An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines

Jörg Liebig, Sven Apel, and Christian Lengauer
University of Passau
{joliebig,apel,lengauer}@fim.uni-passau.de

Christian Kästner and Michael Schulze
University of Magdeburg
{ckaestne,mschulze}@ovgu.de

ABSTRACT

Over 30 years ago, the preprocessor cpp was developed to extend the programming language C by lightweight metaprogramming capabilities. Despite its error-proneness and low abstraction level, the preprocessor is still widely used in present-day software projects to implement variable software. However, not much is known about *how* cpp is employed to implement variability. To address this issue, we have analyzed forty open-source software projects written in C. Specifically, we answer the following questions: How does program size influence variability? How complex are extensions made via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of extension occur? These questions revive earlier discussions on program comprehension and refactoring in the context of the preprocessor. To provide answers, we introduce several metrics measuring the variability, complexity, granularity, and types of extension applied by preprocessor directives. Based on the collected data, we suggest alternative implementation techniques. Our data set is a rich source for rethinking language design and tool support.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

General Terms

Empirical Study

Keywords

Software Product Lines, C Preprocessor

1. INTRODUCTION

The C preprocessor (cpp) is a popular tool for implementing variable software. It has been developed to enhance C by lightweight metaprogramming capabilities and is commonly used to merge files, make arbitrary textual substitutions, and define conditional code fragments (a.k.a. conditional inclusion) [21]. As the cpp tool is line-based, it can be used with any text artifact including other programming languages such as Java or C#. In the past, it has been observed that the use of cpp causes various problems: (1) the occurrence of syntactic and semantic errors during the generation of software products [23]; (2) code pollution due to scattered and tangled #ifdefs (a.k.a. #ifdef hell) [32]; (3) a decrease in maintainability and in ability to evolve [13].

The implementation of variable software is also a major goal of software product line engineering. A *software product line* (*SPL*) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that have been developed from a common set of core assets in a prescribed way [8]. *Software product line engineering* aims at managing variability among the different software products (a.k.a. *variants*) of a product line and facilitating reuse in that the products of the product line share as many common core assets as possible, such as source code artifacts [8, 11, 27].

It is widely assumed that the variability mechanism of the cpp tool is used quite frequently in the implementation of SPLs [1, 12, 18, 33]. We believe this assumption to be true and contribute to the discussions on the connection between preprocessors and SPLs started in prior work with a substantial set of case studies. We answer the following questions: How does program size influence variability of SPLs? How complex are the extensions applied by features via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of extension occur? We argue that insights into the implementation problems of features solved with cpp help to judge whether cpp usage causes problems with regard to code pollution, error-proneness, and reduced maintainability and ability to evolve. An analysis of cpp usage also allows developers to estimate the effort and benefits of migrating to other, more well-founded implementation techniques for SPL engineering, such as aspects [17, 24] or various flavors of feature modules [5, 6].

To answer the questions above, we analyzed forty open-source software systems of different sizes (thousands to millions lines of code) taken from different domains including operating systems, database systems, and compilers. We propose a set of metrics that allow us to infer and classify cpp usage patterns and to map the patterns to common SPL implementation concepts. Our analysis reveals that cpp is used to a large extent to implement and control variability

ICSE '10, May 2-8 2010, Cape Town, South Africa

in SPL code bases (on the average 23 % of each case study's code base is variable). In general, the preprocessor can be used at every level of granularity, but we found that most extensions occur at a high level of granularity (e.g., adding a whole function). The patterns of cpp usage, which we have identified, indicate that alternative, more systematic SPL implementation techniques are feasible and can improve code quality and reduce error-proneness. The degree of detail of the usage patterns even enables us to suggest specific techniques (e.g., aspects and feature modules) that match the properties of the implementation problems found in the software projects analyzed.

Our greater goal is to raise the awareness of the problems of implementing variability with ad-hoc mechanisms like preprocessor directives and to initiate a discussion on the feasibility of well-founded SPL implementation techniques.

To summarize, we make the following contributions:

- We discuss patterns for implementing variability using cpp.
- We propose a set of metrics for identifying and classifying cpp usage patterns that map to well-known concepts in SPL engineering.
- We present data collected with our tool suite cppstats on forty open-source software systems of different sizes and from different domains.
- Based on the data obtained, we analyze correlations between the size of a software system and the variability issues stated earlier.
- We discuss the feasibility and benefits of using alternative SPL implementation techniques.

2. BACKGROUND

Before we begin with the variability analysis and the evaluation of the collected data, we introduce SPLs, the preprocessor cpp, and cpp's role in SPL development.

2.1 Software Product Lines

An important concept in SPL engineering is that of a feature. A *feature* represents an optional or incremental unit of functionality [6, 11, 16]. Different programs, called *variants*, can be generated by selecting different features. For example, features can be optional or can form a group of alternative features.

Usually, a feature's implementation extends a program in one or more places, called *extension points*. If a feature extends multiple points and is not modular, its code is scattered across the software system (*code scattering*). The code of features is often tangled with the base code and possibly with code of other features (*code tangling*). Extensions made by a feature can be classified into homogeneous and heterogeneous extensions [5, 10]. A *homogeneous extension* adds the same piece of code at different extension points and a *heterogeneous extension* adds different pieces of code at different extension points. Previous studies indicate that scattered and tangled feature code as well as homogeneous and heterogeneous extensions occur frequently in SPLs [4, 9, 17].

2.2 The C Preprocessor (cpp)

The preprocessor cpp is a stand-alone tool for text processing, which enhances C by lightweight metaprogramming capabilities [21]. Although initially invented for C, the preprocessor is not limited to a specific language and can be used for arbitrary text and source code transformations [13].[1] The cpp tool works on the basis of directives (a.k.a. macros), that control syntactic program transformations.

The directives supported by the cpp tool can be divided into four classes: file inclusion, macro definition, macro substitution, and conditional inclusion. In cpp-based SPL development, macro definitions (#define) and conditional inclusions (#ifdef[2]) are most important.

2.3 SPL Development with cpp

In order to illustrate how to implement variability with cpp, we use a simple example of a list data structure with different features, such as SORTALGO (sorting algorithms), SORTORDER (ascending or descending order), and DLINKED (doubly linked list), shown in Figure 1. Common source code of the SPL is represented, using the capabilities of the programming language C, in terms of data abstraction (data types; e.g., struct T_node on Line 11) and procedural abstraction (functions; e.g., insert on Line 24). To implement variable source code using the cpp tool, two things are necessary: (1) the definition of a *feature constant* that can be referred to in the source code and (2) the inclusion of additional source code representing the incremental functionality (*feature code*).

A programmer uses the macro #define to set a feature constant (e.g., DLINKED on Line 4). Feature constants can also be defined externally in makefiles, in configuration files, or during the compiler invocation. The cpp tool provides logical operators (e.g., &&) and bit operators (e.g., &) to combine multiple feature constants to complex *feature expressions*.[3] A feature expression represents the condition that controls the inclusion or exclusion of feature code. That is, based on the evaluation of a feature expression, all subsequent lines of source code up to the next #ifdef are included or excluded depending on whether the expression evaluates to true or false (e.g., Line 15 is included if feature DLINKED is selected). We call the use of #ifdef also *source code annotation*. Based on the definition of feature constants, the programmer is able to influence the evaluation of the feature expression and, consequently, the presence or absence of feature code. Furthermore, #ifdef macros can be nested, and the evaluation of a nested #ifdef depends on the evaluation of the enclosing #ifdefs.

Technically, every source code fragment that is enclosed by #ifdef is an optional feature. The specification of alternative features relies either on (1) multiple #ifdefs (e.g., feature SORTALGO on Line 5 with the mutually exclusive options NOSORT, INSERTIONSORT, and BUBBLESORT) or (2) the #if-#elif-#else combination for specifying alternative source code (e.g., feature SORTORDER on Lines 40, 42, and 44).

The code which implements a feature is often scattered across the SPL's code base. Examples are the implementations of the features DLINKED (e.g., Line 14 and 20) and SORTALGO (e.g., Line 25, 29, 36, and 46). Since the introduction of their feature constants and the annotation of their feature code are simple, both features are easy to implement. Source code tangling arises from the mix of source code of

[1] cpp works on lines of text and is oblivious to the underlying language. cpp directives may break existing tool support for languages such as Java or C# but can be used nonetheless for conditional compilation.

[2] For simplicity, we refer to the various conditional inclusion macros, such as #ifdef, #ifndef, and #if, summarily as #ifdef.

[3] Beside feature constants, feature expressions may also contain numbers.

```
1  #define NOSORT 0
2  #define INSERTIONSORT 1
3  #define BUBBLESORT 2
4  #define DLINKED 1
5  #define SORTALGO BUBBLESORT
6  #if SORTALGO != NOSORT
7  #define SORTORDER 1
8  #endif
9  ...
10
11 typedef struct T_node {
12   int item;
13   struct T_node *next;
14 #if DLINKED
15   struct T_node *prev;
16 #endif
17 } node;
18
19 node *first = NULL;
20 #if DLINKED
21 node *last = NULL;
22 #endif
23
24 void insert(node *elem) {
25 #if SORTALGO == BUBBLESORT || SORTALGO == INSERTIONSORT
26   node *a = NULL;
27   node *b = NULL;
28 #endif
29 #if SORTALGO == BUBBLESORT
30   node *c = NULL;
31   node *e = NULL;
32   node *tmp = NULL;
33 #endif
34   if (NULL == first) first = elem;
35   else {
36 #if SORTALGO == INSERTIONSORT
37     a = first;
38     b = first->next;
39     if (first->item
40 #if SORTORDER == 0
41       >
42 #else
43       <
44 #endif
45 ...
46 #if SORTALGO == BUBBLESORT
47 ...
48 #endif
49 }
```

Figure 1: A variable list implementation with cpp

several features at one extension point. A programmer specifies this mixture using the operator && or nested #ifdefs. One example of tangling is feature SORTORDER on Line 40 that tangles with feature SORTORDER from Line 36.

3. METHODOLOGY

Before we present and discuss the results of our analysis, let us have a closer look at the questions stated in the introduction regarding variability. Let us explain why these questions are important and describe how we obtain answers to them.

3.1 Research Questions

Our analysis lays the ground for answering a wide spectrum of questions regarding several areas in SPL engineering. We concentrate on four questions covering two areas: (1) program comprehension and (2) refactoring. The first area contributes to discussions on program understanding, whereas the latter refers to the applicability of alternative SPL implementation techniques. Prior case studies on the preprocessor with respect to comprehension and refactoring either focused on cpp's variability mechanisms at a theoretical level [18, 19] or aimed already at refactorings of individual applications [1]. In contrast to these case studies, we are interested in the general picture of the practical use of cpp's variability mechanisms.

Comprehension

1. How does program size influence variability? Usually, a large software system provides more features than a small software system. A large code base increases the potential of variability. We are interested in how many feature constants occur in the source code because they mark possible configuration parameters and define the configuration space of an SPL. Furthermore, we are interested in the amount of feature code because it represents variability at the source code level at which the programmer operates. A high variability increases significantly the chances that the problems stated previously (e.g., syntactic and semantic errors or code pollution) occur. We argue that ad-hoc variability mechanisms are only manageable up to a certain scale.

2. How complex are extensions made via cpp's variability mechanisms? This question addresses the presence of scattered and tangled feature code. We are interested in whether a higher number of features increases the degrees of scattering and tangling. Furthermore, we are interested in the number of nested #ifdefs, a special case of source code tangling. It is reasonable to expect that a high degree of scattering, tangling, and nested #ifdefs impair comprehension.

Refactoring

3. At which level of granularity are extensions applied? This question is motivated directly by prior discussions on the granularity of extensions in SPLs [18, 19]. These discussions address the necessity of modularization techniques applied at a fine grain, such as statement and expression extensions or function signature changes. Although the cpp tool allows a programmer to annotate code even at the finest level of granularity [18], not much is known about the necessity to make such fine-grained extensions. Further discussions on the modularization of features also motivate this question [17, 26, 34], because most modular SPL implementation techniques either lack the ability to make fine-grained extensions [18, 26] or require workarounds [17, 30]. To this end, we are interested in the level of and extent to which features cause fine-grained extensions. A high number of fine-grained extensions incur the necessity of modularization techniques, whereas a small number may not.

4. Which types of extension occur? This question targets the strengths and weaknesses of alternative SPL implementation techniques. For example, homogeneous extensions can be implemented easily with aspect-oriented language extensions of C because they provide a quantification mechanism for extending multiple places in the source code at a time. Heterogeneous extensions can be specified by simpler mechanisms such as mixins or feature modules [5, 6]. Furthermore, this question is motivated by a prior case study on the use of AspectJ [4], in which it has been observed that most extensions in AspectJ source code are heterogeneous; we want to find out, whether this observation also applies to cpp-based SPL implementations.

3.2 Metrics

We cannot measure cpp usage in terms of comprehension and refactoring directly. Hence, we introduce a set of metrics

that represent these objectives. We measure each metric after normalizing the source code of each software system (i.e., removing comments and so on). Next, we explain each metric, how we measure it, and why it is useful for our evaluation and for subsequent discussions.

Lines of Code (LOC). The LOC metric represents the size of a software system. We measure it by counting the number of newlines of every normalized source code file and use it later to discuss the influence of program size on the remaining metrics.

Comprehension

Number of Features Constants (NOFC). The NOFC metric reflects directly the configuration dimension of an SPL and, to this end, provides insights into the variability and complexity of the SPL. We measure this metric by extracting feature constants from feature expressions in the source code and sum them per project. Our list SPL (Figure 1) contains six feature constants (INSERTIONSORT, BUBBLESORT, DLINKED, SORTALGO, and SORTORDER).[4]

Lines of Feature Code (LOF). The LOF metric is the number lines of feature code that are linked to feature expressions. It tells us whether a small or a large fraction of the code base is variable. We extract this metric by counting the number of lines between two #ifdefs in source code files and sum them per project.[5]

Scattering Degree (SD) and Tangling Degree (TD). The SD metric is the number of the occurrences of feature constants in different feature expressions. We measure this metric by extracting feature constants from feature expressions and calculate the average and standard deviation per project of all occuring feature constants. This metrics tells us about the complexity of feature implementations. A widely scattered feature that extends a software system in several files and at multiple extension points is more complex (e.g., for maintenance tasks) than a feature that makes only a few extensions in a single file.

The TD metric is the number of different feature constants that occur in a feature expression. A low TD is preferable, because a high number of tangled feature constants in feature expressions may impair program comprehension.[6]

Average Nesting Depth of #ifdefs (AND). The AND metric reflects the average nesting depth of #ifdefs. We calculate the average and the standard deviation of all #ifdefs in a file and compute, based on these values, the average and standard deviation for a project. Since nested #ifdefs form feature expressions, this metric is useful for discussions on program comprehension.

[4]We do not count all macros as feature constants. For example, feature NOSORT is not a feature constant in our example because it is not used by an #ifdef.

[5]We omit lines of code that are enclosed by include guards. An include guard is a common preprocessor pattern that frames the entire content of a file with #ifdef to avoid duplicate definitions due to the multiple inclusion of files [1]. It does not represent an increment in functionality.

[6]The term tangling has a non-standard meaning here. Usually, tangling refers to the mixture of several features with each other (side by side) and/or with the base code.

Refactoring

Granularity (GRAN). Since cpp can be used with different host languages, arbitrary changes such as coarse- and fine-grained extensions are possible. Coarse-grained extensions add new functions or data structures, whereas fine-grained extensions add source code pieces, such as statement and expression extensions or function signature changes [18]. To this end, we introduce the GRAN metric, which is the number of #ifdefs that occur at particular levels in the source code.

Based on prior work [18] and on the capabilities of alternative SPL implementation techniques, we measure the GRAN metric at six granularity levels of interest: the global level (GL; e.g., adding a structure or function; Figure 1, Line 20), function or type level (FL; e.g., adding an if-block or statement inside a function or a field to a structure; Figure 1, Line 25), block level (BL; e.g., adding a block; Figure 1, Line 36), statement level (SL; e.g. varying the type of a local variable), expression level (EL; e.g., changing an expression; Figure 1, Line 40), or function signature level (ML; e.g., adding a parameter to a function). The metric provides insight into the granularity of cpp-based SPLs and is used in the discussion section to evaluate alternative SPL implementation techniques. We measure the metric by counting the number of occurrences of #ifdefs at each GRAN level and sum them up for each project.

Type (TYPE). The programmer labels several parts in the source code as feature code either with distinct extensions (heterogeneous) or with the same extension using code duplicates (homogeneous). The TYPE metric is the number of occurrences of particular extensions in the source code. We distinguish three types, homogeneous extension (HOM), heterogeneous extension (HET), and their combination (HEHO) by comparing subsequent lines of source code that belong to the same feature expression using exact string comparison; we discuss this threat to validity later). We use this metric to discuss possible refactorings.

4. ANALYSIS

We analyzed forty different open-source software systems written in C. We limited our analysis to C, because it is used widely in software development and the range of public available open-source projects varies from small ($\sim$10 KLOC) to very large ($>$1,000 KLOC). This section describes the selected software systems, the setup of our analysis, and the collected data.

We list the selected software systems in Table 1. Our selection covers a variety of different domains, such as operating systems and application software, to give as complete an overview of cpp usage as possible. We consider these software systems SPLs, because all of them contain several optional and alternative features, such as support for different platforms and application-specific configuration options.

To make the data of the different software systems comparable, we applied first some syntactic source code adjustments to the system's code base: we deleted blank lines and comments and formatted the code uniformly. Furthermore, we used the tool src2srcml[7] to generate an XML representation of the source code [25] for measuring the granularity of extensions made with cpp. The XML representation has all information of the basic language C in the form of an *abstract syntax tree* (*AST*) with additional information on the pre-

[7]`http://www.sdml.info/projects/srcml/`

software system	version	domain
apache[1]	2.2.11	Web server
berkeley db[1]	4.7.25	database system
cherokee[1]	0.99.11	Web server
clamav[1]	0.94.2	antivirus program
dia[1]	0.96.1	diagramming software
emacs[1]	22.3	text editor
freebsd[1]	7.1	operating system
gcc[1]	4.3.3	compiler framework
ghostscript[1]	8.62.0	postscript interpreter
gimp[1]	2.6.4	graphics editor
glibc[1]	2.9	programming library
gnumeric[1]	1.9.5	spreadsheet appl.
gnuplot[1]	4.2.5	plotting tool
irssi[1]	0.8.13	IRC client
libxml 2[1]	2.7.3	XML library
lighttpd[1]	1.4.22	Web server
linux[1]	2.6.28.7	operating system
lynx[1]	2.8.6	Web browser
minix[1]	3.1.1	operating system
mplayer[1]	1.0rc2	media player
mpsolve[2]	2.2	mathematical software
openldap[1]	2.4.16	LDAP directory service
opensolaris[3]	(2009-05-08)	operating system
openvpn[1]	2.0.9	security application
parrot[1]	0.9.1	virtual machine
php[1]	5.2.8	program interpreter
pidgin[1]	2.4.0	instant messenger
postgresql[1]	(2009-05-08)	database system
privoxy[1]	3.0.12	proxy server
python[1]	2.6.1	program interpreter
sendmail[1]	8.14.2	mail transfer agent
sqlite[1]	3.6.10	database system
subversion[1]	1.5.1	revision control system
sylpheed[1]	2.6.0	e-mail client
tcl[1]	8.5.7	program interpreter
vim[1]	7.2	text editor
xfig[1]	3.2.5	vector graphics editor
xine-lib[1]	1.1.16.2	media library
xorg-server[4]	1.5.1	X server
xterm[1]	2.4.3	terminal emulator

[1] `http://freshmeat.net/`; [2] `http://www.dm.unipi.it/cluster-pages/mpsolve/`; [3] `http://opensolaris.org/os/`; [4] `http://x.org/`; Development versions of software systems are marked with the date of download in brackets.

Table 1: Analyzed software systems

processor statements. The two levels of programming (the metalevel of cpp and the source code level of C) have separate namespaces in XML, which gave us the opportunity to conduct a coherent and separate analysis of the source code.

We measured the metrics introduced in the Section 3.2 on the XML representation. Basically, the analysis rests on the traversal of the XML-annotated source code using our self-written tool cppstats. This tool and the comprehensive data for each system are available at our project's Web site.[8] Table 2 on page 10 depicts the condensed data of all forty software systems analyzed with our tool cppstats.

5. INTERPRETATION & DISCUSSION

The collected data provide answers to various research questions. Here, we focus on program comprehension and refactoring). During the refactoring discussion, we concentrate on two alternative SPL implementation techniques: (1) aspects [22] and (2) feature modules [5]. We limit our discussion to these two because, in our view, they have been receiving most attention from the academic community regarding implementation of variability in recent years. Subsequent to the data interpretation and refactoring discussion, we discuss possible threats to validity.

The percentages given in this section are the average and the standard deviation (a±s). All plots illustrate one of the metrics LOC and NOFC compared with some other metric. Additionally, we calculated the correlation coefficient between the metrics being compared using the method of Kendal [20], because all input data are not normally distributed.

Comprehension

1. How does program size influence variability? The data reveal that the variability of a software system increases with its size (Figure 2 a). This is confirmed by the correlations between the metrics LOC and NOFC as well as LOC and LOF, which correlate highly. We can explain this correlation with the observation that larger software systems usually exhibit more configuration parameters and, consequently, are more variable. The amount of variable source code (LOF metric) in each project correlates with its size and is on the average 23±17 % (Figure 2 b).

The LOF metric reveals two interesting issues. First, we found that, in some mid-size software systems such as libxml2, openvpn, sqlite, and vim, the amount of feature code exceeds 50 % of the code base. Second, the four largest software systems (freebsd, gcc, linux, and opensolaris) contain a smaller percentage of variable source code compared to the average. A reason for both issues may be that the specification of configurable features is more complex in larger systems than in smaller ones. The higher complexity aligns with possibly more scattered and tangled features, a correlation which we address next.

2. How complex are extensions via cpp's variability mechanisms? The complexity of cpp-based SPL implementations increases with the increasing use of feature constants in feature expressions and of #ifdef nesting (SD, TD, and AND metric). We observed that the size of a software system either correlates only at a very low level (SD and TD metric) or does not correlate with the number of features (AND metric). All data points are widely scattered and each correlation coefficient is close to zero. That is, we argue that there is no relationship between the number of features in a software system and the complexity in terms of feature constants. Initially, we expected that, in software systems with a high number of feature constants, the complexity of feature expressions (more feature constants are involved) to be higher than in smaller software systems, but the complexity stays the same.

Notably, the standard deviation of the scattering degree is in most systems quite high (e.g., emacs, freebsd, lynx, and python). That is, a significant number of feature constants incur a high scattering degree and the respective implementation scatters possibly across the entire system. However, we cannot infer from the scattering degree the places at which feature constants occur in the source code. The scattered feature constant may only appear in a subsystem (e.g., a group of files). In the future, this should be investigated in more detail.

The mean and the standard deviation of the tangling degree are quite small in most systems (1 to 3 on average) and,

[8] `http://fosd.de/cppstats/`

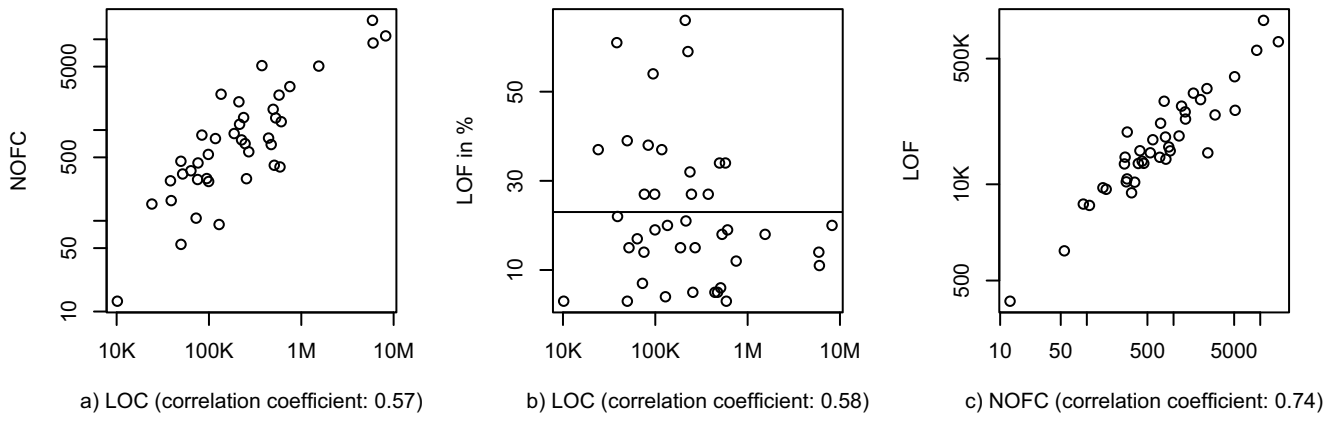

Figure 2: a) plot LOC/NOFC; b) plot LOC/LOF with the average of variable source code in all software systems; c) plot NOFC/LOF

consequently, the complexity of feature expressions is low. A lower complexity is preferable, because feature expressions that consist of a high number of feature constants impair program comprehension.

In addition to the scattering and tangling degree, we measured the average depth of nested #ifdefs (Figure 3 c; AND metric). Notably, in all software systems, the average AND is approx. 1, which means that nesting is used moderately (i.e., the number of nested #ifdefs does not grow with the number of feature constants – NOFC metric). A lower AND is preferable, because the programmer has to be aware of outer #ifdefs when reasoning about inner code. We also determined the maximum number of nested #ifdefs in a file. Two projects (freebsd and gcc) reached a maximum number of 24. The rest of the systems remained at 2 to 9. We argue that high numbers of nested #ifdefs are not manageable, impair program comprehension, and increase the potential for errors. Furthermore, a high AND may reduce the potential for refactorings. Since a nested #ifdef depends on the enclosing one, the dependency between both #ifdefs has to be taken into consideration when a refactoring should be applied.

Refactoring

3. At which level of granularity are extensions applied? The data reveal that programmers use fine-grained extensions (e.g., statement or expression extensions) infrequently.[9] The overall occurrence of these extensions is 1.8±1.8 % on the average. Two projects use them slightly more frequently: (1) lighttpd with 6 % statement extensions and (2) vim with 6 % expression extensions. Although specific modularization techniques have been proposed for implementing fine-grained extensions [30, 14, 26], their usefulness seems to be limited.

Most extensions occur at the global level (GL metric; 46±12 %): enclosing functions, type declarations/definitions, or (re-)definitions of feature constants. These extensions can be realized by SPL implementation techniques, such as aspects or feature modules, in which the introduction of functions or types is supported [6].

Beneath the global level the second largest set of extensions occur at the function and block level (FL and BL metric; 33±9 % and 19±7 %): enclosings (e.g., an if-block or a statement) inside a function. Both extension types are harder to apply, because extensions can appear at every point of a function, and not all proposed techniques provide particular patterns for matching them. The most promising SPL implementation technique is the aspect, which enables extensions beneath the function or type level by addressing particular extension points. However, the data do not reveal whether an implementation technique is applicable and workarounds for refactorings using either feature modules or aspects may be necessary [17, 29].

4. Which types of the extension occur? Our data reveal that 89±6 % of the extensions are heterogeneous (HET metric). Homogeneous extensions (HOM metric) add up to 5±5 % and the combination of both extension types (HEHO metric) makes up to 4±2 %. Aspects are well known for their ability to implement homogeneous extensions [5]. We observed that 5 % of the extensions would benefit from aspects; 89 % would suffice with simpler mechanisms, such as mixins or feature modules; for the rest of the extensions a combination of aspects and feature modules would be profitable [5].

The data coincide with an analysis conducted by Apel [4], which revealed that most extensions are heterogeneous. Apel analyzed the use of AspectJ rather than cpp, but the results are similar.

Threats to Validity

Limitation to a single language. Programming languages provide different mechanisms for the implementation of SPLs. For example, C++ provides template metaprogramming, which can replace most cpp macros for SPL implementations [11, 31]. We limit our analysis to C to make the results of the analyzed software systems comparable.

[9] Less than 1 % of the extensions did not match the patterns of our GRAN metric.

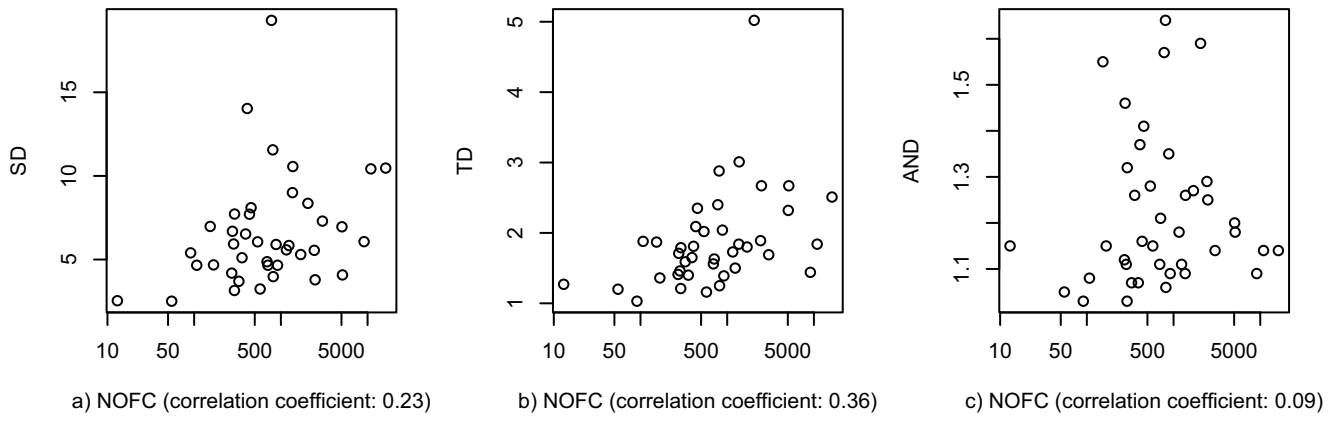

Figure 3: a) plot NOFC/SD; b) plot NOFC/TD; c) plot NOFC/AND

Selection of the software systems. A major problem with case studies is the selection of the objects of study because a biased selection can render the results useless. We are aware of this problem; to minimize it, we selected a large number of software systems of different domains for our analysis.
Source code. Different coding conventions used in software systems may lead to wrong conclusions. For this reason, we preprocessed the analyzed software systems by eliminating comments, empty lines, and include guards, and by applying source code pretty printing.
Feature detection. Our analysis is limited to the source code and to #ifdefs. But the representation, selection, and implementation of features is not limited to #ifdefs. Additional configuration layers, like configuration scripts or tools are also used in SPL engineering. A comprehensive analysis of these configuration layers is out of scope. Our focus on the source code coincides with the programmer's point of view in feature implementation.

We expect the amount of configurable source code to increase when we include the additional configuration layers. This traces back to the selection of files, modules, etc.
High-level and low-level features. An automated analysis of the source code cannot distinguish well between high- and low-level features. High-level features represent requirements of stakeholders, whereas low-level features reflect design decisions made by programmers (e.g., tracing or portability issues like different types of signed integers for different compilers or platforms), which are not of interest to most stakeholders [3]. Making this distinction is not possible without additional expertise regarding the software systems and the domains. Here, we are not interested in it, because we are only looking at the usage of cpp's variability mechanisms at the implementation level.
Feature expression equality. We use string comparison to check the equality of different feature expressions to determine which code fragments belong to the same expression. Our analysis misses the semantic equivalence of feature expressions like A && B and B && A. However, we found that these occur rarely in the SPLs analyzed. In a random inspection of 12 smaller SPLs, we found the error for not considering the equivalence to be below 2.5 %.[10]
Heterogeneous and homogeneous extensions. The classification of extensions into heterogeneous and homogeneous is common in the software engineering literature [5, 9]. Our tool distinguishes heterogeneous from homogeneous source code fragments by string comparison. Thus, character-based, syntactic changes in the source code are not classified correctly. The additional information we gather on the AST does not help here, because semantically equivalent source code fragments can differ in syntax and are thus not recognized by a comparison of subtrees of the AST. The problem is even more serious, because arbitrary extensions that destroy the source code structure are possible with the preprocessor. Generally, the problem is related to the determination of code clones [7]. However, to our knowledge, there is no code clone detection tool capable of comparing arbitrary source code fragments for equality. A combination of our analysis tool with code clone detection tools may lead to more precise results. We expect a higher number of homogeneous extensions. Our measurement marks a lower bound of homogeneous extensions.
Mapping of #ifdefs to AST elements. For creating an AST and mapping #ifdefs to AST elements, we rely on the tool src2srcml. Especially, the GRAN and TYPE metrics rely heavily on the correctness of this mapping. The authors of src2srcml use an extensive test suite to verify the relation of #ifdefs and source code in the XML representation.

6. PERSPECTIVE

Our analysis provides a substantial amount of data that is valuable for further research on language design and tool support. The data are (1) a basis for discussions on the feasibility of implementation techniques for implementing

[10]The problem arises from checking the equality of predicates, a problem which is in the class of NP problems. We used the tool Maple (`http://maplesoft.com/`) to check the equivalence of feature expressions in some smaller SPL. The equality check takes hours even for a small software system, and the equivalence check of a large software system, such as the Linux kernel, would take an estimated time of 13 years.

SPLs and (2) valuable input for language designers and tool writers. Nevertheless, while performing the analysis, several issues came up which we plan to address in further research.

Although we selected a large number of software systems covering a variety of domains, we cannot infer valuable information on cpp usage regarding a specific domain, because the number of systems for each domain is too small. However, we believe that the usage of cpp's variability mechanisms is not just a question of program size, but also of the domain. For example, the variability of operating systems (e.g., driver implementations) to support different hardware platforms may be different than in a Web server. Our results reveal that the Web servers analyzed are more variable in terms of configuration parameters (NOFC metric) and source code (LOF metric) than the operating systems analyzed, but the sample is too small to draw a general conclusion. We plan to investigate particular domains in isolation in terms of comprehension and refactoring and to look for differences and similarities to the case study presented here.

Our analysis covers only one specific release of each software system. We believe that gathering data of a software system over time reveals interesting insights into its adaptation and evolution. These may involve the support of a new platform, a new functionality, or the adaptation to structural changes. To this end, we plan to apply cppstats also to consecutive versions of a software system to contribute to the findings of previous work [1, 28].

Finally, based on the granularity and homogeneous/heterogeneous metrics, we plan to explore the possibility of *automated refactorings* of #ifdefs. To this end, we look for patterns of cpp-based extensions and map them to alternative SPL implementation techniques. Our goal is to provide tool support to programmers with suggestions on possible refactorings in legacy applications [1].

7. RELATED WORK

The cpp tool has been the subject of several papers in the past. We group them according to program comprehension and refactoring and discuss the relation to our work.

Comprehension

The most comprehensive analysis of the cpp tool was conducted by Ernst et al. [12]. The authors presented results of an analysis covering mainly the facilities and possible pitfalls of macro expansion. We complement their case study with detailed information about variability implementation.

Krone and Snelting proposed a tool that extracts #ifdefs from software systems to compute the configuration structure [23]. The tool covers only the coupling of feature constants; it neglects the implementation part (feature code) in terms of homogeneity and granularity.

Favre covered cpp usage in software development from a general point of view without looking at it empirically [13].

Refactoring

Adams et al. proposed a set of patterns for refactorings of #ifdefs into aspects [1]. The authors tested their approach on one software system. Other researchers also addressed the use of aspects for refactoring [2]. Our metrics enable a discussion of different SPL implementation techniques for refactorings. Furthermore, we provide data obtained from several software systems, which provides a more realistic view of cpp's usage.

Kästner et al. raised questions of granularity at a theoretical level [18]. The authors could not draw conclusions on the necessity of certain implementation techniques. Our work contributes to this discussion.

There is a significant amount of work on cpp-aware refactorings [15, 33, 35]. This work aims at refactorings within C programs. In contrast, our analysis provides data for discussions on the applicability of other SPL implementation techniques, such as aspects or feature modules, to replace #ifdefs.

8. CONCLUSION

We presented a comprehensive analysis of variability mechanisms (usage of conditional inclusion; #ifdefs) of the preprocessor cpp. We highlighted the connection between the capabilities of cpp and the concepts of SPL engineering. To this end, we formulated four research questions regarding the variability of software systems in general, the complexity, granularity, and types of extension. We proposed a set of metrics for answering these questions and analyzed forty publically available open-source software systems with more than thirty million lines of code.

We found that cpp's mechanisms are used frequently for implementing source code that is optional or incremental, as it occurs in SPLs (on the average 23 % of the code base in a project is variable). Furthermore, we observed that the complexity of these extensions is independent of the size of the software system. Another result is that most extensions occur at a high level of granularity (programmers use #ifdefs mostly to enframe functions or entire blocks, such as if-statements or for-loops). This is promising, especially, when considering refactorings using alternative SPL implementation techniques, such as aspects or feature modules, that provide similar mechanisms for implementing such variabilities. Finally, we found that most extensions are heterogeneous and, consequently, that the quantification mechanisms of aspect-oriented languages are not needed frequently.

The data we have collected can serve as input for the research area on language design and tool support and may permit answers to further questions on SPL engineering.

Acknowledgments

We are grateful to Janet Feigenspan for help with statistical evaluations. Apel's work is supported in part by DFG project #AP 206/2-1.

9. REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. Hassan. Can we Refactor Conditional Compilation into Aspects? In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 243–254. ACM Press, 2009.

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

[3] S. Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, School of Computer Science, University of Magdeburg, 2007.

[4] S. Apel. How AspectJ is Used: An Analysis of Eleven AspectJ Programs. *Journal of Object Technology (JOT)*, 9(1):117–142, 2010.

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

[6] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering (TSE)*, 33(9):577–591, 2007.

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

[9] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 56–65. ACM Press, 2004.

[10] A. Colyer, A. Clement, and G. Blair. On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Lancaster University, January 2004.

[11] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[12] M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering (TSE)*, 28(12):1146–1170, 2002.

[13] J. Favre. Preprocessors from an Abstract Point of View. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 329–339. IEEE CS, 1996.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[15] A. Garrido and R. Johnson. Challenges of Refactoring C Programs. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 6–14. ACM Press, 2002.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, November 1990.

[17] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Feature Using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 223–232. IEEE CS, 2007.

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

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

[20] M. Kendall and B. Babington Smith. The Problem of m Rankings. *The Annals of Mathematical Statistics*, 10(3):275–287, 1939.

[21] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

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

[23] M. Krone and G. Snelting. On the Inference of Configuration Structures from Source Code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 49–57. IEEE CS, 1994.

[24] K. Lee, K. Kang, M. Kim, and S. Park. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 103–112. IEEE CS, 2006.

[25] J. Maletic, M. Collard, and H. Kagdi. Leveraging XML Technologies in Developing Program Analysis Tools. In *Proceedings of the ICSE Workshop on Adoption-Centric Software Engineering (ACSE)*, pages 80–85, 2004. `http://www.acse2004.cs.uvic.ca/`.

[26] G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 275–284. IEEE CS, 2001.

[27] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[28] A. Reynolds, M. Fiuczynski, and R. Grimm. On the Feasibility of an AOSD Approach to Linux Kernel Extensions. In *Proceedings of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 1–7. ACM Press, 2008.

[29] M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 68(12):1493–1512, 2009.

[30] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier. Avoiding Variability of Method Signatures in Software Product Lines: A Case Study. In *Proceedings of the GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, pages 20–25, 2007. `http://www.softeng.ox.ac.uk/aople/`.

[31] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag, 1998.

[32] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *Proceedings of the USENIX Technical Conference*, pages 185–197. USENIX Association Berkeley, 1992.

[33] D. Spinellis. Global Analysis and Transformations in Preprocessed Languages. *IEEE Transactions on Software Engineering (TSE)*, 29(11):1019–1030, 2003.

[34] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the European Software Engineering Conference and of the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 166–175. ACM Press, 2005.

[35] M. Vittek. Refactoring Browser with Preprocessor. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 101–110. IEEE CS, 2003.

name	LOC	NOFC	LOF	AND	SD	TD	TYPE			GRAN					
							HOM	HET	HOHE	GL	FL	BL	SL	EL	ML
apache	214,607	1,158	45,075	1.18±0.18	5.58±15.35	1.73±1.35	134	2,041	98	2,015	2,186	828	43	35	13
berkeley db	187,429	1,537	28,341	1.09±0.12	4.66±13.80	1.39±0.82	189	1,507	108	1,246	1,138	1,584	2	19	12
cherokee	51,878	328	7,679	1.07±0.08	3.70±7.28	1.59±1.12	42	460	10	534	272	218	0	7	0
clamav	75,345	285	10,809	1.11±0.15	5.93±12.14	1.46±0.80	41	479	41	864	375	411	1	0	2
dia	128,762	91	5,406	1.03±0.04	5.40±16.69	1.03±0.55	3	144	16	439	222	76	7	0	5
emacs	237,341	1,373	76,160	1.26±0.28	10.56±122.58	3.01±1.55	118	3,223	119	3,371	2,344	1,515	15	119	30
freebsd	5,898,723	16,167	845,936	1.14±0.17	10.47±464.34	2.51±1.50	2,750	29,916	2,299	48,420	32,520	19,137	454	691	525
gcc	1,545,302	5,063	284,547	1.20±0.23	6.96±19.95	2.32±3.45	482	9,380	323	11,723	5,780	2,743	284	369	51
ghostscript	442,021	816	21,864	1.06±0.08	3.98±9.40	1.25±1.12	145	1,141	67	2,162	1,206	694	16	12	23
gimp	587,681	392	19,111	1.07±0.08	6.53±15.48	1.65±1.64	163	462	32	1,219	630	344	2	0	1
glibc	751,585	3,012	86,641	1.14±0.16	7.30±31.57	1.69±1.20	432	5,821	391	10,080	4,143	2,218	161	98	55
gnumeric	254,815	291	11,833	1.03±0.05	3.15±4.05	1.21±0.95	9	505	35	350	809	633	66	15	17
gnuplot	76,061	434	20,543	1.16±0.20	7.71±20.10	2.09±2.11	35	947	48	1,091	605	710	50	19	4
irssi	49,798	55	1,262	1.05±0.07	2.51±2.43	1.20±0.64	4	73	3	75	96	24	1	0	0
libxml2	210,903	2,047	139,722	1.59±0.39	8.36±90.09	5.02±3.02	161	2,693	76	6,923	1,954	939	0	29	0
lighttpd	38,983	167	8,565	1.15±0.19	4.68±6.12	1.36±0.97	22	222	35	290	258	289	51	1	16
linux	5,986,427	9,102	647,969	1.09±0.11	6.07±47.96	1.44±1.36	434	17,368	876	30,611	20,024	7,072	497	77	120
lynx	117,521	806	43,585	1.64±0.46	11.56±106.68	2.88±1.21	22	1,771	62	1,355	1,699	1,222	9	69	8
minix	63,759	356	10,689	1.26±0.23	5.11±8.49	1.40±0.84	31	367	18	632	310	254	3	1	0
mplayer	606,101	1,236	114,068	1.11±0.13	5.84±14.51	1.50±1.36	83	2,427	153	3,016	3,130	1,696	130	19	26
mpsolve	10,181	13	263	1.15±0.04	2.54±2.30	1.27±0.45	0	18	0	20	10	4	0	0	0
openldap	246,175	708	66,922	1.21±0.20	4.66±7.83	1.63±1.15	36	1,241	65	1,193	1,334	756	45	26	15
opensolaris	8,197,042	10,901	1,644,723	1.14±0.14	10.42±93.99	1.84±1.47	2,498	22,696	1,390	38,866	43,652	16,404	305	308	376
openvpn	38,285	276	23,288	1.46±0.41	6.69±18.12	1.71±1.20	5	445	11	563	476	225	30	7	13
parrot	98,371	539	26,680	1.28±0.26	6.06±12.67	2.02±1.65	131	908	45	1,297	601	205	1	2	0
php	574,411	2,426	196,830	1.29±0.25	5.55±19.43	1.89±1.38	310	4,741	155	5,181	3,030	1,996	53	53	30
pidgin	269,590	576	40,035	1.15±0.14	3.24±11.44	1.16±0.87	60	854	63	1,090	1,062	369	0	9	39
postgresql	471,751	692	23,204	1.11±0.12	4.87±18.37	1.56±1.22	63	1,031	70	1,591	1,391	756	0	23	12
privoxy	24,079	153	8,984	1.55±0.41	6.98±13.60	1.87±0.99	7	393	12	332	311	187	6	9	0
python	374,160	5,127	100,098	1.18±0.21	4.08±110.60	2.67±1.09	234	6,218	111	2,274	6,730	861	37	31	75
sendmail	83,744	880	31,996	1.35±0.29	5.90±18.54	2.04±1.26	45	1,913	55	1,383	1,044	1,012	6	65	25
sqlite	94,517	292	50,887	1.32±0.25	7.72±13.64	1.79±1.16	8	874	11	823	666	326	14	6	3
subversion	509,337	409	28,433	1.37±0.33	14.03±38.10	1.81±1.20	186	491	44	2,939	1,421	415	3	4	4
sylpheed	99,132	271	18,847	1.12±0.12	4.19±10.37	1.41±1.32	5	446	18	732	399	156	13	5	11
tcl	135,183	2,481	26,618	1.25±0.18	3.79±36.43	2.67±1.41	33	3,220	32	2,976	680	629	3	18	4
vim	225,381	779	132,678	1.57±0.51	19.30±82.09	2.40±1.47	108	4,083	205	2,677	4,336	3,867	116	713	171
xfig	72,549	107	5,184	1.08±0.12	4.66±6.63	1.88±1.99	1	152	10	198	157	98	2	6	1
xine-lib	495,355	1,692	170,602	1.27±0.23	5.30±17.23	1.80±1.36	59	3,499	92	3,361	2,882	1,219	64	11	18
xorg-server	527,871	1,360	95,227	1.09±0.11	9.00±28.26	1.84±1.97	264	2,790	224	5,088	2,986	2,332	51	84	17
xterm	49,621	453	19,208	1.41±0.35	8.10±19.11	2.35±1.63	26	1,199	37	1,007	884	461	3	21	4

LOC: lines of code; **NOFC**: number of feature constants; **LOF**: lines of feature code; **AND**: average nesting depth of #ifdefs; **SD**: average scattering degree; **TD**: average tangling degree; **TYPE**: number of extensions measuring the type (**HOM**: number of homogeneous extensions; **HET**: number of heterogeneous extensions; **HEHO**: number of extensions with heterogeneous and homogeneous portions); **GRAN**: number of extensions measuring the granularity (**GL**: global level; **FL**: function or type level; **BL**: block level; **SL**: statement level; **EL**: expression level; **ML**: method signature level)

Table 2: Data of the analysis