Towards a Better Understanding of Feature Dependencies in Preprocessor-based Systems

Felipe Queiroz¹  Márcio Ribeiro¹,²  Sérgio Soares¹  Paulo Borba¹

¹ Federal University of Pernambuco, Av. Prof. Luís Freire, 50.740-540, Recife, Brazil
² Federal University of Alagoas, Av. Lourival de Melo Mota, 57.072-970, Maceió, Brazil
{fbq, mmr3, scbs, phmb}@cin.ufpe.br

Abstract—Preprocessors are a common mechanism to implement features. In this context, features usually share elements, like variables and methods, which leads to code level feature dependencies like when a feature declares a variable used by another feature. Thus, developers are susceptible to maintain a feature and cause problems to another. To better understand feature dependencies, in this paper we complement previous work by providing an empirical study about them where we investigate if there are correlations with software complexity. In addition, we investigate where they occur, i.e., in which kinds of preprocessor directives we can find them. In particular, we answer the following research questions: How does program size influence feature dependencies? How does the number of different kinds of preprocessor directives influence feature dependencies? How often do feature dependencies occur for each kind of preprocessor directive? To answer them, we analyze 45 preprocessor-based systems, software families, and product lines of different domains, size, and languages. The data we collect complement previous work and show that (i) there are correlations with complexity and (ii) we can find feature dependencies in many kinds of preprocessor directives.

I. INTRODUCTION

Preprocessors are a common mechanism to implement software variability in systems, software families, and software product lines [1], [2]. However, despite their widespread use, preprocessors have several drawbacks, including no support for separation of concerns [3].

In this context, when using preprocessors we can define feature boundaries by using conditional compilation directives. Features usually share code level elements such as variables and methods. In general, this leads to subtle feature dependencies like when a feature declares a variable which is used by another feature. This way, a developer responsible for feature A may introduce problems to another feature B that may not even be under her responsibility. So, such dependencies may compromise feature modularization, which aims at achieving independent comprehensibility and changeability [4].

Previous studies provide data on preprocessor usage [5], [6], which is important to assess how developers use preprocessors in real projects. Although we have provided data on how often feature dependencies may occur in practice [7], we still need to complement all these studies to better understand feature dependencies and to analyze to how extent they might be a problem in practice.

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

1. How does program size influence feature dependencies?
2. How does the number of features influence feature dependencies?
3. How does the number of different kinds of preprocessor directives influence feature dependencies?

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

4. How often do feature dependencies occur for each kind of preprocessor directive?

Answering the three first questions is important to better understand how software complexity correlates with the occurrence of feature dependencies. Answering the last question is important for assessing how often we can face a feature dependency on each preprocessor directive. To answer them, we analyze 45 projects from different domains, size, and languages (C and Java). In particular, we built a tool to compute data with respect to feature dependencies.

In general, this paper presents an empirical study with the following contributions:

- We analyze correlations between software complexity and feature dependencies;
- We complement previous work with different data on preprocessor usage and on feature dependencies;
- We provide tool support for collecting the aforementioned data.

The data we report in this paper improve discussion about a widely used mechanism. Besides, researchers may use them as input to their studies, such as programming languages design and preprocessors tool support.
II. FEATURE DEPENDENCIES

Features usually share elements such as variables and methods among each other. In this context, when maintaining product lines, due to this sharing, developers may compromise feature modularization, which aims at achieving comprehensibility and changeability [4]. In other words, this means that a developer can introduce problems to another feature that is not under her responsibility [8]. For example, she can modify an element—variable or method—in one feature that is declared or used by another feature. Whenever we have this sharing, we say that there is a feature dependency between the involved features [7].

In this paper, we focus on simple dependencies. We characterize them as the following: (i) a dependency consisting of def-use, that is, variables that are defined in one feature and used in another; and (ii) intra-procedural dependencies, that is, feature dependencies which only occur within methods boundaries. Despite some other kinds of feature dependencies [7], we focus on the simple ones throughout this paper.

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

A. Dependency Occurrence on Optional Feature

Listing 1 illustrates a variable that is declared in one feature and used in another. The code snippet was extracted from the Vim product line. Vim is an advanced text editor that seeks to provide the power of the de-facto Unix editor Vi, with a more complete feature set. In this example, we declare the echo variable in a mandatory feature—there is no #ifdef statement encompassing the variable declaration—and we use such a variable in the GOTO_FROM_WHERE_INCLUDED feature. In this particular case, we use the variable as a parameter of the echogets method to compose a conditional structure whose result will be assigned to the ok variable. Notice that, because of the feature dependency, when maintaining the echo variable in the mandatory feature (i.e., by changing it type or its name), developers may introduce problems to the GOTO_FROM_WHERE_INCLUDED feature. Unfortunately, these problems are not always easy to detect, since maintenance tasks may lead not only to compilation problems, but also to behavioral ones [7], [8].

Listing 1: Feature dependency occurrence on Vim text editor.

```c
int main(...) {
    ...
    int echo = ...
    #ifdef GOTO_FROM_WHERE_INCLUDED
        ok = (echogets(Reason, echo) != NULL);
    ...
    #endif
    ...
}
```

B. Dependency Occurrence on Alternative Feature

Different from the previous example, Listing 2 illustrates another way in which feature dependencies can occur: between alternative features and a mandatory one. The code snippet was extracted from Irssi3. The Irssi is a terminal based IRC client for UNIX systems. In this example, we declare the key variable in two alternative features (encompassed by the #ifdef and #else statements), with different types. Again, because of such a feature dependency, changing key may introduce problems to its use in the g_array_append_val method, which is in the mandatory feature.

Listing 2: Feature dependency occurrence on Irssi.

```c
void term_gets(...)
{
    ...
    #ifdef WIDEC_CURSES
        win_t key;
    #else
        int key;
    #endif
    ...
    g_array_append_val(buffer, key);
    ...
}
```

III. EMPIRICAL STUDY

So far, we presented feature dependencies extracted from real systems. Also, we mentioned that these dependencies can lead developers to introduce problems into other features they might even not be responsible for.

Therefore, given the importance of dealing with feature dependencies, in this section we provide an empirical study to better understand how they occur in practice. Recent work [5], [6] have focused on understanding how developers use variability mechanisms of C preprocessor (cpp) in practice. Another recent work [7] focused on how often feature dependencies may occur in practice. In particular, we complement these studies by providing data on cpp preprocessor usage and on feature dependencies by answering the research questions stated in Section I.

A. Sample Projects & Collecting Data

We answer our research questions by analyzing 45 software projects. They consist of single systems, software families, and software product lines.

In terms of complexity, they vary from simple and small, such as mpisolve, to complex and very large ones such as kernel linux. Moreover, the projects belong to a variety of different domains, such as database systems, web servers, programming libraries, mobile games, and operating systems. They are written in two different programming languages: C and Java. The majority of programs is written in C and all of them contain several features implemented using preprocessor directives.

We use a subset of directives supported by Antenna4—a simple Java preprocessor. This subset comprises the following

1http://www.vim.org/

2http://ex-vi.sourceforge.net/

3http://irssi.org/

4http://antenna.sourceforge.net/
directives: #ifdef, #ifndef, #elifdef, #elif, #else, #debug, and #mdebug. Notice that this subset is also supported by cpp.

To perform our analysis we use the tool src2srxml\(^5\) which parses the unprocessed code and generates a XML representation of it. This representation has the AST form, which allows us to navigate between annotations and perform computations. For C projects, we analyze .c files. Analogously, we analyze .java files, but in this case we perform some adjustments in the parsing process, aiming to give the XML the same representation as for C projects. To compute feature dependencies, we developed a tool based on a previous work [5]. We use this tool to compute a set of metrics regarding feature dependencies.

During our analysis, we excluded 77 from 56180 files—comprising 0.13% of all files analyzed—from our computations. This occurred because src2srxml could not correctly parse these files, preventing us to do the measurements.

B. Metrics

To better understand how feature dependencies occurs and answer our research questions, we introduce the following set of metrics:

**Number of Methods (NoM).** The NoM metric represents the total number of methods of a project, representing the size of a software program in terms of such a number.

**Source Lines of Code (SLoC).** The SLoC metric represents the size of a software program regarding lines of code. To measure it, we use the CLoC\(^6\) tool. We use this metric with the NoM metric to better understand the influence of program size on feature dependencies.

**Number of Directives Occurrence (NoDiO).** The NoDiO metric indicates the occurrence of each directive on systems. We count how many times each directive appear on systems’ methods.

**Number of Dependencies in a Directive (NoDDi).** The NoDDi metric represents the occurrence of feature dependencies in each directive. We measure it by counting the directives involved in a dependency relation. For instance, if a variable is declared in a mandatory feature and is used in a feature encompassed by #ifdef (see Listing 1), we count one feature dependency occurrence in #ifdef directive.

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

**Number of Feature Expressions (NoFE).** The NoFE metric represents the number of feature expressions existing in a project. Feature expressions are the logical expressions evaluated by the preprocessor when it finds a directive declaration in code processing. In Listing 2 for example, the feature expression is represented by the term WIDE_CURSES after #ifndef directive declaration. The expressions may be formed by several terms through logical operators combinations, i.e. A && B or A || B.

**Number of Dependencies (NoDe).** The NoDe metric is the total number of dependencies occurrence in the software program.

C. Results

We present the 45 projects and their respective computed metrics in Table II (at the end of the paper). We present all plots with the correlation coefficient of their respective metrics. Because the data are not normally distributed, we compute the correlation coefficient using the Kendall method [9].

We now answer our research questions. Firstly, we focus on the questions related to the correlation analysis regarding complexity and feature dependencies (questions 1, 2, and 3). Then, we answer the fourth question, related to where—i.e., which kinds of preprocessor directives—feature dependencies occur.

1. **How does program size influence feature dependencies?**

   Not surprisingly, our data reveal that dependencies of a software system increases with its size. We confirm this by the correlations between the metrics NoM and NoDe as well as SLoC and NoDe (see Figure 1). It turns out that large software systems usually have more method definitions. So, the probability of finding #ifdefs increases. Consequently, the probability of finding dependencies increases as well.

2. **How does number of features influences feature dependencies?**

   Our data reveal that the number of dependencies increases with the increasing of feature expressions definitions. Like program size, we confirm this by the correlation between the NoFE and NoDe metrics, which correlate highly (see Figure 2).

![Fig. 2: Plot NoFE/NoDe. Correlation coefficient: 0.77.](image-url)
3. How does number of different kinds of preprocessor directives influence feature dependencies?

We can answer this question from data presented in Figure 3. The number of feature dependencies increases with the use of different kinds of directives in software programs. We confirm this by the correlation between the NoDiDe and NoDe metrics—0.57.

Fig. 3: Plot NoDiDe/NoDe. Correlation coefficient: 0.57.

For all systems, we notice that feature dependencies occurred in at least two kinds of directives. Also, no system presented feature dependencies in more than five kinds of preprocessor directives (see last column in Table II). Even in large projects such as kernel linux and freebsd there were no dependencies involving some kinds of directives, as we shall see in the next section.

4. How often feature dependencies occur for each kind of preprocessor directive?

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

TABLE I: Dependency occurrence on cpp’s variability mechanisms.

<table>
<thead>
<tr>
<th></th>
<th>#elif</th>
<th>#else</th>
<th>#if</th>
<th>#ifdef</th>
<th>ifndef</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>1.6%</td>
<td>10.6%</td>
<td>48.8%</td>
<td>37.9%</td>
<td>7.9%</td>
</tr>
<tr>
<td>SD</td>
<td>3.6%</td>
<td>8.3%</td>
<td>19.6%</td>
<td>12.7%</td>
<td>4.3%</td>
</tr>
</tbody>
</table>

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

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

D. Threats to Validity

Another kinds of dependencies. In this study we disregard some kinds of feature dependencies. Our assumptions about simple dependencies did not consider other kinds of feature dependencies, such as the ones that cross methods boundaries (inter-procedural dependencies).

Feature expression equality. Semantic equivalence of feature expressions like A && B and B && A are missing in our analysis. Our tool performs the analysis by using string comparison to check if different code fragments belong to the
same expression. However, such cases are not common [5] and does not significantly influence the results.

Converting of source code on XML documents. For converting the source code in the AST form, we rely on the src2xml tool. To compute the dependencies and measure the NoFE metric, our tool relies blindly on this mapping. We believe that the extensive test suite used by the authors of src2xml is reliable to verify it satisfactorily.

IV. RELATED WORK

In a prior study [7] we analyzed the occurrence frequency of feature dependencies on real systems. We developed a tool to analyze 43 product lines of different domains, sizes and languages. We found that 65.92% ± 18.54% of the methods with directives have dependencies. Moreover, we performed an empirical study to assess the impact of these feature dependencies on maintenance effort when using VSoC and emergent interfaces [8]. We complement this work by taking the complexity aspect into consideration, refining and detailing the occurrence of feature dependencies on real programs.

There is research on assessing the way developers use preprocessor directives. Recently, researchers [5] created and computed many metrics to analyze the feature code scattering and tangling when using conditional compilation directives. To do so, they analyzed 40 software product lines implemented in C. They formulated research questions and answered them with the aid of a tool. We complement this work by taking feature dependencies into consideration. Also, we provide data in different product lines (the ones written in Java).

Researchers [6] examined the use of preprocessor-based code in systems written in C. Directives like #ifdefs are indeed powerful, so that programmers can make all kinds of annotations using them. Hence, developers can introduce subtle errors like annotating a closing bracket but not the opening one. This is an "undisciplined" annotation. Disciplined annotations hold properties useful for preprocessor-aware parsing tools so we can represent annotations as nodes in the AST. They found that the majority of the preprocessor usage are disciplined. Specifically, they found that the case studies have 84.4% of their annotations disciplined. We also analyzed several systems, but we focus on dependencies among features implemented with preprocessors.

Another study concerning preprocessor usage in C systems [10] studied the occurrence of conditional compilation directives focusing on the entire code by analyzing many other kinds of preprocessors (like macros). We also analyzed several kinds of conditional compilation directives, but no macros.

V. CONCLUDING REMARKS

This paper presented an empirical study that analyzed the correlation between software complexity and feature dependencies. Also, we provided data with respect to where feature dependencies occur. By where we mean the different kinds of preprocessor directives we can find feature dependencies.

Not surprisingly, we found that the number of feature dependencies increases with the system complexity (in terms of lines of code, number of methods, and number of features). Also, we found that the higher the number of different kinds of directives in a system, the higher is the number of feature dependencies.

Our work complements previous work on preprocessor usage. Researchers can use our data as input to their studies. For example, we have a tool for Eclipse IDE that computes feature dependencies. Because we use data-flow analysis, sometimes it takes a long time to compute them. Now that we know that the feature dependencies occur more frequently in ifdef and if statements (when compared to ifndef, for instance), our tool can focus on these directives, exchanging precision for better performance.

ACKNOWLEDGMENT

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, and by CAPES/PROCAD 2007/090. Felipe Queiroz was supported by FAPEAL, grant 20091235501-3. Sérgio Soares is partially supported by CNPq and FACEPE, grants 305085/2010-7 and APQ-0093-1.03/08.

REFERENCES


<table>
<thead>
<tr>
<th>System</th>
<th>NoM</th>
<th>SLoC</th>
<th>NoDiDe</th>
<th>NoDi</th>
<th>NoDDiDe</th>
<th>NoDDi</th>
<th>NoDiDi</th>
<th>NoDE</th>
<th>NoDiDiDe</th>
<th>Blacklag</th>
<th>FeedBack</th>
<th>NoDiDiDe</th>
</tr>
</thead>
<tbody>
<tr>
<td>cherokee-1.0.8</td>
<td>324</td>
<td>13575</td>
<td>0</td>
<td>98</td>
<td>0</td>
<td>0</td>
<td>12209</td>
<td>96</td>
<td>66</td>
<td>0</td>
<td>1822</td>
<td>159</td>
</tr>
<tr>
<td>db-5.1.19</td>
<td>408</td>
<td>98151</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>68225</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1006</td>
<td>129</td>
</tr>
<tr>
<td>dia-0.97.1</td>
<td>326</td>
<td>13547</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>96126</td>
<td>96</td>
<td>66</td>
<td>0</td>
<td>1245</td>
<td>194</td>
</tr>
<tr>
<td>linux-2.6.36</td>
<td>980</td>
<td>98980</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>58982</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2432</td>
<td>2854</td>
</tr>
<tr>
<td>openldap-2.4.23</td>
<td>326</td>
<td>13547</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>96126</td>
<td>96</td>
<td>66</td>
<td>0</td>
<td>1245</td>
<td>194</td>
</tr>
<tr>
<td>subversion-1.6.13</td>
<td>326</td>
<td>13547</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>96126</td>
<td>96</td>
<td>66</td>
<td>0</td>
<td>1245</td>
<td>194</td>
</tr>
<tr>
<td>xinelib-1.1.19</td>
<td>326</td>
<td>13547</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>96126</td>
<td>96</td>
<td>66</td>
<td>0</td>
<td>1245</td>
<td>194</td>
</tr>
<tr>
<td>xterm-2.6</td>
<td>980</td>
<td>98980</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>58982</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2432</td>
<td>2854</td>
</tr>
</tbody>
</table>

**TABLE II: Data analysis.**

**NoM:** number of methods;  
**SLoC:** source lines of code;  
**NoDiDe:** number of directives occurrence;  
**NoFe:** number of feature expressions;  
**NoDe:** number of dependencies;  
**NoDDiDe:** number of dependency occurrence on each directive;  
**NoDiDi:** number of directives with dependencies.