UNDERSTANDING THE SHAPE OF FEATURE CODE

RODRIGO BARBOSA QUEIROZ

UNDERSTANDING THE SHAPE OF FEATURE CODE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais – Departamento de Ciência da Computação como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente

Belo Horizonte

Julho de 2015

RODRIGO BARBOSA QUEIROZ

UNDERSTANDING THE SHAPE OF FEATURE CODE

Dissertation presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais – Departamento de Ciência da Computação in partial fulfillment of the requirements for the degree of Master in Ciência da Computação.

Advisor: Marco Túlio de Oliveira Valente

Belo Horizonte

July 2015

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Queiroz, Rodrigo Barbosa.

Q3u Understanding the shape of feature code / Rodrigo Barbosa Queiroz — Belo Horizonte, 2015.
xix, 82f. : il. ; 29cm.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação

Orientador: Marco Túlio de Oliveira Valente

1. Computação - Teses. 2. Engenharia de linha de produto de software - Teses. 3. Software - Reutilização - Teses. I. Orientador. II. Título.

519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Understanding the shape of feature code

RODRIGO BARBOSA DE QUEIROZ

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

Dr. André Cavalcante Hora
Pós-Doutorando DCC

[illegible]

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

[illegible]

Profa. Mariza Andrade da Silva Bigonha
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 31 de julho de 2015.

Resumo

Feature annotations (por exemplo, diretivas do preprocessador C, na forma de *#ifdefs*) são usadas para controlar extensões de código relacionadas a uma *feature*. Por muito tempo, tais anotações têm sido consideradas indesejáveis. Seu uso excessivo pode aumentar o risco de *ripple effects*, desorganizar o código e dificultar sua compreensão e manutenção. Para prevenir esses problemas, desenvolvedores devem monitorar o uso de *feature annotations*, por exemplo, estabelecendo *thresholds* (valores de referência). No entando, pouco se sabe sobre como extrair *thresholds* na prática, ou quais valores são representativos para métricas relacionadas a *features*. Para contribuir com uma solução para esse problema, nós analizamos a distribuição estatística de métricas relacionadas a *feature annotations*, extraídas de um *corpus* de 20 sistemas baseados no preprocessador C, amplamente conhecidos, com longo histórico de evolução e que abrangem diferentes domínios de função. O estudo considera 3 métricas: *scattering degree* (espalhamento de *feature constants*), *tangling degree* (entrelaçamento de *feature expressions*) e *nesting depth* (profundidade das anotações). Os resultados mostram que *feature scattering* possui uma distribuição com elevada assimetria. Em 14 sistemas (70 %), a distribuição de *scattering degree* segue uma distribuição *power-law*, tornando medidas de média e desvio padrão não confiáveis para estabelecer limites. Em relação a *tangling* e *nesting*, os valores tendem a seguir uma distribuição uniforme. Embora existam *outliers*, eles pouco impactam a média, sugerindo que medidas de tendência central podem gerar *thresholds* confiáveis. Com base nestes resultados, nós propomos *thresholds* gerados a partir de nosso *benchmark* como base para trabalhos futuros. Adicionalmente, nós realizamos uma revisão sistemática da literatura para identificar descobertas e suposições reportadas na literatura sobre o uso de *ifdefs*. Os resultados mostram que os estudos disponíveis não realizam análise estatística de métricas relacionadas a *features*, e nem propõem *thresholds*.

Palavras-chave: Linhas de Produto de Software, features, variabilidade, cpp, ifdef.

Abstract

Feature annotations (e.g., code fragments guarded by *ifdef* C-preprocessor directives) are widely used to control code extensions related to features. Feature annotations have long been said to be undesirable. When maintaining features guarded by annotations, there is a high risk of ripple effects. Also, excessive use of feature annotations may lead to code clutter, hinder program comprehension and harden maintenance. To prevent such problems, developers should monitor the use of feature annotations, for example, by setting acceptable thresholds. Interestingly, little is known about how to extract thresholds in practice, and which values are representative for feature-related metrics. To address this issue, in this master dissertation we analyze the statistical distribution of three feature-related metrics collected from a corpus of 20 well-known and long-lived C-preprocessor-based systems from different domains. We consider three metrics: scattering degree of feature constants, tangling degree of feature expressions, and nesting depth of preprocessor annotations. Our findings show that feature scattering is highly skewed; in 14 systems (70%), the scattering distributions match a power law, making averages and standard deviations unreliable limits. Regarding tangling and nesting, the values tend to follow a uniform distribution; although outliers exist, they have little impact on the mean, suggesting that central statistics measures are reliable thresholds for tangling and nesting. Following our findings, we then propose thresholds from our benchmark data, as a basis for further investigations. We also report in this work the result of a systematic literature review, conducted to identify empirical findings and assumptions on the usage of *ifdefs* as reported in the literature. The inspection of the assumptions and findings shows that studies do not investigate the statistical distributions that better describe feature-related metric values, and also do not propose thresholds for such metrics.

Keywords: Software Product Lines, features, variability, cpp, ifdef.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In this chapter, we state the problem and present this dissertation's motivation (Section 1.1). We then provide an overview of our study (Section 1.2). Finally, we present the outline of the dissertation (Section 1.3) and our publications (Section 1.4).

1.1 Motivation

Feature annotations, such as *ifdefs* (`#ifdef`, `#ifndef`, `#elif`, and `#if` C-preprocessor directives), are long said to be undesirable in source code [Favre, 1996; Krone, 1994; Spencer and Collyer, 1992]. Since annotations are often spread across the entire code base, they clutter source code, hinder program comprehension, and, consequently, complicate maintenance. These annotations are used to relate code fragments to corresponding features. When maintaining the features of the system, each related extension is a potential code fragment that has to be maintained, increasing the likelihood of ripple effects.

Despite these drawbacks, feature annotations are widely used in practice [Apel et al., 2008a, 2013a; Favre, 1996; Kiczales et al., 1997; Krone, 1994; Liebig et al., 2010, 2011; Spencer and Collyer, 1992], mainly, due to limitations of existing programming languages (e.g., see the tyranny of the dominant decomposition [Sullivan et al., 2005; Kästner et al., 2011]). In any case, annotations provide a simple way to include new features into the code base, avoiding the upfront investment on creating modules and interfaces [Kästner et al., 2008]. Still, to prevent an excessive use of feature annotations, developers should monitor their use, for example, by setting *thresholds* (a typical limit).

To reveal how feature annotations are used in source code, metrics quantifying properties, such as scattering, tangling, or nesting, have been proposed in the literature [Liebig et al., 2010]. However, different from other standard code metrics (e.g., size, complexity, coupling) [Alves et al., 2010; Oliveira et al., 2014b], these feature-annotation metrics have never been studied using rigorous statistical methods. At best, researchers report averages and standard deviations over large sets of system, as done by Liebig et al. [2010]. Central tendency and dispersion measures (e.g., mean and standard deviation), however, might not result in representative values.

Recent work [Baxter et al., 2006; Louridas et al., 2008; Wheeldon and Counsell, 2003; Ferreira et al., 2012; Filó et al., 2014], suggests that some code metrics follow *heavy-tailed* distributions, often matching a power-law distribution. In such distributions, the probability that an entity measure deviates from a typical value (e.g., arithmetic mean) is not negligible. That is, a significant fraction of code entities do not follow typical metric values, making centrality and dispersion statistics unreliable.

The central goal of this master dissertation is to understand how feature code is implemented in practice. To achieve this goal, we initially search for studies in the literature regarding the usage of *ifdefs*. Next, we investigate twenty well-known C-preprocessor-based open-source software systems regarding the distribution of three metrics. We analyze the distribution of feature scattering degree (SD), tangling degree (TD) and the nesting depth (ND) of *ifdef* annotations in these systems. These metrics are based on metrics proposed by Liebig et al. [2010]: SD counts the number of *ifdefs* that refer to a given feature; TD counts the number of features that occur in a given feature expression; ND is the depth of the tree of nested *ifdefs*.

We found that feature scattering has highly skewed distributions, and that reporting metrics in terms of averages and standard deviations is unreliable, although commonly done so. Hence, we raise awareness that feature scattering thresholds based on central measures are not reliable in practice. However, regarding tangling degree and nesting depth, the extracted metric values tend to follow a uniform distribution in all systems, with most values equal to one. Although outliers exist, these distributions are not as skewed as the ones seen in the scattering degree metric. This result suggests that mean values for tangling degree and nesting depth are in fact robust. Based on our analysis, we propose thresholds for the metrics we studied, which are derived such that they respect the statistical distributions we have observed.

1.2 An Overview of the Study

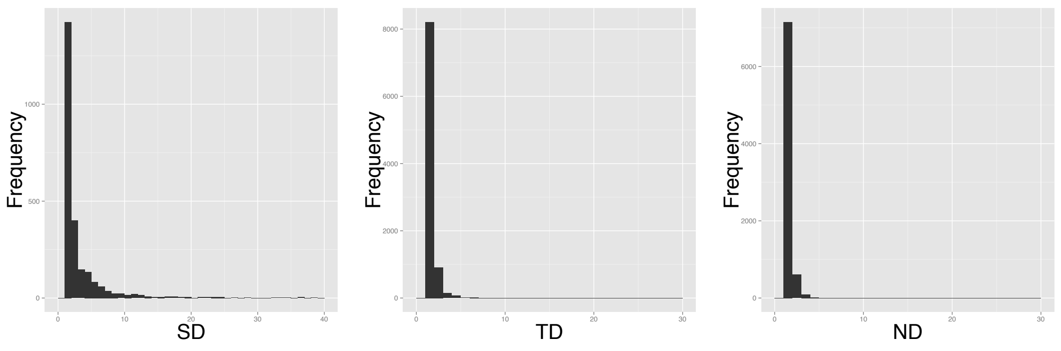
In the first part of this study, we conducted a systematic literature review based on the Kitchenham and Charters [2007] guidelines to search for studies containing assumptions and findings regarding the usage of *ifdefs*. The first goal was to strength our knowledge of the literature about *ifdef* usage. We performed an automatic search from 12/10/2014 to 12/11/2014 in the most relevant digital libraries and inspected the results to select studies that have at least one finding or assumption about the use of *ifdefs*. Next, we extracted findings and assumptions directly from the papers, classified them, and merged the similar ones (whenever possible), keeping the mapping with their studies. We then prioritized them in a ranking system to identify the most common assumptions and findings regarding the usage of *ifdefs*, as reported in the literature.

The inspection of the assumptions and findings shows that there are few studies that rely on metrics to reason about preprocessor-based systems. Specifically, they do not investigate the statistical distributions that better describe preprocessor-based metric values and also do not propose thresholds for such metrics.

In the second part of the study, we performed an analysis of twenty C-preprocessor-based software systems (e.g.,Git, php, MySQL, Linux kernel). To select the subjects, we aimed at covering multiple application domains, and therefore, avoiding bias toward an specific domain. Each system has substantial history of development and use. We used a custom-made tool (fscat) to parse the source code and to compute three metrics: scattering degree (SD) of feature constants, tangling degree (TD) of feature expressions, and nesting depth (ND) of preprocessor annotations. The proposed metrics are based on the metrics of Liebig et al. [2010] and can be used to evaluate complex usages of *ifdefs* in source-code.

After collecting the metrics, we performed the statistical analysis, inspecting the distributions of SD, TD, and ND for each of our 20 subject systems. We inspected the histograms, standard descriptive statistics, and also the Gini coefficient [Gini, 1921] to measure the degree of concentration of the metric values inside each distribution. In this initial analysis step, we check whether the collected distributions have characteristics of a power-law distribution. Then, we proceed with a rigorous test of the power-law hypothesis following Clauset et al. [2009].

Our analysis revealed that feature scattering, as measured by the SD metric, follows a heavy-tailed distribution in all subject systems. In 14 systems (70%), these heavy-tailed distributions matched a power law. Regarding tangling and nesting degrees, the metric values in all systems tend to a uniform distribution, with most values equal to one for both metrics and a few occurrences of slighter higher values. As an example of the results, Figure 1.1 shows the histograms for SD, TD and ND distributions in php, one of our subject systems. The histograms suggest a highly skewed distribution and a high level of inequality for SD. Figure 1.2 shows the Empirical Cumulative Density Function (CDF) of SD in php and the fitted power-law function in the red line (a power-law function appears as a decreasing line when plotted on a logarithmic scale in both axes). The plot reveals that the points approximate the line, strengthening our understanding that feature scattering indeed follows a power-law distribution.



(a) SD histogram (b) TD histogram (c) ND histogram

Figure 1.1: Histogram of Scattering Degree (SD), Tangling Degree (TD), and Nesting Depth (ND) in php

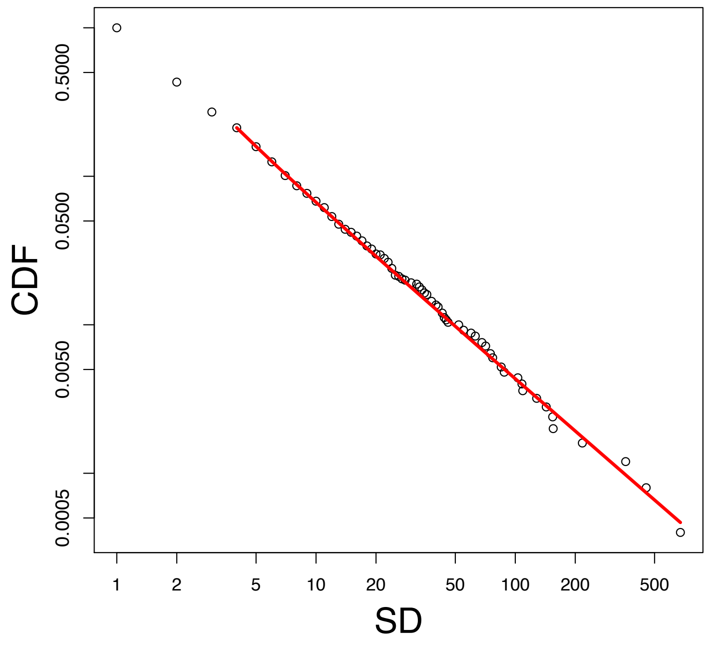


Figure 1.2: Empirical CDF of the Scattering Degree (SD) in php and the fitted power-law function in red, both in logarithmic scale.

Based on our analysis, we proposed thresholds for the metrics we studied, which are derived such that they respect the statistical distributions we have observed. We followed a technique proposed by Oliveira et al. [2014b] to extract thresholds from a set of subject systems for metrics that follow heavy-tailed distributions. Taking data skew into account we used Oliveira et al. [2014a] functions to extract thresholds for Scattering Degree (SD). Tangling degree (TD) and Nesting Depth (ND) approximate an uniform distribution, allowing to directly define thresholds from the mode and its relative frequency (%).

Finally, to illustrate how the proposed thresholds can be used to check whether a system implementation includes a complex usage of *ifdefs* we applied them on XTERM 3.1.8. Existing research [Liebig et al., 2010] shows that XTERM makes a heavy and complex usage of *ifdefs*. In our evaluation, showed in Section 4.3.5, the derived thresholds indeed indicate that XTERM has a complex usage of *ifdefs*.

1.3 Outline of the Dissertation

We organized the remainder of this work as follows:

- Chapter 2 covers central concepts related to this master dissertation, including a discussion on the following concepts and tools: Software Product Lines, C Preprocessor, Software Metrics, and Power-Law Distributions.

- Chapter 3 presents a Systematic Literature Review (SLR) conducted to identify empirical findings and assumptions on the usage of *ifdefs* to implement variability. We detail the protocol that guided our search strategy, the study selection, the data extraction process, and how we organized and classified the findings and assumptions extracted from the literature on *ifdefs*. Finally, we discuss our findings.

- Chapter 4 presents a study analyzing twenty well-know C-preprocessor-based open-source software systems to reveal how feature annotations are used in source code, and how to extract thresholds to monitor and prevent excessive usage. In this study we analyzed the statistical distribution of scattering degree, tangling degree, and nesting depth. Finally, we propose relative thresholds for the metrics we studied.

- Chapter 5 presents the final considerations of this dissertation, including related work, a summary of our contributions, and suggestions of future work.

1.4 Publications

This master dissertation generated the following publications and therefore contains material from them:

- Rodrigo Queiroz; Leonardo Passos; Marco Tulio Valente; Claus Hunsen; Sven Apel; Krzysztof Czarnecki. The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. Journal on Software and Systems Modeling, pages 1–29, 2015. Qualis B1. JCR 2015 = 1 408.

- Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Sven Apel, and Krzysztof Czarnecki. Does Feature Scattering Follow Power-Law Distributions? An Investigation of Five Pre-Processor-Based Systems. In 6th International Workshop on Feature-Oriented Software Development (FOSD), pages 1–7, 2014.

Chapter 2

Background

In this chapter, we provide the background for understanding this master dissertation. We discuss the following concepts and tool: Software Product Lines (Section 2.1), C Preprocessor (Section 2.2), Software Metrics (Section 2.3), and Power-Law Distributions (Section 2.4).

2.1 Software Product Lines

A Software Product Line (SPL) is a family of related program variations that are generated from a common code base [Czarnecki and Eisenecker, 2000; Liebig et al., 2010; Apel et al., 2013b]. The aim of SPL engineering is to facilitate the reuse of common software artifacts in different variants. Software Product Lines introduce the possibility to incorporate individual requirements on the software production such as functionality, target platforms, performance, and energy consumption, but with the benefits of mass production. For example, the Linux kernel runs on a wide variety of different platforms (embed devices, desktops, large scale servers, etc.) and supports different application domains (office software, high performance computing, server software, and many others) [Apel et al., 2013b].

Apel et al. [2013b] highlight the most important promised benefits of Software Product Lines:

- Tailor-made: SPL-based software development approaches facilitate tailoring products to individual customers instead of providing a standardized product or a small set of preconfigured products.

- Reduced costs: instead of paying the costs of designing and developing each product from scratch, product-line vendors develop reusable parts that can be combined in different ways. The required upfront investment is larger than developing a single software product, but the approach pays off specially when multiple tailored products are requested.

- Improved quality: SPLs are constructed from standardized parts. Compared with software developed from scratch, these standardized parts are systematically checked and tested in many products. Parts that are used in multiple products can lead to more stable and reliable products.
- Time to market: software vendors can quickly produce a software product by assembling existing parts. Building a product on top of existing well-designed reusable parts is much faster than developing it from scratch.

However, the benefits of a product-line approach come at a price: it raises the complexity of development, requiring a variability management and a significant upfront investment [Apel et al., 2013b].

2.1.1 Features

The concept of *feature* is very important to SPL engineering. In their seminal work, Kang et al. [1990] introduce FODA (Feature-oriented domain analysis), a SPL approach based on feature diagrams, and the notion of feature as *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"*. However, this concept is inherently hard to define, and consequently there are many definitions. On the one hand, some definitions capture the intentions of stakeholders (end users, managers, programmers, etc.) of a product-line. On the other hand, some definitions capture implementation-level concepts used to structure and reuse software artifacts. Some of the more common definitions are listed below:

> *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"* [Kang et al., 1990]
>
> *"a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder"* [Czarnecki and Eisenecker, 2000]
>
> *"an optional or incremental unit of functionality"* [Zave, 2003]
>
> *"a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement a design decision, and to offer a configuration option"* [Apel et al., 2008b]

The combination of features generates distinguished programs, called *variants* [Liebig et al., 2010; Kästner et al., 2009]. However, not every feature combination is meaningful. For example, some features are mutual exclusive. A *feature model* [Kästner et al., 2009; Apel et al., 2013b] documents the features of a product line and their relationships, defining which combinations are valid. In practice, a feature model contains hundreds or thousands of features, and the number of potential variants can grow exponentially. The idea of *feature orientation* is to organize and structure all software artifacts in terms of features. It makes a feature explicit in the entire life cycle: requirements, design, coding, and testing [Apel et al., 2013b].

2.1.2 Implementation Approaches

Software Product Lines can be implemented in two different ways: the compositional approach and the annotative approach [Kästner et al., 2008]. With the compositional approach, features are implemented as distinct modules, and a set of modules is composed to generate a product. A classic example is a framework that can be extended with plug-ins (ideally one plug-in per feature). There are several examples of compositional techniques using specialized architectures and languages like *Mixin Layers* [Smaragdakis and Batory, 2002], *AHEAD* [Batory et al., 2003], and *Aspects* [Kiczales et al., 1997].

With the annotative approach, features are implemented with explicit or implicit annotations of the source code. A typical example for annotative approach is the use of `#ifdef` directives of the *C preprocessor* (Section 2.2). Other examples include *Gears*, *XVCL*, and *CIDE* [Kästner et al., 2008]. Depending on how they are used, some approaches like Aspects can be included in both groups. However, approaches based on tool support like *Generative Programming* [Czarnecki and Eisenecker, 2000] do not fit into either group.

Compositional approaches typically support coarse-grained extensions (e.g., adding new methods or classes, extending explicit extension points). In contrast, fine-grained extensions (e.g., adding new statements on existing methods, extending expressions, extending a method signature) are better supported by annotative approaches because they can mark arbitrary code fragments. However, these annotations provide no perceptible form of modularity. Instead, they obfuscate and raise complexity of the source code [Kästner et al., 2008].

2.2 C Preprocessor

The C preprocessor (CPP) enriches the C language with simple meta-programming facilities, supporting the implementation of software families [Liebig et al., 2010; Apel et al., 2013a; Passos et al., 2013]. In particular, CPP introduces three capabilities: file inclusion (#include directive), macro definition (#define directive) and expansion, and conditional compilation (#ifdef directive). Here, we concentrate on conditional compilation to support variability in source code and on problems related to this capability.

The conditional-compilation mechanisms of the C preprocessor provide an easy approach to implement variability in software product-lines. The concept is very simple: features are denoted by macro names, which in turn are referenced by different compilation-guard conditions annotated in code fragments. Depending on the feature selection, the preprocessor removes annotated code fragments before the compilation. There are different types of guard conditions: #ifdef, #ifndef, #elif, and #if. For brevity, we refer to all these constructs as *ifdefs*.

In Figure 2.1, we exemplify a preprocessor-based implementation taken from the PYTHON INTERPRETER source code, with fragments of code framed with #ifdef and #endif directives. In file mmapmodule.c, developers introduce some extensions conditionally, depending on the choice of the target operating system. This conditional code is controlled by the presence or absence of certain features. Lines 438–449, for instance, depend on the presence of feature MS_WINDOWS, while lines 453–464 depend on the presence of feature UNIX. Nested in the code of UNIX feature, there is further variability that is related to the support of large files. Feature HAVE_LARGEFILE_SUPPORT implements this nested variable behavior at line 460. In Figure 2.2, there is another use of feature MS_WINDOWS, also taken from the PYTHON INTERPRETER source code. In this case, the presence of either MS_WINDOWS or ___CYGWIN___ enables the compilation of the guarded code at line 428.

The C preprocessor is widely used in projects written in C and C++, and some other language like FORTRAN. Many well-known systems like APACHE, MYSQL, PYTHON, and LINUX heavily rely on the C preprocessor to implement variability in source code. In any case, annotations provide a simple way to include new features into the code base, avoiding the upfront investment on creating modules and interfaces [Kästner et al., 2008].

```c
mmap_size_method(mmap_object *self, PyObject *unused)
{
    CHECK_VALID(NULL);
#ifdef MS_WINDOWS
    if ( self->file_handle != INVALID_HANDLE_VALUE) {
        DWORD low,high;
        PY_LONG_LONG size;
        low = GetFileSize(self->file_handle, &high);
         (...)
        if (!high && low < LONG_MAX)
            return PyLong_FromLong((long)low);
        size = (((PY_LONG_LONG)high)<<32) + low;
        return PyLong_FromLongLong(size);
    } else {
        return PyLong_FromSsize_t(self->size);
    }
#endif /* MS_WINDOWS */

#ifdef UNIX
    {
        struct stat buf;
        if (-1 == fstat(self->fd, &buf)) {
            PyErr_SetFromErrno(PyExc_OSError);
            return NULL;
        }
#ifdef HAVE_LARGEFILE_SUPPORT
        return PyLong_FromLongLong(buf.st_size);
#else
        return PyLong_FromLong(buf.st_size);
#endif
    }
#endif /* UNIX */
}
```

Figure 2.1: Feature implementation example using *ifdefs* (mmapmodule.c file from the PYTHON INTERPRETER)

```c
#if defined(MS_WINDOWS) || defined(__CYGWIN__)
    _setmode(self->fd, O_BINARY);
#endif
 (...)
```

Figure 2.2: Tangled feature implementation example using *ifdefs* (fileio.c from the PYTHON INTERPRETER)

However, preprocessors such as CPP are heavily criticized in the literature. Most of the criticism around CPP claim that it is error-prone, lacks modularity, obfuscates the code and hardens maintenance [Spencer and Collyer, 1992; Kästner et al., 2008, 2009]. In fact, the flexibility of the C preprocessor allows programmers to make all kinds of annotations, including individual tokens such as a closing bracket, leading to hard-to-find syntax errors. Annotations that do not align with the syntactic code structure, (e.g., with entire statements, functions, and type declarations) are called *undisciplined annotations* [Liebig et al., 2011]. The presence of preprocessor directives also hinders the use of supporting tools and creates a great challenge for refactoring [Medeiros et al., 2014; Overbey et al., 2014; Kästner et al., 2009; Garrido and Johnson, 2013]. As Adams et al. [2008] state, *"cpp is a necessary evil for every C programmer and maintainer"*.

2.3 Software Metrics

According to Fenton and Pfleeger [1998], *"Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules"*. A software can be measured using *software metrics*, a quantitative measure of the degree to which a system, component, or process possess a given attribute [IEEE, 1990].

Software metrics can be divided into three categories [Fenton and Pfleeger, 1998]:

- Process metrics: measure attributes of a development process itself.
- Product metrics: measure documents and software artifacts that were produced as part of the software development process.
- Resource metrics: measure resources used as part of a process.

They can also be divided into metrics that measure internal or external attributes. A metric can measure an internal attribute by observing only the process or product itself, or an external attribute related to the behavior of the software. In this work we focus on internal product metrics that measure source code properties.

Over decades, hundreds of source code metrics have been proposed by researchers and practitioners, in both theoretical and empirical studies. Well-known source code metrics include cyclomatic complexity, number of attributes (NOA), number of methods (NOM), response for a class (RFC), number of references to a class (FAN-IN), number of other classes referenced by a class (FAN-OUT), and weighted method count (WMC) [Chidamber and Kemerer, 1994; Brito e Abreu and Carapuca, 1994; Lanza and Marinescu, 2010].

However, metrics are rarely used to control in an effective way the quality of software products [Fenton and Neil, 2000]. To use software metrics as an effective measurement instrument, developers should define meaningful thresholds. In this way, software engineers can rely on metrics, for example, to monitor the evolution of components or the quality degradation.

In general, a threshold defines an upper bound. Values greater than a threshold value are considered to be problematic, and the values lower are considered to be acceptable. Most metric thresholds proposed in literature are based on the experience of software experts on what constitutes desirable software properties [Caltech, 2010]. Other thresholds are computed using a more transparent method, for example, from benchmark data [Alves et al., 2010; Oliveira et al., 2014b].

2.4 Power-law Distributions

Many empirical quantities cluster around a typical value. For example, the heights of human beings, the speeds of cars on a highway, air pressure. Even with some variations, their distributions place a negligible amount of probability far from the typical value. However, not all distributions fit this pattern. Among such distributions, power-law distributions widely appear in a diverse range of natural and man-made phenomena. Examples include the intensity of earthquakes, solar flares, moon craters, and people's personal fortunes [Clauset et al., 2009; Newman, 2005].

Studies around the power law as a descriptive device has been around for more than a century and have appeared in various contexts [Louridas et al., 2008]:

- 1897: The Italian economist Vilfredo Pareto described a power law distribution in the 19th century.
- 1925: G. Udny Yule observed the power law model in his study on the creation of biological species.
- 1935: George Kingsley Zipf observed the frequency of the most common words in natural language.
- 1951: Benoit Mandelbrot came upon power laws in a theory of word frequencies.

A distribution is said to be a power-law when the probability of measuring a particular value varies inversely as a power of that value [Newman, 2005]. The population of towns and cities is a classic example of this type of distribution. Figure 2.3 shows a histogram with the distribution of the US city populations, extracted from the 2000 census,[1] as analyzed by Clauset et al. [2009] and Newman [2005]. The histogram is

[1]Data available at `http://tuvalu.santafe.edu/~aaronc/powerlaws/data.htm`

highly right-skewed: while the bulk of the distribution refers to small-sized cities, a small number of very large cities produces the heavy-tail to the right of the histogram. Another important characteristic of a power-law distribution concerns its visualization: when plotted on a logarithmic scale in both axes, a power-law function appears as a decreasing line, as shown in Figure 2.4.

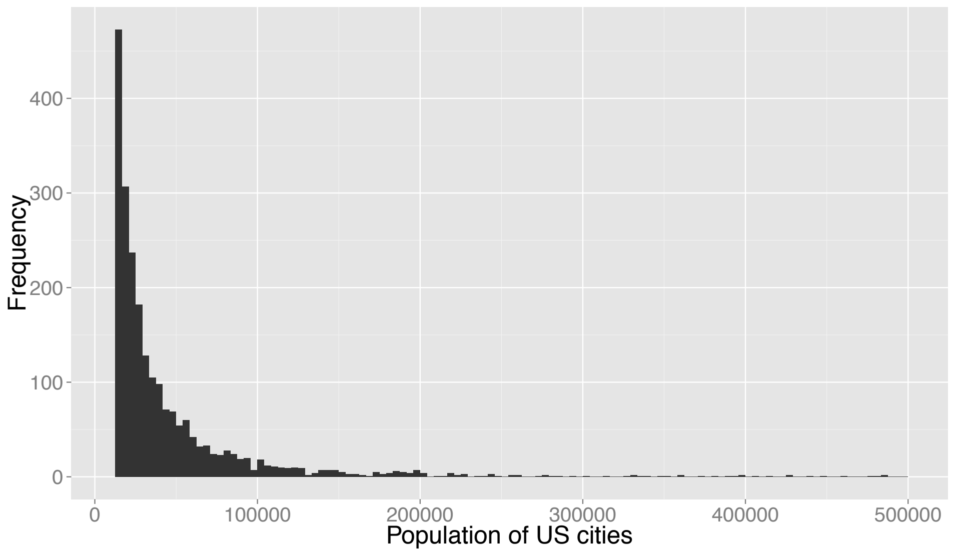


Figure 2.3: Histogram of the population of US cities with population of 10 000 or more (data from the 2000 US Census)

In formal terms, a discrete power-law distribution (which we refer henceforth as power-law) is a distribution in which the probability that a discrete random variable X assumes a value x is proportional to x raised to the negative power of a positive constant k:

$$P(X = x) \propto cx^{-k} \qquad where \quad c > 0, k > 0 \tag{2.1}$$

A power law, as given by this equation, diverges when $x = 0$. In fact, it requires a lower-bound value $x_{min} > 0$ to define a cut-off value as starting point ($x > x_{min}$) from which a power-law behavior occurs [Clauset et al., 2009]. As we shall see later in Chapter 4, the parameters k and x_{min} play an important role when performing a goodness-of-fit analysis.

From a software engineering point of view, different researchers examined the distribution of different source-code metrics and network relationships among software components. The conclusions from these studies is that power-laws appear to be quite common [Baxter et al., 2006; Louridas et al., 2008; Oliveira et al., 2014b; Wheeldon and Counsell, 2003; Ferreira et al., 2012]. However, to the best of our knowledge, it has not

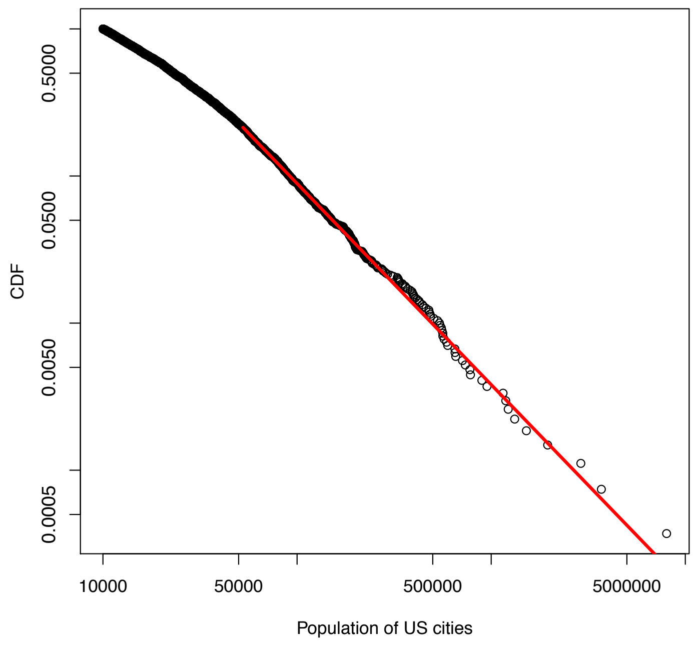


Figure 2.4: Empirical Cumulative Density Function of the population of US cities (points) and the fitted power-law function in red, both in logarithmic scale (data from the 2000 US Census)

been investigated whether the same holds for feature-related metrics. Understanding the distribution of these metrics affects our understanding of how Software Product Lines implemented with CPP are built in practice, and how we can extract thresholds from real software systems. For example, if they do follow a power-law distribution, the central limit theorem does not apply, and the sample mean and variance can not be used as estimators of the population mean and variance [Baxter et al., 2006].

2.5 Final Remarks

This chapter presented essential concepts for understanding this master dissertation. Initially, we discussed the main concepts of Software Product Lines, Features and Implementation Approaches. Next, we presented the C Preprocessor as a simple and flexible method to implement variability in SPL with some examples from a real software system (Python Interpreter). Finally, we presented the concept of Power-Law distributions, and how they are common in a diverse range of phenomena, including software-related metrics and relationships between software artifacts.

Chapter 3

Findings and Assumptions on the Usage of *ifdefs*

In this chapter, we present a systematic literature review conducted to identify empirical findings and assumptions on the usage of *ifdefs*. We start by detailing the protocol that guided our search strategy, the study selection, the data extraction process, and how we organized and classified the findings and assumptions extracted from the literature on *ifdefs* (Section 3.1). Finally, Section 3.2 discusses our findings.

3.1 Study Design

To identify empirical findings and assumptions on the usage of *ifdefs* to implement variability in practice, we follow a Systematic Literature Review (SLR) protocol based on the Kitchenham and Charters [2007] guidelines. This SLR aims to address the following research questions:

RQ #1: What are the most common findings related to the use of *ifdefs*?
RQ #2: What are the most common assumptions related to the use of *ifdefs*?

3.1.1 Search Strategy

We performed an automatic search in the most relevant digital libraries for potentially relevant studies without restriction on the year of publication to increase the coverage of the review [Brereton et al., 2007; Kitchenham and Charters, 2007]. The selected libraries are as follows:

- ACM Digital Library (`http://dl.acm.org`)
- Ei Compendex and Inspec (`www.engineeringvillage.com`)
- IEEEXplore (`http://ieeexplore.ieee.org`)
- ScienceDirect (`http://www.sciencedirect.com`)
- Scopus (`http://www.scopus.com`)
- SpringerLink (`http://link.springer.com`)

The search was based on the keywords *"c preprocessor"* and *"ifdef"*, including the variations found in a pilot search. The general search string is given as follows:

> "c preprocessor" OR "c pre processor" OR "c++ preprocessor" OR "c++ pre processor" OR "preprocessor for c" OR "pre processor for c" OR ("cpp AND preprocessor") OR ("cpp AND pre processor") OR "ifdef"".

The search was performed within titles or abstracts, following conventional practices [Brereton et al., 2007; Kitchenham and Charters, 2007]. When the library does not provide an option to limit the search within titles or abstracts, the search was performed within the full-text. Moreover, the search string was adapted to suit specific requirements or limitations of the different libraries. For example, the filters that can be included in the search string may differ. The corresponding strings used for each library are given in Appendix A.

We conducted the search from 12/10/2014 to 12/11/2014 and collected the results in a spreadsheet recording title, authors, journal/conference name, year of publication, abstract, and the library in which the article was found. This search resulted in 396 articles, of which 245 were distinct. We converted the spreadsheet into a PostgreSQL database and performed a query according to the search string, applied to specific fields (title and abstract). This search aims to make the selection process uniform, independent from which library a paper came from (e.g., Spring does not support searching for the contents of the abstract; rather, it also returns occurrences of the search string in the whole text—see Appendix A). We also used the query to find and eliminate redundant papers inside the same library results (e.g., the same paper published in a conference proceeding and in a journal). This process resulted in 116 potentially relevant papers, as presented in Table 3.1. This table shows the number of articles resulting from each step of the process, from each Library, including overlapping and distinct results.

Digital Library	**Library search**	**SQL Query**
ACM	30	30
Ei Compendex and Inspec	127	103
IEEEXplore	27	26
ScienceDirect	5	4
Scopus	78	73
SpringerLink	129	2
Total	396	238
Distinct	245	116

Table 3.1: Number of articles resulting from the search process. The first column lists the search results from the digital libraries; the second column lists the number of papers resulting from the SQL filter.

3.1.2 Study Selection

To select studies from the potentially relevant papers, we applied the following inclusion criteria: (i) studies published in peer-reviewed journals or conference proceedings; (ii) studies in English; (iii) studies that have at least one finding or assumption about the use of *ifdefs*. All the inclusion criteria had to be satisfied to ensure that the selected study was within our targeted area of research. Some digital libraries provide filtering mechanisms that can be used to restrict the search to the inclusion criteria (i) and (ii) (details of each filter are listed in Appendix A).

To evaluate the inclusion criterion (iii), we manually inspected the 116 potentially relevant papers, as follows. Two researchers read the following sections of each paper: Introduction, Conclusion, and any section that reports findings and/or discusses them. They kept track of papers that claim at least one finding concerning *ifdef* usage, or that make at least one assumption about this usage. We considered as *finding* any statement about the usage of *ifdefs* that is based on the results of the study reported in the paper (e.g., Liebig et al. [2010] reports that *"programmers use fine-grained extensions infrequently"* based on their analysis of 40 preprocessor-based systems). Statements on the usage of *ifdefs* that are explicitly declared using references to other papers (e.g., Kenner et al. [2010] reports that *"a majority of industrial software product lines are implemented with the C preprocessor"* based on the studies from Pearse and Oman [1997]), or as common knowledge (e.g., Feigenspan et al. [2013] reports that *"Preprocessor directives are easy to use"*) are considered as *assumptions*. Papers that do not report any finding, nor make explicit assumptions were excluded.

Both researchers agreed to include 41 papers, and they disagreed regarding 16 papers. The final decision about the 16 cases of disagreements was made during a consensus meeting. A consensus was reached and the two researchers decided to include

14 articles, resulting in 55 selected studies (see Appendix B). The distribution of the resulting articles among the digital libraries with overlapping and distinct results is given in Table 3.2.

Digital Library	**Selected Studies**
ACM	24
Ei Compendex and Inspec	47
IEEEXplore	19
ScienceDirect	0
Scopus	47
SpringerLink	1
Total	138
Distinct	55

Table 3.2: Number of articles resulting from the selection process after inspection and consensus by two researchers.

3.1.3 Data Extraction

As the first step of the extraction, the author of this master dissertation read the selected papers entirely, keeping record of all the findings and assumptions regarding the use of *ifdefs*, as originally stated in the paper. In a second step, he rewrote the findings to eliminate redundant results in the same paper and to break different findings or assumptions from a single statement. An example is given as follows:

- Saebjoernsen et al. originally state that *"C preprocessor (CPP) is generally considered a source of difficulty for understanding and maintaining C/C++ programs"*. [Saebjoernsen et al., 2009]

This assumption regards two different problems: maintainability and understandability. We then split it into two different assumptions:

- *"CPP is a source of difficulty for understanding C/C++ programs"*
- *"CPP is a source of difficulty for maintaining C/C++ programs"*

This was necessary to allow direct comparison between statements from different studies. We collected the results in a spreadsheet, keeping the mapping between papers and their findings/assumptions.

3.1.4 Data Classification and Ranking

With all the results in a spreadsheet, we identified 17 groups of findings/assumptions. Each finding or assumption could be classified in one or more group (when applicable). These groups with examples of findings or assumptions are listed next:

- Code comprehension (e.g., *"Extensive use of the CPP across the code causes adverse consequences for code comprehension"*, S44)
- Maintainability (e.g., *"CPP leads to source code that is hard to maintain"*, S37)
- Error-proneness (e.g., *"CPP eases the introduction of subtle syntax errors"*, S48)
- Testing (e.g., *"As the number of variant features grows, programs become difficult to test"*, S24)
- Feature scattering (e.g., *"Feature scattering often occurs in practice"*, S53)
- Tangling (e.g., *"CPP leads to extremely tangled preprocessor code"*, S18)
- Nesting (e.g., *"Code with nested ifdef directives are hard to identify"*, S45)
- Discipline (e.g., *"Most of the #ifdefs are disciplined"*, S35)
- Granularity (e.g., *"Most extensions occur at a high level of granularity, such as if-statements or for-loops)"*, S29)
- Performance (e.g., *"CPP is often used to tune performance using in-line code"*, S22)
- Code replication (e.g., *"A minor fraction of all code clones occur within #ifdef"*, S36)
- Portability/reuse (e.g., *"Portability accounts for almost half of conditional compilation directives"*, S11)
- Tooling (e.g., *"There are no refactoring tools that can completely and safely transform C code because of the CPP"*, S47)
- Variability model (e.g., *"Inconsistencies between feature models and feature implementations in CPP are common"*, S30)
- Coding guidelines (e.g., *"Many programmers follow conventions on the use of cpp"*, S25)
- CPP alternatives (e.g., *"Applicability of alternative mechanisms such as aspects is hard to envision"*, S44)
- Miscellaneous (e.g., *"For reverse engineers, the connection between the preprocessor input and output is hard to precisely understand"*, S12)

From each group, we tried to merge, whenever possible, similar assumptions or similar findings, updating the mapping accordingly. For example, consider the following three assumptions from the group *Code comprehension*:

- *"CPP files can be very difficult to understand"* (S07)
- *"CPP usage impairs readability of the base system"* (S15)
- *"CPP mechanisms are known to challenge code comprehension"* (S38)

These three assumptions were merged in a single assumption:

- *"CPP has a negative effect on code readability and comprehension"* (S07, S15, S38)

From the mapping between findings/assumptions and their papers, we created a bipartite graph[1], where papers denote nodes on the left of the graph, and assumptions or findings denote nodes on the right of the graph. The lists of assumptions and findings were prioritized according to the number of its incoming edges as a ranking criteria. As an example, Figure 3.1 shows a graph including only four findings from four studies: S25 [Padioleau, 2009], S29 [Liebig et al., 2010], S35 [Liebig et al., 2011], and S44 [Jbara and Feitelson, 2013]—see Appendix B. The number of incoming edges is used as a ranking criteria, prioritizing the findings mentioned by more studies (e.g., finding F1 and finding F4 are both mentioned by two studies). The resulting ranking of this graph is given in Table 3.3. A similar procedure was performed for the list of assumptions. The final ranking including all findings and assumptions is discussed in Section 3.2.

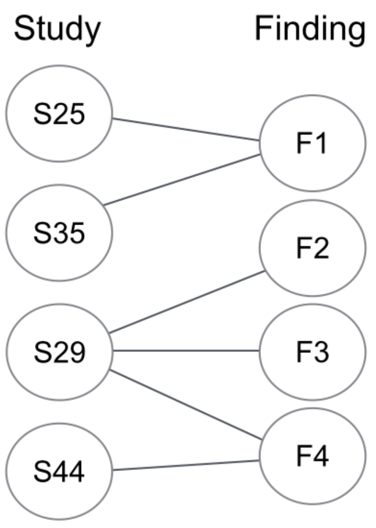


Figure 3.1: Bipartite graph connecting selected studies and their findings (e.g., finding F1 is reported in studies S25 and S35; study S29 reports findings F2, F3 and F4)

[1]A bipartite graph is a graph whose vertices can be divided into two disjoint sets U and V, such that every edge connects a vertex in U to one in V.

Table 3.3: Ranking of findings extracted from selected studies. The number of studies reporting the finding is used as a ranking criteria

Pos	ID	Finding	Degree	Studies
1	F1	Most of the #ifdefs are disciplined	2	S25, S35
1	F4	Nested #ifdefs are used moderately	2	S29, S44
2	F2	Few #ifdef extensions would benefit from alternatives like aspects	1	S29
2	F3	Variability management with cpp does not cause excessive code degradation	1	S29

3.2 Results

3.2.1 Overview

The search and selection processes resulted in 55 studies, which are listed in Appendix B. Figure 3.2 shows the distribution of selected studies over the years. The first study we considered appeared in 1990 and the last one appeared in 2014. As the figure shows, empirical studies regarding the use of the C-preprocessor started to attract more research attention after 2009 (65% of the selected studies were published after 2009), which indicates that this area of research is currently highly active.

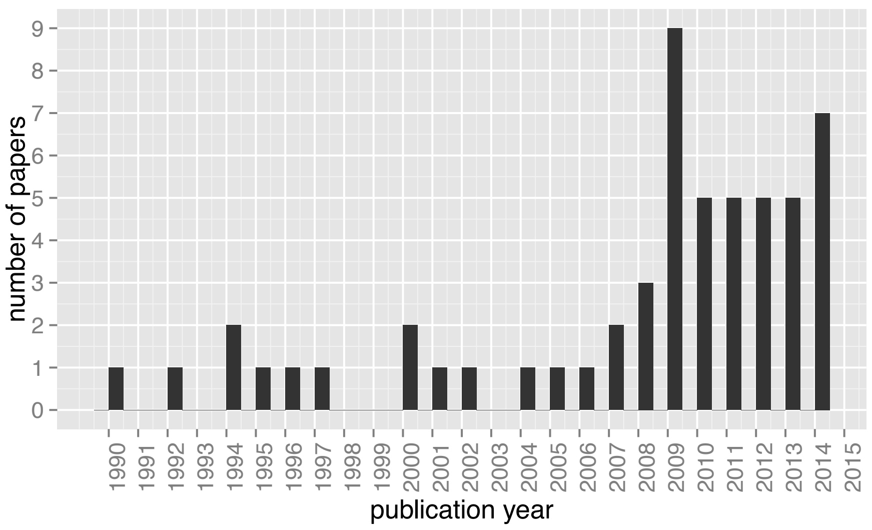
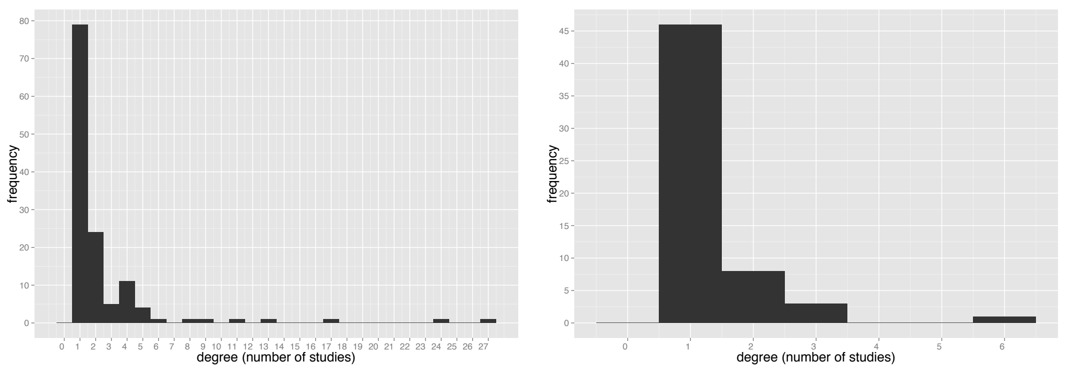


Figure 3.2: Selected studies over the years

The data extraction from these studies resulted in a list of 89 findings and 345 assumptions, from which we performed the classification and ranking process detailed in Section 3.1. Appendix C shows both rankings, with 131 distinct assumptions and 58 distinct findings that resulted from the unification process. Figure 3.3 shows the distribution of assumptions and findings degrees. While 79 assumptions (60%) have degree=1 (i.e, they are cited by only one study), 52 assumptions (40%) are cited by, at least, two different studies. Most of the findings (46, 79%) are cited by only one study (degree=1). This was expected, since many results are very specific. However, we did find similar results regarding 12 findings (21%).

Interestingly, the most common assumptions identified in our study (assumptions with degree>10) are confirmed or strengthened by some of the most common findings. The best ranked assumptions, findings, and their correlations are discussed in Sections 3.2.2 and 3.2.3.



(a) assumptions (b) findings

Figure 3.3: Distribution of assumptions degree (a) and findings degree (b)

3.2.2 Best Ranked Assumptions

The most common assumptions of our study are as follows:

Assumption A001: *"CPP is heavily used to implement variability"* (Studies S06, S07, S09, S15, S17, S18, S21, S23, S24, S25, S28, S29, S30, S32, S33, S34, S38, S39, S41, S42, S45, S46, S47, S48, S49, S52, S55)

This assumption has degree 27 and is shared by almost half of our studies (49%), which are distributed over a wide range of years, from 1996 (S06) to 2014 (S55). All the studies are based on the general assumption that CPP is heavily used and is the first choice to implement variability in the source code of variable systems. As an example, a recent study (S55) states that *"preprocessor directives are still widely used as no real size program with configurations exists without them"*.

Assumption A002: *"CPP has a negative effect on code readability and comprehension"* (Studies S01, S02, S06, S07, S10, S11, S12, S15, S21, S22, S23, S27, S28, S33, S34, S36, S37, S38, S44, S45, S46, S47, S48, S55)

The second ranked assumption (A002) has degree=24 and is also shared by almost half of our studies (44%). These studies are distributed over an even wider range of years, from 1990 (S01) to 2014 (S55). This assumption concerns a recurrent problem on CPP usage: code comprehension. While some studies (S34, S36, S44) consider that CPP reduces readability or comprehensibility, other studies consider that preprocessor directives can be very hard or impossible to understand. For example, Study S06 states that *"the presence of numerous #if directives makes the structure almost impossible to follow. Human readers are not able to take into account all the variants at a time"*.

Assumption A003: *"CPP impairs maintainability of code"* (Studies S02, S06, S07, S08, S11, S12, S17, S21, S23, S27, S28, S29, S34, S36, S37, S49, S54)

The third ranked assumption (A003) has degree=17 and is also distributed over a wide range of years, from 1992 (S02) to 2014 (S54). This assumption is related to assumption A002, and highlights the lack of maintainability as a consequence of many problems caused by CPP directives in source code. As an example, S02 states that *"preprocessor commands can obfuscate the program's mechanics and, consequently, maintainability"*. S06 states that *"maintenance of complex preprocessor files is a nightmare"*.

Assumption A004: *"CPP is error-prone"* (Studies S09, S13, S14, S21, S28, S29, S36, S38, S41, S43, S48, S49, S55)

The fourth ranked assumption (A004) has degree=13 and ranges from 2000 (S09) to 2014 (S55). Many studies state that the preprocessor can introduce subtle syntax errors in the host language, making variability implementation error-prone.

Assumption A005: *"CPP is often used to achieve portability"* (Studies S02, S03, S08, S16, S17, S18, S22, S27, S41, S51, S52)

The fifth ranked assumption (A005) has degree=11 and ranges from 1992 (S02) to 2014 (S52). As the studies suggest, portability has traditionally been obtained with *ifdefs*. S16 suggests that C/C++ programs invariably achieve portability using the C-preprocessor: *"in C and C++ programs portability is invariably managed using the CPP"*.

3.2.3 Best Ranked Findings

The most common findings of our study are given as follows:

Finding F01: *"CPP is heavily used to implement variability"* (Studies S11, S22, S29, S38, S44, S52)

The first ranked finding (F01) is also the most common assumption A001, and is supported by six studies (degree=6). All these studies evaluate the preprocessor usage in real systems. Combined, they analyzed the preprocessor usage in more than 100 systems and packages, including implementations in C and FORTRAN, from different functional domains. The results confirm that despite all the criticism, the C preprocessor and its conditional compilation mechanisms are widely used to implement variability in program families, supporting assumption A001.

Finding F02: *"Alternative mechanisms to cpp such as aspects are hard to envision"* (Studies S18, S29, S44)

The second ranked finding F02 has degree=3. Although the benefits of migrating to more modern implementation techniques for Software Product Lines are well known, theses studies found that this can be hard to envision for all *ifdef* extensions and usage patterns. Study S18 explored the process of mining and extracting aspects from preprocessor-driven systems, Study S29 investigated two alternative SPL implementation techniques: aspects and feature modules, and finally Study S44 performed a detailed analysis on the CPP usage of the Linux kernel. All three studies have similar results, indicating that the interaction, nesting and fine-grainedness of preprocessor directives complicate the identification of separable concerns and their extraction to aspects.

Finding F03: *"CPP introduces errors to code"* (Studies S13, S48, S49)

The second ranked finding F03 also has degree=3 and is similar to the assumption A004. Study S48 performs an empirical study in 41 program family releases, and more than 51 thousand commits of 8 program families. They found 20 preprocessor-based syntax errors in commits, and 7 in releases. Study S49 analyzes releases of 12 C program families and found that incomplete annotations can cause semantic errors and also memory leaks. These findings confirm assumption A004.

Finding F04: *"Most of the #ifdefs are disciplined"* (Studies S25, S35, S52)

The second ranked finding F04 has degree=3. Although preprocessors are frequently criticized for their undisciplined usage, these findings reveal that most of the *ifdefs* are in disciplined form. These findings confirm assumption A012—*"In practice most annotations are already in a disciplined form"* (S28, S20, S32, S46, S49).

The following findings have degree=2, but are related to the high ranked assumptions A002, A003 and A005:

Finding F05: *"CPP annotations complicate program comprehension"* (Studies S44, S46)

Finding F05 confirms assumptions A002 and A003. Ranked in the third position, this finding is shared by two studies. Study S44 performs an analysis of the Linux kernel, including the configurability model and the source code. They found nearly 5000 real config options, suggesting extensive use of the CPP across the code. Study S46 is a controlled experiment with human subjects. They measured program comprehension evaluating the performance of the subjects regarding correctness and response time for solving tasks in annotated code. The experiment confirms that finding errors in the presence of preprocessor annotations is a tedious and time-consuming task.

Finding F10: *"Portability accounts for a great part of #ifdef usage"* (Studies S11, S32)

The third ranked finding F10 confirms assumption A005. Study S11 analyzes 26 packages comprising 1.4 million lines of source code to evaluate C-preprocessor usage. The study reveals that portability accounts for 37% of conditional compilation directives (e.g., *ifdefs* used to enclose specific capabilities of the target machine or operating system). Study S32 evaluate an open-source web server with a type-checking tool (TypeChef). They found *ifdefs* implementing variability that can be both considered a feature in the sense of a product line and low-level portability. These findings confirm assumption A005.

3.3 Final Remarks

In the systematic literature review reported in this chapter, we searched and selected studies containing assumptions and findings regarding the usage of *ifdefs*. The high number of studies found in recent years reveals how *ifdefs* are attracting more research attention. Based on data extracted from these studies, we ranked the results to identify the most common assumptions and findings reported in the literature. The results reveal how the most common assumptions identified in our study are confirmed or strengthened by some of the most common findings.

The inspection of the assumptions and findings shows that there are few studies that rely on metrics to reason about preprocessor-based systems. Specifically, the existing studies do not investigate the statistical distributions that better describe preprocessor-based metric values. They also do not propose thresholds for such metrics, to better assess for example when *ifdef* directives in fact hamper readability and comprehensibility (A002) or impair maintainability (A003). In the following chapter, we therefore report a study conducted to fill this gap detected in the literature.

Chapter 4

The Shape of Feature Code

To reveal how feature annotations are used in source code, and how to extract thresholds to monitor and prevent excessive usage, we performed a study analyzing twenty C-preprocessor-based open-source software systems. In this study we analyzed the statistical distribution of scattering, tangling, and nesting degrees and propose thresholds for the metrics we studied, which are derived such that they respect the statistical distributions we have observed.

In this chapter, we start by presenting the methodology to perform our study, including the subject systems, the process and tools we used to compute the feature-related metrics, and the procedure we followed in the statistical analysis of the collected data (Section 4.1). Next, Section 4.2 presents our results for the collected metrics, including a discussion on the statistical distributions that best describe our data. Section 4.3 discusses implications of our findings, in particular, regarding the extraction of thresholds for feature-related metrics. Section 4.4 reports threats to validity. Finally, Section 4.5 summarizes our findings and states our final remarks.

The study presented in this chapter has been preliminary published at a workshop [Queiroz et al., 2014] and later extended to a journal [Queiroz et al., 2015].

4.1 Methodology

In this section, we discuss the selection of subject systems (Section 4.1.1), the process and tools to compute feature-related metrics (Section 4.1.2), the statistical analysis of the data we collect (Section 4.1.3), and the method we use to propose thresholds (Section 4.1.4).

Table 4.1: Subject systems

System	Version	Year	Since	Domain	SLOC
VI[2]	50325	2005	2000	Text editor	22 275
LIGHTTPD	1.4.35	2014	2003	Web server	39 991
XFIG	3.2.5c	2013	1985	Graphics editor	74 713
SENDMAIL	8.14.9	2014	1983	Network service	92 204
SYLPHEED	3.4.2	2014	2000	E-mail client	116 454
GIT	2.1.0	2014	2005	Version control	152 018
APACHE	2.4.10	2014	1995	Web server	155 846
LIBXML2	2.9.1	2014	1999	Programming library	222 009
EMACS	24.3	2013	1985	Text editor	249 932
OPENLDAP	2.4.39	2014	1998	Network service	291 781
SUBVERSION	1.8.10	2014	2000	Version control	328 878
IMAGEMAGICK	6.8.9-7	2014	1987	Image editor	333 048
PYTHON	3.4.1	2014	1989	Program interpreter	353 485
PHP	5.6.0	2014	1985	Program interpreter	664 259
POSTGRESQL	9.3.5	2014	1995	Database system	676 435
GIMP	2.8.14	2014	1996	Image editor	703 435
GLIBC	2.20	2014	1987	Programming library	826 502
MYSQL	5.6.19	2014	1995	Database system	1 577 874
GCC	4.9.0	2014	1987	Compiler framework	3 209 684
LINUX KERNEL	3.15	2014	1991	Operating system	11 964 075

4.1.1 Selection of Subject Systems

To analyze the statistical distributions describing feature-related metrics, we selected 20 open-source software systems that use C preprocessor directives to annotate feature code; Table 4.1 provides information on all subject systems.

Three criteria guided the selection of our subjects: First, we aimed at covering multiple application domains, therefore avoiding bias toward an specific domain. In Table 4.1, the 20 systems are distributed across 12 different domains. Second, each system has substantial history of development and use, as given by columns *Year* (the year of the release of the system under analysis) and *Since* (the year of the first release). The rationale is that mature systems are more likely to have found a practical balance to when and how much to scatter, tangle, and nest preprocessor annotations than immature systems. Third, the selection includes systems of different sizes, to avoid bias toward a particular system size. We measured size using Source Lines of Code (SLOC), which is the total number of source lines of code of a given system. These numbers excludes blank lines and comments. Moreover, sequences of multilines (lines ending with a backslash) are counted as a single line.[1]

[1]Multilines are convenient when spanning a long line across multiple ones; during compilation, sequences of multilines are taken as a single line.

[2]The VI system we use is a port of the original VI (late 1970's) to modern Unix systems.

As shown in Table 4.1, our set of subjects includes four small systems ($<$ 100 KSLOC), nine medium sized systems (100 to 400 KSLOC), and seven large software systems ($>$ 400 KSLOC).

4.1.2 Data Collection and Metrics

To extract feature-related metric values, we first parse the source files (C implementation and header files) of each subject system. Parsing is performed using the tool SRC2SRCML,[3] which creates an XML representation of the code. The resulting XML files preserve all the code, including preprocessor annotations (SRC2SRCML does not perform any preprocessing). With all annotations in place, we run a custom-made tool (FSCAT[4]) to process the XML files produced by SRC2SRCML and to compute the following system-level metrics:

1. Number of Feature Constants (NOFC): The total number of macro names that are referred in, at least, one *ifdef*.

2. Number of Feature Expressions (NOFE): The total number of conditional expressions used in *ifdef* directives to control the inclusion or exclusion of variable code.

3. Number of Top-level Branches (NOTLB): The total number of top-level *ifdef* branches, including `#else` preprocessor directives. An *ifdef* branch is a block of code that is delimited by `#ifdef`, `#ifndef`, `#if`, or `#else` and closed by its `#endif` or followed by a `#else` or `#elif` (when applicable). A top-level *ifdef* branch is an *ifdef* or `#else` that is not inside an enclosing *ifdef* branch.

In addition, FSCAT also computes metrics for each feature constant, feature expression, and top-level *ifdef* branch of the system under analysis, as follows:

1. Scattering Degree (SD): This degree is calculated per feature constant of a system. It counts the number of *ifdefs* that refer to a given feature constant. For example, considering the examples in Figure 2.1 and Figure 2.2, the scattering degree of MS_WINDOWS is 2.

2. Tangling Degree (TD): This degree is calculated per feature expression of a system. It counts the number of feature constants that occur in a given feature expression. For example, in Figure 2.2, the tangling degree of the feature expression in Line 427 is 2.

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

[4] https://bitbucket.org/lpassos/fscat

3. Nesting Depth (ND): This metric is defined for each top-level *ifdef* branch in a system. ND is the depth of the tree of nested annotations in a given top-level *ifdef* branch. In this tree, the nodes are *ifdefs* and `#else` annotations, while the edges represent nested relations between such nodes. The root node is a top-level *ifdef* or `#else`, and the children of a node are the annotations it directly encloses. The depth of a node is the depth of its parent plus 1. By definition, the depth of the root node is one. The depth of the tree is the depth of its node with the maximal depth. For example, in Figure 2.1, the ND of the top-level `#ifdef` in Line 437 is 1, and the ND of the top-level `#ifdef` in Line 452 is 2.

The proposed metrics are based on the metrics of Liebig et al. [2010]. SD is based on the number of maintenance program locations that one potentially has to consider to maintain a feature and has been already used in other studies [Hunsen et al., 2015; Jbara and Feitelson, 2013; Liebig et al., 2010; Passos et al., 2015]. Likewise, our definition of TD is also based on Liebig et al. However, the tool used by Liebig et al. for computing SD and TD (cppstats) applies transformations on the annotations of the code (e.g., by propagating the condition of outer *ifdefs* to inner ones and conjoining each `#elif`/`#else` condition with the condition of preceding branches). These transformations influence the scattering and tangling values, causing higher values. In contrast, the tooling we use when collecting these metrics (fscat) does not perform any transformation on annotations, taking them as explicitly defined in the source code. As an example, Figure 4.1 shows two fragments of the same code. In the original code (a), as computed by fscat, FEATURE_A has SD=1 and the feature expression at line 3 has TD=1. In the fragment (b), we show the code after the transformations performed by cppstats. In this case, cppstats returns that FEATURE_A has SD=3, and that the feature expression at line 3 has TD=2.

```
1 #ifdef FEATURE_A
2     (...)
3 #ifdef FEATURE_B
4     (...)
5 #endif
6 #elif FEATURE_C
7     (...)
8 #endif
```

(a) original code

```
1 #ifdef FEATURE_A
2     (...)
3 #ifdef FEATURE_A && FEATURE_B
4     (...)
5 #endif
6 #elif !FEATURE_A && FEATURE_C
7     (...)
8 #endif
```

(b) code with transformations

Figure 4.1: Example of code, as considered by fscat (a) and after the transformations performed by cppstats (b)

The third metric, however, differs from the ones proposed by Liebig et al. Originally, Liebig et al. define the Average Nesting Depth (AND) as the average depth of nested *ifdefs*. Instead, we propose the metric Nesting Depth (ND) representing the depth of each top-level *ifdef* branch. We argue that ND is more robust than AND, because it is not based on averages. Furthermore, while performing maintenance tasks, a programmer has to be aware of inner *ifdefs* when reasoning about the code. ND supports the developer to estimate the complexity of the code fragment inside a top-level branch. The AND metric, however, gives only a rough estimation of the complexity of the whole file. As an example, in Figure 4.1, AND is the average nesting depth of each *ifdef* block in this file (AND $= (1 + 2)/2 = 1.5$). In contrast, ND captures the nesting of each top-level branch, e.g., ND of the top-level *ifdef* branch at lines 1–5 is 2, and the ND of the top-level *ifdef* branch at lines 6–8 is 1.

Liebig et al. also use the *Granularity* and *Type* metrics to reveal the number of extensions in a software system by the level of granularity e.g., 6% of extensions in LIGHTTPD are fine-grained) and type (e.g., 36 extensions in OPENLDAP are homogeneous, i.e., they use duplicated code). However, both metrics are computed at a system level, and therefore do not generate a distribution of metric values for each system.

4.1.3 Statistical Analysis

After collecting the metrics, we inspect the histograms and standard descriptive statistics describing the distributions of SD, TD, and ND for each of our 20 subject systems. In addition, we computed the Gini coefficient [Gini, 1921] to measure the degree of concentration of the metric values inside each distribution. The Gini coefficient has been proposed as an economic indicator to measure and compare income distributions, but can be adapted to the distribution of software metrics, providing an aggregated metric that is system-size independent [Vasa et al., 2009; Serebrenik and van den Brand, 2010; Vasilescu et al., 2011].

In this initial analysis step, we check whether the collected distributions have characteristics of a power-law distribution. Then, we proceed with a rigorous test of the power-law hypothesis. To decide whether the metric values follow a power-law distribution we used the POWERLAW package [Gillespie, 2014a] from the R statistical environment.[5] First, we define the two parameters k and x_{min} of the best-fit power law ($\mathbb{P}_0$) that approximates as close as possible the empirical CDF (cumulative distribution function) of the distribution ($\mathbb{P}$) of a system under analysis. When searching for $\mathbb{P}_0$, we rely on the maximum-likelihood estimator method (MLE), while choosing x_{min} as

[5]http://www.r-project.org/

the value minimizing the Kolmogorov-Statistic (KS). The KS is given by the maximum distance $|\mathbb{P}_0(x) - \mathbb{P}(x)|$, for all $x > x_{min}$. For further details, the reader is referred to elsewhere [Clauset et al., 2009; Gillespie, 2014b,a].

Once we estimated k and x_{min}, we perform a hypothesis test to verify whether a power-law distribution is a plausible model for the behavior of each empirical CDF. Following Clauset et al. [2009], we perform a goodness-of-fit test via a bootstrap procedure following the algorithms and steps outlined by Gillespie [2014b]. Simply put, we generate 2 500 datasets from the $\mathbb{P}_0$ model and then try to re-infer a new best-fit power law for each generated dataset. The *p-value* of the simulation process corresponds to the fraction of times in which the KS of the best-fit model of the generated dataset is higher than the one obtained for $\mathbb{P}$. Our hypotheses are the following:

- Null hypothesis: $\mathbb{P}_0$ is a plausible fit for $\mathbb{P}$
- Alternative hypothesis: $\mathbb{P}_0$ is not a plausible fit for $\mathbb{P}$

As Clauset et al. argue, if the test reports a *p-value* larger than 0.1, one shall accept the null hypothesis. Otherwise, it should be rejected in favor of the alternative hypothesis. In the latter case, $\mathbb{P}$ is unlikely to conform to a power-law distribution; rather, the empirical CDF function better fits another model, and may or may not be heavy-tailed.

For each metric of each subject system, the fit is initially done based on the whole set of metric data. However, when we do not have strong evidence that the data fit a power law (*p-value* < 0.1), we verify whether it is possible to fit, at least, part of the data, which is valid as we want to prove that the distribution follows a power law, asymptotically. To this end, we iteratively crop single data points from the right of the tail, removing at each step the highest metric value. The iteration continues until we find a fit up to a certain value x_{max}. The procedure stops if we remove 1% of the data points and we still do not find strong evidence for a power law. A similar procedure has been reported by Baxter et al. [2006], when analyzing standard metrics for Java software.

4.1.4 Threshold Extraction

Based on our statistical analysis, we derive thresholds for feature scattering, tangling, and nesting in a way that respects the distributions found in our study. Taking data skew into account, we rely on the notion of *relative thresholds* and on the functions introduced by Oliveira et al. [2014b] to calculate thresholds for our metrics with heavy-tailed distribution. We also illustrate how the proposed thresholds can be used to check whether a system implementation includes a complex usage of *ifdefs*. Figure 4.2 summarizes the main steps of the methodology of this study.

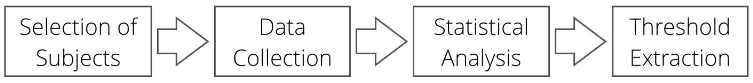


Figure 4.2: Main steps of our methodology

4.2 Results

In this section, we describe the results of our distribution fitting analysis for the three metrics: Scattering Degree (Section 4.2.1), Tangling Degree (Section 4.2.2), and Nesting Depth (Section 4.2.3).

4.2.1 Scattering Degree

To assess the distribution of the feature-scattering degrees, we initially plotted the histograms of the extracted SD values for each system, shown in (Figure 4.3). We can observe that all histograms are right-skewed, meaning that, while most SD values are small (equal to one, in most cases), we also observe high and very high SD values, suggesting a heavy-tailed—possibly a power-law—distribution. Table 4.2 shows the collected measures for SD per system, including the number of feature constants, the mean, median, 95th percentile, maximum value, mode, the percentage of entities following the mode, and also the Gini coefficient. The scattering degree reaches extreme values in GCC (max SD=1 867, for the feature __cplusplus) and in the LINUX KERNEL (max SD=2 698, for the feature __BIG_ENDIAN_BITFIELD), while other systems have lower degrees, for example, VI (max SD=53, for the feature BIT8) and LIGHTTPD (max SD=48, for the feature USE_OPENSSL). Furthermore, the mean SD value is, in all cases, too far apart from the values in the last 5% of the features. For instance, in the LINUX KERNEL, the mean SD is 5.40, and the 95th percentile of SD is 14. This nicely illustrates that the mean should not be used as a reference or centrality measure when analyzing SD. For instance, in the LINUX KERNEL, the mean is not only very different from the small SD values (equal to 1) that represent the bulk of the distribution, but it also does not represent the high SD values in the right part of the histogram (≥ 5.4). Therefore, it is fundamental to know the statistical distribution describing the collected SD data, before investigating reference values, thresholds, and similar quantitative guidelines for this metric.

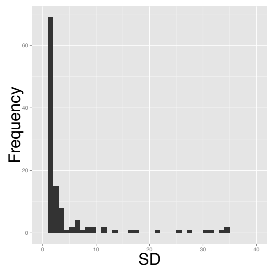
Finally, as presented in Table 4.2, the SD distribution's Gini coefficients range from 0.56 (GIT) to 0.77 (SYLPHEED), which suggests a high level of inequality inside each distribution. All aforementioned characteristics are necessary, yet not sufficient, to claim that feature scattering follows a power-law distribution.

Table 4.2: Scattering degree (SD) descriptive measures (NOFC: Number of Feature Constants)

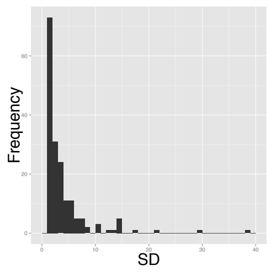
System	NOFC	Mean	Median	95th	Max	Mode	Mode %	Gini
VI	118	4.72	1	27.45	53	1	58.4	0.66
LIGHTTPD	179	4.20	2	14.00	48	1	40.7	0.57
XFIG	110	3.87	1	14.10	83	1	54.5	0.64
SENDMAIL	905	3.91	2	11.00	204	1	45.0	0.59
SYLPHEED	121	8.90	2	35.00	242	1	47.1	0.77
GIT	383	2.62	1	7.90	92	1	71.2	0.56
APACHE	606	3.30	1	12.00	114	1	57.2	0.58
LIBXML2	2 095	4.14	1	13.00	379	1	86.0	0.73
EMACS	1 970	3.50	1	9.00	211	1	66.8	0.64
OPENLDAP	784	4.10	1	14.00	85	1	50.6	0.62
SUBVERSION	217	5.63	1	11.00	339	1	53.4	0.72
IMAGEMAGICK	636	5.46	1	13.00	429	1	53.9	0.72
PYTHON	2 849	2.71	1	7.00	322	1	69.5	0.56
PHP	2 502	4.37	1	12.00	674	1	56.9	0.67
POSTGRESQL	1 264	4.49	1	13.85	569	1	62.7	0.70
GIMP	557	4.66	1	18.00	156	1	56.7	0.66
GLIBC	3 370	4.94	1	14.00	662	1	56.5	0.71
MYSQL	1 990	6.93	2	18.00	652	1	47.9	0.75
GCC	8 898	4.38	1	13.00	1 867	1	57.1	0.68
LINUX KERNEL	12 661	5.40	1	14.00	2 698	1	52.1	0.71

Following the methodology described in Section 4.1.3, as a next step we estimate the parameters of the power-law model (k and x_{min}) that best fit our empirical CDF. In Figure 4.4, we plot the empirical CDF and the fitted power law in logarithmic scale, of which the latter appears as a red decreasing line in each graph of figure. As we stated in Section 2.4, the resulting line is a distinct characteristic of power-law distributions. The resulting plot reveals that the points approximate the line of the power law, strengthening our understanding that feature scattering indeed follows a power-law distribution.

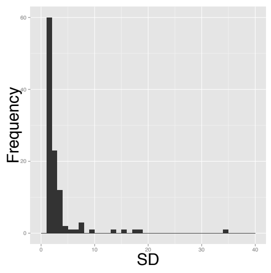
To statistically check whether the fitted power laws are plausible models (null hypothesis), we perform a bootstrapping hypothesis test, following the methodology described in Section 4.1.3. Table 4.3 shows the *p-values* obtained for each test. In the case of 14 systems, we found statistically significant models (*p-value* > 0.1), leading us to accept the null hypothesis—the power-law model is a plausible explanation model. Similar to the estimation of x_{min}, in the case of three systems (SUBVERSION, GIMP,



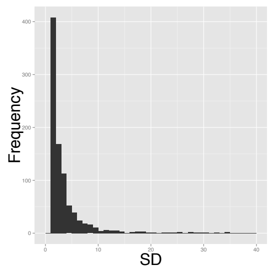
(a) vi



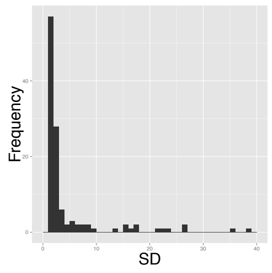
(b) lighttpd



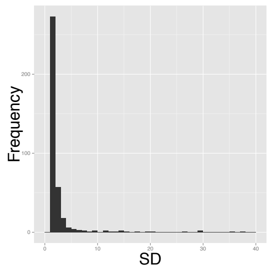
(c) xfig



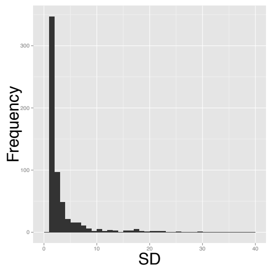
(d) sendmail



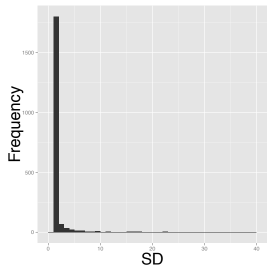
(e) sylpheed



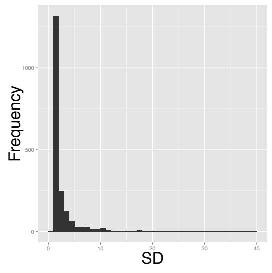
(f) git



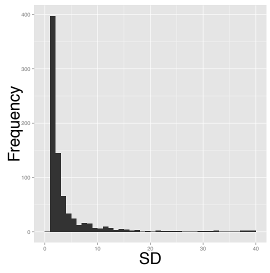
(g) apache



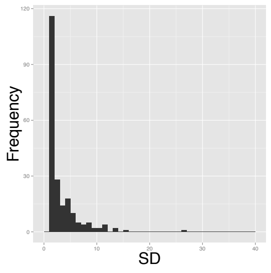
(h) libxml2



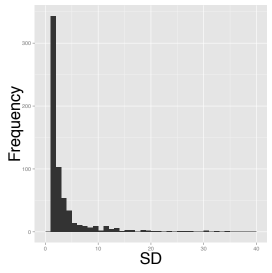
(i) emacs



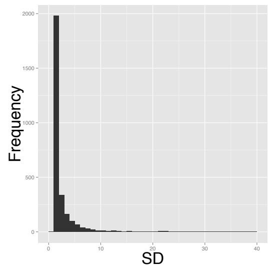
(j) openldap



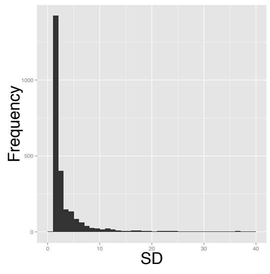
(k) subversion



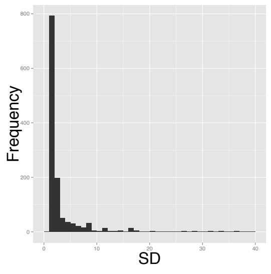
(l) imagemagick



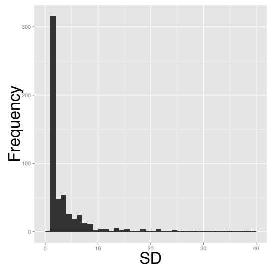
(m) python



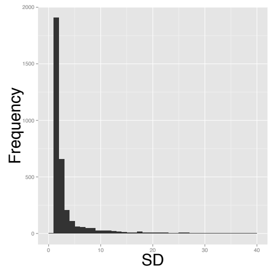
(n) php



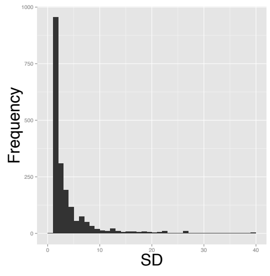
(o) postgresql



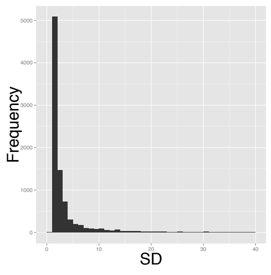
(p) gimp



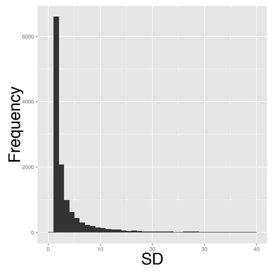
(q) glibc



(r) mysql

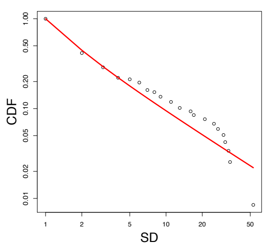


(s) gcc

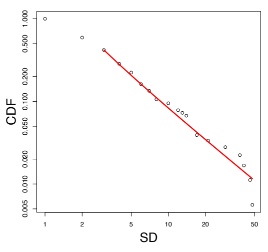


(t) linux kernel

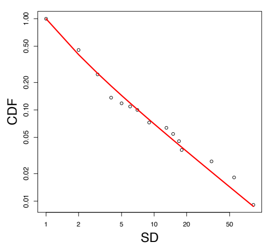
Figure 4.3: Histograms of scattering degrees (SD)



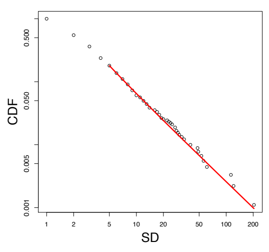
(a) VI



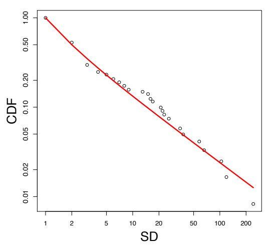
(b) LIGHTTPD



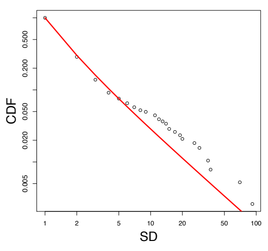
(c) XFIG



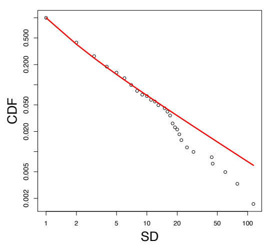
(d) SENDMAIL



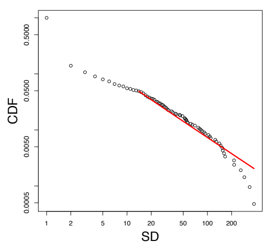
(e) SYLPHEED



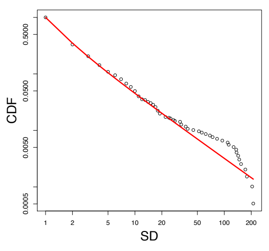
(f) GIT



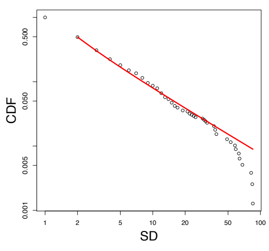
(g) APACHE



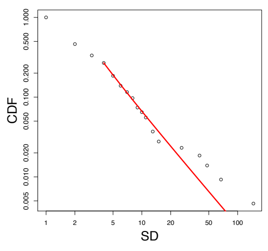
(h) LIBXML2



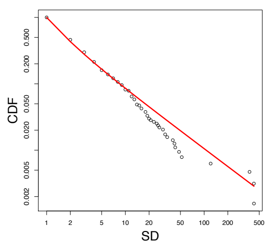
(i) EMACS



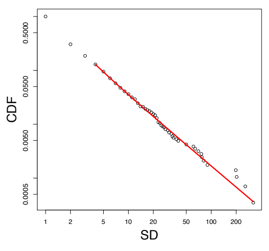
(j) OPENLDAP



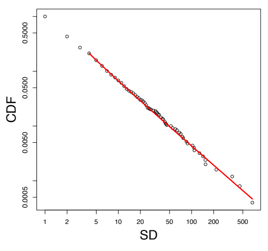
(k) SUBVERSION



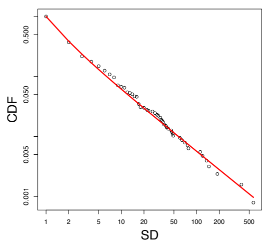
(l) IMAGEMAGICK



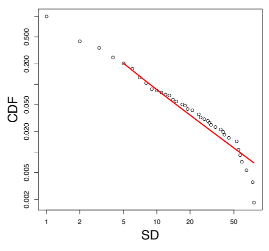
(m) PYTHON



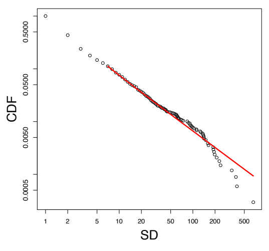
(n) PHP



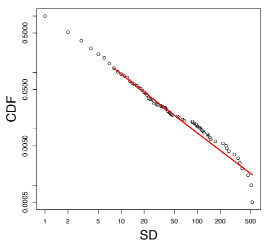
(o) POSTGRESQL



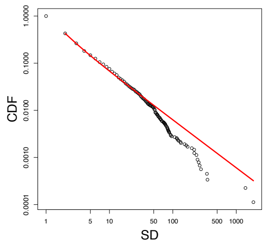
(p) GIMP



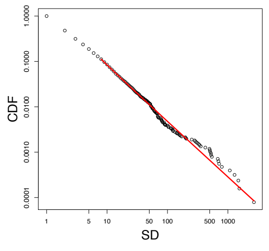
(q) GLIBC



(r) MYSQL



(s) GCC



(t) LINUX KERNEL

Figure 4.4: Empirical CDFs of the scattering degrees (points) and the fitted power law (red line), both in logarithmic scale

Table 4.3: Power-law best-fit analysis for scattering degree (SD). Significant results (*p-value* > 0.1) are bold.

System	k	x_{min}	x_{max}	% crop	p-value
VI	1.8542	1	53	0	**0.2692**
LIGHTTPD	2.2239	3	48	0	**0.7928**
XFIG	1.9645	1	83	0	**0.1388**
SENDMAIL	2.3729	5	204	0	**0.7392**
SYLPHEED	1.7286	1	242	0	**0.1728**
GIT	2.3003	1	92	0	**0.1592**
APACHE	1.9704	1	114	0	0.0632
LIBXML2	1.9535	14	379	0	**0.1880**
EMACS	2.1288	1	211	0	0.0020
OPENLDAP	2.0032	2	85	0	0.0392
SUBVERSION	2.4064	4	146	0.46	**0.4984**
IMAGEMAGICK	1.8915	1	429	0	0.0220
PYTHON	2.2993	4	322	0	**0.7960**
PHP	2.1652	4	674	0	**0.9636**
POSTGRESQL	2.0145	1	569	0	0.0292
GIMP	2.1997	5	76	0.35	**0.1272**
GLIBC	2.0358	7	662	0	**0.7840**
MYSQL	2.0255	8	528	0.05	**0.1128**
GCC	2.0146	2	1867	0	0.0000
LINUX KERNEL	2.2216	8	2698	0	**0.5516**

and MYSQL), the best fit to a power law requires an upper bound value (x_{max}). This cropping of a small number of features (less than 0.5%) in the end of the distribution is necessary when the power-law behavior seems to fit most of the distribution, but does not hold for some few higher values. For the remaining six systems, we reject the null hypothesis in favor of the alternative hypothesis (the power-law model is not a plausible explanation model). Note that this is not the same as concluding that the scattering distribution of these six systems is not heavy-tailed. In fact, Figure 4.4 suggests a heavy-tailed distribution for all 20 systems, some of which possibly follow an alternative distribution (e.g., stretched exponential or log-normal).

4.2.2 Tangling Degree

To analyze the distribution of tangling degrees, we also plotted the histograms of the extracted TD values for each system, as shown in Figure 4.5. We observe that the histograms follow a different pattern from the ones for scattering (Figure 4.3). In particular, most TD values are equal to one, and we have very few feature expressions with higher TD values. Table 4.4 shows the collected measures for TD per system, including the number of feature expressions (NOFE), as well as the mean, median,

Table 4.4: Tangling degree (TD) descriptive measures (NOFE: Number of Feature Expressions)

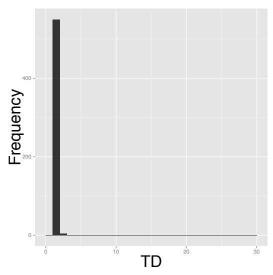
System	NOFE	Mean	Median	95th	Max	Mode	Mode %	Gini
VI	554	1.00	1	1	2	1	99.2	0.01
LIGHTTPD	686	1.09	1	2	7	1	93.5	0.08
XFIG	378	1.12	1	2	7	1	94.4	0.10
SENDMAIL	3 176	1.11	1	2	7	1	90.5	0.09
SYLPHEED	986	1.09	1	1	6	1	95.1	0.08
GIT	885	1.13	1	2	10	1	91.0	0.11
APACHE	1 788	1.12	1	2	7	1	93.3	0.10
LIBXML2	8 127	1.06	1	2	8	1	94.8	0.06
EMACS	5 565	1.23	1	2	12	1	82.2	0.16
OPENLDAP	2 930	1.09	1	2	8	1	91.7	0.08
SUBVERSION	1 113	1.09	1	2	6	1	92.5	0.08
IMAGEMAGICK	2 732	1.27	1	2	5	1	76.3	0.16
PYTHON	6 969	1.11	1	2	7	1	91.5	0.09
PHP	9 373	1.16	1	2	8	1	87.5	0.12
POSTGRESQL	4 717	1.20	1	2	11	1	88.5	0.15
GIMP	2 118	1.22	1	2	6	1	83.8	0.16
GLIBC	13 345	1.24	1	2	14	1	80.6	0.16
MYSQL	12 359	1.11	1	2	8	1	91.0	0.09
GCC	29 842	1.30	1	3	21	1	82.1	0.20
LINUX KERNEL	63 482	1.07	1	2	12	1	93.5	0.06

95th percentile, maximum value, mode, and the percentage of entities following the mode. The table also provides the Gini coefficients computed over the TD values of a system. First of all, we can observe that, in all systems, the mean is close to one and both the median and the mode are equal to one. Second, the 95th percentile is less or equal to two in 19 systems (only in GCC it is equal to three). In 13 systems, the mode represents, at least, 90% of the measured values. For example, 99.2% of the feature expressions in VI have a TD value equal to one. IMAGEMAGICK is the system with the lowest frequency of TD values that are equal to one (76.3%).

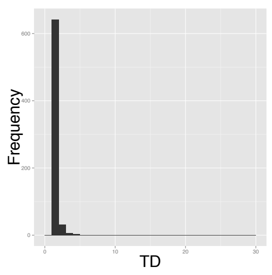
Finally, the Gini coefficients for TD—across all subjects—are less than 0.21, which shows that tangled degree is nearly equally distributed. For all systems, we found that a power-law distribution is not a plausible model for the reported TD values. Although it is possible to fit a power law to the tail of each TD distribution, the number of data points in the right of the tail is too small to claim statistical power.

4.2.3 Nesting Depth

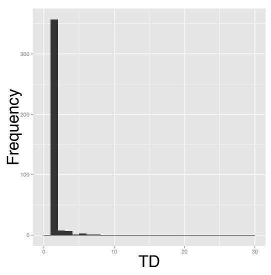
As shown in Figure 4.6, the histograms of the ND values for each system are similar to the ones for tangling: most ND values are equal to one, and we have very few



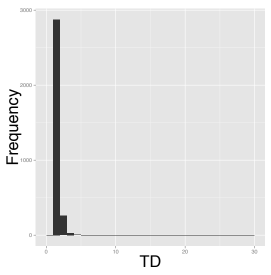
(a) vi



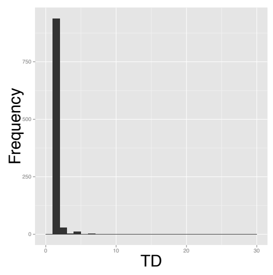
(b) lighttpd



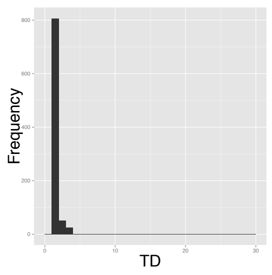
(c) xfig



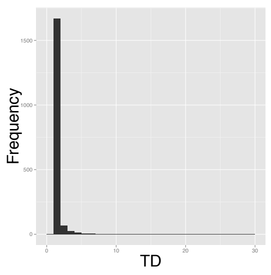
(d) sendmail



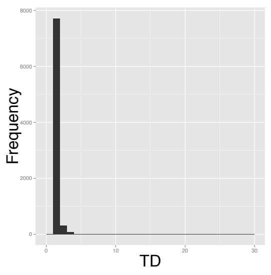
(e) sylpheed



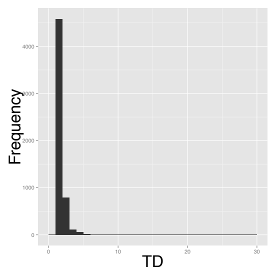
(f) git



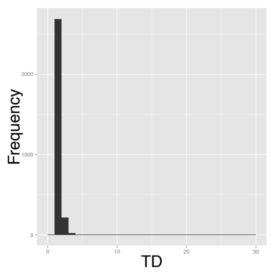
(g) apache



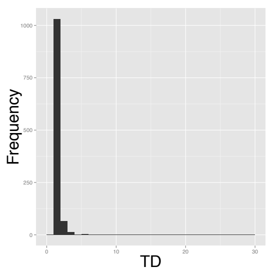
(h) libxml2



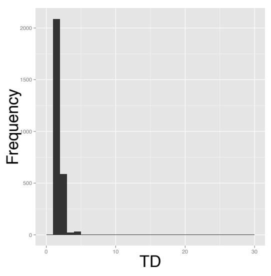
(i) rmacs



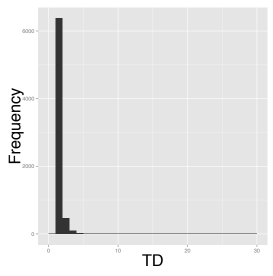
(j) openldap



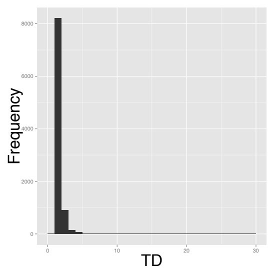
(k) subversion



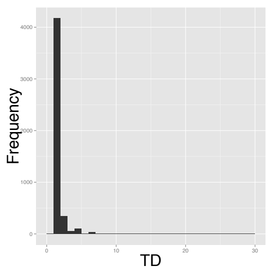
(l) imagemagick



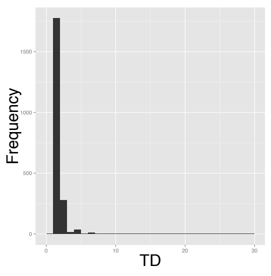
(m) python



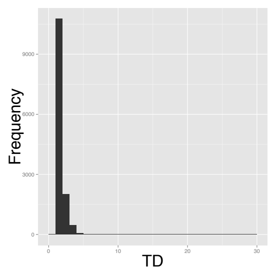
(n) php



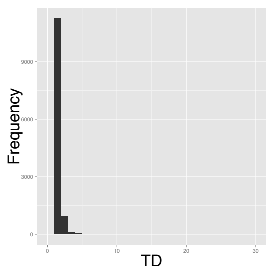
(o) postgresql



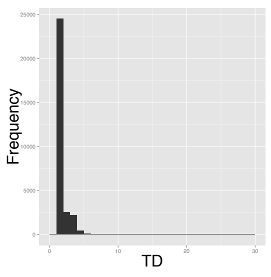
(p) gimp



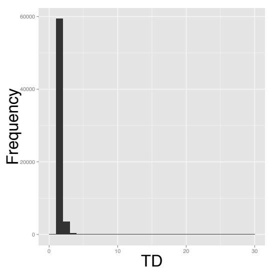
(q) glibc



(r) mysql



(s) gcc



(t) linux kernel

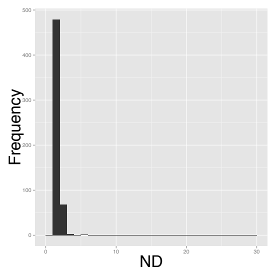
Figure 4.5: Histogram of tangling degrees (TD)

branches with nested *ifdefs* and `#else` annotations. Table 4.5 shows the number of top-level branches (NOTLB) in the subject systems, as well as the mean, median, 95th percentile, maximum value, mode, the percentage of entities following the mode, and also the Gini coefficients computed over the ND values extracted for each system. Similarly to TD, the mode is one in all systems and it represents, at least, 86% of the measured ND values. For example, 94.7% of the *ifdef* branches in APACHE have ND values equal to one, and GIT is the system with the lowest number of branches without nested *ifdefs* (86.2%).

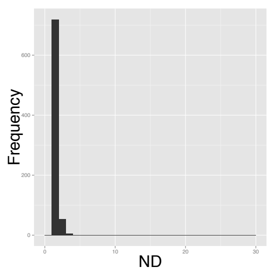
Much like for TD, the Gini coefficients are quite low, less than 0.15, suggesting a uniform distribution, even more than the ones observed for tangling. As a consequence, we found that a power-law distribution is not a plausible model for the reported ND values. As we concluded for TD, although it is possible to fit a power law to the tail of each ND distribution, the number of data points in the right of the tail is too small to claim statistical power.

Table 4.5: Nesting Depth (ND) descriptive measures (NOTLB: Number of Top-Level Branches)

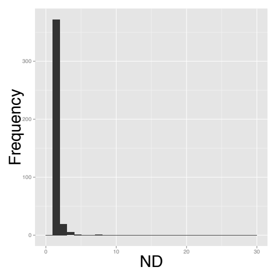
System	**NOTLB**	**Mean**	**Median**	**95th**	**Max**	**Mode**	**Mode %**	**Gini**
VI	551	1.14	1	2	5	1	86.9	0.10
LIGHTTPD	780	1.08	1	2	5	1	92.1	0.07
XFIG	398	1.09	1	2	7	1	93.4	0.08
SENDMAIL	2 290	1.17	1	2	5	1	86.2	0.13
SYLPHEED	1 197	1.06	1	2	6	1	94.4	0.05
GIT	809	1.17	1	2	9	1	86.2	0.12
APACHE	1 799	1.05	1	2	6	1	94.7	0.05
LIBXML2	2 585	1.14	1	2	7	1	88.8	0.11
EMACS	3 106	1.18	1	2	15	1	87.3	0.14
OPENLDAP	2 626	1.12	1	2	5	1	89.5	0.09
SUBVERSION	1 277	1.04	1	1	4	1	95.7	0.04
IMAGEMAGICK	2 139	1.09	1	2	5	1	93.0	0.08
PYTHON	6 416	1.12	1	2	9	1	89.8	0.09
PHP	7 868	1.10	1	2	6	1	90.8	0.09
POSTGRESQL	4 044	1.11	1	2	6	1	91.8	0.09
GIMP	2 216	1.09	1	2	6	1	92.8	0.08
GLIBC	12 062	1.14	1	2	6	1	88.0	0.11
MYSQL	11 850	1.08	1	2	6	1	92.8	0.07
GCC	26 888	1.11	1	2	24	1	90.6	0.09
LINUX KERNEL	71 591	1.05	1	2	5	1	94.6	0.05



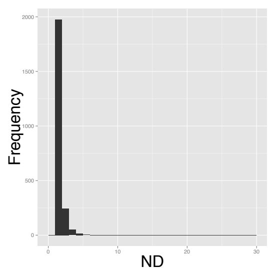
(a) VI



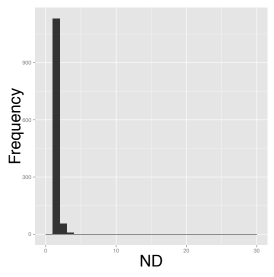
(b) LIGHTTPD



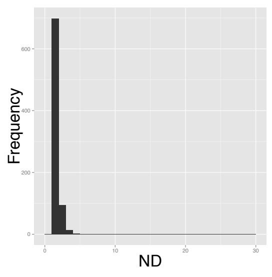
(c) XFIG



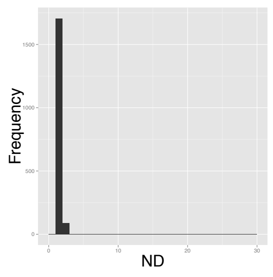
(d) SENDMAIL



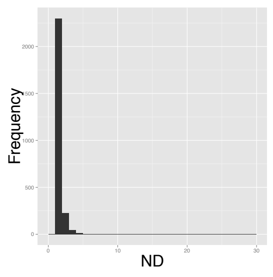
(e) SYLPHEED



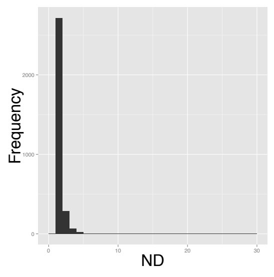
(f) GIT



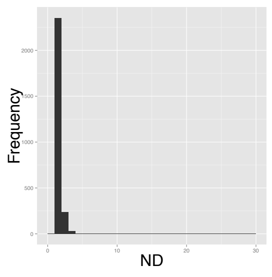
(g) APACHE



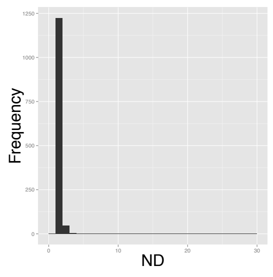
(h) LIBXML2



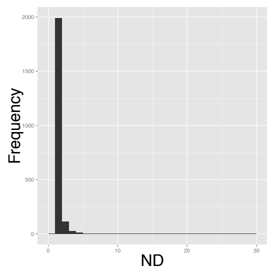
(i) EMACS



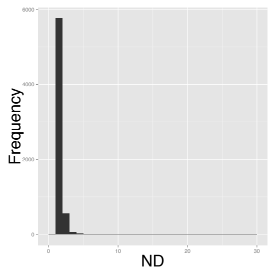
(j) OPENLDAP



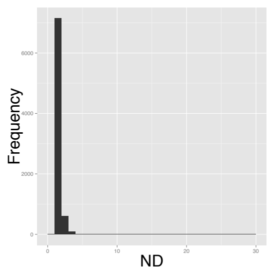
(k) SUBVERSION



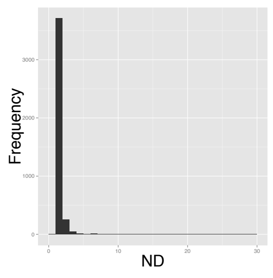
(l) IMAGEMAGICK



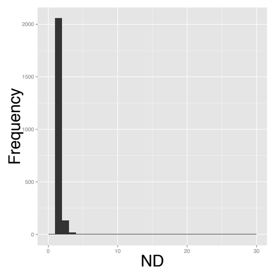
(m) PYTHON



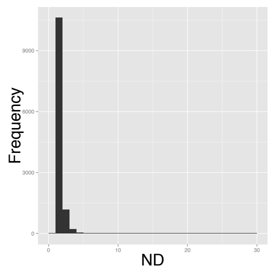
(n) PHP



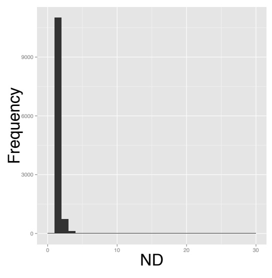
(o) POSTGRESSQL



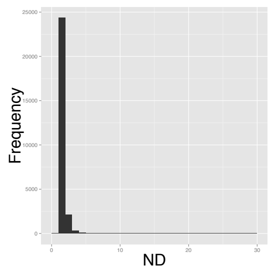
(p) GIMP



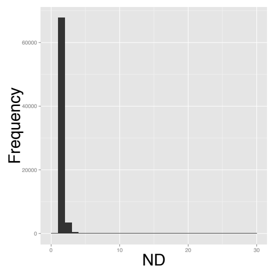
(q) GLIBC



(r) MYSQL



(s) GCC



(t) LINUX KERNEL

Figure 4.6: Histogram of Nesting Depth (ND)

4.3 Thresholds for Feature-Related Metrics

Our empirical study shows that feature scattering is highly skewed. In fact, 14 out of 20 systems show strong evidence that feature scattering is heavy-tailed, and the underlying distributions follow a power-law. Tangling and nesting have a more uniform distribution for all subject systems, with most values equal to one. These findings are of great relevance for the extraction of thresholds for the studied metrics. Most importantly, feature-scattering thresholds should not be based on centrality statistic measures (e.g., mean and standard deviation). In contrast, tangling and nesting have only a small number of outliers, which are not too far apart from the most typical values of these two metrics. Thus, mean values are robust thresholds for the tangling and nesting metrics.

Based on our statistical analysis, we derive thresholds for feature scattering, tangling, and nesting in a way that respects the distributions found in our study. Taking data skew into account, we rely on the notion of *relative thresholds*, which we explain next.

4.3.1 Relative Thresholds

Several code metrics, measuring properties such as size, coupling, and cohesion, are well known following heavy-tailed distributions [Baxter et al., 2006; Louridas et al., 2008; Wheeldon and Counsell, 2003; Ferreira et al., 2012; Filó et al., 2014]. For this reason, previous work proposed techniques to extract thresholds that do not rely on the mean or the standard deviation. For example, Oliveira et al. [2014b] proposed the notion of relative thresholds for evaluating heavy-tailed metric values, along with a set of functions that obtain such thresholds from a set of subject systems (*Corpus*). Relative thresholds have the following format:

$$\textit{at least } p\% \textit{ of the entities should have } M \leq k$$

where M is a metric calculated for a given source code entity, k is an upper limit, and p is the minimal percentage of entities that should be below this upper limit. The goal is to establish upper limits for metric values that should be followed by most entities, not necessarily all, though. The reason is that, in heavy-tailed distributions, the high metric values of the distribution make it challenging to define thresholds for all entities. Thus, relative thresholds attempt to balance two forces: (i) on one hand, relative thresholds should reflect real design rules, followed by most subjects in the target system; (ii) on the other hand, the prescribed thresholds should not be based on lenient upper limits.

$$
\begin{aligned}
\mathit{ComplianceRate}[p,k] &= \frac{\mid \{\, S \in \mathit{Corpus} \mid p\% \text{ of the entities in } S \text{ have } M \leq k \} \mid}{\mid \mathit{Corpus} \mid} \\
\mathit{penalty}_1[p,k] &= \begin{cases} \dfrac{90 - \mathit{ComplianceRate}[p,k]}{90} & \text{if } \mathit{ComplianceRate}[p,k] < 90 \\ 0 & \text{otherwise} \end{cases} \\
\mathit{penalty}_2[k] &= \begin{cases} \dfrac{k - \mathit{Median90}}{\mathit{Median90}} & \text{if } k > \mathit{Median90} \\ 0 & \text{otherwise} \end{cases} \\
\mathit{CompliancePenalty}[p,k] &= \mathit{penalty}_1[p,k] + \mathit{penalty}_2[k]
\end{aligned}
$$

Figure 4.7: *ComplianceRate* and *CompliancePenalty* functions [Oliveira et al., 2014b]

For example, a threshold stating that "95% of the feature constants in a system should have a scattering degree less than 3K" is probably satisfied by most systems.

Figure 4.7 presents the functions introduced by Oliveira et al. to calculate the parameters p and k that define the relative threshold for a given metric M. First, function *ComplianceRate*$[p, k]$ returns the percentage of systems in the *Corpus* that follows the relative threshold defined by the pair $[p, k]$. To determine the best p and k, *ComplianceRate* is maximized, while accounting for a minimal *CompliancePenalty*. The latter is the sum of penalties introduced by two functions:

- *penalty*$_1[p, k]$: a *ComplianceRate*$[p, k]$ less than 90% receives a penalty proportional to its distance to 90%. This penalty fosters thresholds followed by, at least, 90% of the systems in the *Corpus*.
- *penalty*$_2[k]$: a *ComplianceRate*$[p, k]$ receives the second penalty proportional to the distance between k and the median of the 90th percentiles of the values of M in each system in the *Corpus*, denoted by *Median90*.

4.3.2 Thresholds for Scattering Degree

As we found that feature scattering degrees follow a heavy-tailed distribution, we computed relative thresholds for this metric, using our sample of 20 systems and the compliance functions described in Section 4.3.1, obtaining the following result:

$\boxed{\text{at least, 85\% of the feature constants in a system should have SD} \leq 6}$

In fact, this threshold holds for all systems in our corpus except VI and SYLPHEED, which exceed the threshold only marginally. In VI, we observe that 83% of the feature constants have a SD ≤ 6 and, in SYLPHEED, this percentage is 82%. However, the proposed relative threshold holds for large and complex systems, with thousands of *ifdefs*, such as the LINUX KERNEL, GCC, and MYSQL. Figure 4.8 shows the percentile functions for the SD values of each subject system. The x-axis represents the percentiles, and the y-axis represents the upper SD values of the feature constants matching the percentile. The plot nicely illustrates that SD values are heavy-tailed, as already concluded in Section 4.2.1 . However, there are two systems whose SD values begin to grow earlier, around the 85th percentile, which are exactly VI and SYLPHEED.

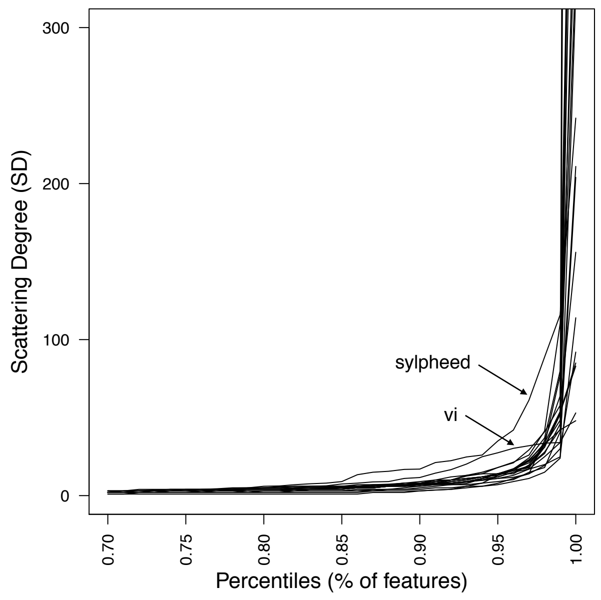


Figure 4.8: Percentile plots of scattering degrees (SD)

These results suggest that, the proposed relative threshold reflects the most common scattering distributions found in our corpus. Another corpus, however, may yield a different threshold (e.g., a corpus with systems of a particular domain). However, assuming that we selected a representative sample of C-preprocessor-based systems, including small, medium, and large systems, we expect that different corpora would not produce radically different thresholds. In other words, not following the thresholds by a small margin—like in VI and SYLPHEED—does not necessarily mean a serious design flaw. However, if only 50% of the feature constants in a system have SD ≤ 6, this would certainly raise more serious concerns on the quality of the feature-implementation structure. For example, Spencer and Collyer [Spencer and Collyer, 1992] claim that ifdef-based implementation should follow basic principles of software engineering, including clean interfaces and information hiding. More specifically, *ifdefs* should be hidden behind interfaces, making it possible to implement the bulk of the software as a single version using these interfaces. Code that do not follow the proposed thresholds for scattering degree might, for example, have many *ifdefs* that do not follow this general principle.

4.3.3 Thresholds for Tangling Degree

Tangling degrees approximate an uniform distribution, allowing to directly define thresholds. After inspecting the results in Table 4.4, specifically the mode and its relative frequency (%), we propose the following threshold:

$\boxed{\text{at least, 80\% of the feature expressions in a system should have TD = 1}}$

To define this threshold we assume that the mode of the TD distributions (which is equal to one in all systems) should correspond to, at least, 80% of the feature expressions in each system. We propose this threshold based on the percentage of feature expressions that follow the mode in most subjects, minimizing the number of outliers. In other words, we assume that systems where the mode corresponds to less than 80% of the feature expressions deviate from an uniform distribution and therefore are outliers. All systems in our sample follow this threshold, except IMAGEMAGICK.

4.3.4 Thresholds for Nesting Depth

After inspecting the measures of Table 5, specially the mode and its frequency, we propose the following threshold:

$\boxed{\text{at least, 85\% of the top-level branches in a system should have ND = 1}}$

To define this threshold we followed the assumptions of the TD threshold, based on the percentage of entities that follow the mode in most subjects. However, in this case we are requiring the mode of the ND distribution (ND=1) to correspond to, at least, 85% of the top-level *ifdef* branches (and not 80%, as in the TD threshold). The reason is that in our sample the mode of ND corresponds to, at least, 86.2% of the top-level *ifdef* branches in each system. Therefore, this threshold is followed by all systems in our sample. The absence of systems not following the proposed threshold is explained by the fact that the ND distributions are quite similar across all subject systems.

4.3.5 Discussion

The proposed thresholds can be used to check whether a system implementation includes a complex usage of *ifdefs*, at least when compared with other relevant systems (i.e., the systems considered in our Corpus). To illustrate this usage, we applied our thresholds on XTERM (version 3.1.8), the standard terminal emulator for the XWindow system. Existing research shows that XTERM makes a heavy and complex usage of *ifdefs*. For example, Liebig et al. show that almost 40% of XTERM's lines of code are enclosed by *ifdefs* Liebig et al. [2010]. Moreover, almost 10% of the *ifdefs* in XTERM are undisciplined annotations, i.e., they delimit tokens that do not align with the syntactic code structure, e.g., with entire statements, functions, and type declarations [Liebig et al., 2011]. Furthermore, we inspected the description of 318 patches of XTERM, from 1996 to 2015.[6] We found that 82 patches (26%) included 110 changes in *ifdefs*, to correct bugs, to implement new features, or due to refactorings. We provide three examples of such changes:

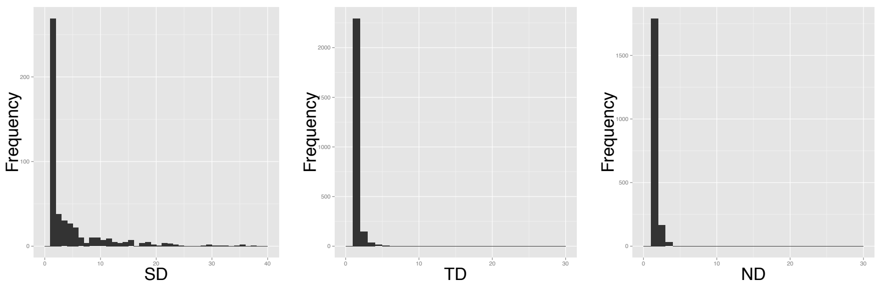
- Patch #315: *"fix an ifdef'ing problem, where –disable-dec-locator would turn off logic needed for DECIC and DECDC."*
- Patch #275: *"adjust ifdef's for putenv and unsetenv in case only one of those is provided on a given platform."*
- Patch #216: *"ifdef'd Sun function-key feature to make it optional, like HP and SCO."*

Therefore, we hypothesize that XTERM *should be classified as an outlier system*, according to the thresholds for SD, TD, or ND derived in the previous sections. To check this hypothesis, we used FSCAT to compute the distribution of the SD, TD, and ND values in XTERM. These distributions are presented in Figure 4.9.

We then checked whether XTERM follows the proposed thresholds, with the following results:

- The threshold derived for SD states that a system should have, at least, 85% of the *ifdefs* with $SD \leq 6$. However, in XTERM, only 79% of the *ifdefs* have $SD \leq 6$. Therefore, XTERM is indeed an outlier regarding SD.
- The threshold for TD states that a system should have, at least, 80% of the *ifdefs* with TD=1. Indeed, XTERM has 91% of the *ifdefs* with TD=1. Therefore, it is not an outlier for TD.

[6]XTERM change log is available at http://invisible-island.net/xterm/xterm.log.html



(a) SD (b) TD (c) ND

Figure 4.9: Histogram of Scattering Degree (SD), Tangling Degree (TD), and Nesting Depth (ND) in XTERM

- The threshold for ND states that a system should have, at least, 85% of the *ifdefs* with ND=1. Indeed, XTERM has 89% of the *ifdefs* with ND=1. Therefore, it is not an outlier for ND.

To conclude, the derived thresholds indeed indicate that XTERM has a complex usage of *ifdefs*, which manifests in scattering. Regarding tangling and nesting, XTERM is not different from the systems in our corpus.

4.4 Threats to Validity

A threat to external validity of our conclusions is the selection of the subject systems. We acknowledge that the current selection does not support us to conclude that our findings are applicable to every C-preprocessor-based system. Specially, the proposed thresholds for SD, TD, and ND should be used with caution, as they heavily depend on context, as usual with software metrics [Zhang et al., 2013; Souza and Maia, 2013]. However, we attempted to increase external validity by carefully selecting mature systems of different sizes from different application domains.

The mechanisms in which features are implemented also pose threat to external validity. Since features may be implemented in different ways depending on the programming language, scattering, tangling, and nesting may not have the same behavior as observed in C-preprocessor-based systems.

Different programming styles used by developers to write *ifdefs* may affect the measured degrees, a threat to construct validity. As an example, Listing 4.10 shows two fragments of *ifdef* code, with exactly the same behavior. However, since they have different *ifdef* structures, the measured metric values are different (in Style 1, `FEATURE_A` has SD=1, but in Style 2, `FEATURE_A` has SD=2). We attempt to

```
//Style 1:

#ifdef FEATURE_A
 //code a
#elif FEATURE_B
 //code b
#endif

//Style 2:

#ifdef FEATURE_A
 //code a
#endif
#if !defined(FEATURE_A) && defined (FEATURE_B)
// code b
#endif
```

Figure 4.10: Implementing *ifdefs* with different programming styles

mitigate this threat to validity by analyzing different systems, from different application domains. This way, we are not favouring one style over the other.

Another threat to internal validity arises when computing the three metrics we considered. When using FSCAT, we consider all the C source code of each system, and we do not distinguish files that are automatically generated (e.g., those produced by parser generators) from those that are not. We also do not discard unit test files. Thus, our results are, to some extent, subject to the influence of the file type. We argue, however, that the majority of the files we take for analysis are not automatically generated. We performed a manual inspection on random files, and also on the files containing feature expressions with higher values of tangling and nesting degree to ensure they are automatically generated code. Last, but not least, our results indicate that feature scattering follows a power-law distribution in 14 out of 20 of our subjects. However, it might be the case that other distributions different from power laws are in fact a better fit (e.g., log-normal or stretched exponential). Even if that turns out to be true, conclusions will be the same (i.e., scattering will still be a heavy-tailed distribution).

4.5 Final Remarks

In the empirical study reported in this chapter, we analyzed the statistical distribution of the scattering, tangling, and nesting degrees in 20 open-source C preprocessor-based systems. Our study revealed that feature scattering, as measured by the SD metric, follows a heavy-tailed distribution in all subject systems. In 14 systems (70%), these heavy-tailed distributions matched a power law. Regarding tangling and nesting degrees, the metric values in all systems tend to a uniform distribution, with most values equal to one for both metrics and a few occurrences of slighter higher values. Based on these findings, we proposed thresholds for all three metrics.

All datasets, R scripts, and associated tooling used in this study are publicly available in a website.[7]

[7] http://rodrigoqueiroz.bitbucket.org/sosym2015.html

Chapter 5

Conclusion

We organized this chapter as follows. First, Section 5.1 provides a brief description of the study. Second, Section 5.2 reviews the contributions of our research. Next, Section 5.3 discuss related work. Finally, Section 5.4 suggests further work.

5.1 Overview

In the first part of this master dissertation, we conducted a systematic literature review to search for studies containing assumptions and findings regarding the usage of *ifdefs*. The first goal was to strength our knowledge about *ifdef* usage. Our results show that there are few studies that rely on metrics to reason about preprocessor-based systems. Moreover, the existing studies do not investigate the statistical distributions that better describe preprocessor-based metric values and also do not propose thresholds for such metrics.

In the second part of the study, we performed an analysis of twenty C-preprocessor-based software systems. We extracted and inspected the statistical distribution of three metrics: scattering degree (SD) of feature constants, tangling degree (TD) of feature expressions, and nesting depth (ND) of preprocessor annotations. Our study revealed that feature scattering follows a heavy-tailed distribution in all subject systems, and 14 (70%) of them matched a power-law distribution. Regarding tangling and nesting degrees, the metric values in all systems tend to a uniform distribution, with most values equal to one for both metrics. Based on these findings, we proposed thresholds for all three metrics.

As a practical consequence, our work suggests that scattering, tangling and nesting must be controlled in order to avoid impact on quality and maintainability. However, although high values must be avoided, they should be expected in real software systems. Particularly, high and very high values of scattering can be impracticable to eliminate entirely. The proposed thresholds reveal how preprocessor-based annotations are used in source code and reflect design practices that are common in our subject systems. They can be used to control and monitor the evolution of a system, and raise quality alerts when a system reaches a dangerous level if the proposed thresholds are violated.

5.2 Contributions

The contributions of this master dissertation are as follows:

- The ranking of common assumptions and findings reported in the literature and related to the usage of *ifdefs*.
- An empirical study with long-lived open-source and mature systems that revealed how feature annotations are implemented in practice and the characterization of three metrics considered in the study: Scattering Degree (SD), Tangling Degree (TD), and Nesting Depth (ND).
- A publicly available dataset with three metrics considered in this study and extracted from 20 open-source systems.
- The thresholds proposed for the three metrics considered in the study, which can help practitioners to better understand and evaluate the shape of their feature code.

5.3 Related Work

5.3.1 Metrics for C-preprocessor Annotations

Liebig et al. [2010] analyzed 40 systems written in C showing how developers use the C preprocessor when implementing features and their associated *ifdefs* in source code. The authors consider not only scattering, tangling, and nesting, but also metrics measuring the granularity of annotations (the syntactic location where an *ifdef* occurs—e.g., at a global level, inside a function, and inside a block) and the type of annotated code (homogeneous, meaning that a verbatim copy of the annotated block also appears in other annotated code; heterogeneous, with distinct extensions; or a mix of the two). The authors report their results using centrality and dispersion statistics, including mean and standard deviations. However, the properties of the underlying distributions have not been analyzed (e.g., whether they are symmetric, as in Gaussian distributions, or whether they are heavy-tailed, as in power-law distributions), which may turn results not representative of true typical values.

Hunsen et al. [2015] used the same metrics and tools, including the transformations on the ifdef conditions, as Liebig et al. to compare metric values for open-source and industrial systems. While the authors report the metrics for the individual systems using centrality and dispersion statistics, they used distribution-independent statistical tests (i.e., the Mann-Whitney U test) to check their hypotheses regarding the difference between open-source and closed-source systems.

Couto et al. [2011] extracted a software product line from the ArgoUML tool, using *ifdefs* to annotate feature code. As part of their study, the authors report similar metrics to the ones used by Liebig et al., including the scattering and tangling degree. With the observed scattering values, the authors link the corresponding features to specific patterns reported by Figueiredo et al. [Figueiredo et al., 2009]. These patterns formalize rules on how to identify specific kinds of scattered features.

Eaddy et al. [2008] investigated the relation between scattering and bugs, but do not prescribe a threshold limiting the degree of scattering. Nonetheless, they provide evidence that simple metrics, such as the scattering degree (a.k.a. *concern diffusion metric*), correlate with the number of bugs in a system, independent of its size.

Passos et al. [2015] conducted a longitudinal case study of scattered features in the Linux kernel focusing on driver features. They analyze their evolution by considering scattering thresholds, linking findings to the kernel architectural decomposition, and studying how scattered driver features differ from non-scattered ones.

5.3.2 Characterization of Software Metrics Distribution

Beyond the feature-oriented and product-line communities, there are different pieces of work checking the characteristic distribution of size, coupling, and cohesion-related metrics. Louridas et al. [2008] studied the existence of power-law distributions in different kinds of software components, including Java classes, Perl packages, shared Unix Libraries, and Windows dynamic linked libraries (DLLs). The authors conclude that heavy-tailed distributions, usually power laws, appear at various levels of abstraction, in many domains, operating systems, and languages.

Concas et al. [2007] studied ten different properties related to classes and methods of a large Smalltalk system, consistently finding non-Gaussian distributions of these properties. The authors then conclude that “the usual evaluation of systems based on mean and standard deviation of metrics can be misleading”.

Baxter et al. [2006] reported that some structural properties of Java software follow power-law distributions, while others do not. They conjecture that metrics measuring local properties that programmers are inherently aware about (e.g., out-degree distributions or number of method parameters) tend to follow distributions that are not power laws. In fact, this is the case for tangling (TD) and nesting (ND) considered here.

Taube-Schock et al. [2011] studied connectivity in 97 open-source software systems, and they found that all these systems exhibit a similar scale-free dependency structure, with regard to both the overall connectivity and between-modules connectivity. For this reason, they concluded that high coupling can never be entirely eliminated from software design and that, in fact, some degree of high coupling might be quite reasonable. A similar conclusion appears to apply to scattering in C preprocessor-based systems.

Ferreira et al. [2012] studied the structure of 40 open-source software systems developed in Java, including tools, libraries, and frameworks. They proposed thresholds for six object-oriented software metrics: COF, LCOM, DIT, afferent couplings, number of public methods, and number of public fields. The study concluded that values of those metrics, except DIT, follow a heavy-tailed distribution.

5.4 Future Work

This master dissertation work can be complemented with the following future work:

- We can extend our analysis to a larger set of systems, covering more functional domains, and investigating how a subset of a particular size or specific functional domain affect the proposed thresholds.
- We can assess feature-related metrics in systems written in languages other than C (e.g., in object-oriented languages) or using other preprocessor mechanisms, such as visual annotations [Kästner et al., 2008; Valente et al., 2012].
- We can perform a longitudinal study, investigating the evolution history of scattering, tangling and nesting for each subject system.
- We can explore other techniques for extracting thresholds, such as the one proposed by Alves et al. [2010], or boxplots adjusted to skewed distributions [Hubert and Vandervieren, 2008].

- We can validate the proposed thresholds, by checking whether following (or not following) them has an impact on other software properties, such as bugs, and maintenance effort.

Bibliography

Adams, B., Van Rompaey, B., Gibbs, C., and Coady, Y. (2008). Aspect Mining in the Presence of the C Preprocessor. In *Proceedings of the AOSD Workshop on Linking Aspect Technology and Evolution*, pages 1:1--1:6. ACM.

Alves, T. L., Ypma, C., and Visser, J. (2010). Deriving Metric Thresholds from Benchmark Data. In *Proceedings of the International Conference on Software Maintenance*, pages 1--10. IEEE.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013a). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013b). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag.

Apel, S., Leich, T., and Saake, G. (2008a). Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, 34(2):162–180.

Apel, S., Lengauer, C., Möller, B., and Kästner, C. (2008b). An Algebra for Features and Feature Composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology*. Springer-Verlag.

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2003). Scaling Step-wise Refinement. In *Proceedings of the International Conference on Software Engineering*, pages 187--197. IEEE Computer Society.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. (2006). Understanding the Shape of Java Software. In *Proceedings of the International Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 397--412. ACM.

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. (2007). Lessons from Applying the Systematic Literature Review Process Within the Software Engineering Domain.

Brito e Abreu, F. and Carapuca, R. (1994). Object-Oriented Software Engineering: Measuring and Controlling the Development Process. In *Proceedings of the International Conference of Software Quality*.

Caltech (2010). JPL institutional coding standard for the Java programming language. Technical report.

Chidamber, S. R. and Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476--493.

Clauset, A., Shalizi, C. R., and Newman, M. E. J. (2009). Power-Law Distributions in Empirical Data. *Society for Industrial and Applied Mathematics Review*, 51(4):661--703.

Concas, G., Marchesi, M., Pinna, S., and Serra, N. (2007). Power-Laws in a Large Object-Oriented Software System. *IEEE Transactions on Software Engineering*, 33(10):687--708.

Couto, M., Valente, M., and Figueiredo, E. (2011). Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 191–200. IEEE.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co.

Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. (2008). Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering*, 34(4):497--515.

Favre, J.-M. (1996). Preprocessors from an Abstract Point of View. In *Proceedings of the International Conference on Software Maintenance*, page 329. IEEE.

Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., and Saake, G. (2013). Do Background Colors Improve Program Comprehension in the #Ifdef Hell? *Empirical Software Engineering*, 18(4):699--745.

Fenton, N. E. and Neil, M. (2000). Software Metrics: Roadmap. In *Proceedings of the International Conference on Software Engineering*, pages 357--370. ACM.

Fenton, N. E. and Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 2nd edition.

Ferreira, K. A. M., Bigonha, M. A. S., Bigonha, R. S., Mendes, L. F. O., and Almeida, H. C. (2012). Identifying Thresholds for Object-oriented Software Metrics. *Journal of Systems and Software*, 85(2):244--257.

Figueiredo, E., da Silva, B. C., Sant'Anna, C., Garcia, A. F., Whittle, J., and Nunes, D. J. (2009). Crosscutting Patterns and Design Stability: An Exploratory Analysis. In *Proceedings of the International Conference on Program Comprehension*, pages 138–147. IEEE.

Filó, T. G., Bigonha, M. A., and Ferreira, K. A. (2014). Statistical Dataset on Software Metrics in Object-oriented Systems. *ACM SIGSOFT Software Engineering Notes*, 39(5):1--6.

Garrido, A. and Johnson, R. E. (2013). Embracing the C preprocessor During Refactoring. *Journal of Software: Evolution and Process*, 25:1285--1304.

Gillespie, C. S. (2014a). *Fitting Heavy-Tailed Distributions: The poweRlaw Package.* R package version 0.20.5.

Gillespie, C. S. (2014b). *The poweRlaw Package: A General Overview.*

Gini, C. (1921). Measurement of Inequality of Incomes. *The Economic Journal*, 31(121):124--126.

Hubert, M. and Vandervieren, E. (2008). An Adjusted Boxplot for Skewed Distributions. *Computational Statistics & Data Analysis*, 52(12):5186--5201.

Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., and Apel, S. (2015). Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering*, pages 1–34. To appear.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1--84.

Jbara, A. and Feitelson, D. (2013). Characterization and assessment of the linux configuration complexity. In *International Working Conference on Source Code Analysis and Manipulation*, pages 11–20. IEEE.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report.

Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering*, pages 311--320. ACM.

Kästner, C., Apel, S., and Kuhlemann, M. (2009). A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 157--166. ACM.

Kästner, C., Apel, S., and Ostermann, K. (2011). The Road to Feature Modularity? In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 1--8. ACM.

Kenner, A., Kästner, C., Haase, S., and Leich, T. (2010). TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 25--32. ACM.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242. Springer.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical report.

Krone, M.; Snelting, G. (1994). On the Inference of Configuration Structures from Source Code. In *Proceedings of the International Conference on Software Engineering*, pages 49--57. IEEE.

Lanza, M. and Marinescu, R. (2010). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer Publishing Company, Incorporated, 1st edition.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the International Conference on Software Engineering*, pages 105--114. ACM.

Liebig, J., Kästner, C., and Apel, S. (2011). Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 191--202. ACM.

Louridas, P., Spinellis, D., and Vlachos, V. (2008). Power Laws in Software. *ACM Transactions on Software Engineering and Methodology*, 18:1--26.

Medeiros, F., Ribeiro, M., Gheyi, R., and Fonseca, B. (2014). A catalogue of Refactorings to Remove Incomplete Annotations. *Journal of Universal Computer Science*, 20:746--771.

Newman, M. (2005). Power Laws, Pareto Distributions and Zipf's Law. *Contemporary Physics*, 46:323–351.

Oliveira, P., Lima, F., Valente, M. T., and Alexander, S. (2014a). RTTOOL: A Tool for Extracting Relative Thresholds for Source Code Metrics. In *Proceedings of the International Conference on Software Maintenance and Evolution (Tool Demo Track)*, pages 1--4.

Oliveira, P., Valente, M., and Paim Lima, F. (2014b). Extracting Relative Thresholds for Source Code Metrics. In *Proceedings of the International Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 254–263. IEEE.

Overbey, J. L., Behrang, F., and Hafiz, M. (2014). A Foundation for Refactoring C with Macros. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 75--85. ACM.

Padioleau, Y. (2009). Parsing C/C++ Code Without Pre-processing. In *Proceedings of the International Conference on Compiler Construction*, pages 109--125. Springer-Verlag.

Passos, L., Guo, J., Teixeira, L., Czarnecki, K., Wasowski, A., and Borba, P. (2013). Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *Proceedings of the International Software Product Line Conference*, pages 91--100. ACM.

Passos, L., Padilla, J., Berger, T., Apel, S., Czarnecki, K., and Valente, M. T. (2015). Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *Proceedings of the International Conference on Modularity*, pages 1--12. ACM.

Pearse, T. and Oman, P. (1997). Experiences Developing and Maintaining Software in a Multi-platform Environment. In *Proceedings of the International Conference on Software Maintenance*, pages 270--277. IEEE.

Queiroz, R., Passos, L., Valente, M. T., Apel, S., and Czarnecki, K. (2014). Does Feature Scattering Follow Power-Law Distributions? An Investigation of Five Pre-Processor-Based Systems. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 23--29. ACM.

Queiroz, R., Passos, L., Valente, M. T., Hunsen, C., Apel, S., and Czarnecki, K. (2015). The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. To appear.

Saebjoernsen, A., Jiang, L., Quinlan, D., and Su, Z. (2009). Static Validation of C Preprocessor Macros. In *Proceedings of the International Conference on Automated Software Engineering*, pages 149--160. IEEE Computer Society.

Serebrenik, A. and van den Brand, M. (2010). Theil Index for Aggregation of Software Metrics Values. In *Proceedings of the International Conference on Software Maintenance*, pages 1–9. IEEE.

Smaragdakis, Y. and Batory, D. (2002). Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215--255.

Souza, L. and Maia, M. (2013). Do software Categories Impact Coupling Metrics? In *Proceedings of the Working Conference on Mining Software Repositories*, pages 217--220. IEEE.

Spencer, H. and Collyer, G. (1992). #ifdef Considered Harmful, or Portability Experience with C News. In *Proceedings of the USENIX Summer Technical Conference*, pages 185--197. USENIX Association.

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. (2005). Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 166--175. ACM.

Taube-Schock, C., Walker, R. J., and Witten, I. H. (2011). Can We Avoid High Coupling? In *Proceedings of the European Conference on Object-Oriented Programming*, pages 204–228. Springer.

Valente, M. T., Borges, V., and Passos, L. (2012). A Semi-automatic Approach for Extracting Software Product Lines. *IEEE Transactions on Software Engineering*, 38(4):737--754.

Vasa, R., Lumpe, M., Branchand, P., and Nierstrasz, O. (2009). Comparative Analysis of Evolving Software Systems Using the Gini Coefficient. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188. IEEE.

Vasilescu, B., Serebrenik, A., and van den Brand, M. (2011). You Can't Control the Unfamiliar: A Study on the Relations Between Aggregation Techniques for Software Metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 313--322. IEEE.

Wheeldon, R. and Counsell, S. (2003). Power Law Distributions in Class Relationships. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, pages 45--54. IEEE.

Zave, P. (2003). An experiment in feature engineering. In *Programming Methodology*, pages 353–377. Springer New York.

Zhang, F., Mockus, A., Zou, Y., Khomh, F., and Hassan, A. E. (2013). How does Context affect the Distribution of Software Maintainability Metrics? In *Proceedings of the International Conference on Software Maintainability*, pages 1–10. IEEE.

Appendix A

Search Strings and Filters

In this Appendix, we present the search strings used to select the papers revised in Chapter 3. The strings are adapted to suit specific requirements or limitations of the different libraries. Additional filtering options were used to limit the search by our inclusion criteria.

A.1 ACM Digital Library

The ACM Digital Library is a full-text collection of all articles published by ACM (Association for Computing Machinery) in its articles, magazines and conference proceedings.

Search String: (Title: "c preprocessor" OR Title: "c pre processor" OR Title: "c++ preprocessor" OR Title: "c++ pre processor" OR Title: "preprocessor for c" OR Title: "pre-processor for c" OR (Title: "cpp" AND Title:"preprocessor") OR (Title: "cpp" AND Title:"pre processor") OR Title: "ifdef" OR Abstract: "c preprocessor" OR Abstract: "c pre processor" OR Abstract: "c++ preprocessor" OR Abstract: "c++ pre processor" OR Abstract: "preprocessor for c" OR Abstract: "pre-processor for c" OR (Abstract: "cpp" AND Abstract:"preprocessor") OR (Abstract: "cpp" AND Abstract:"pre processor") OR Abstract: "ifdef") AND (PublishedAs:journal OR PublishedAs:proceeding).

URL: `http://dl.acm.org/`

A.2 IEEEXplore

IEEEXplore is a research database that provides full-text for articles and papers on computer science, electrical engineering and electronics. The database mainly covers material from the Institute of Electrical and Electronics Engineers (IEEE) and the Institution of Engineering and Technology. Since the web tool limits the size of the search string, we performed two searches and merged the results. String A was performed within *Document Titles*, and String B was performed within *Abstracts*.

Search String A: (""Document Title"": ""c preprocessor"" OR ""c pre processor"" OR ""c++ preprocessor"" OR ""c++ pre processor"" OR ""preprocessor for c"" OR ""pre processor for c"" OR ""ifdef"") OR (""Document Title"": ""cpp"" AND ""preprocessor"") OR (""Document Title"": ""cpp"" AND ""pre processor"").

Search String B: (""Abstract"": ""c preprocessor"" OR ""c pre processor"" OR ""c++ preprocessor"" OR ""c++ pre processor"" OR ""preprocessor for c"" OR ""pre processor for c"" OR ""ifdef"") OR (""Abstract"": ""cpp"" AND ""preprocessor"") OR (""Abstract"": ""cpp"" AND ""pre processor"").

Filtering: Limited to publication type (Conference publications, Journals & Magazines).

URL: `http://ieeexplore.ieee.org/`

A.3 ScienceDirect

ScienceDirect is a platform operated by the publisher Elsevier, and provides access to academic journals and e-books full-text.

Search String: TITLE("c preprocessor" OR "c pre processor" OR "c++ preprocessor" OR "c++ pre processor" OR "preprocessor for c" OR "pre processor for c" OR "ifdef" OR ("cpp" AND "preprocessor") OR ("cpp" AND "pre processor")) OR ABSTRACT("c preprocessor" OR "c pre processor" OR "c++ preprocessor" OR "c++ pre processor" OR "preprocessor for c" OR "pre processor for c" OR "ifdef" OR ("cpp" AND "preprocessor") OR ("cpp" AND "pre processor")).

Filtering: Limited to publication type (Journal).

URL: http://www.sciencedirect.com/

A.4 Ei Compendex and Inspec

Ei Compendex is an engineering bibliographic database that indexes scientific literature pertaining to engineering materials. The name "Compendex" stands for COMPuterized ENgineering inDEX. Inspec is an indexing database of scientific and technical literature, published by the Institution of Engineering and Technology (IET), and formerly by the Institution of Electrical Engineers (IEE). The search was performed using Engineering Village web-based discovery platform, which searches both libraries at the same time.

Search String: (("c preprocessor" OR "c pre processor" OR "c++ preprocessor" OR "c++ pre processor" OR "preprocessor for c" OR "pre processor for c" OR "ifdef" OR ("cpp" AND "preprocessor") OR ("cpp" AND "pre processor")) WN TI) OR (("c preprocessor" OR "c pre processor" OR "c++ preprocessor" OR "c++ pre processor" OR "preprocessor for c" OR "pre processor for c" OR "ifdef" OR ("cpp" AND "preprocessor") OR ("cpp" AND "pre processor")) WN AB).

Filtering: Limited to publication type (conference article, journal article), and language (english).

URL: www.engineeringvillage.com

A.5 SpringerLink

SpringerLink is an online collection of peer-reviewed journals and book series published by Springer, covering a variety of topics in the sciences, social sciences, and humanities.

Search String: "c preprocessor" OR "c pre processor" OR "c++ preprocessor" OR "c++ pre processor" OR "preprocessor for c" OR "pre processor for c" OR "ifdef" OR("cpp" AND "preprocessor") OR ("cpp" AND "pre processor").

Filtering: Limited to publication type (article), and language (english).

URL: `http://link.springer.com/`)

A.6 Scopus

Scopus is a bibliographic database containing abstracts and citations for academic journal articles and is owned by Elsevier.

Search String: TITLE-ABS("c preprocessor" OR "c pre processor" OR "c++ preprocessor" OR "c++ pre processor" OR "preprocessor for c" OR "pre processor for c" OR "ifdef") OR TITLE-ABS("cpp" AND "preprocessor") OR TITLE-ABS("cpp" AND "pre processor").

Filtering: Limited to publication type (Conference Paper or Article).

URL: `http://www.scopus.com/`

Appendix B

Selected Studies

In this Appendix, we present the full list of selected papers from the Systematic Literature Review detailed in Chapter 3. Table B.1 lists the year of publication, the title of the article and an acronym for the name of the Journal or Conference. Table B.2 correlates the name of the Journal or Conference with its respective acronym.

Table B.1: Full list of selected studies

ID	Year	Title	Journal/Conference
S1	1990	The C preprocessor and Jensenś device	ACM-SE
S2	1992	#ifdef Considered Harmful, or Portability ExperienceWith C News	USENIX STC
S3	1994	SPP-low tech, practical, UNIX software portability	USENIX UADS
S4	1994	Understanding code containing preprocessor constructs	IWCPC
S5	1995	Design and implementation aspects of an experimental C++ programming environment	SPE
S6	1996	Preprocessors from an abstract point of view	WCRE
S7	1997	A rigorous approach to support the maintenance of large portable software	EUROMICRO
S8	2000	Framework for preprocessor-aware C source code analyses	SPE
S9	2000	Object-oriented preprocessor fit for C++	IEE P-S
S10	2001	Folding: an approach to enable program understanding of preprocessed languages	WCRE
S11	2002	An empirical analysis of c preprocessor use	TSE
S12	2004	Columbus schema for C/C++ preprocessing	CSMR
S13	2005	ASTEC: a new approach to refactoring C	ESEC/FSE
S14	2006	A simple generic library for C	CC
S15	2007	C-CLR: A Tool for Navigating Highly Configurable System Software	ACP4IS

ID	Year	Title	Journal/Conference
S16	2007	How We Manage Portability and Configuration with the C Preprocessor	ICSM
S17	2008	A tale of four kernels	ICSE
S18	2008	Aspect mining in the presence of the C preprocessor	LATE
S19	2008	TBCppA: A Tracer Approach for Automatic Accurate Analysis of C Preprocessor""s Behaviors	SCAM
S20	2009	A model of refactoring physically and virtually separated features	GPCE
S21	2009	Avoiding Some Common Preprocessing Pitfalls with Feature Queries	APSEC
S22	2009	C preprocessor use in numerical tools: an empirical analysis	ACM-SE
S23	2009	How to compare program comprehension in FOSD empirically - An experience report	FOSD
S24	2009	Increasing usability of preprocessing for feature management in product lines with queries	ICSE
S25	2009	Parsing c/c++ code without pre-processing	CC
S26	2009	Refactoring of C/C++ preprocessor constructs at the model level	ICSOFT
S27	2009	Static Validation of C Preprocessor Macros	ASE
S28	2009	Virtual separation of concerns - A second chance for preprocessors	JOT
S29	2010	An analysis of the variability in forty preprocessor-based software product lines	ICSE
S30	2010	Efficient extraction and analysis of preprocessor-based variability	GPCE
S31	2010	Leviathan: SPL support on filesystem level	LNCS
S32	2010	TypeChef: toward type checking #ifdef variability in C	FOSD
S33	2010	Visual Support for Understanding Product Lines	ICPC
S34	2011	#ifdef confirmed harmful: Promoting understandable software variation	VL/HCC
S35	2011	Analyzing the discipline of preprocessor annotations in 30 million lines of C code	AOSD
S36	2011	Analyzing the Effect of Preprocessor Annotations on Code Clones	SCAM
S37	2011	FeatureCommander: colorful #ifdef world	SPLC
S38	2011	Partial preprocessing C code for variability analysis	VaMoS
S39	2012	A Variability-aware Module System	OOPSLA
S40	2012	Characterization of the Linux configuration system	ICPC
S41	2012	Marco: safe, expressive macros for any language	ECOOP
S42	2012	Static flow-sensitive & context-sensitive information-flow analysis for software product lines: position paper	PLAS
S43	2012	The demacrofier	ICSM
S44	2013	Characterization and assessment of the linux configuration complexity	SCAM
S45	2013	Do background colors improve program comprehension in the #ifdef hell?	ESE

ID	**Year**	**Title**	**Journal/Conference**
S46	2013	Does the discipline of preprocessor annotations matter?: a controlled experiment	GPCE
S47	2013	Embracing the C preprocessor during refactoring	JSEP
S48	2013	Investigating preprocessor-based syntax errors	GPCE
S49	2014	A catalogue of refactorings to remove incomplete annotations	JUCS
S50	2014	A Foundation for Refactoring C with Macros	FSE
S51	2014	An approach to safely evolve program families in C	SPLASH
S52	2014	Climate models: challenges for Fortran development tools	SEHPCCSE
S53	2014	Does feature scattering follow power-law distributions?: an investigation of five pre-processor-based systems	FOSD
S54	2014	Projectional editing of variational software	GPCE
S55	2014	Service layer for IDE integration of C/C++ preprocessor related analysis	ICCSA

Table B.2: Conference and Journal Acronyms

Acronym	**Journal/Conference Name**
ACM-SE	ACM Southeast Regional Conference
ACP4IS	Workshop on Aspects, Components, and Patterns for Infrastructure Software
AOSD	International Conference on Aspect-Oriented Software Development
APSEC	Asia-Pacific Software Engineering Conference
ASE	IEEE/ACM International Conference on Automated Software Engineering
CC	International Conference on Compiler Construction
CSMR	European Conference on Software Maintenance and Reengineering
ECOOP	European Conference on Object-Oriented Programming
ESE	Empirical Software Engineering
ESEC/FSE	European Software Engineering Conference/ACM SIGSOFT Symposium on the Foundations of Software Engineering
EUROMICRO	Euromicro Conference on Software Maintenance and Reengineering
FOSD	International Workshop on Feature-Oriented Software Development
FSE	ACM SIGSOFT International Symposium on the Foundations of Software Engineering
GPCE	International Conference on Generative Programming and Component Engineering
ICCSA	International Conference on Computational Science and Its Applications
ICPC	International Conference on Program Comprehension
ICSE	International Conference on Software Engineering
ICSM	International Conference on Software Maintenance
ICSR	International Conference on Software Reuse
ICSOFT	International Conference on Software and Data Technologies
IEE P-S	IEE Proceedings - Software
IWCPC	International Workshop on Program Comprehension
JOT	Journal of Object Technology
JSEP	Journal of software: Evolution and Process
JUCS	Journal of Universal Computer Science
LATE	AOSD Workshop on Linking Aspect Technology and Evolution
OOPSLA	International Conference on Object-Oriented Programming, Systems, Languages & Applications
PLAS	Workshop on Programming Languages and Analysis for Security
SCAM	International Working Conference on Source Code Analysis and Manipulation
SEHPCCSE	International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering
SPE	Software - Practice and Experience
SPLASH	ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity
SPLC	International Software Product Line Conference
TSE	IEEE Transactions on Software Engineering
USENIX STC	USENIX Summer Technical Conference
USENIX UADS	USENIX UNIX Applications Development Symposium
VaMoS	Workshop on Variability Modeling of Software-Intensive Systems
VL/HCC	IEEE Symposium on Visual Languages and Human-Centric Computing
WCRE	Working Conference on Reverse Engineering

Appendix C

Ranking

In this Appendix, we present the full ranking extracted from the studies selected for the Systematic Literature Review detailed in Chapter 3. Table C.1 lists the Ranking of Assumptions and Table C.2 lists the Ranking of Findings.

Table C.1: Ranking of Assumptions. The column Pos indicates the position in our ranking; the column Degree indicates the number of studies reporting the assumption; and the column Studies lists the related studies. The details of each study can be found in Table B.1

Pos	ID	Assumption	Degree	Studies
1	A001	CPP is heavily used to implement variability	27	S06, S07, S09, S15, S17, S18, S21, S23, S24, S25, S28, S29, S30, S32, S33, S34, S38, S39, S41, S42, S45, S46, S47, S48, S49, S52, S55
2	A002	CPP has a negative effect on code readability and comprehension	24	S01, S02, S06, S07, S10, S11, S12, S15, S21, S22, S23, S27, S28, S33, S34, S36, S37, S38, S44, S45, S46, S47, S48, S55
3	A003	CPP impairs maintainability of code	17	S02, S06, S07, S08, S11, S12, S17, S21, S23, S27, S28, S29, S34, S36, S37, S49, S54
4	A004	CPP is error-prone	13	S09, S13, S14, S21, S28, S29, S36, S38, S41, S43, S48, S49, S55
5	A005	CPP is often used to achieve portability	11	S02, S03, S08, S16, S17, S18, S22, S27, S41, S51, S52
6	A006	CPP is simple to use	9	S23, S24, S28, S32, S34, S38, S45, S46, S48
7	A007	ifdef directives are typically scattered across the code base	8	S06, S10, S18, S20, S29, S39, S45, S53
8	A008	CPP directives makes it hard to refactor C code	6	S13, S19, S26, S29, S47, S50

Pos	ID	Assumption	Degree	Studies
9	A009	CPP complicates the analysis of source code	5	S13, S18, S19, S32, S38
9	A010	CPP is a constant problem to tool builders	5	S08, S11, S43, S52, S55
9	A011	CPP provides great flexibility	5	S10, S19, S34, S45, S47
9	A012	In practice most annotations are already in a disciplined form.	5	S28, S20, S32, S46, S49
10	A013	CPP annotations have a negative impact on code quality	4	S28, S36, S49, S55
10	A014	CPP can reduce code reuse	4	S15, S21, S24, S28
10	A015	CPP increases expressiveness of C	4	S08, S10, S28, S45
10	A016	CPP introduces problems to reverse engineering	4	S10, S12, S19, S55
10	A017	CPP is often used to improve performance	4	S02, S18, S22, S31
10	A018	Extensive use of preprocessors can lead to serious maintenance problems	4	S06, S12, S29, S55
10	A019	Fine-grained extensions complicates the use of CPP	4	S18, S21, S45, S46
10	A020	Heavy usage of prepprocessor directives impairs program comprehension	4	S02, S24, S29, S44
10	A021	High degree of feature scattering impairs comprehension	4	S21, S29, S30, S45
10	A022	On average, almost 10% of source code contains preprocessor directives.	4	S12, S18, S19, S55
10	A023	Transition from preprocessors to other languages and tools is hard or impossible	4	S02, S29, S33, S45
11	A024	CPP is useful for development	3	S08, S11, S55
11	A025	High degree of tangling impairs comprehension	3	S21, S24, S29
11	A026	It is possible to refactor conditional compilation into aspects	3	S18, S33, S49
11	A027	There is a lack of adequate tools to support CPP usage	3	S07, S26, S47
11	A028	Undisciplined annotations makes it hard to read and analyze code	3	S35, 46, 49
12	A029	Academics recommend to limit or entirely abandon the use of preprocessors and instead implement SPLs with modular implementation mechanisms	2	S28, S33
12	A030	Annotations increase productivity	2	S02, S20
12	A031	CPP can create portability problems	2	S11, S17
12	A032	CPP conditional compilation makes analysis tasks (e.g., type checking) difficult	2	S32, S52
12	A033	CPP errors are hard to detect	2	S02, S27
12	A034	CPP facilities are a constant problem to programmer	2	S08, S11
12	A035	CPP increases maintenance costs	2	S34, S45
12	A036	CPP leads to code that is difficult to test.	2	S21, S24
12	A037	CPP leads to dead code	2	S06, S30
12	A038	CPP usage leads to tangled code	2	S18, S20
12	A039	Developers prefer disciplined annotations	2	S20,S35
12	A040	Fine-grained usage of CPP hinders tool support for code analysis	2	S32, S46
12	A041	Incomplete annotations aggravates CPP problems.	2	S46, S49
12	A042	Nested #ifdefs impairs comprehension	2	S29, S45
12	A043	Preprocessor directives are easy to use.	2	S28, S45
12	A044	Programs containing CPP directives are difficult to parse	2	S19, S52
12	A045	Scattered features can cause ripple effects	2	S21, S53

Pos	ID	Assumption	Degree	Studies
12	A046	Scattered features cause difficulties in maintenance	2	S44, S53
12	A047	Scattering within certain limits is not necessarily bad	2	S24, S53
12	A048	Some developers follow coding conventions or guidelines for CPP usage	2	S25, S46
12	A049	The issues relating to the CPP arise with virtually all C programs	2	S06, S08
12	A050	The wide use of ifdefs is “harmful” and should be avoided	2	S02, S12
12	A051	Tool support is needed to cope with #ifef	2	S06, S31
12	A052	Undisciplined annotations reduce code replication	2	S46, S49
13	A053	#ifdef doesn’t hide anything, and the interface it creates is arbitrarily complex and almost never documented.	1	S02
13	A054	#ifdef legitimate uses are fairly narrow, and it gets abused almost as badly as the notorious goto statement.	1	S02
13	A055	#ifdef nesting depth with level 5 is not rare	1	S02
13	A056	A high number of fine-grained extensions incur the necessity of modularization techniques	1	S29
13	A057	A legitimate use of #ifdef is in protecting header files against multiple inclusion.	1	S02
13	A058	Accommodating other features into product variants requires fine granular code changes in many components, at many variation points.	1	S21
13	A059	Arbitrary usage of CPP leads to maintainability problems	1	S46
13	A060	As software is ported to more systems, #ifdefs proliferate, nest, and interlock	1	S02
13	A061	C preprocessor is almost never separated from C code	1	S08
13	A062	C preprocessor makes variability implementation difficult	1	S48
13	A063	Code movement refactorings are sensitive to conditional directives	1	S50
13	A064	Conditional compilation is also used for include guards	1	S38
13	A065	Conditional compilation is popular avoiding accidental redefinition of a preprocessor flag	1	S18
13	A066	Conditional compilation takes up half of the preprocessor directives	1	S18
13	A067	Conditionally compiled configurations results in non-explicit representation of configuration compositions.	1	S15
13	A068	Consistent and thoughtful scoping of macro identifiers can greatly ease program comprehension efforts, and reduce maintenance cost.	1	S16
13	A069	CPP can create reliability problems	1	S17
13	A070	CPP hinders structural information from C++ programs.	1	S05
13	A071	CPP is a potential source of suboptimal coding practices.	1	S18
13	A072	CPP is effective	1	S48
13	A073	CPP is infamous for its obtrusive syntax	1	S34
13	A074	CPP is powerful and unstructured	1	S019
13	A075	CPP leads to inconsistencies with respect to the intended (modeled) and the implemented variability of the software.	1	S30
13	A076	CPP provides an intuitive mechanism to implement variability at source code.	1	S30
13	A077	Developers use incomplete annotations to select alternative statements	1	S49
13	A078	Disciplined annotations alleviate the drawbacks of annotations on source-code quality.	1	S36

Pos	ID	Assumption	Degree	Studies
13	A079	Disciplined annotations are sufficient for most problems in software development	1	S35
13	A080	Disciplined annotations limit expressiveness	1	S36
13	A081	Disciplined annotations may require more effort from developers.	1	S28
13	A082	Disciplined use of the preprocessor can improve performance.	1	S11
13	A083	Disciplined use of the preprocessor can improve portability.	1	S11
13	A084	Disciplined use of the preprocessor can improve readability.	1	S11
13	A085	Features that have no relationship in the feature model often interact with each other in the implementation.	1	S30
13	A086	For a virtually separated implementation we could often find an equivalent physically separated one and vice versa	1	S20
13	A087	High numbers of nested #ifdefs are not manageable and increase the potential for errors	1	S29
13	A088	if #ifdef is needed at all, it is best confined to declarations to try to preserve some explicit notion of interfaces.	1	S02
13	A089	if one must use #ifdef, one should test for specific features or characteristics, not for specific machines.	1	S02
13	A090	If the code contains macro occurrences or is enclosed in conditional directives, a local knowledge is not sufficient.	1	S06
13	A091	In general it is always possible to expand undisciplined annotations to disciplined ones	1	S32
13	A092	In nested #ifdef code, maintenance is reduced to hit-or- miss patching	1	S02
13	A093	In nested #ifdef code, not many alternate code paths is tested	1	S02
13	A094	In nested #ifdef code, not many alternate code paths makes sense	1	S02
13	A095	Incomplete annotations have a negative impact on code quality.	1	S49
13	A096	Industrial practice revealed serious problems with preprocessor use	1	S24
13	A097	It is difficult to trace feature-related code.	1	S21
13	A098	It's often possible to avoid system-dependent area well enough that the same code will run on all systems	1	S02
13	A099	Limitations in existing programming languages can lead to scattered features	1	S53
13	A100	Macros pervade the code with 0.28 uses per line	1	S19
13	A101	Modularizing the conditional code will improve program understanding.	1	S18
13	A102	Most of the problems are not specific to CPP and occur with other preprocessors as well	1	S06
13	A103	Nested #ifdef code have more alternate paths to consider	1	S02
13	A104	Occurrences of #include inside #ifdef should always be viewed with suspicion.	1	S02
13	A105	Physical separation of features is better suited for long term development and maintenance.	1	S20
13	A106	Portability is generally the result of advance planning rather than trench warfare involving #ifdef.	1	S02
13	A107	Portability with #ifdefs requires considerable effort	1	S03
13	A108	Preprocessors are commonly used in the domain of embedded and operating systems.	1	S31
13	A109	Preprocessors are frequently criticized for their undisciplined usage	1	S20

Pos	ID	Assumption	Degree	Studies
13	A110	Preprocessors neglect separation of concerns	1	S28
13	A111	Problems emerge when each feature maps to many variation points in many base program components (scattering).	1	S21
13	A112	Problems emerge when the number of inter-dependent features grows (tangling).	1	S21
13	A113	Programmers can visually rely on conventions to easily recognize CPP usages	1	S25
13	A114	Programmers use #ifdef directives when programming security primitives in languages such as C.	1	S42
13	A115	Programming guidelines recommend moderation in the use of preprocessor constructs	1	S17
13	A116	Simple use of #ifdef works acceptably well when differences are localized and only two versions are present	1	S02
13	A117	Some coding guidelines suggest disciplined over undisciplined annotations	1	S35
13	A118	The best method of managing system-specific variants is to define a portable interface to suitably-chosen primitives, and then implement different variants of the primitives for different system	1	S02
13	A119	The effort required to construct "portable" software can discourage authors from distributing their software	1	S03
13	A120	The impulse to use #ifdefs to implement portability is usually a mistake	1	S02
13	A121	The intrinsic problem of CPP is the differences between the code the programmer sees and what the compiler compiles	1	S08
13	A122	The majority of errors occur in incomplete annotations.	1	S49
13	A123	The pitfalls of CPP are now widely known and thus programmers are more conservative in their use of CPP.	1	S25
13	A124	The portability scheme adopted can make it more difficult to port it to other architecture	1	S03
13	A125	Time-pressure can lead to scattered features	1	S53
13	A126	Tools that provides views on variants, completely hiding variability from developers can increase 40% increase in productivity.	1	S33
13	A127	Undisciplined annotations are frequently used	1	S46
13	A128	Undisciplined annotations contribute to unstructured, tangled source code	1	S36
13	A129	Usually, a large software system provides more features than a small software system.	1	S29
13	A130	Virtual separation of feature approaches are common in industry	1	S20
13	A131	Without incomplete annotations, developers can use more tools to analyze the code and find bugs	1	S49

Table C.2: Ranking of Findings. The column Pos indicates the position in our ranking; the column Degree indicates the number of studies reporting the finding; and the column Studies lists the related studies. The details of each study can be found in Table B.1

Pos	**ID**	**Finding**	**Degree**	**Studies**
1	F01	CPP is heavily used to implement variability	6	S11, S22, S29, S38, S44, S52
2	F02	Alternative mechanisms to CPP such as aspects are hard to envision	3	S18, S29, S44
2	F03	CPP introduces errors to code	3	S13, S48, S49
2	F04	Most of the #ifdefs are disciplined.	3	S25, S35, S52
3	F05	CPP annotations complicate program comprehension	2	S44, S46
3	F06	Feature scattering have characteristics of heavy-tailed distributions	2	S44, S53
3	F07	High Feature scattering is concentrated in specific features	2	S44, S53
3	F08	Inconsistencies between feature models and feature implementations in CPP are common	2	S30, S44
3	F09	Nested ifdefs are used moderately	2	S29, S44
3	F10	Portability accounts for a great part of #ifdef usage	2	S11, S32
3	F11	The benefits of disciplined annotations outweigh the drawbacks of code cloning	2	S35, S36
3	F12	Undisciplined annotations are not more problematic than undisciplined annotations	2	S46, S48
4	F13	#undef is rarely used	1	S11
4	F14	A representation of variation based on background coloring can improve understandability over software implemented with the CPP	1	S34
4	F15	A tool for understanding CPP cannot focus on just a subset of directives.	1	S11
4	F16	Alternative macro expansions occur in all analyzed project, but are not very frequent.	1	S38
4	F17	Annotations are able to implement fine grained extensions.	1	S20
4	F18	Both virtual and physical separation of features can express the same programs.	1	S20
4	F19	Climate simulation code with fewer lines of code also contain fewer preprocessing directives	1	S52
4	F20	Configuration options typically provided by CPP directives are most commonly used to: (1) redefine macros for alternative configurations, (2) compose multiple configuration options.	1	S15

Pos	ID	Finding	Degree	Studies
4	F21	CPP behavior looks simple, but not in practice.	1	S19
4	F22	CPP is used in exceptionally broad and diverse ways.	1	S11
4	F23	CPP syntax errors are not common in family releases	1	S48
4	F24	CPP usage complicates the development of C programming support tools.	1	S11
4	F25	CPP-aware program analyses will be applicable to Fortran programs as well.	1	S52
4	F26	Developers are aware of the problems of undisciplined annotations and deliberately limit their use.	1	S35
4	F27	Disciplined annotations bear the potential to improve the situation for tool developers that aim at unprocessed code.	1	S35
4	F28	Disciplined annotations may lead to code clones	1	S36
4	F29	Feature scattering follow a power-law distribution	1	S53
4	F30	Feature-scattering thresholds based on central measures are not reliable in practice.	1	S53
4	F31	Ifdefs expressions are usually simple and have only one config option	1	S44
4	F32	Ifdefs have a minor effect on code cloning	1	S36
4	F33	In the Linux kernel, the situation is not so bad and the code remains largely comprehensible.	1	S44
4	F34	Incomplete annotations can cause memory leaks.	1	S49
4	F35	Many conditional compilation directives would be unnecessary if the CPP had a "define only if not already defined" directive	1	S11
4	F36	Many conditional compilation directives would be unnecessary if the CPP had an #import facility that automatically avoids multiple inclusions	1	S11
4	F37	More systematic SPL implementation techniques are feasible and can improve code quality and reduce error-proneness.	1	S29
4	F38	Most CPP usage follows fairly simple patterns.	1	S11
4	F39	Most extensions occur at a high level of granularity (such as if- statements or for-loops) or are heterogeneous	1	S29
4	F40	Most files have few variations, some have an extremely large number of variations.	1	S44
4	F41	Numerical tools often have a limited need to implement performance optimization using expression macros.	1	S22
4	F42	Programmers use fine-grained extensions infrequently	1	S29
4	F43	Refactoring can introduce variable code that either do not compile or have different behavior.	1	S50
4	F44	Simple extensions of concepts and tools can avoid many pitfalls of preprocessor usage	1	S28
4	F45	Some organizations discourage the use of preprocessor.	1	S17

Pos	ID	Finding	Degree	Studies
4	F46	The average (geometric) scattering degree of config options is relatively high as well as the number of code variations within a file.	1	S44
4	F47	The majority of syntax errors occur because of ill-formed constructions (e.g., missing brackets).	1	S48
4	F48	There are many CPP constants that do not reflect real variability	1	S44
4	F49	There are no differences in program comprehension between Java-CPP and Java-CIDE.	1	S23
4	F50	There is considerable variation in the distribution of directive types in numerical tools	1	S22
4	F51	There is no correlation between the file size and the time developers need to fix the preprocessor-based syntax errors.	1	S48
4	F52	There is no correlation between the time to fix (CPP) syntax errors and the number of developers that commit a file with syntax error.	1	S48
4	F53	There is no difference in CPP usage between numerical tools and general tools	1	S22
4	F54	There is no relationship between the number of features in a software system and the complexity in terms of feature constants.	1	S29
4	F55	Tools that analyze Fortran source code must be able to handle embedded C preprocessor directives.	1	S52
4	F56	Undisciplined systems have more annotated code than disciplined systems.	1	S36
4	F57	Variability management with CPP does not cause excessive code degradation	1	S44
4	F58	Variability of a software system increases with its size	1	S29