Emergent Interfaces**

Emergent Interfaces is an approach to deal with maintenance tasks on a product line, when developers are maintaining feature code. Emergent Interfaces capture dependencies between the feature being maintained and others, giving information about features which might impact with the maintenance task. From that, developers can be aware of dependencies and, consequently, might avoid maintainability problems as late error detection and hard navigation.

The evidence which included the study in this topic is as follows:

- **PS05** — *“In this context, emergent interfaces can capture dependencies between the features we are maintaining and others, making developers aware of them.”*

### **Portability**

An application is portable across a class of environments to the degree that the effort required to transport and adapt it to a new environment in the class is less than the

effort of redevelopment. In the context of `#ifdefs`, its use works acceptably well when differences are localized and only two versions of the code are present. Unfortunately, as software using this approach is ported to more and more systems, the `#ifdefs` proliferate, nest, and interlock. After a while, the result is usually an unreadable and unmaintainable mess.

The evidence which included the study in this topic is as follows:

- **PS06** — *“We believe that a C programmer’s impulse to use `#ifdef` in an attempt at portability is usually a mistake.”*

### Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm which aims to increase modularity by allowing the separation of cross-cutting concerns. AOP complements OO programming by allowing developer to dynamically modify the static OO model to create a system that can grow to meet new requirements through some concepts as joinpoints, pointcuts and advices. Just as objects in the real world can change their states during their lifecycles, an application can adopt new characteristics as it develops.

The evidence which included the study in this topic is as follows:

- **PS07** — *“This paper applies preprocessor blueprints to find out whether or not the refactoring of conditional compilation patterns into advice is technically feasible, independent from the semantics or purpose of the conditional code.”*
- **PS09** — *“This paper presents a case study that quantitatively and qualitatively assesses the positive and negative impacts of AOP on a number of changes applied to both the core architecture and variable features of SPLs ... Conditional compilation was the variability mechanism used ... in turn with the goal of supporting an analysis of the positive and negative impacts of AOP.”*
- **PS10** — *“This section presents and discusses ... the feasibility of using aspect-oriented development techniques to extract the eight optional features of ArgoUML-SPL.”*

#### 3.3.3 RQ2 - Kinds of Evaluation

The goal of this question was categorize kinds of evaluation used by research on conditional compilation use. It is important to identify which kinds of evaluation were

employed by the authors because this might be valuable information to researchers that intend to replicate the study analysis or apply some evaluation to their own research. The results from the current analysis will help us to better define our evaluation strategy.

Only one paper did not provide any information about the evaluation performed (PS06). All other seven studies applied some kind of automatic analysis, aided by some toolchain, developed by the authors or existing on the literature.

The evidences which show use of automatic analysis on the studies are presented below:

- **PS01** — *“As frontend, we have written the tool `source2rsf`, which extracts `cpp` directives from source files. ... The backend is implemented with our tool `undertaker`...”*
- **PS02** — *“We have performed empirical evaluations of our tool on the top test subjects used in Ernst et al.’s study.”*
- **PS03** — *“Furthermore, we used the tool `src2srcml` to generate an XML representation of the source code for measuring the granularity of extensions made with `cpp`.”*
- **PS04** — *“Our analysis requires representing source code as an AST. We use the tool `src2srcml` for this task.”*
- **PS05** — *“We built a tool to compute some metrics such as number of methods with preprocessor directives (`MDi`) and number of methods with feature dependencies (`MDe`).”*
- **PS07** — *“We built a prototype implementation (`R3V3RS3`) for the blueprint model and pattern matching facility, based on a robust `C` parser and regular expression matching... To validate the preprocessor blueprints’ ability to express and query for all occurrences of a conditional compilation pattern, we have applied `R3V3RS3` to the Parrot VM.”*
- **PS08** — *“We used programs we wrote to analyze 26 publicly available `C` software packages that represent a mix of application domains, user interface styles (graphical versus text-based, command-line versus batch), authors, programming styles, and sizes.”*

- **PS09** — *“The design stability evaluation of the Java and AspectJ versions were based on three conventional metrics suites for modularity, change impact, and feature interaction ... an independent group of five post-graduate students was responsible for implementing the successive evolution scenarios ... ”*
- **PS10** — *“We have proposed a framework for the evaluation and characterization of preprocessor-based product lines ... we have extended this framework with new metrics, such as those related to scattering and tangling. The extended framework supports the characterization of features according to different perspectives, including size, crosscutting behavior, granularity, and static location in the code.”*

### 3.3.4 RQ3 - Main Results

The objective of this section is to map the main results obtained by research on conditional compilation use. The results from the studies concerned to the research topics identified and described in Section 3.3.2. As results were found with respect to different research topics and present different data from different evaluations, they could not be summarized and are described as follow:

- **PS01** conducts two case studies: on Graph Product Line (GPL) and on Linux Kernel code. On the first one, it found out that variability described by the feature model covers less than 2% of the variability described in the source code, demonstrating the “semantic poorness” of *c++*-based variability. On the second one, it implements a consistency check which detects conditional blocks that are not selected under any possible input configuration (dead blocks) through SAT and BDD variants of undertaker. It found 4 dead blocks, two of which have been confirmed as new bugs.
- **PS02** introduces a characterization of inconsistent macro usage as a strong indicator of macro errors. It claims that all applications of the same macro should behave similarly. To validate this, the authors implemented an algorithm (CppChecker) to statically validate macro usage and applied it to 4 software packages. The results point that the authors found it difficult to detect macro errors using their notion of inconsistent macro usage. Despite that, the tool was effective in detecting common macro-related errors, reporting some false positives (4/6) on analysis, making it a practical tool for validating macro usage.

- **PS03** tries to answer some questions regarding program comprehension and refactoring through the analysis of forty preprocessor-based product lines. Concerned to program comprehension, it found that the variability of a software system increases with its size. Furthermore, the complexity of cpp-based SPL implementations increases with the increasing use of feature constants in feature expressions and of `#ifdef` nesting. Concerning refactoring too, data reveals that programmers use fine-grained extensions infrequently (1.8% in the average). Moreover, 89% of the extensions are heterogeneous, that is, it would suffice to use simpler mechanisms, such as mixins or feature modules.
- **PS04** aims to treat the question of how frequently programmers use undisciplined annotations and whether it is feasible to change them to disciplined annotations. It shows empirically, through analysis of 30 million lines of code, that programmers use *cpp* mostly in a disciplined way: about 84% of all annotations. To take advantage of disciplined annotations, it is feasible to accept certain kinds of and a certain amount of code replication as a means to promote a better tool support and a better readability of code.
- **PS05** proposes the concept of emergent interfaces to help developers to maintain feature tasks. To do so, it analyzed 3 metrics (number of fragments, number of features and lines of code) on methods of 44 SPLs, comparing the effort on maintenance tasks using VSoC and emergent interfaces. Authors conclude that, when using emergent interfaces, the maintainance effort reduces 37% for fragments, 25% for features and 26% for lines of code. Considering the total of methods analyzed, it achieved effort reduction in 36% of methods. From SPLs, the gain was 82%.
- **PS06** discusses the use of `#ifdefs` as a means to improve the portability. The authors claims that use `#ifdefs` as way to turn software portable is a bad idea. By means of examples of C News code, they discuss some aspects such as portable interfaces and levels of abstraction, and conclude that the use of `#ifdefs` often degrades modularity and readability. There are better ways to turn software portable, given some advance planning.
- **PS07** discusses the refactoring of conditional compilation use into aspects. It investigates some issues regarding that, as which patterns of conditional compilation aspects make sense and whether or not current aspect technology is able to express these patterns. The paper presents a graphical “preprocessor blueprint” model

which offers a queryable representation of the syntactical interaction of conditional compilation and the source code. Through the prototype R3V3RS3, the authors validate the approach in Parrot VM, concluding that eleven patterns matched capture conditional compilation usage in up to 99% of the source files, and that two of the six most popular patterns are hard to refactor into advice. Yet, authors claimed that, given the trade-offs derived for most of the discussed aspect implementations, conditional compilation is often still the preferred implementation technique to manage variability in C/C++ systems, despite the tangling and scattering.

- **PS08** analyzes *cpp* usage through various aspects, such as macro definitions, macro uses and dependencies upon macros. It analyzes 26 software packages and provides a set of data regarding mentioned previously aspects. From the results, it was possible to identify the difficulty of eliminating the use of preprocessor through translation to C++. It shows the way to development of preprocessor-aware tools, and provides tools as an extensible preprocessor, a *cpp* partial evaluator and a lint-like *cpp* checker. The resulting data can provide value to language designers, tool writers, programmers and software engineers.
- **PS09** presents a case study that assesses the positive and negative impacts of AOP on a number of changes applied to both the core architecture and variable features of two SPLs. To do this, the authors used three metrics suites (modularity, change impact and feature interaction) with conditional compilation in turn with the goal of supporting the assessment. From the results, the authors conclude that AO implementations of the studied SPLs tended to have more stable design particularly when a change target optional and alternative features, indicating that aspectual decompositions are superior in those situations. Furthermore, they found that AO mechanisms did not cope with the introduction of widely-scoped mandatory features or when changing a mandatory feature into alternatives.
- **PS10** describes an experiment involving the extraction of an SPL from ArgoUML, called ArgoUML-SPL. The authors used conditional compilation to extract eight complex and relevant features from it and used four set of metrics to evaluate the implementation of these features: size metrics, crosscutting metrics, granularity metrics, and localization metrics. From the results, it was possible to discuss some points about ArgoUML-SPL, such as crosscutting patterns found on its extraction and the feasibility of using aspect-oriented development techniques to extract the eight optional features. The authors made ArgoUML publicly available in intention

to promote its use among researchers and practitioners interested in product line related topics.

### 3.3.5 Other Results

As a way to increase the main results discussed in Section 3.3.4, other results are presented as follows. Table 3.5 shows the main tools developed and/or used in studies. These tools helped authors to provide data about their research.

Table 3.5: Summary of main tool developed/used in research papers.

Reference - PS: Primary Study	Toolchain
PS01	source2rsf, undertaker
PS02	CppChecker (ROSE, Boost Wave)
PS03	src2srcml
PS04	src2srcml
PS05	Feature Sensitive, Emergent Interface
PS06	Unknown
PS07	R3V3RS3
PS08	Not described
PS09	Unknown
PS10	Not described

Most of papers' authors developed their own tool to help on data collection and analysis. Some authors made use of other tools to help on their research, such as PS02. Two papers (PS06 and PS09) present collected data, conclude something about that, but did not specify which tool was used for this purpose. Furthermore, two papers (PS08 and PS10) mentioned the use of a developed tool, but did not describe it.

## 3.4 LIMITATIONS

As we stated at the beginning of this chapter, we perform a mapping study based on systematic mapping guidelines. The objective of these guidelines is to introduce a methodology for performing rigorous reviews of empirical evidence in software engineering studies. But in our mapping study, we simplify some steps in the systematic mapping process, aiming to meet the time constraints imposed and focus on the main contribution of this work. As the restrictions presented earlier in this chapter affect the rigour established by the systematic mapping guidelines, we are aware that our mapping study presents limitations and that these limitations can be rectified by future studies.

### **3.5 SUMMARY**

In this chapter we present a mapping study concerning preprocessor usage on software projects. We define a review protocol, when we specify the research questions and the methods that were used to undertake the mapping, as well as the terms used to compose the search string. The results from the mapping execution were presented, answering the research questions stated on the protocol, and describing other results concerning the study.



## CHAPTER 4

# DEPENDENCY ANALYSIS

Chapter 3 presented a mapping study about the use of conditional compilation in real software projects, describing the protocol used and showing the results about that. Motivated by the mapping results, in this chapter we present an analysis of feature dependency occurrence in real software projects regarding the use of the annotative approach called conditional compilation. We aim to complement other studies about conditional compilation use, as well as to provide intuitive data which can be used by researchers as input to their studies, such as programming languages design and preprocessors tool support.

Prior, we specify the concept of feature dependency in Section 4.1. We make some assumptions aiming to perform an initial assessment of dependency relations. At Section 4.2 we describe our empirical study, presenting our research questions, the process performed to answer them and showing the results of it, considering the threats to study validity.

### 4.1 FEATURE DEPENDENCIES

Software Product Lines share a common set of features that satisfy the needs of a particular market segment. By reusing assets, it is possible to construct products through features defined according to customers' requirements. In this context, features can be considered the semantic units by which we can differentiate programs in an SPL [TBD06].

Features are essential abstractions that both customers and developers understand. A feature is an end-user-visible program characteristic that is relevant to the stakeholders of the application domain. It represents important distinguishing aspects, qualities or characteristics of a family of systems. Customers and engineers usually speak of product characteristics in terms of the features the product has or delivers, so it's natural and intuitive to express any commonality or variability in terms of features.

Features usually share elements such as variables and methods among each other. In this context, when maintaining product lines, due to this sharing, developers may compromise feature modularization, which aims at achieving comprehensibility and changeability [Par72]. In other words, this means that a developer can introduce problems

to another feature that is not under his responsibility [RPTB10]. For example, he can modify an element—variable or method—in one feature that is declared or used by another feature. Whenever we have this sharing, we say that there is a feature dependency between the involved features [RQB<sup>+</sup>11].

In our research, we target some specific kind of feature dependency, which we call *simple dependencies*. These *simple dependencies* may be characterized by the following assumptions: (i) a dependency consists of a def-use attribute, that is, variables that are defined in one feature and used in another; and (ii) intra-procedural dependencies, that is, feature dependencies which only occur within methods boundaries. Despite some other kinds of feature dependencies [RPTB10], we focus on the simple ones to perform an initial assessment of dependencies occurrence.

To better explain the feature dependencies we consider here, we provide two scenarios where we use preprocessor directives to encompass feature code. We expose them in the following.

### Dependency Occurrence on Optional Feature

Listing 4.1 illustrates a variable that is declared in one feature and used in another. The code snippet was extracted from *Vim*<sup>1</sup> product line. *Vim* is an advanced text editor that seeks to provide the power of the de-facto Unix editor *Vi*<sup>2</sup>, with a more complete feature set. In this example, we declare the `echo` variable in a mandatory feature—there is no `#ifdef` statement encompassing the variable declaration—and we use such a variable in the `GOTO_FROM_WHERE_INCLUDED` feature. In this particular case, we use the variable as a parameter of the `echogets` method to compose a conditional structure whose result will be assigned to the `ok` variable. Notice that, because of the feature dependency, when maintaining the `echo` variable in the mandatory feature (i.e., by changing its type or its name), developers may introduce problems to the `GOTO_FROM_WHERE_INCLUDED` feature. Unfortunately, these problems are not always easy to detect, since maintenance tasks may lead not only to compilation problems, but also to behavioral ones [RPTB10, RQB<sup>+</sup>11].

---

Listing 4.1: Feature dependency occurrence on *Vim* text editor.

---

```

1 ...
2 int main (...) {
3     ...
4     int echo = ...

```

---

<sup>1</sup><http://www.vim.org/>

<sup>2</sup><http://ex-vi.sourceforge.net/>

```

5     ...
6     #ifdef GOTO.FROM.WHERE.INCLUDED
7         ...
8         ok = (echogets(Reason, echo) != NULL);
9         ...
10    #endif
11    ...
12 }
13 ...

```

---

Another example of feature dependency occurrence on optional feature is illustrated in Listing 4.2. The code snippet was extracted from *Kernel Linux*<sup>3</sup>. The *kernel Linux* is the operating system kernel used by the Linux family of Unix-like operating systems. It is an important piece of code composed by many features, configurable through its *kconfig* tool. In this case, notice that the variable `KERN_INFO` has been used on two different features: `CONFIG_WL1271` and `!CONFIG_WL1271`, encompassed by directives `#ifdef` and `#else`, respectively. Thus, any maintenance task in `KERN_INFO` variable can affect the result of their use on both features.

---

Listing 4.2: Feature dependency occurrence on *Kernel Linux*.

---

```

1 ...
2 #ifdef CONFIG_WL1271
3     printk(KERN_INFO "%s: CONFIG_WL1271 detected", __func__);
4 #else
5     printk(KERN_INFO "%s: CONFIG_WL1271 not detected", __func__);
6 #endif
7 ...

```

---

### Dependency Occurrence on Alternative Feature

Different from the two previous examples, Listing 4.3 illustrates another way in which feature dependencies can occur: between alternative features and a mandatory one. The code snippet was extracted from *Irssi*<sup>4</sup>. The *Irssi* is a terminal based IRC client for UNIX systems. It provides some great features such as autologging and configurable keybindings, and supports SILC and ICB protocols via plugins. In this example, we declare the `key` variable in two alternative features (encompassed by `#ifdef` and `#else` statements), with different types. Again, because of such a feature dependency, if the

---

<sup>3</sup><http://www.kernel.org/>

<sup>4</sup><http://irssi.org/>

developer change the `key` variable type, for instance, problems may be introduced to its use in the `g_array_append_val` method, which is in the mandatory feature. The method `g_array_append_val` would be prepared to accept the variable `key` as param, free of its type, and the developer would be aware of this.

---

Listing 4.3: Feature dependency occurrence on *Irssi*.

---

```
1 ...
2 void term_gets (...) {
3     ...
4     #ifdef WIDEC_CURSES
5         wint_t key;
6     #else
7         int key;
8     #endif
9     ...
10    g_array_append_val(buffer, key);
11    ...
12 }
13 ...
```

---

## 4.2 EMPIRICAL STUDIES

So far, we presented the concept of feature dependency as well as some examples extracted from real systems. Also, we mentioned that these dependencies can lead developers to introduce problems into other features they might not even responsible for.

Therefore, given the importance of dealing with feature dependencies, in this section we provide three empirical studies to better understand how these feature dependencies occur in practice. Recent work [LAL<sup>+</sup>10, LKA11] has focused on understanding how developers use *c++*'s variability mechanisms in practice. Complementing these studies, we are interested in understanding the occurrence of feature dependencies in real case studies.

In the next sections, we present our three empirical studies: we state the research questions, present how we collect the data to answer these research questions and show the results for each one.

### 4.2.1 Sample Projects & Collecting Data

For all three studies, we try to answer their research questions by analyzing 45 software projects. They vary from simple and small product lines, such as *mpsolve*, to complex and very large ones such as the Linux Kernel. Moreover, the projects belong to a variety of different domains, such as database systems, web servers, programming libraries, mobile games and operating systems, also different programming languages — C and Java. The majority of programs is written in C and all of them contain several features implemented using conditional compilation directives.

In order to provide compatibility in the analysis of projects developed in C and Java, our research considered conditional compilation directives based on the *Antenna*<sup>5</sup> project. *Antenna* provides a set of *Ant* tasks for developing wireless Java applications. With these tasks, its possible compile, preverify, package, obfuscate and run applications. It is also possible to manipulate a Java Descriptor, as well as convert JAR files to PRC files. Among other things, *Antenna* provides a simple Java preprocessor, similar to the ones known from C and other languages, that allows for conditional compilation and including source files. We used a subset of directives defined by *Antenna* which are also supported by *cpp*. This subset of directives comprises the following: `#ifdef`, `#ifndef`, `#elifdef`, `#elifndef`, `#if`, `#elif`, `#else`, `#debug` and `#mdebug`.

To perform our analysis we used a set of tools that helped us to do it and will be described through this section. By the use of these tools, we could apply a set of metrics and generate a large amount of projects data. The process for project data generation is described as follows. At first, all 45 projects were downloaded manually from repositories. Most of projects were hosted at *Sourceforge*<sup>6</sup>, which is a known web-based source code repository, acting as a centralized location for software developers to control and manage free and open source software development. Some projects, such as *Totem*, were downloaded from their own repositories, through their own official websites.

Next, we used the tool *src2srcml*<sup>7</sup> which parses the unprocessed code and generates an XML representation of it [MCK04]. This representation has the AST form, which allows us to navigate between annotations and perform computations. For C projects, we analyzed *.c* files, where we could find dependencies. Similarly, we did for Java projects, but some adjustments were necessary in the parsing process, aiming to normalize the XML as the representations of C projects.

---

<sup>5</sup><http://antenna.sourceforge.net/>

<sup>6</sup><http://sourceforge.net/>

<sup>7</sup><http://www.sdml.info/projects/srcml/>

With XMLs generated, we could compute the feature dependencies. For this, we developed a tool based on a recent work [LAL<sup>+</sup>10] called *pl-stats*<sup>8</sup>. We use our tool to compute a set of metrics regarding feature dependencies. Our tool computes the occurrence of dependencies for each project’s method, considering directives specified previously, generating a *.csv* data file for each project. Thereafter, all projects data are summarized in a one *.csv* file, which is used to provide the answers to our research questions. Figure 4.1 give us an overview for the process of project data generation.

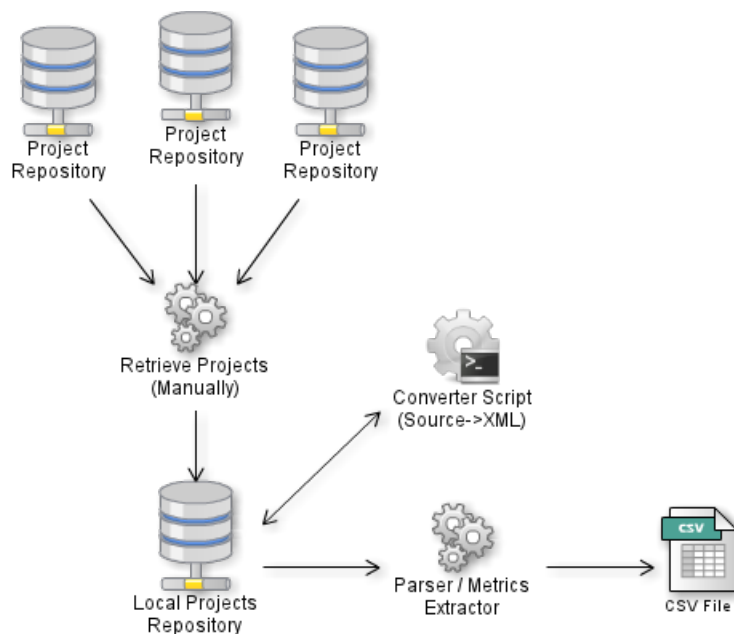


Figure 4.1: Overview for the process of projects data generation.

During our analysis, we had to exclude 77 from 56049 files (0.13% of all analyzed files) from computations. This occurred because *src2srcml* could not correctly parse these files, preventing us from performing the measurements.

### 4.2.2 Metrics

Aiming to understand how feature dependencies occur and answer the research questions stated in our studies (Sections 4.2.3 - 4.2.5), we introduce the following set of metrics:

**Number of Methods (NoM).** The NoM metric is used to calculate the average count of all class operations per class. It represents the size of a software program regarding number of methods.

<sup>8</sup><https://github.com/felipebuarque/PL-Stats>

**Source Lines of Code (SLoC).** The SLoC metric represents the size of a software program regarding lines of code. To measure it, we used the *CLoC*<sup>9</sup> tool. We use this metric with the NoM metric to better understand the influence of program size on feature dependencies.

**Methods with Directives (MDi).** The MDi metric is the number of methods which contains at least one directive. This metric will help us to understand the preprocessor usage on analyzed software.

**Methods with Dependencies (MDe).** The MDe metric is the number of methods exhibiting occurrences of dependencies. With this metric we will understand the impact of dependencies on projects' methods.

**Number of Fragments (NoFrag).** The metric NoFrag is the number of fragments that would have to be analyzed in feature dependency scenarios. We used this metric to understand the effort of maintenance tasks occurring in interprocedural dependencies.

**Number of Features (NoFeat).** The metric NoFeat is the number of features that would have to be analyzed in feature dependency scenarios. We used this metric to understand the effort of maintenance tasks occurring in interprocedural dependencies too.

**Number of Directive Occurrence (NoDiO).** The NoDiO metric indicates the occurrence of each directive on systems. We count how many times each directive appear on system methods. This metric will help us to understand the proportionality in which dependencies occur in each directive.

**Number of Dependencies in a Directive (NoDDi).** The NoDDi metric represents the occurrence of feature dependencies in each directive. We measure it by counting the directives involved in a dependency relation. For instance, if a variable is declared in a mandatory feature and is used in a feature encompassed by `#ifdef`, we count one feature dependency occurrence in `#ifdef` directive. If the variable is used another time in the same feature or in another feature encompassed by `#ifdef`, we count two feature dependency occurrences, and so forth.

**Number of Kinds of Directives (NoKDi).** The NoKDi metric represents the occurrence of different kinds of directives in a software program. If a project con-

---

<sup>9</sup><http://cloc.sourceforge.net>

tains directives like `#ifdef`, we count 1. If the same project contains another kind of directive like `#elifdef`, we increase this count to 2, and so fourth. In other words, we check how many different kinds of directives are used in each software program.

**Number of Directives with Dependencies (NoDiDe).** The NoDiDe metric indicates the number of different kinds of directives involved in dependency relations. If there are dependencies involving `#ifdef` directive, for example, we count 1. If there are dependencies involving another kind of directive like `#elif`, we increase this count to 2, and so forth. In other words, we check if each kind of directive is present at least one dependency relation.

**Number of Feature Expressions (NoFE).** The NoFE metric represents the number of feature expressions existing in a project. Feature expressions are the logical expressions evaluated by the preprocessor when it finds a directive declaration in code processing. These expressions may be formed by several terms through logical operators combinations, i.e. `A && B` or `A || B`.

**Number of Dependencies (NoDe).** The NoDe metric is the number of def-use dependencies occurrences in the software program. We calculate the dependencies for all methods based on assumptions stated on section 4.1.

We present the 45 projects and their respective computed metrics in Table 4.3 and Table 4.4. The percentage of data presented following the convention “average  $\pm$  standard deviation” and all plots are presented with the correlation coefficient of their respective metrics. Because the data are not normally distributed, we compute the correlation coefficient using the method of Kendall [KBS39]. The value of Kendall correlation coefficient ( $\tau$ ) vary from  $-1$  to  $1$ . The signal indicates if the relation direction between two variables is positive or negative, and the value suggests the relation strength between the variables analyzed. A perfect correlation ( $-1$  or  $1$ ) indicates that the value of a variable can be known by knowing the value of another one. For the other values, we relied on classification defined by Dancey & Reidy [DR11], which is presented in Table 4.1.

Based on classification schema presented above and the metrics defined previously, we can answer our research questions. The description of each study and their respective results are presented below.



Value	Strength
0	None
$0, 1 < \tau \leq 0, 3$	Weak
$0, 3 < \tau \leq 0, 6$	Moderate
$0, 6 < \tau \leq 1$	Strong

Table 4.1: Classification of correlation values.

### 4.2.3 First Study: Frequency of Feature Dependencies Occurrence

In our first study, we investigate how feature dependencies impact on SPL maintenance tasks. This was part of a major study comparing emergent interfaces [RPTB10] with VSoC [KAK08] regarding the benefits of the first over the second on SPL maintenance tasks. The idea was to measure the effort of both approaches when developers perform maintenance tasks on SPL by computing some metrics like lines of code, number of fragments and number of features.

To understand the frequency of dependencies occurrence on SPL, we answer the following research questions:

**Research Question 1:** How often do methods contain preprocessor directives?

**Research Question 2:** How often do methods with preprocessor directives contain feature dependencies?

To assess the research questions stated above, we analyzed the data provided from MDi and MDe metrics on Table 4.3. According to the results, these metrics vary significantly across the product lines. Some product lines have few directives in their methods. For instance, only 57 (2%) *Irssi* methods have directives. On the other hand, this number is much bigger in other ones, like *Python*, where 3473 (27.59%) methods have directives and *Mobile-RSS*, where 243 (26.94%) methods have directives. Following the convention “average  $\pm$  standard deviation”, our data reveals that  $7.61\% \pm 7.22\%$  of the methods use preprocessor directives. This answers our first research question concerning how often methods contains preprocessor directives.

Notice that the MDe metric is low in many product lines. However, we compute this metric with respect to all methods. Rather, if we take only methods with directives into consideration, we conclude that, when maintaining features—in other words, when maintaining code with preprocessor directives—the probability of finding dependencies increases a lot. Taking the *Sylpheed* product line as example, only 274 (7.54%) of its methods have directives and 197 (5.42%) have feature dependencies. Therefore, 71.9% of

methods with directives have feature dependencies (see MDe/MDi column in Table 4.3). Our data reveals that  $72.49\% \pm 17.69\%$  of the methods with directives have dependencies. This answers our second research question related to how often methods with preprocessor directives contain feature dependencies. All results of the major study can be verified on the published paper [RQB<sup>+</sup>11].

#### 4.2.4 Second Study: Complexity and Feature Dependencies

Although we have provided data on how often feature dependencies may occur in practice, we still need to complement this study to better understand feature dependencies and to analyze to what extent they might be a problem in practice.

Firstly, we focus on complexity. By complexity, we mean metrics such as (i) source lines of code; (ii) number of methods; (iii) number of features; and (iv) number of different kinds of preprocessor directives, like `#ifdef`, `#ifndef`, `#elif` etc. By using these metrics we then investigate if there are correlations between software complexity and feature dependencies. So, our second study consists of answering the following research questions:

**Research Question 1:** How program size is related to feature dependencies?

**Research Question 2:** How number of features is related to feature dependencies?

**Research Question 3:** How number of different kinds of preprocessor directives is related to feature dependencies?

Besides this correlation study, we also provide data with respect to *where* feature dependencies occur. By “where” we mean the different kinds of preprocessor directives. Therefore, we also answer the following research question:

**Research Question 4:** How often do feature dependencies occur for each kind of preprocessor directive?

Answering the three first questions, its important to better understand how software complexity correlates with the occurrence of feature dependencies. Answering the last question is important for assessing how often we can face a feature dependency on each preprocessor directive.

To assess the research questions stated above, we analyzed the data of Table 4.4. Not surprisingly, our data reveals that dependencies of a software system increase with its size. If we take a small program like *BestLapCC* (342 methods) and a large software

like *FreeBSD* (130320 methods), we notice that the number of dependencies increase substantially: 408 to 210408, respectively. We confirm this by the correlations between the metrics NoM and NoDe as well as SLoC and NoDe (see Figure 4.2), that correlates with a moderate degree. It turns out that large software systems usually have more method definitions. So, the probability of finding directives increases. Consequently, the probability of finding dependencies increases as well. This answer our first question about influence of program size on feature dependencies.

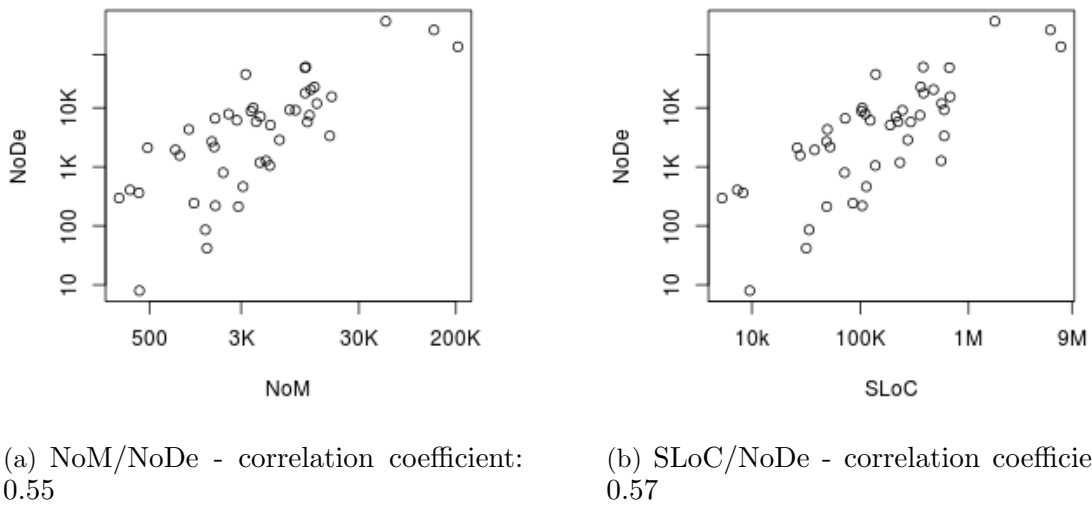


Figure 4.2: Correlations between NoM and NoDe, and SLoC and NoDe metrics.

To answer the second research question we stated above, we look for the NoFE metric. Our data reveals that the number of dependencies increases with the increasing of feature expressions definitions. Like program size, we confirm this by the correlation between the NoFE and NoDe metrics, which correlate tightly—0.77 (see Figure 4.3).

We can answer the third question from data presented in Figure 4.4. The number of feature dependencies increases with the use of different kinds of directives in software programs. For example, if we take a program with four different kinds of directives like *sendmail*, we can notice that the number of dependencies is low: 243. On other hand, if we take a program with the same size and use five kinds of directives like *Openvpn*, we can notice that the number of dependencies increases substantially: 2700. We confirm this by the correlation between the NoKDi and NoDe metrics, which correlate moderately—0.32.

Complementing the answer for the third question, we can notice that for all systems, feature dependencies occurred in at least two kinds of directives. Also, no system

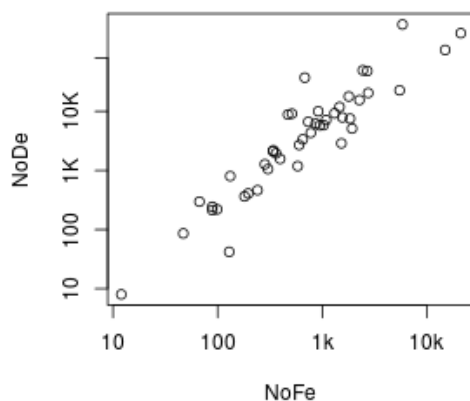


Figure 4.3: Correlation between NoFE and NoDe metrics. Correlation coefficient: 0.77.

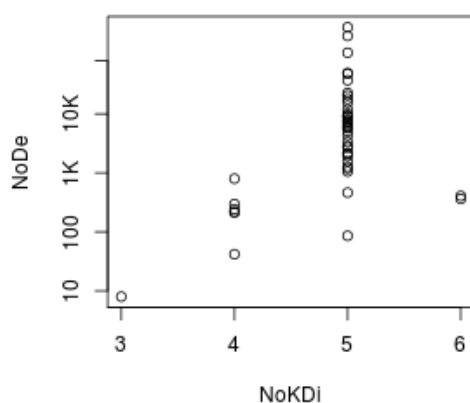


Figure 4.4: Correlation between NoKDi and NoDe metrics. Correlation coefficient: 0.32.

presented feature dependencies in more than five kinds of preprocessor directives (see NoDiDe column in Table 4.4). Even in large projects such as *kernel linux* and *freebsd* there were no dependencies involving some kinds of directives.

We provide data to answer the fourth question in Table 4.2. Notice that feature dependency occurrence is most frequent when using the `#if` directive. On average, 48.8% of dependencies from 45 projects we analyzed occur in this directive. Furthermore, as the use of `#else` and `#elif` directives has to be preceded by `#if` or `#ifdef` declaration, the occurrence of feature dependencies on these directives is lower—10.6% and 1.6% respectively.

Table 4.2: Dependency occurrence on *c++*'s variability mechanisms.

	<code>#elif</code>	<code>#else</code>	<code>#if</code>	<code>#ifdef</code>	<code>#ifndef</code>
<b>Mean</b>	1.6%	10.6%	48.8%	37.9%	7.9%
<b>SD</b>	3.6%	8.3%	19.6%	12.7%	4.3%

We do not present data about some directives—`#debug`, `#elifdef`, `#elifndef` and `#mdebug`—in Table 4.2, since the number of feature dependencies when using these directives is not relevant. The `#debug` and `#elifdef` directives have 12 and 104 dependency occurrences, respectively, which means less than 0.01% of the total number of dependencies. There were no dependency occurrences involving `#mdebug` and `#elifndef` directives.

In our analysis, we notice that developers implement debug system messages in most cases by using standard output functions (`System.out.print()` in Java and `printf()` in C) within other directives, such as `#if` or `#ifdef`, instead of using the `#debug` or `#mdebug` directives. Moreover, the use of `#elifdef` and `#elifndef` may be replaced by combinations between `#ifdef` and `#ifndef` with language logical operators, reducing their use and, consequently, the dependency occurrence.

#### 4.2.5 Third Study: Interprocedural Analysis

Additionally to the two studies we have just presented, we intend to perform an interprocedural analysis regarding the occurrence of feature dependencies on SPL. The idea is to complement the first study regarding the occurrence of feature dependencies through an interprocedural analysis. By interprocedural we can understand the occurrence of feature dependencies in two or more methods of the same file<sup>10</sup>, as we show in the example below.

Listing 4.4 illustrates an example where there is an occurrence of an interprocedural feature dependency. The code snippet was extracted from *Clamav* product line. *Clamav* is an open source antivirus engine designed for detecting trojans, viruses, malwares and other malicious treats<sup>11</sup>. In this example, we can see in line 5 that the variable `len` is assigned with the result of a call for the method `send_fdpass`. The `send_fdpass` method receive two params: `sockd` and `filename`. Notice that this call is held on the body of `dsresult` method and is encompassed by `#ifdef` directive, which represent the feature `HAVE_FD_PASSING`. Considering the variable `sockd`, we can notice that it is used on the body of `send_fdpass` method, in a mandatory feature (see line 13), that is: we use the

<sup>10</sup>At first, we intend to consider only interprocedural dependencies on methods of the same file.

<sup>11</sup><http://www.clamav.net/>

`sockd` variable as param of `send_fdpass` method, which is called on body of `dsresult` method. The call for `send_fdpass` method is a statement of `HAVE_FD_PASSING` feature and the variable is used on the mandatory feature of the method body. This characterizes an interprocedural feature dependency.

---

Listing 4.4: Interprocedural Feature dependency occurrence on *Clamav*.

---

```

1 int dsresult(int sockd, int scantype, const char *filename, int *printok,
   int *errors) {
2     ...
3     #ifdef HAVE_FD_PASSING
4         case FILDES:
5             len = send_fdpass(sockd, filename);
6             break;
7     #endif
8     ...
9 }
10 ....
11 static int send_fdpass(int sockd, const char *filename) {
12     ...
13     if(sendln(sockd, "zFILDES", 8)) {
14         ...
15     }
16     ...
17 }
```

---

This study aims to emphasize the benefits of using emergent interfaces concerning SPL maintenance tasks. We conduct a study based on comparing the effort when the developer is using emergent interfaces and when he is not using emergent interfaces approach to perform SPLs maintenance tasks. To do this, we measure the effort through the number of fragments (NoFa) and the number of the features (NoFe) which the developer would have to analyze to perform a maintenance on a variable shared among features and involved in an interprocedural dependency.

Table 4.5 present the data about our analysis. According to the results, the metrics data vary significantly across the product lines. Some product lines have few or have no data about interprocedural analysis (see *BestLapCC* and *Juggling* product lines). This occur mainly on Java projects: as a file tends to comprise only one Java class, the method calls tend to reference methods declared on other classes, which are defined on another files. And in this study, we are concerned about feature dependencies on methods of the same file.

Thus, following the convention “average  $\pm$  standard deviation”, our data reveals that when using emergent interfaces approach, developers has a gain of  $56.28\% \pm 21.70\%$  related to the number of fragments and  $46.29\% \pm 19.73\%$  related to the number of features they would have to analyze, compared to not using emergent interfaces. This reinforce the benefits of using emergent interfaces concerning SPL maintenance tasks, mainly when interprocedural dependencies scenarios are considered.

#### 4.2.6 Threats to Validity

In this section we discuss the threats to validity of the studies, presenting how the results could be affected.

##### Other Kinds of Dependencies

In our research, we care about dependencies caused by the sharing of an attribute declared into a method body. Our assumptions about *simple dependencies* did not consider other kinds of feature dependencies, such as chain of assignments, control flow dependencies, dependencies related to concurrent access, among others. The idea was to provide an initial assessment on feature dependencies and to motivate researchers to extend the work aiming to perform other analysis covering other kinds of dependencies, as we cite previously.

##### Counting of Lines of Code - SLoC Metric

For measuring the SLoC metric, we rely on tool *CLoC*. Although some tool limitations stated by the authors, as to identify some cases of comments constructs, we believe that these are not common and do not influence significantly the results.

##### Feature Expression Equality

Semantic equivalence of feature expressions like  $A \ \&\& \ B$  and  $B \ \&\& \ A$  are missing by our analysis. Our tool performs the analysis by using string comparison to check if different code fragments belong to the same expression. However, such cases are not common [LAL<sup>+</sup>10] and does not significantly influence the results.

##### Converting of Source Code on XML Documents

For converting the source code in AST form, we rely on tool *scr2srcml*. To compute the dependencies and measure the NoFE metric, our tool relies blindly on this mapping. We believe that the extensive test suite used by the authors of *scr2srcml* is reliable to verify it satisfactorily.

### **Number of Samples**

Despite the use of several projects to perform the assessments, the number of analyzed systems limits the power of the correlations, since each one is equivalent to the sample.

## **4.3 SUMMARY**

In this chapter we present the studies we performed concerning feature dependencies. We stated our assumptions about feature dependencies, presented the concept of *simple dependencies*, and showed examples of occurrence of these dependencies in real software programs. We then presented our empirical studies: we defined the metrics used on the studies, presented the research questions which guided the studies and explained how we collected the data about the projects. The results of each study were presented, showing a considerable amount of data about the projects and answering the research questions stated for each study.



Table 4.3: Data analysis - Part I.

**Version:** system version; **Language:** primary system language; **SLoC:** source lines of code; **NoM:** number of methods; **MDi:** methods with directives; **NoDiO:** number of directives occurrences; **MDe:** methods with dependencies; **MDe/MDi:** frequency of dependencies in methods with directives.

System	Version	Language	SLoC	NoM	MDi	#debug	#elif	#elifdef	#elifmdef	NoDiO				#mdebug	MDe	MDe/MDi
										#if	#ifdef	#ifmdef	#ifmdef			
cherokee	1.0.8	C	52776	1773	158	0	19	0	0	116	80	246	23	0	120	75.95%
clamav	0.96.4	C	139054	3284	307	0	38	0	0	230	157	496	64	0	226	73.62%
db	5.1.19	C	381315	10636	965	0	22	0	0	646	831	1054	740	0	819	84.87%
dia	0.97.1	C	137937	5262	160	0	2	0	0	96	128	128	11	0	103	64.38%
emacsc	23.2	C	232728	4333	242	0	8	0	0	167	166	370	67	0	105	43.39%
freebsd	8.1.0	C	5694620	130320	11712	0	407	0	0	5210	7503	16123	2887	0	8563	73.11%
gcc	4.5.1	C	1746963	50777	3021	0	175	0	0	3330	5200	2557	682	0	2010	66.53%
ghostscript	9.00	C	677020	17648	1279	0	13	0	0	569	772	1895	99	0	1035	80.92%
gimp	2.6.11	C	596081	16992	487	0	12	0	0	174	240	439	65	0	310	63.66%
glibc	2.12.1	C	595525	7748	777	0	50	0	0	558	446	852	214	0	445	57.27%
gnurmic	1.10.11	C	244968	8711	428	0	1	0	0	75	291	343	60	0	170	39.72%
gnuplot	4.4.2	C	72556	1804	278	0	16	0	0	206	221	534	100	0	179	64.39%
htpdp	2.2.17	C	214000	4379	534	0	15	0	0	297	555	616	70	0	411	76.97%
irssi	0.8.15	C	49085	2843	57	0	0	0	0	24	4	59	20	0	42	73.68%
libxml2	2.7.7	C	188960	5324	1433	0	36	0	0	160	954	1325	22	0	1220	85.14%
lighttpd	1.4.28	C	37953	831	139	0	9	0	0	105	81	256	42	0	97	69.78%
linux	2.6.36	C	7121949	208048	10198	0	147	0	0	2458	4440	12014	1091	0	7637	74.89%
lynx	2.8.7	C	111478	2349	503	0	11	0	0	435	458	1291	258	0	343	68.19%
minix	3.1.1	C	113768	3114	141	0	0	0	0	90	170	73	27	0	77	54.61%
Mplayer	1.0rc2	C	475160	11730	1408	0	164	0	0	751	782	2007	210	0	999	70.95%
MPSolve	2.2	C	9562	411	7	0	0	0	0	3	10	1	0	0	4	57.14%
openldap	2.4.23	C	223409	4026	516	0	54	0	0	239	199	994	58	0	401	77.71%
openvpn	2.1.3	C	48850	1694	304	0	44	0	0	101	207	387	12	0	257	84.54%
parrot	2.9.1	C	104322	1813	111	0	0	0	0	90	36	50	2	0	31	27.93%
php	5.3.3	C	664683	10436	1229	0	94	0	0	821	1222	1461	503	0	927	75.43%
pligim	2.7.5	C	292311	10965	577	0	8	0	0	281	329	545	89	0	373	64.64%
postgressql	8.4.5	C	564961	13199	835	0	49	0	0	568	170	1087	198	0	600	71.86%
privoxy	3.0.16	C	26222	482	101	0	27	0	0	121	78	266	40	0	87	86.14%
Python	2.7	C	362071	12590	3473	0	71	0	0	601	533	2183	2824	0	632	18.20%
sendmail	8.14.4	C	85600	1195	54	0	0	0	0	47	84	42	1	0	9	16.67%
sqlite	3.7.3	C	104594	3807	405	0	9	0	0	200	207	310	383	0	347	85.68%
subversion	1.6.13	C	558746	4894	197	0	4	0	0	101	154	169	18	0	132	67.01%
syphheed	3.0.3	C	102983	3634	275	0	3	0	0	187	286	206	37	0	192	69.82%
tcl	8.5.9	C	123778	2761	294	0	11	0	0	152	202	652	119	0	232	78.91%
totem	3.1.0	C	33746	1492	35	0	1	0	0	10	17	24	1	0	21	60.00%
vim	7.3	C	274858	6354	702	0	10	0	0	277	311	1448	88	0	345	49.15%
xfig	3.2.5b	C	71740	2112	83	0	0	0	0	42	28	132	35	0	47	56.63%
xinelib	1.1.19	C	383727	10501	1037	0	56	0	0	552	574	1372	192	0	751	72.42%
xorgserver	1.7.1	C	356300	11425	1160	0	30	0	0	335	902	1227	132	0	836	72.07%
xterm	2.6.1	C	49752	1080	266	0	27	0	0	136	679	255	41	0	221	83.08%
BestLapCC	1.0	Java	7340	342	70	0	91	2	0	44	79	17	54	0	34	48.57%
Juggling	1.0	Java	8279	407	63	0	39	2	0	29	135	17	7	0	36	57.14%
lampiro	1.0	Java	31774	1538	124	0	0	0	0	0	0	26	10	128	10	8.06%
mobile-rss	1.0	Java	27879	902	243	0	0	6	0	39	0	731	39	0	194	79.84%
MobileMedia	0.9	Java	5305	276	22	0	4	0	0	1	16	61	0	0	13	59.09%

Table 4.4: Data analysis - Part II.

**SLoC**: source lines of code; **NoM**: number of methods; **NoFE**: number of feature expressions; **NoDe**: number of dependencies; **NoDDi**: number of dependency occurrence on each directive; **NoFE**: number of feature expressions; **NoDe**: number of dependencies; **NoDDi**: number of dependency occurrence on each directive; **NoKDi**: number of kinds of directives; **NoDiDe**: number of directives with dependencies.

System	SLoC	NoM	NoFE	NoDe	#debug	#elif	#elifdef	#elifndef	NoDDi				#mdebug	NoKDi	NoDiDe
									#else	#if	#ifdef	#ifndef			
cherokee	52776	1773	341	2184	0	147	0	0	688	168	1259	154	0	5	5
clamav	139054	3284	681	37240	0	249	0	0	1919	677	35415	194	0	5	5
db	381315	10636	2442	49495	0	123	0	0	2587	8374	14837	28431	0	5	5
dia	137937	5262	303	1055	0	0	0	0	469	239	369	24	0	5	4
emacs	232728	4333	583	1186	0	4	0	0	403	172	431	252	0	5	5
freeltd	5694620	130320	21091	210408	0	2975	0	0	35924	67319	113966	12310	0	5	5
gcc	1746963	50777	5846	295167	0	1157	0	0	6722	284464	6385	2634	0	5	5
ghostscript	677020	17648	2265	15550	0	386	0	0	2138	4365	9484	403	0	5	5
gimp	596081	16992	651	3361	0	49	0	0	889	988	1510	182	0	5	5
glibc	595525	7748	1311	9286	0	363	0	0	2135	2392	4890	1440	0	5	5
gnome	244968	8711	513	9150	0	1	0	0	311	99	7944	864	0	5	5
gnuplot	72556	1804	735	6662	0	47	0	0	712	1000	5098	200	0	5	5
htpdt	214000	4379	1087	7141	0	81	0	0	1321	2619	3648	260	0	5	5
irssi	49085	2843	89	213	0	0	0	0	62	0	144	11	0	4	3
libxml2	188960	5324	1928	5125	0	57	0	0	471	964	4053	37	0	5	5
lighttpd	37953	831	361	1947	0	179	0	0	195	40	1362	219	0	5	5
linux	7121949	208048	14964	109394	0	537	0	0	10803	20989	76382	4018	0	5	5
lynx	111478	2349	1558	7917	0	8	0	0	2045	1647	4359	737	0	5	5
minix	113768	3114	241	464	0	1	0	0	87	237	103	40	0	5	5
MPlayer	475160	11730	2748	20503	0	1388	0	0	5518	3261	11019	823	0	5	5
MPSolve	9562	411	12	8	0	0	0	0	2	6	0	0	0	5	2
openldap	223409	4026	1030	5856	0	216	0	0	777	845	4600	107	0	5	5
openvpn	48850	1694	602	2700	0	387	0	0	186	806	1359	54	0	5	5
parrot	104322	1813	99	220	0	0	0	0	99	47	81	7	0	4	4
php	664683	10436	2694	48275	0	547	0	0	7373	28780	12914	7169	0	5	5
pidgin	292311	10965	946	5794	0	16	0	0	1032	613	4010	522	0	5	5
postgresql	564961	13199	1464	11907	0	273	0	0	2203	4956	4725	760	0	5	5
privoxy	26222	482	338	2118	0	91	0	0	297	511	1365	27	0	5	5
Python	362071	12590	5489	22694	0	4582	0	0	1953	10472	7403	544	0	5	5
sendmail	85600	1195	89	243	0	0	0	0	52	274	105	2	0	4	4
sqlite	104594	3807	916	10014	0	58	0	0	937	1191	2161	7118	0	5	5
subversion	558746	4894	283	1274	0	50	0	0	141	423	636	47	0	5	5
syphed	102983	3634	470	8841	0	16	0	0	2695	5242	1135	77	0	5	5
tel	123778	2761	866	6184	0	55	0	0	504	2632	1784	1736	0	5	5
totem	33746	1492	47	86	0	0	0	0	7	18	61	0	0	5	3
vim	274858	6354	1525	2878	0	0	0	0	549	438	2139	184	0	5	4
xfig	71740	2112	132	801	0	0	0	0	183	141	371	131	0	4	4
xinefb	383727	10501	1793	17879	0	519	0	0	5400	3326	7339	2179	0	5	5
xorgserver	356300	11425	1846	7530	0	171	0	0	999	1734	4741	569	0	5	5
xterm	49752	1080	777	4338	0	110	0	0	693	3293	709	190	0	5	5
BestLapCC	7340	342	196	408	0	146	0	0	29	144	76	98	0	6	5
Juggling	8279	407	181	364	0	40	0	0	59	157	108	27	0	6	5
lampiro	31774	1538	129	42	12	0	0	0	0	0	0	30	0	4	2
mobile-rss	27879	902	396	1571	0	0	104	0	81	0	1384	64	0	5	4
MobileMedia	5305	276	67	297	0	0	0	0	0	19	334	0	0	4	2

Table 4.5: Data analysis - Part III.

**SLoC**: source lines of code; **NoM**: number of methods; **EI**: emergent interfaces; **Non-EI**: non-emergent interfaces; **NoFrag**: number of fragments; **NoFeat**: number of features.

System	SLoC	NoM	EI		Non-EI	
			NoFrag	NoFeat	NoFrag	NoFeat
cherokee	52776	1773	182	91	627	208
clamav	139054	3284	262	250	735	504
db	381315	10636	2785	2590	7552	5920
dia	137937	5262	23	22	89	68
emacs	232728	4333	443	385	1428	925
freetsd	5694620	130320	9272	7385	27353	15486
gcc	1746963	50777	2300	2048	7381	4860
ghostscript	677020	17648	549	462	1899	1358
gimp	596081	16992	235	203	761	522
glibc	595525	7748	393	279	1959	938
gnumeric	244968	8711	302	285	722	512
gnuplot	72556	1804	279	242	624	440
httpd	214000	4379	382	293	1297	787
irssi	49085	2843	32	29	49	41
libxml2	188960	5324	8624	6991	25288	13675
lighttpd	37953	831	1558	1072	3251	3132
linux	7121949	208048	175	173	339	262
lynx	111478	2349	824	776	2976	1923
mix	113768	3114	152	122	365	262
Mplayer	475160	11730	936	862	2351	1788
MPSolve	9562	411	0	0	0	0
opendap	223409	4026	239	225	859	574
openvpn	48850	1694	110	94	235	174
parrot	104322	1813	13	10	106	88
php	664683	10436	2461	2146	5531	3843
pidgin	292311	10965	406	328	1371	660
postgresql	564961	13199	1016	972	1823	1341
privoxy	26222	482	97	73	478	249
Python	362071	12590	1177	1008	3564	2211
sendmail	85600	1195	342	299	1440	966
sqlite	104594	3807	721	657	1241	1015
subversion	558746	4894	424	405	708	586
sympheed	102983	3634	286	248	713	429
tcl	123778	2761	104	99	198	161
totem	33746	1492	4	4	18	12
vim	274858	6354	1197	1091	2900	1765
xfig	71740	2112	35	30	98	40
xine-lib	383727	10501	914	822	1958	1401
xorgserver	356300	11425	376	372	1129	699
xterm	49752	1080	352	295	919	600
BestLapCC	7340	342	0	0	0	0
Juggling	8279	407	0	0	0	0
lampiro	31774	1538	22	22	61	35
mobile-rss	27879	902	0	0	0	0
MobileMedia	5305	276	9	8	11	11

# CONCLUDING REMARKS

This work presented an initial analysis of feature dependencies occurrence on SPL. We develop a tool based on a previous work which compute a set of metrics were defined among this work, providing a large amount of data regarding preprocessor usage and feature dependency occurrence on SPL.

Initially, to know about the relevant studies on preprocessor usage, we perform a literature review based on systematic mapping features. We define a review protocol, when we specify the research questions and the methods that were used to undertake the mapping. As result, we identify 8 empirical studies on use of conditional compilation in software projects. From these, we answer the research questions prior defined, describing how each one answer the questions and pointing other results identified among study.

Beyond the mapping study concerning preprocessor usage, we concentrate to analyze the occurrence of feature dependencies on SPL. We provide an initial analysis of feature dependencies occurrence through three empirical studies. The first one focuses on the frequency of feature dependencies occurrence on SPL. The data show that  $72.49\% \pm 17.69\%$  of methods with some directive have feature dependencies.

The second study focus on understand the correlation between complexity and feature dependencies, and where these dependencies occur. By *complexity*, we mean metrics such as source lines of code, number of methods, number of features and number of different kinds of preprocessor directives. By *where* we mean the different kinds of preprocessor directives we can find feature dependencies. The data show that complexity and feature dependencies are closely related and we present this through charts and coefficient correlation for each metric defined. Moreover, we show that feature dependency occurrence is most frequent when using the `#if` directive. On average,  $48.8\%$  of dependencies from 45 projects we analyzed occur in this directive.

The third study complements the first two through an interprocedural analysis, where we compare the effort to maintain dependent features using emergent interfaces and non emergent interfaces approaches in SPLs. We found that, when using emergent interfaces approach, developers has a gain of  $56.28\% \pm 21.70\%$  related to the number of fragments and  $46.29\% \pm 19.73\%$  related to the number of features they would have to analyze,

compared to not using emergent interfaces.

This large amount of data, as well as the analysis performed can be a good starting point to understand the occurrence of feature dependencies on SPL and might be used for researchers as input to their studies in preprocessors support tools, for example.

## 5.1 CONTRIBUTIONS

The main contributions of this research are detailed as follows:

- A large dataset regarding preprocessor usage and feature dependencies on software projects. The total of 45 software projects from different domains, sizes and languages were analyzed and provided a rich amount of data. Researchers can use our data as input to their studies on programming languages or preprocessor support tools. For example, we have a tool for Eclipse IDE that computes feature dependencies. Because we use data-flow analysis, sometimes it takes a long time to compute them. Now that we know that the feature dependencies occur more frequently in `#if` and `#ifdef` statements (when compared to `#ifndef`, for instance), our tool can focus on these directives, exchanging precision for better performance;
- A tool to compute a set of metrics regarding preprocessor usage on software projects. Our tool extend a previous tool [LAL<sup>+</sup>10] by considering conditional compilation directives in C and Java projects. Furthermore, our tool compute a set of metrics regarding the occurrence of *simple dependencies* on software projects. Although compute metrics about occurrence of *simple dependencies*, our tool can be extended to compute other kinds of feature dependencies, as *chain of assignments*. Also, other metrics computations regarding these dependencies can be implemented;
- A mapping study regarding preprocessor usage on software projects. As the study was conducted following systematic mapping features, this allows other researchers to replicate it, adjusting some variables, like the terms of the search string, according the goals of their researches.

## 5.2 RELATED WORK

The first related work discussed here [LAL<sup>+</sup>10] investigate the way developers use preprocessors to implement variability. To do so, the authors analyzed forty preprocessor-based product lines implemented in C of different domains. They formulated research questions focused on two research area on software development: program comprehension

and refactoring, in the context of preprocessor-based product lines. For each area mentioned before, they created and computed many metrics that represent these objectives, for instance, *Number of Feature Constants (NOFC)* and *Lines of Feature Code (LOF)* to represent the program comprehension and *Granularity (GRAN)* and *Type (TYPE)* to represent refactoring. Thus, to answer the research questions, they were aided by a tool which was able to compute the metrics. We complement this work by taking feature dependencies into consideration. We extended the tool provided by these researchers so that we can compute metrics like *number of methods with directives (MDi)* and *number of methods with feature dependencies (MDe)*. Moreover, we also complement this work with respect to product lines of different languages. Beyond analyze the product lines written in C, we perform an analysis in five product lines written in Java. This allowed us to expand our results to another class of products.

Another related work [STL10] examined the use of preprocessor-based code in systems written in C. Directives like `#ifdefs` are indeed powerful, so that programmers can make all kinds of annotations using them. Hence, developers can introduce subtle errors like annotating a closing bracket but not the opening one. This implies in which authors called of “undisciplined” annotation. They distinguish between “disciplined” annotations, which hold properties useful for preprocessor-aware parsing tools so we can represent annotations as nodes in the AST, and “undisciplined” annotations, which do not align with this structure and complicate tool development. To do that, a set of metrics was defined and used to compute the projects data. The authors found that the majority of the preprocessor usage are disciplined. Specifically, they found that the case studies have 84.4% of their annotations disciplined. We also analyzed several systems, but we focus on dependencies among features implemented with preprocessors.

Still on the studies concerning preprocessor usage, Ernst et al. analyzed the preprocessor usage covering mainly macro definitions using `#define` directives [EBN02]. The authors points out that, despite their evident shortcomings, the controlled use of preprocessors can improve portability, performance or even readability. While `#define` directives are also important for tool support, conclusions based on their usage don't give a complete view of how other directives also effects tool support. Like our work, they compute the occurrence of some conditional compilation directives as well. But, differently, we did not find a lot of preprocessor usage. We focus only on methods with conditional compilation directives, while they focus on entire software code (not only methods). We also analyzed several kinds of conditional compilation directives, but we focus on those that could be in a dependency relation.

Sincero et al. performed a study aiming to exploit the implementation variability provided for the code with *cpp* directives [STLSP10]. In this study, is presented an approach to extract a variability model directly from source code that uses conditional compilation directives. The authors formalize the *cpp* directives using propositional logic through the extraction of a boolean formula from the preprocessor-based source code, which represents the variability of compilation units. To do that, they build a tool chain which produces propositional formulas that can be further processed with standard Binary Decision Diagram (BDD) packages or SAT solvers. With this, the authors conduct two case studies. In the first, they compare the feature model provided by the *cpp*-based version of the Graph Product Line (GPL) with the variability model extracted from its source code, regarding the number of variants, blocks and directives per file. They could find that the variability described by the feature model covers less than 2% of the variability described in the source code, demonstrating the semantic poorness of the *cpp*-based variability. In the second, they apply their approach on a very large SPL, the linux kernel code. They could generate the boolean formulas for all source code files in less than 30 minutes, verifying the tool performance. We also build a tool, that compute a set of metrics regarding *cpp*-based variability. But we focus in understand the dependency relations between the features implemented with this resources.

In addition to these previous studies, Adams et al. investigate if is a good idea to refactor the use of conditional compilation into aspects [ADMTH09]. Despite the known advantages of this technique to handle some issues of conditional compilation as code tangling and code scattering, the authors were interested in for which patterns of conditional compilation aspects make sense and whether or not current aspect technology is able to express these patterns. They present a graphical model which offers a queryable representation of the syntactical interaction of conditional compilation and the source code. To evaluate their model, the authors conduct a case study on a virtual machine, which shows that preprocessor blueprints are able to express and query for the four commonly known patterns of conditional compilation usage, and that they allow to discover seven additional important patterns. By correlating each pattern's potential for refactoring into advice and each pattern's evolution of the number of occurrences, their show that refactoring into advice in the Parrot VM is a good alternative for three of the eleven patterns, whereas for the other patterns trade-offs have to be considered. Although we analyze a virtual machine too, we try to understand the occurrence of dependencies between features that are implemented with use of conditional compilation directives.

### 5.3 FUTURE WORK

As future work, we intend to perform a deeper analysis of the resulting data, trying to understand the relations between feature dependencies and other projects aspects, such as the programming language. Additionally, we intend to improve our tool to download the projects and identify if they use preprocessor directives automatically, enhancing the process of data sampling. Furthermore, we intend to improve our tool to compute other metrics, as well as use other techniques like data-flow analysis to improve our analysis regarding preprocessor usage and feature dependency occurrence. More specifically, we intend to analyze other projects aspects, as follows:

- The nature of features - the features are representative (are big or small)? With respect to granularity, they are fine or coarsened? This can categorize better the features and, consequently, the dependencies computed;
- Other kinds of dependencies - *chain of assignments* and feature dependencies among different files of the project intend to be considered to cover more dependencies occurrences and improve the resulting data.

The idea is to provide a tool to support SPL projects which use preprocessors to implement variability. The focus is get developers to be aware of feature dependencies occurrence, since some variability model like configuration knowledge is used as input to the tool.

In summary, this work provide a large amount of data regarding preprocessor usage and feature dependency occurrence on SPL, as well as the tool used to compute this data. Furthermore, we provide an initial analysis of feature dependencies occurrence through three empirical studies. We believe that these analysis are a good start point to understand the occurrence of feature dependencies on SPL and might be used together the amount of data for researchers, language designers, tool writers, programmers and software engineers, as input to their studies and projects.



# Appendices

## A.1 PRIMARY STUDIES

ID	Year	Source	Reference
PS01	2010	EI Compendex	Julio Sincero, Reinhard Tartler, Daniel Lohmann and Wolfgang Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. <i>SIGPLAN Not.</i> ,46(2):33-42, October 2010.
PS02	2009	EI Compendex	Andreas Saebjoernsen, Lingxiao Jiang, Daniel Quinlan, and Zhendong Su. Static validation of c preprocessor macros. In <i>Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09</i> , pages 149-160, Washington, DC, USA, 2009. IEEE Computer Society.
PS03	2010	Scopus	Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In <i>Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)</i> , pages 105-114, New York, NY, USA, 2010. ACM.
PS04	2011	Scopus	Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In <i>Proceeding of the 10th International Conference on Aspect Oriented Software Development (AOSD'11)</i> , pages 191-202, New York, NY, USA, March 2011. ACM.
PS05	2011	Scopus	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 <i>Proceedings of the 10th ACM international conference on Generative programming and component engineering, GPCE '11</i> , pages 23-32, New York, NY, USA, 2011. ACM.
PS06	1992	Scopus	Henry Spencer and Geoff Collyer. <code>#ifdef</code> considered harmful, or portability experience with C news. In <i>Proceedings of the Usenix Summer 1992 Technical Conference</i> , pages 185-198, Berkeley, CA, USA, June 1992. Usenix Association.

PS07	2009	Scopus	Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In <i>Proceedings of the 8th ACM international conference on Aspect-oriented software development</i> , AOSD'09, pages 243-254, New York, NY, USA, 2009. ACM.
PS08	2002	Scopus	Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. <i>IEEE Trans. Softw. Eng.</i> , 28:1146-1170, December 2002.
PS09	2008	Other	Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uirá Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. <i>Proceedings of the 30th international conference on Software engineering</i> , ICSE'08, pages 261-270, New York, NY, USA, 2008. ACM.
PS10	2011	Other	Marcus Vinicius Couto, Marco Tulio Valente and Eduardo Figueiredo. Extracting Software Product Lines: A Case Study Using Conditional Compilation. <i>Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering</i> , CSMR'11, pages 191-200, Washington, DC, USA. IEEE Computer Society.

## BIBLIOGRAPHY

- [ADMTH09] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 243–254, New York, NY, USA, 2009. ACM.
- [AJC<sup>+</sup>05] 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'05)*, volume 3714 of *LNCS*, pages 70–81. Springer-Verlag, September 2005.
- [Alv07] Vander Alves. *Implementing Software Product Line Adoption Strategies*. PhD thesis, Federal University of Pernambuco, Recife, Brazil, March 2007.
- [BC90] Gilad Bracha and William Cook. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA/ECOOP'90)*, pages 303–311, New York, NY, USA, 1990. ACM Press.
- [Boe86] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11:14–24, August 1986.
- [Bos02] Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 257–271, London, UK, UK, 2002. Springer-Verlag.
- [CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

- [dABMN<sup>+</sup>07] Jorge Calmon de Almeida Biolchini, Paula Gomes Mian, Ana Candida Cruz Natali, Tayana Uchôa Conte, and Guilherme Horta Travassos. Scientific research ontology to support systematic review in software engineering. *Adv. Eng. Inform.*, 21:133–151, April 2007.
- [DR11] Prof Christine Dancey and John Reidy. *Statistics without Maths for Psychology*. Prentice Hall, 5th edition, April 2011.
- [DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *J. Syst. Softw.*, 74(2):173–194, January 2005.
- [EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng.*, 28:1146–1170, December 2002.
- [Fav96] Jean-Marie Favre. Preprocessors from an abstract point of view. In *Proceedings of the 1996 International Conference on Software Maintenance, ICSM '96*, pages 329–, Washington, DC, USA, 1996. IEEE Computer Society.
- [FBB<sup>+</sup>99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1st edition edition, July 1999.
- [F.P87] Jr. Brooks F.P. No silver bullet essence and accidents of software engineering. *Computer*, 20:10–19, 1987.
- [FP98] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 2nd edition, February 1998.
- [GA01] Critina Gacek and Michalis Anastasopoulos. Implementing product line variabilities. In *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context, SSR '01*, pages 109–117, New York, NY, USA, 2001. ACM.
- [Jar07] Stanislaw Jarzabek. *Effective Software Maintenance and Evolution*. Auerbach Publications, May 2007.

- [JBZZ03] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. Xvcl: Xml-based variant configuration language. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 810–811, Washington, DC, USA, 2003. IEEE Computer Society.
- [Jon86] Caper Jones. *Programming Productivity*. Mcgraw-Hill, January 1986.
- [JV04] Doug Janzen and Kris De Volder. Programming with crosscutting effective views. In *18th ECOOP*, pages 195–218, 2004.
- [KA09] Christian Kästner and Sven Apel. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.
- [KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 311–320, New York, NY, USA, 2008. ACM.
- [KBS39] M. G. Kendall and B. Babington Smith. The problem of m rankings. *The Annals of Mathematical Statistics*, 10(3):275–287, September 1939.
- [Kit07] Barbara Kitchenham. Guidelines for Performing Systematic Literature Reviews in Software Engineering. EBSE Technical Report Version 2.3, Keele University, UK, July 2007.
- [KKHL10] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: toward type checking #ifdef variability in c. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD'10)*, pages 25–32, New York, NY, USA, 2010. ACM.
- [KLM<sup>+</sup>97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
- [KMPY05] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. A Case Study in Refactoring a Legacy Component for Reuse in a Product Line. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 369–378, Washington, DC, USA, 2005. IEEE Computer Society.

- [KR88] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.
- [Kru02] Charles W. Krueger. Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering, PFE '01*, pages 282–293, London, UK, UK, 2002. Springer-Verlag.
- [Kru06] Charles W. Krueger. Introduction to the emerging practice of software product line development. *Practical Knowledge for the Software Developer, Tester and Project Manager*, 4, 2006.
- [KS94] 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 '94*, pages 49–57, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [LAL<sup>+</sup>10] 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'10)*, pages 105–114, New York, NY, USA, 2010. ACM.
- [LKA11] 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'11)*, pages 191–202, New York, NY, USA, March 2011. ACM.
- [LSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- [MCK04] Jonathan I. Maletic, Michael Collard, and Huzefa Kagdi. Leveraging xml technologies in developing program analysis tools. In *in Proceedings of 4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 80–85, 2004.
- [McM06] Robert McMillan. Intel reissues buggy patch, August 2006.

- [McM10] Robert McMillan. After buggy patch, criminals exploit windows flaw., June 2010.
- [MLWR01] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: an exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 275–284, Washington, DC, USA, 2001. IEEE Computer Society.
- [Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972.
- [PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [PBvdL05] Klaus Pohl, Gunter Bockle, and Frank J. van der Linden. *Software Product Line Engineering*. Springer, 2005.
- [PFMM08] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering, EASE'08*, pages 68–77, Swinton, UK, UK, 2008. British Computer Society.
- [Pig97] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley, October 1997.
- [PO97] T. Troy Pearce and Paul W. Oman. Experiences developing and maintaining software in a multi-platform environment. In *Proceedings of the International Conference on Software Maintenance, ICSM '97*, pages 270–, Washington, DC, USA, 1997. IEEE Computer Society.
- [RPTB10] Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent Feature Modularization. In *Onward! 2010, affiliated with ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'10)*, pages 11–18, New York, NY, USA, 2010. ACM.
- [RQB<sup>+</sup>11] Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Tárzis 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 '11, pages 23–32, New York, NY, USA, 2011. ACM.
- [SC92] Henry Spencer and Geoff Collyer. `#ifdef` considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer 1992 Technical Conference*, pages 185–198, Berkeley, CA, USA, June 1992. Usenix Association.
- [SGC<sup>+</sup>03] Claudio Sant’anna, Alessandro Garcia, Christina Chavez, Carlos Lucena, and Arndt v. von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.
- [SJQS09] Andreas Saebjoernsen, Lingxiao Jiang, Daniel Quinlan, and Zhendong Su. Static validation of c preprocessor macros. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 149–160, Washington, DC, USA, 2009. IEEE Computer Society.
- [SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, February 2003.
- [STL10] Julio Sincero, Reinhard Tartler, and Daniel Lohmann. An Algorithm for Quantifying the Program Variability Induced by Conditional Compilation. Technical report, University of Erlangen, Dept. of Computer Science, January 2010.
- [STLSP10] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. *SIGPLAN Not.*, 46(2):33–42, October 2010.
- [TBD06] 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'06)*, pages 191–200, New York, NY, USA, 2006. ACM.

- [tio12] Tiobe software. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>, May 2012.

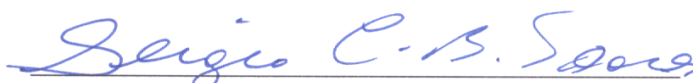
Dissertação de Mestrado apresentada por **Felipe Buarque de Queiroz** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Analysing Feature Dependencies in Preprocessor-Based Systems**" orientada pelo Prof. **Sérgio Castelo Branco Soares** e aprovada pela Banca Examinadora formada pelos professores:



Prof. Fernando José Castor de Lima Filho  
Centro de Informática / UFPE



Prof. Uirá Kulesza  
Deptº de Informática e Matemática Aplicada



Prof. Sérgio Castelo Branco Soares  
Centro de Informática / UFPE

Visto e permitida a impressão.  
Recife, 24 de agosto de 2012



**Prof. Nelson Souto Rosa**

Coordenador da Pós-Graduação em Ciência da Computação do  
Centro de Informática da Universidade Federal de Pernambuco.