

UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS GRADUAÇÃO EM INFORMÁTICA

IRAN RODRIGUES GONZAGA JUNIOR

Empirical Studies on Fine-Grained Feature Dependencies

Maceió
2015

Iran Rodrigues Gonzaga Junior

Empirical Studies on Fine-Grained Feature Dependencies

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Orientador: Prof. Dr. Márcio de Medeiros
Ribeiro

Coorientador: Prof. Dr. Balduino Fonseca
dos Santos Neto

Maceió
2015

Catálogo na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

G642e Gonzaga Junior, Iran Rodrigues.
Empirical studies on fine-grained feature dependencies / Iran Rodrigues
Gonzaga Junior. – Maceió, 2015.
59 f. : il.

Orientador: Márcio de Medeiros Ribeiro.
Coorientador: Baldoino Fonseca dos Santos Neto.
Dissertação (Mestrado em Informática) – Universidade Federal de Alagoas.
Instituto de Computação. Programa de Pós-Graduação em Informática.
Maceió, 2015.

Bibliografia: f. 56-59.

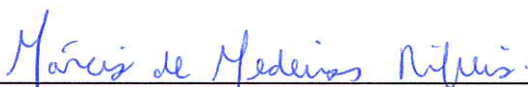
1. Engenharia de software. 2. Famílias de programas. 3. Preprocessadores.
4. Dependências entre features. 5. Erros de variabilidade. I. Título.

CDU: 004.416:004.4'427



Membros da Comissão Julgadora da Dissertação de Mestrado de Iran Rodrigues Gonzaga Júnior, intitulada: “Empirical Studies on Fine-Grained Feature Dependencies”, apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas em 10 de agosto de 2015, às 14h00min, no Miniauditório do Instituto de Computação da UFAL.

COMISSÃO JULGADORA



Prof. Dr. Márcio de Medeiros Ribeiro
UFAL – Instituto de Computação
Orientador



Prof. Dr. Balduino Fonseca dos Santos Neto
UFAL – Instituto de Computação
Orientador



Prof. Dr. Patrick Henrique da Silva Brito
UFAL – Instituto de Computação
Examinador



Prof. Dr. Paulo Henrique Monteiro Borba
UFPE – Universidade Federal de Pernambuco
Examinador

Ao meu filho Guilherme, meu maior professor.

AGRADECIMENTOS

Agradeço a Deus, acima de tudo, por toda a força e inspiração que obtive no decorrer dessa jornada, possibilitando a superação de todos os obstáculos encontrados no caminho, culminando não só na conquista de mais um degrau em minha formação acadêmica, mas na realização de um sonho.

À minha família, por todo o apoio e incentivo recebido nesse período. Em especial, agradeço à minha esposa, Bianca, e ao meu filho, Guilherme, pela compreensão nos momentos em que tive que abdicar de suas companhias para me dedicar às atividades acadêmicas. Prometo que irei compensar todo esse tempo da melhor forma possível. Também agradeço aos meus pais, Iran e Mônica, meus primeiros educadores, sem os quais eu não chegaria até aqui.

Agradeço imensamente ao meu orientador, Professor Márcio Ribeiro, por me guiar na realização deste trabalho, exigindo nada menos que o meu melhor. Márcio, a forma como você se fez presente durante toda essa caminhada, não me permitiu acomodar ou esmorecer. Agradeço por toda a confiança depositada em mim, do início ao fim dessa jornada. Espero poder atender às suas expectativas ao te entregar este trabalho e que possamos manter a parceria em projetos futuros.

Agradeço ao meu coorientador, Professor Baldoino Fonseca, por ter me recebido no *EASY Group* e por ter contribuído diversas vezes com o amadurecimento de minha pesquisa. A oportunidade de ter apresentado o meu trabalho a pesquisadores da Universidade de Coimbra, como o Professor Marco Vieira, foi especialmente proveitosa.

Ao colega de trabalho e de grupo de estudo, Flávio Medeiros, por toda a ajuda técnica na realização dos estudos empíricos. Cada vez em que precisei de sua colaboração, fui prontamente atendido, sempre com muita presteza e paciência. Flávio, também me coloco à sua disposição no que eu puder ajudar.

A todos os professores das disciplinas cursadas no mestrado, pois muito aprendi com cada um deles no decorrer do curso. Cada lição aprendida certamente contribuiu bastante para a minha formação.

Agradeço aos membros da banca da proposta de dissertação, apresentada em março de 2015, formada pelos Professores Patrick Brito e Rodrigo Paes, pelo excelente retorno recebido, que me motivou a buscar melhorar cada vez mais o presente trabalho.

Agradeço a todos os amigos que fiz no mestrado, com quem pude trocar as mais diversas experiências e ideias, como Diogo, David, Gilton, Marcos Paulo, Daniel, Maria "Bia", entre tantos outros. Espero poder reencontrá-los ao retomarmos os estudos no doutorado.

Agradeço também a todo o pessoal da secretaria do Instituto de Computação, em especial à Flor, por toda a atenção dispensada sempre que precisei de sua ajuda nos mais

diversos trâmites acadêmicos.

Por fim, agradeço ao Instituto Federal de Alagoas, especialmente a todos os meus chefes nesse período: Fernando, David, Emanuel e Rogério, pelo reconhecimento da importância da capacitação da equipe, possibilitando uma jornada de trabalho flexível. Assim, pude conciliar minhas obrigações profissionais e minhas atividades acadêmicas, sem prejuízos a nenhuma delas. Toda a experiência adquirida durante o mestrado contribuiu significativamente para meu desenvolvimento pessoal e profissional. Assim, espero retribuir o favor realizando um melhor trabalho para o instituto.

RESUMO

Manter famílias de programas não é uma tarefa trivial. Desenvolvedores comumente introduzem erros quando não consideram dependências existentes entre *features*. Quando tais *features* compartilham elementos de programa, como variáveis e funções, utilizar estes elementos inadvertidamente pode resultar em erros de variabilidade. Neste contexto, trabalhos anteriores focaram apenas na ocorrência de dependências intraprocedurais, ou seja, quando *features* compartilham elementos de programa dentro de uma mesma função. Mas ao mesmo tempo, ainda não temos estudos investigando dependências que extrapolam os limites de uma função, já que esses casos também podem causar erros. De fato, neste trabalho nós trazemos evidências de que erros de variabilidade também podem ocorrer devido a dependências interprocedurais ou a variáveis globais. Para avaliar até que ponto esses diferentes tipos de dependências entre *features* existem na prática, nós realizamos um estudo empírico abrangendo 40 famílias de programas de diferentes domínios e tamanhos. Nossos resultados mostram que dependências intraprocedurais e interprocedurais são comuns na prática: $51,44\% \pm 17,77\%$ das funções com diretivas de pré-processamento têm dependências intraprocedurais, enquanto $25,98\% \pm 19,99\%$ de todas as funções possuem dependências interprocedurais. Após estudar a ocorrência de dependências entre *features* na prática, nós conduzimos outro estudo empírico, desta vez direcionado a encontrar erros de variabilidade relacionados a dependências entre *features*. Aqui nos concentramos em variáveis não declaradas, funções não declaradas, variáveis sem uso e funções sem uso. Este estudo utiliza 15 sistemas para responder a questões de pesquisa relacionadas a como os desenvolvedores introduzem esses erros e sua frequência. Nós detectamos e confirmamos a existência de 32 erros de variabilidade. O conjunto de erros que coletamos é uma fonte valiosa para fundamentar a pesquisa sobre esse tema e para auxiliar desenvolvedores de ferramentas, de forma que eles forneçam meios em suas ferramentas de evitar tais problemas.

Palavras-chaves: Famílias de programas, Pré-processadores, Dependências entre *features*, Erros de variabilidade.

ABSTRACT

Maintaining program families is not a trivial task. Developers commonly introduce bugs when they do not consider existing dependencies among features. When such implementations share program elements, such as variables and functions, inadvertently using these elements may result in bugs. In this context, previous work focuses only on the occurrence of intraprocedural dependencies, that is, when features share program elements within a function. But at the same time, we still lack studies investigating dependencies that transcend the boundaries of a function, since these cases might cause bugs as well. Indeed, in this work we bring evidence that variability bugs can also occur due to interprocedural dependencies and global variables. To assess to what extent these different types of feature dependencies exist in practice, we perform an empirical study covering 40 program families of different domains and sizes. Our results show that the intraprocedural and interprocedural feature dependencies are common in practice: $51.44\% \pm 17.77\%$ of functions with preprocessor directives have intraprocedural dependencies, while $25.98\% \pm 19.99\%$ of all functions have interprocedural dependencies. After studying the feature dependencies occurrence in practice, we perform another empirical study now focusing on finding actual bugs related to feature dependencies. Here we focus on undeclared variables, undeclared functions, unused variables, and unused functions. This study uses 15 systems to answer research questions related to how developers introduce these bugs and their frequency. We detect and confirm 32 variability bugs. The corpus of bugs we gather is a valuable source to ground research on this topic and to help tool developers, so they can provide means in their tools to avoid these problems.

Keywords: Program families, Preprocessors, Feature dependencies, Variability bugs.

CONTENTS

1	INTRODUCTION	9
2	MOTIVATING EXAMPLES	12
2.1	Scenario 1: Intraprocedural dependency	12
2.2	Scenario 2: Global dependency	14
2.3	Scenario 3: Interprocedural dependency	15
2.4	Summary	18
3	FEATURE DEPENDENCIES IN THE WILD	19
3.1	Study Settings	19
3.1.1	Goal, Question, and Metrics	19
3.1.2	Instrumentation	22
3.1.3	Operation	23
3.2	Results and discussion	24
3.2.1	Question 1: How often do program families contain intraprocedural dependencies?	25
3.2.2	Question 2: How often do program families contain global dependencies?	25
3.2.3	Question 3: How often do program families contain interprocedural dependencies?	27
3.2.4	Question 4: How often do dependencies of different directions (mandatory-to-optional, optional-to-mandatory, and optional-to-optional) occur?	29
3.2.5	Question 5: What is the dependency depth distribution for interprocedural dependencies?	32
3.2.6	Threats to validity	33
4	FEATURE DEPENDENCIES CAUSING VARIABILITY BUGS	38
4.1	Strategy to detect variability bugs	39
4.2	Study settings	42
4.2.1	Subject selection	42
4.2.2	Instrumentation	43
4.3	Results and discussion	43
4.3.1	Question 1: What are the frequencies of undeclared variables, unused variables, undeclared functions, and unused functions?	44
4.3.2	Question 2: How do developers introduce variability bugs related to feature dependencies?	46
4.3.3	Summary	48

4.3.4	Threats to validity	48
5	RELATED WORK	50
5.1	Feature dependencies	50
5.2	Variability bugs	51
5.3	C preprocessor usage	52
6	CONCLUDING REMARKS	53
6.1	Review of the contributions	53
6.2	Limitations	54
6.3	Future work	54
	Bibliography	56

1 INTRODUCTION

Developers commonly introduce errors when they fail to recognize dependencies among the software modules they are maintaining (1). The same situation happens in configurable systems in terms of program families and product lines, where features share program elements such as variables and functions. This way, features might depend on each other and developers can miss such dependencies as well. Consequently, by maintaining one feature implementation, they might introduce problems to another, like when assigning a new value to a variable which is correct to the feature under maintenance, but incorrect to the one that uses this variable (2, 3).

In this context, developers often use the C preprocessor to implement variability in software families (4, 5, 6, 7). The C preprocessor allows the use of directives to annotate the code, associating program elements with specific features. When a developer defines a variable in a feature and then uses it in another feature, we have a feature dependency. The same happens with functions. These dependencies might cause problems, such as when an implementation includes a feature that accesses a variable or function, but does not include the feature that defines such identifiers. In this work, we consider only the dependencies we can compute from the analysis of an abstract syntax tree (AST) of a source code. Hence, we refer to these purely syntactic code level feature dependencies simply as feature dependencies.

Previous work (8) reports on how often feature dependencies occur in practice by considering 43 preprocessor-based families and product lines. However, the study focuses only on intraprocedural dependencies, that is, feature dependencies that occur exclusively within the function boundaries. Nevertheless, dependencies that go beyond function boundaries may result in bugs that are harder to detect and fix. In this way, we still lack a study that takes other kinds of feature dependencies into account. Also, we lack a systematic study to better relate feature dependencies and bugs, in the sense we can understand, for example, how developers introduce these bugs in practice, to avoid them in the future.

Therefore, to help fill the gap and better understand feature dependencies, in this work we perform two studies:

- (i) an empirical study to assess to what extent feature dependencies occur in practice, identifying their characteristics and frequency;
- (ii) an empirical study to identify actual variability bugs related to feature dependencies, quantifying such bugs and investigating how developers introduce them.

Before executing the studies, as a first step, we arbitrarily analyze several bug reports

from many open-source software families, like *GCC*,¹ *GNOME*,² and *Linux* kernel.³ The idea of this first step is to learn how the bugs happen in practice and better prepare our studies. Given this knowledge related to the feature dependencies bugs we find, we conduct an empirical study that complements previous work on this topic, in the sense that we take interprocedural dependencies into account. Notice that, during maintenance of preprocessor-based software, these dependencies are even harder to detect: one feature might use data from another and they are in different functions. Because in a typical system we have several method calls passing data, we also compute the depth of such dependencies (from the variable definition to its use). In addition, we consider dependencies based on global variables. We also compute the dependency direction, that is, mandatory-to-optional, optional-to-mandatory, and optional-to-optional. A mandatory-to-optional dependency, for instance, means that the definition of the program element (for instance, a global variable) happens in a mandatory feature—that is, no `#ifdef` encompassing the definition—and its use in an optional feature. In particular, we answer the following research questions: How often do program families contain intraprocedural dependencies? How often do program families contain interprocedural dependencies? What is the average dependency depth for interprocedural dependencies? How often do program families contain global dependencies? How often do dependencies of different directions occur in practice? Answering these questions is important to better understand feature dependencies, assess their occurrence in practice, and enable the development of tools and techniques to guide developers during maintenance tasks in the presence of such dependencies.

To answer our research questions, our first study covers 40 C program families of different domains and sizes. We select these families inspired by previous work (6, 7, 8, 9, 10, 11). We rely on *TypeChef* (12), a variability-aware parser, to compute feature dependencies considering the entire configuration space of each source file of the families we analyze. To detect dependencies that span multiple files, we perform global analysis (instead of per-file analysis).

The data we collect in our first empirical study reveal that the feature dependencies we consider in this work are reasonably common in practice, except the ones regarding global variables. Following the convention “average \pm standard deviation”, our results show that $51.44\% \pm 17.77\%$ of functions with preprocessor directives have intraprocedural dependencies, $11.90\% \pm 12.20\%$ of the functions which use global variables have global dependencies, while $25.98\% \pm 19.99\%$ of all functions have interprocedural dependencies.

Given that feature dependencies are reasonable common in practice, we conduct a second empirical study to better understand the actual variability bugs related to dependencies. For example, here we intend to understand the way developers introduce such

¹ <<https://gcc.gnu.org/bugzilla/>>

² <<https://bugzilla.gnome.org/>>

³ <<https://bugzilla.kernel.org/>>

bugs when maintaining software families. This time, we analyze 15 popular open-source program families written in C, such as *Bash*, *gzip*, and *libssh*. We answer research questions related to the occurrence of variability bugs regarding feature dependencies in the families we analyze and how developers introduce them. Answering these questions is important to quantify and study these issues, understand their peculiarities, and support tool developers, so they can provide means to minimize or even avoid such variability bugs.

To detect variability bugs in a systematic way and go beyond analysis of repositories, in this second study we also consider variability-aware analysis (12, 13) to check the entire configuration space. We also perform a global analysis to detect bugs that span multiple files and take the header files into account. However, to scale our study and minimize setup problems of variability-aware tools, we propose a strategy that only considers the header files of the target platform. We instantiate our strategy with the *TypeChef* (12) variability-aware parser and target headers of the Linux platform.

Our second empirical study confirms that such feature dependencies can cause variability bugs (14). We detect 32 variability bugs of different kinds, such as undeclared variables and functions. This set also includes 10 unused variables and 7 unused functions, which do not cause compilation errors, but might trigger compilation warnings and slightly pollute the code. Thus, they are still considered by developers through several bug reports.⁴ Although some of these bugs might be relatively simple to fix, the variability factor makes them harder to detect, remaining hidden in the families source code.

We organize the remainder of this work as follows:

- Chapter 2 shows motivating examples that illustrate actual variability bugs related to feature dependencies;
- Chapter 3 presents the empirical study to assess feature dependencies in practice;
- Chapter 4 presents the empirical study to quantify and better understand variability bugs regarding feature dependencies;
- Chapter 5 discusses the related work;
- Chapter 6 presents the final considerations of this work.

⁴ <https://bugzilla.gnome.org/show_bug.cgi?id=461011>, 167715, and 401580

2 MOTIVATING EXAMPLES

Developers often use preprocessors to implement variability in software families, even though they might induce to errors (4, 5, 15, 16).

In this work, we refer to bug as a fault, that is, an incorrect instruction in the software code, due to a developer mistake, that leads to an incorrect program state (17).

A variability bug is a fault that happens in some, but not all, feature configurations of a software family (17). A category of variability bugs is related to the *sharing* of elements such as variables and functions among features. Due to this sharing, a maintenance task in one feature might break another one (8). This might happen since there is no mutual agreement (18) between the developers. In this work, we refer to such sharing as a *feature dependency* between the involved features.

To better illustrate that feature dependencies may cause problems, in this chapter we present three scenarios of C program families containing actual variability bugs related to dependencies. First, we present an example of variability bug regarding an intraprocedural dependency (Section 2.1). Then, we present a variability bug related to a global dependency (Section 2.2). Next, we present a variability bug regarding an interprocedural dependency (Section 2.3). Finally, we summarize our findings on this topic (Section 2.4).

2.1 Scenario 1: Intraprocedural dependency

In this work we refer to *intraprocedural* dependencies when features share the same program element inside a function. For example, we may have a variable defined in a feature and used in another one.

To better define intraprocedural dependencies, in this paper we rely on the presence condition definition (17). The presence condition is a boolean formula that denotes the minimum subset of configurations in which a fragment of code is included in the conditional compilation (17). Thus, we have an intraprocedural dependency every time the definition of a local variable has a different presence condition than its use. We refer to every variable in this situation as *dependent variable*.

Figure 1 presents a code snippet from *GLib*,¹ a general-purpose utility library for applications written in C. The figure shows a modification made to the code, by including and removing specific lines, committed to a *Git* repository.² In the figure, the function `g_inet_address_new_from_string` parses a string containing an IP address. Inside this function, there is a call to `g_inet_address_get_type` (see line 10). To ensure that the compiler would not optimize away this function, the developer added a volatile variable,

¹ <https://developer.gnome.org/glib/>

² <https://git.gnome.org/browse/glib/>

`type` (see line 5), assigning the function return value to such a variable (by removing line 10 and adding line 11).

Figure 1 – Adding an intraprocedural dependency, causing a bug in *GLib*.

```

1. GInetAddress *g_inet_address_new_from_string (...) {
2.     #ifdef G_OS_WIN32
3.         struct sockaddr_storage sa;
4.         ...
+ 5.     volatile GType type;
6.     gint len;
7.     #else /* !G_OS_WIN32 */
8.         ...
9.     #endif
- 10.    (void) g_inet_address_get_type ();
+ 11.    type = g_inet_address_get_type ();
12.    ...
13. }

```

+ Including line - Removing line

Source: Author's own elaboration.

The problem is that the definition of `type` is inside an `#ifdef` block, and therefore it is accessible only when we define `G_OS_WIN32`. Notice that the developer introduced an intraprocedural dependency for the variable `type`. We say that the direction of this dependency is optional-to-mandatory, as the presence condition of the variable definition (see line 5) is `G_OS_WIN32`, whereas the presence condition of the variable use (see line 11) is `true`. In case we do not set `G_OS_WIN32`, that is, in a non-*Windows* system, we get an undefined variable error for the variable `type` and cannot compile the code.

Figure 2 shows a new modification to the function, in order to fix this variability bug.³ To do so, the developer relocated⁴ the `type` variable definition to a mandatory portion of code (by removing line 6 and adding line 2). This modification makes the presence condition of both variable definition (see line 2) and its use (see line 11) the same, ceasing the dependency.

In this work, we consider two points for a dependency: the *maintenance point* and the *impact point*. For intraprocedural dependencies, we define a maintenance point as the point where we can change the name, type, or value of a dependent variable. Thus, a variable definition or assignment are possible maintenance points. The impact points are the points we can affect by changing the maintenance point, or, in other words, where the dependent variable is later referenced. Notice that a maintenance point, such as a variable assignment, can also be an impact point, regarding a previous maintenance point. Moreover, to have a dependency, the presence condition of the maintenance point must be different than the impact point. In Figure 1 we have an example of maintenance point at line 5 and an impact point at line 11.

³ https://bugzilla.gnome.org/show_bug.cgi?id=580750

⁴ <https://github.com/GNOME/glib/commit/97fe421518139dcb3477209d3d3c3b6744f54153>

Figure 2 – Removing the dependency to fix the bug in *GLib*.

```

1. GInetAddress *g_inet_address_new_from_string (...) {
+ 2. volatile GType type;
3. #ifdef G_OS_WIN32
4.     struct sockaddr_storage sa;
5.     ...
- 6. volatile GType type;
7.     gint len;
8. #else /* !G_OS_WIN32 */
9.     ...
10. #endif
11.     type = g_inet_address_get_type ();
12.     ...
13. }

```

+ Including line
- Removing line

Source: Author's own elaboration.

2.2 Scenario 2: Global dependency

Dependencies often transcend the boundaries of a function. A *global* dependency is similar to an intraprocedural dependency, except that the dependent variable is global, not local. In a global dependency the global variable appears outside a function and is used within a function. As we can define a global variable in a different file from where we use it, we might overlook these dependencies.

For instance, *libxml2*⁵ is a XML parser written in C. Figure 3 presents a code snippet of the *libxml2* software family. The figure depicts a modification made to the code, committed to a *Git* repository.⁶ In the figure we have a global variable, `xmlout` (see line 3). A preprocessor conditional directive (`#if defined(HTML) || defined(VALID)`) surrounds its definition, which means the variable `xmlout` is available if we define at least one of the macros. Notice that the developer added lines 10 and 11 to the code, using the `xmlout` variable inside the function `parseAndPrintFile` in the mandatory feature (see line 10). As there is no `#ifdef` encompassing the `xmlout` use, we have a global dependency for this variable. The direction of this dependency is again optional-to-mandatory, as the presence condition of the variable definition (the maintenance point) is `HTML || VALID` while the presence condition of its use (the impact point) is `true`.

This dependency triggers a bug⁷ if we do not define any of the macros (`HTML` or `VALID`), as we still reference the variable `xmlout` in the function `parseAndPrintFile` while it is undefined. Figure 4 presents another modification to the code, aiming to solve this variability bug. In the figure, the developer included⁸ the same conditional directive of the variable definition to its use (by adding lines 10 and 13).

⁵ <http://xmlsoft.org/>
⁶ <https://git.gnome.org/browse/libxml2/>
⁷ https://bugzilla.gnome.org/show_bug.cgi?id=611806
⁸ <https://goo.gl/gACFs6>

Figure 3 – Adding a new save option, creating a global dependency and causing a variability bug in *libxml2*.

```

1.  #if defined(HTML) || defined(VALID)
2.  ...
3.  static int xmlout = 0;
4.  #endif
5.  ...
6.  static void parseAndPrintFile(...) {
7.  ...
8.  if (format)
9.      saveOpts |= XML_SAVE_FORMAT;
+ 10. if (xmlout)
+ 11.     saveOpts |= XML_SAVE_AS_XML;
12. ...
13. }

```

+ Including line

Source: Author's own elaboration.

Figure 4 – Removing the dependency to fix the bug in *libxml2*.

```

1.  #if defined(HTML) || defined(VALID)
2.  ...
3.  static int xmlout = 0;
4.  #endif
5.  ...
6.  static void parseAndPrintFile(...) {
7.  ...
8.  if (format)
9.      saveOpts |= XML_SAVE_FORMAT;
+ 10. #if defined(HTML) || defined(VALID)
11.     if (xmlout)
12.         saveOpts |= XML_SAVE_AS_XML;
+ 13. #endif
14. ...
15. }

```

+ Including line

Source: Author's own elaboration.

2.3 Scenario 3: Interprocedural dependency

We refer to *interprocedural* dependencies when features share data among different functions. Consider two functions, f and g . Function f calls g passing x as an argument. If g uses data from x in an different feature than the feature associated to the g call in f , we have an interprocedural dependency. In this case, maintaining the argument of the call to g , for instance, by changing its value, we might break a feature at the points where function g references x .

Figure 5 presents a code snippet from *Lustre*,⁹ a parallel distributed file system for high-performance cluster computing. The figure depicts a modification made to the code, committed to a Git repository.¹⁰ In the figure, the developer added an `#ifdef` block (see lines 4-7) containing a reference to the parameter `nd` (see line 5), which is a pointer to a

⁹ <http://www.lustre.org>

¹⁰ <http://git.whamcloud.com/>

Figure 5 – Adding an interprocedural dependency in *Lustre*.

```

1. int ll_revalidate_nd(struct dentry *dentry,
2.                     struct nameidata *nd) {
3.     ...
+ 4. #ifdef LOOKUP_RCU
+ 5.     if (nd->flags & LOOKUP_RCU)
+ 6.         return -ECHILD;
+ 7. #endif
8.     ...
9. }

```

+ Including line

Source: Author's own elaboration.

struct of type `nameidata`. Developers reported a null pointer dereference bug¹¹ regarding the `nd` parameter. When calling the function `ll_revalidate_nd`, we may face a null pointer dereference accessing `nd->flags` if `nd` is null.

Figure 6 – Fixing the possible null pointer dereference in *Lustre*.

```

1. #ifdef HAVE_IOP_ATOMIC_OPEN
2. int ll_revalidate_nd(struct dentry *dentry,
3.                     unsigned int flags) {
4.     ...
5. }
6. #else /* !HAVE_IOP_ATOMIC_OPEN */
7. int ll_revalidate_nd(struct dentry *dentry,
8.                     struct nameidata *nd) {
9.     ...
10. #ifndef HAVE_DCACHE_LOCK
- 11. if (nd->flags & LOOKUP_RCU)
+ 12. if (nd && (nd->flags & LOOKUP_RCU))
13.     return -ECHILD;
14. #endif
15. ...
16. }
17. #endif /* HAVE_IOP_ATOMIC_OPEN */

```

+ Including line - Removing line

Source: Author's own elaboration.

To solve this problem, the program now checks¹² if `nd` is null, right before accessing `nd->flags` (see Figure 6, line 12). Despite its severity, this bug remained undetected for more than one year. This is because the problematic line of code is guarded by a macro and is only accessible on Linux kernel versions 2.6.38 and up. On older kernels, the code is innocuous. In other words, this variability bug occurs only in configurations that exist on newer versions of Linux kernel.

To verify the existence of an interprocedural dependency in this code, we must check the calls to the function `ll_revalidate_nd`. Although there are no calls to this function in the *Lustre* code, we can find them in the *Linux* kernel. Figure 7 shows such an indirect call in line 26. In this case, the field `d_op` of struct `dentry` corresponds to the struct

¹¹ <<https://jira.hpdd.intel.com/browse/LU-3483>>

¹² <<http://review.whamcloud.com/#/c/6715/5/lustre/llite/dcache.c,cm>>

Figure 7 – Code snippet from *Linux kernel*.

```

1. struct dentry *lookup_one_len(...) {
2.     ...
3.     struct qstr this;
4.     ...
5.     return __lookup_hash(&this, base, NULL);
6. }
7.
8. static struct dentry *__lookup_hash(struct qstr *name,
9.                                     struct dentry *base,
10.                                    struct nameidata *nd) {
11.     ...
12.     struct dentry *dentry;
13.     ...
14.     dentry = do_revalidate(dentry, nd);
15.     ...
16. }
17.
18. static struct dentry *do_revalidate(struct dentry *dentry,
19.                                     struct nameidata *nd) {
20.     int status = d_revalidate(dentry, nd);
21.     ...
22. }
23.
24. static inline int d_revalidate(struct dentry *dentry,
25.                                struct nameidata *nd) {
26.     return dentry->d_op->d_revalidate(dentry, nd);
27. }

```

Source: Author's own elaboration.

`ll_d_ops` (Figure 8). Therefore, `dentry->d_op->d_revalidate` points to the function `ll_revalidate_nd` (see line 2 in Figure 8). Notice that, even though this dependency does not directly cause this variability bug, it might delay the detection and further correction of this bug. For instance, if there was no `#ifdef` encompassing the reference to the parameter `nd` (see Figure 5, line 5), this problem would occur in every implementation, probably being more noticeable.

Figure 8 – Code snippet from *Lustre*.

```

1. struct dentry_operations ll_d_ops = {
2.     .d_revalidate = ll_revalidate_nd,
3.     .d_release = ll_release,
4.     .d_delete = ll_ddelete,
5.     .d_iput = ll_d_iput,
6.     .d_compare = ll_dcompare,
7. };

```

Source: Author's own elaboration.

As this call is in a mandatory section of code and we access the parameter `nd` in an optional feature (see Figure 5, line 5), we have an mandatory-to-optional interprocedural dependency. The presence condition of `nd` at first was `LOOKUP_RCU` (see Figure 5, line 4), but further modifications changed the presence condition to `(!HAVE_IOP_ATOMIC_OPEN && !HAVE_DCACHE_LOCK)` (see Figure 6, lines 6 and 10).

Analogously to the intraprocedural and global dependencies, in which the dependent variable initialization is a possible maintenance point, we consider the function call as a maintenance point regarding an interprocedural dependency, as its arguments initialize

the function formal parameters. Thus, a maintenance task in a function call, such as an argument change, might impact the corresponding parameter use inside the callee function (the impact point). In this example, we have a maintenance point at Figure 7, line 26, and an impact point in Figure 5, line 5. When such points are in different files, or, in this case, in different projects, detecting these dependencies can be more difficult.

Moreover, a function call argument may come from another function. In Figure 7, the null problematic value originates in the function `lookup_one_len`, as an argument when calling the function `__lookup_hash` (see line 5). This argument initializes the parameter `nd` (line 10), which also passes it through the function `do_revalidate` (see line 14) before finally reaching the function `d_revalidate` (see line 20). Function `d_revalidate` includes it as an argument of the call to `dentry->d_op->d_revalidate` (see line 26). We refer to the total of chained function calls that share the same data regarding an interprocedural dependency as the *dependency depth*. In this example, the depth is four, as the null value (Figure 7, line 5) passes through four function calls before the function `ll_revalidate_nd` references it in a different configuration. Interprocedural dependencies with high depths might require more attention from the developer. As there are more functions to consider when maintaining a feature, such dependencies are easier to miss, facilitating the introduction of bugs.

2.4 Summary

Bugs in general contribute to decrease developers productivity and impair software quality. Tasks like submitting bug reports, triaging bugs, developing patches, committing changes to the repository, validating patches, and updating documentation demand time and effort, even for simple bugs (19, 20).

Bugs regarding feature dependencies are even more difficult to deal with, since they might occur only in specific configurations, possibly delaying their detection and subsequent fix. Moreover, global and interprocedural dependencies are particularly problematic, as different features might share data from a variable between different files. The bug in Section 2.3 is rather complicated because it involves two different projects (*Lustre* and *Linux*). The maintenance of a variable in a mandatory section of *Lustre* causes a bug when *Linux* references it in an optional feature. Fixing a bug like this involves coordinating teams of developers of both projects.

This section introduces variability bugs related to three types of feature dependencies. To assess how often these dependencies occur in practice, we present next an empirical study to answer research questions on this topic (Chapter 3). Then, we present another empirical study to better understand how feature dependencies might lead to variability bugs (Chapter 4).

3 FEATURE DEPENDENCIES IN THE WILD

In this chapter we present an empirical study to assess fine-grained feature dependencies in practice. First, we report our empirical study of 40 popular software families, presenting the research questions we address, and our approach to identify feature dependencies in those families (Section 3.1). Then, we present and discuss the results, as well as the threats to validity of our study (Section 3.2).

3.1 Study Settings

In this section we present the settings of our study to investigate feature dependencies on software families. Our study covers 40 C industrial program families. We select these families inspired by previous work (6, 7, 8, 9, 10, 11). We consider this selection to be very appropriate, because it includes well-known families used in industrial practice. Moreover, these families comprise different domains, such as operating systems, databases, text editors, and web servers, varying from small to mid sizes. Finally, all of them contain several features implemented using preprocessor directives, which is a prerequisite for our dependency detection technique. To structure our research, we use the Goal, Question, and Metrics (21) approach.

3.1.1 Goal, Question, and Metrics

The goal of this empirical study is to investigate to what extent feature dependencies occur in practice, their types, and how they might impact maintenance tasks on product families.

To achieve this goal, this study addresses the following research questions:

- **Question 1:** how often do program families contain intraprocedural dependencies?
- **Question 2:** how often do program families contain global dependencies?
- **Question 3:** how often do program families contain interprocedural dependencies?
- **Question 4:** how often do dependencies of different directions (mandatory-to-optional, optional-to-mandatory, and optional-to-optional) occur?
- **Question 5:** what is the dependency depth distribution for interprocedural dependencies?

To answer **Question 1**, we count the number of functions with preprocessor directives, such as `#ifdef`, `#elif` or `#else`, and the number of functions with intraprocedural

dependencies for each family. These metrics allow us to calculate how often functions with preprocessor directives have intraprocedural dependencies.

To answer **Question 2**, we count the functions with impact points regarding global dependencies, that is, direct references to global variables in a different feature than its definition. We do not consider global variable assignments as maintenance points when they are inside a function, because we are unable to track the dataflow of global variables across functions, as we shall see in Section 3.2.6.

To answer **Question 3**, as interprocedural dependencies directly involves two functions, we count separately the number of functions containing maintenance points and the number of functions containing impact points regarding interprocedural dependencies for each family. Notice that the same function may contain both maintenance points and impact points, regarding distinct interprocedural dependencies, so these values may overlap. We also count the number of functions containing either maintenance points or impact points, to total how many functions contribute to interprocedural dependencies.

To answer **Question 4**, we classify every dependency based on its direction (mandatory-to-optional, optional-to-mandatory, or optional-to-optional). Next we count the occurrences of dependencies for each direction. We use this metric in order to verify if some direction is particularly common.

To answer **Question 5**, we create a call graph (22), having functions as nodes, and function calls as arcs pointing the callee function to the caller function. We also consider arguments placement and the presence condition of each function call when creating the graph. We use the well-known depth-first search (23) algorithm to traverse the graph. We make an adjustment in the algorithm to allow revisiting nodes (but avoiding loops), in order to track all possible paths (from the shortest to the longest) from every node regarding functions with interprocedural dependencies. This information is important to foresee situations where an argument in a function call may cause a problem due to an existing dependency later on the code. Also, answering this question is important to better set up dataflow analysis tools, such as Emergo (2) (see Section 5.1).

To better explain the metrics we compute, we refer to the code snippet in Figure 9. We extract this code snippet from *libssh*,¹ a multiplatform C library for SSH protocol implementations. The figure depicts five functions from three different files, *dh.c*, *packet.c*, and *packet1.c*. These functions handle SSH cryptography and packet sending over a SSH session. Function `ssh_crypto_init` initializes the values for the global variables `g` and `p`. Function `ssh_packet_send_unimplemented` calls function `packet_send`. Depending on the configuration regarding SSH protocol version (`WITH_SSH1` or `!WITH_SSH1`), `packet_send` may call either `packet_send1` or `packet_send2`. The code snippet also contains other four macros: `HAVE_LIBZ` and `WITH_LIBZ`, both related to *zlib*,² a compression library;

¹ <http://www.libssh.org/>

² <http://www.zlib.net/>

HAVE_LIBGCRYPT, related to *libgcrypt*,³ a cryptographic library; and DEBUG_CRYPT0, for debugging purposes.

In the figure, there is an intraprocedural dependency in the function `packet_send2` regarding variable `currentlen`, which is defined in a mandatory feature (see line 48) and later referenced in an optional configuration (`HAVE_LIBZ && WITH_LIBZ`, see line 52). Thus, the direction of this dependency is mandatory-to-optional.

Figure 9 – Code snippet from *libssh*.

```

1. static bignum g;
2. static bignum p;
3.
4. int ssh_crypto_init(void) {
5.     ...
6.     g = bignum_new();
7.     ...
8. #ifdef HAVE_LIBGCRYPT
9.     ...
10.    if (p == NULL) {
11.        bignum_free(g);
12.        g = NULL;
13.    }
14.    ...
15.    ...
16. #endif
17.    ...
18. }
19.
20. int ssh_packet_send_unimplemented(...) {
21.     int r;
22.     ...
23.     r = packet_send(session);
24.     ...
25. }
26.
27. int packet_send(ssh_session session) {
28. #ifdef WITH_SSH1
29.     if (session->version == 1) {
30.         return packet_send1(session);
31.     }
32. #endif
33.     return packet_send2(session);
34. }
35.
36. #ifdef WITH_SSH1
37. int packet_send1(ssh_session session) {
38.     ...
39. #ifdef DEBUG_CRYPT0
40.     ssh_print_hexa(..., ssh_buffer_get_begin(session->out_buffer), ...);
41. #endif
42.     ...
43. }
44. #endif
45.
46. static int packet_send2(ssh_session session) {
47.     ...
48.     uint32_t currentlen = ...;
49.     ...
50. #if defined(HAVE_LIBZ) && defined(WITH_LIBZ)
51.     ...
52.     currentlen = buffer_get_rest_len(session->out_buffer);
53.     ...
54. #endif
55.     ...
56. }

```

Source: Author's own elaboration.

There are two global variables, `g` and `p`, both declared in a mandatory section of code

³ <http://www.gnu.org/software/libgcrypt/>

(see lines 1 and 2) and referenced multiple times within the function `ssh_crypto_init`. We have global dependencies regarding both variables, as function `ssh_crypto_init` references them in an optional configuration (`HAVE_LIBGCRYPT`, see lines 10 – 12). The direction of these global dependencies is also mandatory-to-optional.

In addition, notice that there are four functions involved with interprocedural dependencies. Considering line 23 as a maintenance point, we may impact lines 29 and 30 in another function: `packet_send`. This happens due to the data from the `session` variable that flows out the function `ssh_packet_send_unimplemented` and flows into the function `packet_send`. As the impact points in `packet_send` are in an optional configuration (`WITH_SSH1`), we have two interprocedural dependencies involving both functions (as there are two distinct pairs of a maintenance point and an impact point). The direction of both dependencies is mandatory-to-optional. Notice that the reference to variable `session` at line 33 does not result in a dependency among features: both maintenance and impact points are in the mandatory feature. Furthermore, considering line 30 as another maintenance point, we may also impact line 40, implying in one more interprocedural dependency. Its direction is optional-to-optional, as both maintenance and impact points have different (and not `true`) presence conditions. Finally, when considering line 33 as a maintenance point, we may impact line 52, resulting in another mandatory-to-optional interprocedural dependency.

Regarding dependency depths, we track all paths across functions to interprocedural dependencies. For instance, we have two interprocedural dependencies involving functions `ssh_packet_send_unimplemented` and `packet_send`. From `ssh_packet_send_unimplemented` to `packet_send` the depth is one. As we have two interprocedural dependencies in these functions, we count this path (`ssh_packet_send_unimplemented` → `packet_send`) twice. Furthermore, we have another interprocedural dependency involving functions `packet_send` and `packet_send1`, and a last one involving functions `packet_send` and `packet_send2`. In these dependencies, the maintenance points are within function `packet_send`, at lines 30 and 33. In both cases, the value of the argument of these function calls comes from another function: `packet_send_unimplemented`. Hence, besides the obvious paths `packet_send` → `packet_send1` and `packet_send` → `packet_send2`, both with a depth of one, we also consider `packet_send_unimplemented` → `packet_send` → `packet_send1` and `packet_send_unimplemented` → `packet_send` → `packet_send2`, with a depth of two. In short, the average dependency depth for this example is $(1 + 1 + 1 + 1 + 2 + 2)/6 = 1.33$.

Table 1 summarizes the metrics we compute for this example.

3.1.2 Instrumentation

To compute the metrics we consider, we rely on *TypeChef* (12), a variability-aware type checking utility, to create an *Abstract Syntax Tree* (AST) from each source file.

Table 1 – Metrics summary for the code snippet in Figure 9.

Question	Metric	Value
Question 1	Functions with preprocessor directives	4 (80%)
	Functions with intraprocedural dependencies	1 (20%)
Question 2	Functions with global dependencies	1 (20%)
	Functions with maintenance points regarding interprocedural dependencies	2 (40%)
Question 3	Functions with impact points regarding interprocedural dependencies	3 (60%)
	Functions with either maintenance or impact points regarding interprocedural dependencies	4 (80%)
Question 4	Mandatory-to-optional	7 (87.5%)
	Optional-to-mandatory	0 (0%)
	Optional-to-optional	1 (12.5%)
Question 5	Average depth	1.33
	Standard deviation	0.47

Source: Author’s own elaboration.

TypeChef can parse C code containing `#ifdef` directives without generating all possible variants; instead, it creates an AST that preserves all variability information, having each preprocessor directive as a node in the tree. Previous studies (6, 8, 11) use *srcML* (24) to create ASTs represented in the XML format, but this tool usually fails when handling code with non-disciplined annotations (11). Thus, it generates ill-formed XML files as a result. Therefore, we believe that *TypeChef* is a better solution for this task.

In this work, we use *TypeChef* version 0.3.5 to create the ASTs for all program families source code files. We develop a tool to automate both ASTs creation and dependencies computation. We use Java SE 7 to implement this tool.

3.1.3 Operation

We perform the empirical study using a 2 GHz quad-core Intel Core i7-2630QM with 8 GB of RAM, running MS Windows 7 Home Premium SP1 64-bit. We divide this study in two parts: dependency identification and dependency depth analysis. In the first part of the study, our tool analyzes the ASTs generated for all program families source code files, one at a time, searching for intraprocedural, global, and interprocedural dependencies in all functions of each family. We simplistically describe this strategy in Algorithm 1. Notice that we present a simpler version of the actual algorithm, to better explain its operation. Thus, it lacks any optimizations in favor of understandability.

In the algorithm we traverse all ASTs for a program family. For each AST, we look into all functions, searching for variable references. For each variable, we search all uses within the function. For each variable use, we verify the variable scope. If it is a function parameter, we investigate all function calls in all ASTs, and compare the presence condition of each function call against the presence condition of the variable use. Every time such presence conditions differ, we have an interprocedural dependency. Now, if the variable is not a function parameter, it must be either a local or a global variable. So, we get all variable definitions (declarations, assignments, and increments/decrements) inside the current function, comparing their presence condition against the presence condition of the variable use. Again, every time a definition and an use have different presence

Algorithm 1 General algorithm for dependency search

```

ASTS  $\leftarrow$  set of all abstract syntax trees of a program family
FUNCTIONS  $\leftarrow$  set of all function definitions within the current AST
VARIABLES  $\leftarrow$  set of all variables used within the current function
CALLS  $\leftarrow$  set of all function calls to the current function
USES( $v$ )  $\leftarrow$  function that returns the set of all uses of the variable  $v$ 
DEFINITIONS( $v$ )  $\leftarrow$  function that returns the set of all definitions of the variable  $v$ 
IS_LOCAL( $v$ )  $\leftarrow$  function that returns true, if the variable  $v$  is a local variable; false, otherwise
IS_GLOBAL( $v$ )  $\leftarrow$  function that returns true, if the variable  $v$  is a global variable; false, otherwise
IS_PARAMETER( $v$ )  $\leftarrow$  function that returns true, if the variable  $v$  is a function parameter; false, otherwise
PC( $s$ )  $\leftarrow$  function that returns the presence condition of statement  $s$ 

1: for each ast in ASTS do
2:   for each function in FUNCTIONS do
3:     for each variable in VARIABLES do
4:       for each use in USES(variable) do
5:         if IS_PARAMETER(variable) then
6:           for all call in CALLS do
7:             if PC(use)  $\neq$  PC(call) then
8:               — There is an interprocedural dependency
9:             end if
10:          end for
11:         else  $\triangleright$  The variable is either local or global
12:           for each definition in DEFINITIONS(variable) do
13:             if PC(definition)  $\neq$  PC(use) then
14:               if IS_LOCAL(variable) then
15:                 — There is an intraprocedural dependency
16:               else if IS_GLOBAL(variable) then
17:                 — There is a global dependency
18:               end if
19:             end if
20:           end for
21:         end if
22:       end for
23:     end for
24:   end for
25: end for

```

conditions, we have a dependency, that can be intraprocedural or global, depending on the variable scope.

In the second part of the study, our tool analyzes each function call which is a maintenance point to an interprocedural dependency to find out the maximum dependency depth. To do so, we check whether the function call argument comes from another function, thus being a parameter of the current function. If so, we then analyze the caller function, in a recursive manner.

Next, we interpret and discuss the results of our empirical study to assess fine-grained feature dependencies.

3.2 Results and discussion

In this section, we answer the research questions based on the results of our first empirical study and present the threats to validity. All results are available at the companion web site.⁴

⁴ <http://www.iranrodrigues.com.br/masterthesis>

3.2.1 Question 1: How often do program families contain intraprocedural dependencies?

To answer this question, we use the *number of functions with preprocessor directives* (FDi) and the *number of functions with intraprocedural dependencies* ($FIntra$). Table 2 shows FDi and $FIntra$ for all families, expressed as a percentage of the total of functions (NoF) we analyze. According to the table, both metrics differ considerably depending on the family we analyze. For instance, only 1.36% of *libsoup* functions have preprocessor directives (FDi), while *Vim* have preprocessor directives in 37.25% of its functions. Following the convention “average \pm standard deviation”, our results show that $10.09\% \pm 8.63\%$ of the functions have preprocessor directives within.

The same variation occurs for $FIntra$. While *mpsolve* has no intraprocedural dependencies, $FIntra$ reaches 17.04% on *libxml2*, and 18.77% on *Vim*. Notice that the values for $FIntra$ are rather low because we consider the number of functions with dependencies over the total of functions. If we consider only the number of functions with directives, this number grows substantially. For instance, *mptris* family has intraprocedural dependencies in 14.14% of its functions. However, 82.35% of its functions with preprocessor directives also have intraprocedural dependencies. We express this ratio as the column $FIntra/FDi$ in Table 2, which stands for $FIntra$ divided by FDi . Another interesting example is Lua: only one function (0.12% out of the total) has preprocessor directives. But, this function also contains intraprocedural dependencies, meaning that 100% of Lua functions with preprocessor directives also have intraprocedural dependencies. The division is more meaningful than $FIntra$ alone, because only functions with directives can possibly have intraprocedural dependencies. This way, when maintaining code with preprocessor directives, the likelihood of finding a dependency increases. Our data shows that $51.44\% \pm 17.77\%$ of the functions with directives also have intraprocedural dependencies. Therefore, intraprocedural dependencies are rather common in the product families we analyze, confirming the previous study (8) results, in the sense that the majority of functions with preprocessor directives have such dependencies.

3.2.2 Question 2: How often do program families contain global dependencies?

To answer this question we use the *number of functions referencing global variables* ($FGRef$) and the *number of functions with global dependencies* ($FGlobal$). Table 3 shows that, once more, the results vary vastly depending on the family we analyze. Some families do not make much use of global variables. *Lua*, for instance, has only 4.06% of its functions referencing global variables ($FGRef$). On the other hand, *gzip* has a $FGRef$ of 68.42%, meaning that the majority of its functions references global variables. Our results show that $27.24\% \pm 14.24\%$ of the functions do reference global variables.

The number of functions with global dependencies ($FGlobal$) also vary across the families we analyze. According to the table, five of the families do not have any global

Table 2 – Intraprocedural dependencies in the program families.

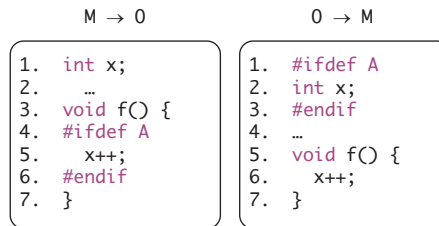
Family	Version	Application Domain	FDi	FIntra	FIntra/FDi	NoF
apache	2.4.3	web server	7.57%	3.58%	47.30%	3910
atlantis	0.0.2.1	operating system	4.27%	1.71%	40.00%	117
bash	2.01	command language interpreter	13.42%	5.89%	43.89%	1647
bc	1.03	calculator	2.41%	0.60%	25.00%	166
berkeley db	4.7.25	database system	11.01%	7.87%	71.47%	3468
bison	2.0	parser generator	2.19%	1.17%	53.33%	684
cherokee	1.2.101	web server	8.11%	3.97%	48.99%	1838
clamav	0.97.6	antivirus	13.37%	7.05%	52.71%	2072
cvs	1.11.21	revision control system	7.66%	4.46%	58.14%	1122
dia	0.96.1	diagramming software	2.33%	1.72%	73.68%	814
expat	2.1.0	XML library	5.71%	1.66%	29.03%	543
flex	2.5.35	lexical analyzer	6.14%	1.44%	23.53%	277
fvwm	2.4.15	window manager	6.45%	3.27%	50.72%	2141
gawk	3.1.4	GAWK interpreter	12.21%	6.58%	53.85%	745
gnuchess	5.06	chess engine	1.84%	0.92%	50.00%	217
gnuplot	4.6.1	plotting tool	12.84%	6.23%	48.54%	1861
gzip	1.2.4	file compressor	21.93%	13.16%	60.00%	114
irssi	0.8.15	IRC client	2.17%	0.53%	24.19%	2853
kin	0.5	database system	6.01%	3.85%	64.00%	1248
libdsmcc	0.6	DVB library	2.00%	1.00%	50.00%	100
libieee	0.2.11	IEEE standards for VHDL library	12.18%	2.54%	20.83%	197
libpng	1.0.60	PNG library	27.31%	13.87%	50.77%	476
libsoup	2.41.1	HTTP library	1.36%	0.61%	45.00%	1475
libssh	0.5.3	SSH library	10.82%	5.51%	50.98%	943
libxml2	2.9.0	XML library	23.91%	17.04%	71.26%	6009
lighttpd	1.4.30	web server	16.73%	10.56%	63.10%	1004
lua	5.2.1	programming language	0.12%	0.12%	100.00%	837
lynx	2.8.7	web browser	29.07%	15.47%	53.21%	1448
m4	1.4.4	macro expander	9.72%	4.63%	47.62%	216
mpsolve	2.2	mathematical software	1.95%	0.00%	0.00%	411
mptris	1.9	game	17.17%	14.14%	82.35%	99
prc-tools	2.3	C/C++ library for palm OS	3.52%	2.71%	76.92%	369
privoxy	3.0.19	proxy server	21.55%	14.02%	65.05%	478
rcs	5.7	revision control system	2.34%	1.00%	42.86%	299
sendmail	8.14.6	mail transfer agent	7.67%	3.72%	48.48%	861
sqlite	3.7.15.3	database system	16.85%	8.81%	52.27%	2612
sylpheed	3.3.0	e-mail client	3.50%	1.85%	52.75%	2597
vim	7.3	text editor	37.25%	18.77%	50.38%	5600
xfig	3.2.3	vector graphics editor	2.96%	1.60%	54.00%	1689
xterm	2.9.1	terminal emulator	8.09%	4.95%	61.25%	989

Notes: **FDi**: % of functions with preprocessor directives; **FIntra**: % of functions with intraprocedural dependencies; **NoF**: Number of functions.

Source: Author's own elaboration.

dependencies: *Atlantis*, *bc*, *Expat*, *libsoup*, and *Lua*. Families with the highest values for *FGlobal* include *Vim* (15.89%) and *libxml2* (15.14%). However, these percentages relate to the total of functions (*NoF*), for instance, from all *Vim* functions, 15.89% have global dependencies. We cannot restrict this number to consider only functions with preprocessor directives, as we do in Section 3.2.1. This is because global dependencies may occur even in a function without any preprocessor directive, simply by declaring a global variable in an optional feature and referencing within such a function, in a mandatory feature. Figure 10 illustrates two functions with global dependencies. While the function in the left-hand side of the figure contains a preprocessor directive (`#ifdef`), the right-hand side shows a function without any directives.

Now, if we consider only functions which reference global variables, we can better

Figure 10 – Global dependencies of different directions.

Source: Author's own elaboration.

estimate the global dependency occurrence. For instance, *mptris* has global dependencies in 8.08% of its functions. But, when considering only the functions which reference global variables, 42.11% of them have global dependencies (see column *FGlobal/FGRef* in Table 3, which stands for *FGlobal* divided by *FGRef*). Our results show that $11.90\% \pm 12.20\%$ of the functions which refer to global variables also have global dependencies. Therefore, we conclude that this type of dependency is less common in the families we analyze. Nevertheless, this value is a lower bound. As we do not track the dataflow of global variables across functions, we cannot consider all possible maintenance points of a global variable, such as assignments or increments/decrements, that may happen inside functions. We further discuss this limitation in Section 3.2.6.

Moreover, we cannot neglect such dependencies, because depending on the family, the total of global dependencies may be reasonably higher. Besides, Section 2.2 shows that this type of dependency can be as problematic as any other dependency. Additionally, such dependencies might be hidden as different files can refer to the same global variable.

3.2.3 Question 3: How often do program families contain interprocedural dependencies?

To answer this question, we use the *number of functions with maintenance points regarding interprocedural dependencies (FM)*, the *number of functions with impact points regarding interprocedural dependencies (FI)*, and the *number of functions with interprocedural dependencies (FInter)*. As an interprocedural dependency involves two functions, one containing a maintenance point and the other containing an impact point, we refer to functions with interprocedural dependencies (*FInter*) as the functions containing either a maintenance point or an impact point regarding interprocedural dependencies. Table 4 shows the values for *FM*, *FI*, and *FInter* for the families we analyze. Not surprisingly, again these values vary significantly across the families. *FM*, for instance, ranges from 0.12% to 56.69%, in *Lua* and *Privoxy*, respectively. Considering the families we analyze, $18.96\% \pm 16.41\%$ of the functions have maintenance points regarding interprocedural dependencies. In other words, this is the number of functions containing function calls that lead to interprocedural dependencies.

In most families we analyze, *FI* is lower than *FM*, with some exceptions. Curiously,

Table 3 – Global dependencies in the program families.

Family	Version	Application Domain	FGRef	FGlobal	FGlobal/FGRef	NoF
apache	2.4.3	web server	30.74%	1.30%	4.24%	3910
atlantis	0.0.2.1	operating system	17.95%	0.00%	0.00%	117
bash	2.01	command language interpreter	51.00%	7.65%	15.00%	1647
bc	1.03	calculator	27.11%	0.00%	0.00%	166
berkeley db	4.7.25	database system	8.39%	0.95%	11.34%	3468
bison	2.0	parser generator	35.38%	0.73%	2.07%	684
cherokee	1.2.101	web server	5.98%	1.52%	25.45%	1838
clamav	0.97.6	antivirus	15.93%	1.64%	10.30%	2072
cvs	1.11.21	revision control system	30.30%	2.23%	7.35%	1122
dia	0.96.1	diagramming software	26.54%	0.37%	1.39%	814
expat	2.1.0	XML library	23.57%	0.00%	0.00%	543
flex	2.5.35	lexical analyzer	23.83%	1.44%	6.06%	277
fvwm	2.4.15	window manager	44.14%	2.29%	5.19%	2141
gawk	3.1.4	GAWK interpreter	30.60%	2.95%	9.65%	745
gnuchess	5.06	chess engine	35.94%	0.92%	2.56%	217
gnuplot	4.6.1	plotting tool	41.97%	6.45%	15.36%	1861
gzip	1.2.4	file compressor	68.42%	9.65%	14.10%	114
irssi	0.8.15	IRC client	26.95%	0.21%	0.78%	2853
kin	0.5	database system	26.84%	1.60%	5.97%	1248
libdsmcc	0.6	DVB library	5.00%	2.00%	40.00%	100
libieee	0.2.11	IEEE standards for VHDL library	26.90%	1.52%	5.66%	197
libpng	1.0.60	PNG library	11.34%	2.73%	24.07%	476
libsoup	2.41.1	HTTP library	12.47%	0.00%	0.00%	1475
libssh	0.5.3	SSH library	8.38%	1.17%	13.92%	943
libxml2	2.9.0	XML library	37.11%	15.14%	40.81%	6009
lighttpd	1.4.30	web server	12.55%	1.00%	7.94%	1004
lua	5.2.1	programming language	4.06%	0.00%	0.00%	837
lynx	2.8.7	web browser	37.22%	10.70%	28.76%	1448
m4	1.4.4	macro expander	38.43%	6.48%	16.87%	216
mpsolve	2.2	mathematical software	10.46%	1.46%	13.95%	411
mptris	1.9	game	19.19%	8.08%	42.11%	99
prc-tools	2.3	C/C++ library for palm OS	43.36%	0.81%	1.88%	369
privoxy	3.0.19	proxy server	22.38%	4.81%	21.50%	478
rcs	5.7	revision control system	40.80%	1.00%	2.46%	299
sendmail	8.14.6	mail transfer agent	31.24%	1.97%	6.32%	861
sqlite	3.7.15.3	database system	20.06%	3.25%	16.22%	2612
sympheed	3.3.0	e-mail client	28.30%	1.04%	3.67%	2597
vim	7.3	text editor	38.57%	15.89%	41.20%	5600
xfig	3.2.3	vector graphics editor	52.40%	1.12%	2.15%	1689
xterm	2.9.1	terminal emulator	17.80%	1.72%	9.66%	989

Notes: **FGRef**: % of functions referencing global variables; **FGlobal**: % of functions with global dependencies; **NoF**: Number of functions.

Source: Author's own elaboration.

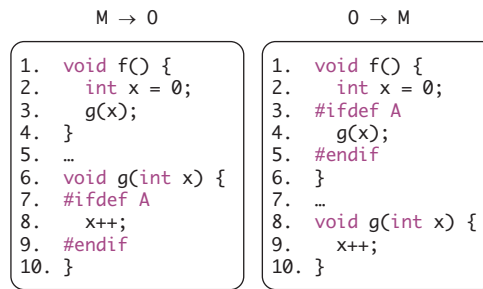
Lua has the same value for *FM* and *FI*: 0.12%, which is also the smallest value for the *FI* in all families. The highest *FI* is in *libpng*, where 42.65% of its functions contain impact points regarding interprocedural dependencies. Our data reveal that $12.14\% \pm 10.46\%$ of the functions references dependent variables in interprocedural dependencies.

According to Table 4, the family with the lowest *FInter* is once again *Lua*, with 0.24% of its functions with interprocedural dependencies. On the other hand, *Vim* is the family with the highest number of functions with interprocedural dependencies: 67.34%. Considering all the families, the average number of functions with interprocedural dependencies is $25.98\% \pm 19.99\%$.

All these metrics consider all the functions (*NoF*) for each family. Once more, we cannot restrict them to consider only the functions with preprocessor directives. The

reason is that both maintenance points and impact points can be in a mandatory feature, provided that their counterparts are in optional features. To better explain this situation, consider Figure 11 as an example. In each side of the figure, we have an interprocedural dependency regarding functions `f` and `g`. In the left-hand side of the figure we have a maintenance point at line 3 in a mandatory feature, and an impact point at line 8 in an optional feature. In the right-hand side of the figure the situation is just the opposite, as the maintenance point at line 4 is now in an optional feature, while the impact point at line 9 is in a mandatory feature. Nevertheless, we conclude that interprocedural dependencies are reasonably common in the families we analyze. This may be a problem if developers are not aware of the existence of such dependencies in the code they maintain. Problems regarding interprocedural dependencies may involve more than one file, or even different software families (see Section 2.3), making maintenance tasks in the presence of such dependencies rather risky.

Figure 11 – Interprocedural dependencies of different directions.



Source: Author's own elaboration.

3.2.4 Question 4: How often do dependencies of different directions (mandatory-to-optional, optional-to-mandatory, and optional-to-optional) occur?

To answer this question, we use the *number of mandatory-to-optional dependencies* ($M \rightarrow O$), the *number of optional-to-mandatory dependencies* ($O \rightarrow M$), and the *number of optional-to-optional dependencies* ($O \rightarrow O$).

Figure 12 shows the distribution of dependency directions according to their types using a beanplot chart (25). Notice that the distribution of intraprocedural dependencies resembles the distribution of global dependencies, with pretty similar averages (the horizontal lines) and estimated density (the bean shape) for each bean. In both types, most of dependencies are mandatory-to-optional ($M \rightarrow O$), followed by optional-to-optional ($O \rightarrow O$) dependencies. Only few are optional-to-mandatory ($O \rightarrow M$). These results mean that developers create most of intraprocedural and global dependencies by defining local and global variables in a mandatory feature, and referencing them in an optional feature. In such cases, these dependencies do not cause build errors (regarding undeclared variables), but they can trigger compilation warnings regarding unused variables, which is

Table 4 – Interprocedural dependencies in the program families.

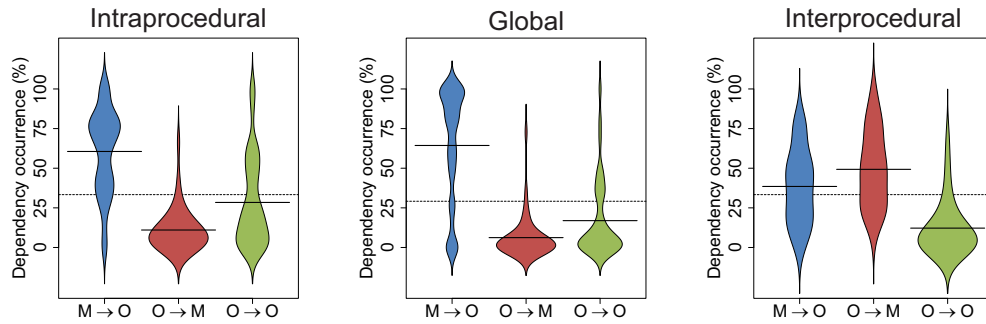
Family	Version	Application Domain	FM	FI	FInter	NoF
apache	2.4.3	web server	6.14%	5.01%	9.87%	3910
atlantis	0.0.2.1	operating system	2.56%	0.85%	2.56%	117
bash	2.01	command language interpreter	37.58%	19.98%	47.78%	1647
bc	1.03	calculator	10.84%	6.02%	15.06%	166
berkeley db	4.7.25	database system	55.82%	18.54%	62.72%	3468
bison	2.0	parser generator	2.92%	6.58%	8.48%	684
cherokee	1.2.101	web server	12.62%	9.85%	18.61%	1838
clamav	0.97.6	antivirus	17.18%	11.00%	23.65%	2072
cvs	1.11.21	revision control system	18.81%	14.35%	28.43%	1122
dia	0.96.1	diagramming software	2.33%	1.97%	4.05%	814
expat	2.1.0	XML library	3.68%	1.84%	5.52%	543
flex	2.5.35	lexical analyzer	1.08%	11.19%	12.27%	277
fvwm	2.4.15	window manager	9.29%	6.45%	14.20%	2141
gawk	3.1.4	GAWK interpreter	16.91%	8.86%	23.49%	745
gnuchess	5.06	chess engine	20.28%	4.61%	23.96%	217
gnuplot	4.6.1	plotting tool	31.22%	13.70%	40.03%	1861
gzip	1.2.4	file compressor	24.56%	15.79%	35.96%	114
irssi	0.8.15	IRC client	2.28%	1.68%	3.72%	2853
kin	0.5	database system	32.29%	5.69%	36.38%	1248
libdsmcc	0.6	DVB library	13.00%	15.00%	22.00%	100
libieee	0.2.11	IEEE standards for VHDL library	8.63%	7.61%	12.69%	197
libpng	1.0.60	PNG library	40.97%	42.65%	59.66%	476
libsoup	2.41.1	HTTP library	0.61%	0.41%	1.02%	1475
libssh	0.5.3	SSH library	35.63%	23.97%	50.69%	943
libxml2	2.9.0	XML library	44.02%	15.74%	52.89%	6009
lighttpd	1.4.30	web server	25.70%	20.22%	37.35%	1004
lua	5.2.1	programming language	0.12%	0.12%	0.24%	837
lynx	2.8.7	web browser	28.04%	24.31%	42.33%	1448
m4	1.4.4	macro expander	11.11%	9.72%	18.98%	216
mpsolve	2.2	mathematical software	0.97%	1.46%	2.43%	411
mptris	1.9	game	27.27%	27.27%	41.41%	99
prc-tools	2.3	C/C++ library for palm OS	12.20%	5.42%	17.34%	369
privoxy	3.0.19	proxy server	56.69%	24.69%	66.74%	478
rcs	5.7	revision control system	6.35%	6.69%	12.04%	299
sendmail	8.14.6	mail transfer agent	21.95%	8.48%	26.71%	861
sqlite	3.7.15.3	database system	44.83%	32.20%	59.49%	2612
sylpheed	3.3.0	e-mail client	4.74%	3.62%	7.78%	2597
vim	7.3	text editor	52.38%	39.88%	67.34%	5600
xfig	3.2.3	vector graphics editor	5.45%	3.02%	7.58%	1689
xterm	2.9.1	terminal emulator	9.50%	9.10%	15.77%	989

Notes: **FM**: % of functions with maintenance points regarding interprocedural dependencies; **FI**: % of functions with impact points regarding interprocedural dependencies; **FInter**: % of functions with interprocedural dependencies (that is, containing either maintenance or impact points); **NoF**: Number of functions.

Source: Author’s own elaboration.

a minor problem. We can also infer that in such types of dependency, when developers define variables in optional features, they reference such variables much more in another optional feature than in a mandatory one.

One might wonder if this situation can cause compilation errors. However, this depends on a number of factors, such as configuration parameters or the family feature model, for instance. To better explain this, we refer to the code snippet in Figure 13. This code contains intraprocedural dependencies regarding variables `dsa` and `bio`. The variable `dsa` has two definitions. The presence condition of its first definition (see line 3) is `HAVE_LIBGCRYPT`. The presence condition of its second definition (see line 6) is `!HAVE_LIBGCRYPT && HAVE_LIBCRYPTO`. Despite such definitions occurring only in op-

Figure 12 – Dependency directions distribution by type.

Source: Author's own elaboration.

tional features, the variable `dsa` is referenced in a mandatory section of the code (see line 17). If we do not define either `HAVE_LIBGCRYPT` or `HAVE_LIBCRYPTO` we would have an undefined variable error. Now, look at `bio` definition at line 8. Its presence condition is `!HAVE_LIBGCRYPT && HAVE_LIBCRYPTO`. This variable is later referenced at line 12, in a different presence condition: `HAVE_LIBCRYPTO`. Now, we would face a similar compilation error if we define both macros at once. In this case, *libssh* configure step prevents the occurrence of such errors, ensuring that either `HAVE_LIBGCRYPT` or `HAVE_LIBCRYPTO` is available (7).

Figure 13 – Code snippet from *libssh*.

```

1.  #ifdef HAVE_LIBGCRYPT
2.      ...
3.      gcry_sexp_t dsa = NULL;
4.      ...
5.  #elif defined HAVE_LIBCRYPTO
6.      DSA *dsa = NULL;
7.      ...
8.      BIO *bio = NULL;
9.  #endif
10.  ...
11.  #ifdef HAVE_LIBCRYPTO
12.      if (bio == NULL) {
13.          ...
14.      }
15.  #endif
16.  ...
17.      privkey->dsa_priv = dsa;

```

Source: Author's own elaboration.

The distribution of interprocedural dependencies is very different (see Figure 12). Regarding interprocedural dependencies, optional-to-mandatory is the most common direction, followed closely by mandatory-to-optional. Optional-to-optional dependencies are rather uncommon. These results show that developers introduce the majority of interprocedural dependencies by calling functions from optional features which reference their parameters in a mandatory feature. Besides, the opposite situation, that is, calling a function from a mandatory feature which references its parameters in optional features, is also common.

3.2.5 Question 5: What is the dependency depth distribution for interprocedural dependencies?

Table 5 summarizes the dependency depths for all interprocedural dependencies we collect. The *Max.* column in the table refers to the maximum depth we find in each family. In this sense, we can see that *bc*, *Flex*, and *MPSolve* do not have interprocedural dependencies with a depth greater than one. In such dependencies, functions share data directly from the caller function to the callee function. On the other hand, *Berkeley DB* and *libxml2* have surprisingly high values for the maximum dependency depth: 23. This means that *Berkeley DB*, for instance, shares data across 23 functions before reaching an impact point of a particular interprocedural dependency. According to Table 5, *Lua* has an even greater maximum depth: 29. However, our results for *Lua* are still indefinite. This particular family has a surprisingly high number of chained functions, which our tool cannot handle in our current equipment, due to memory constraints. Therefore, we limit the maximum number of paths of the call graph for *Lua* functions, thus obtaining a lower bound for the maximum depth. Considering all the families, the maximum dependency depth is 8.48 ± 6.59 . We also present the average (*Avg.*) and the standard deviation (*St. Dev.*) of the depths for all dependencies in each family. These data consider the depths of all possible paths to an interprocedural dependency. To better explain this metric, we refer to Figure 14. The code in the figure illustrates an interprocedural dependency involving functions *f* and *g*: the latter provides the argument (see line 8) which the former references in a different presence condition (see line 3). Between these functions there is a depth of one. Besides, function *h* provides the data to the function *g* (see line 12). Thus, from *h* to *f* we have a depth of two. Furthermore, functions *i* and *j* provide data to *h*, which will ultimately reach *f*. Now, we have two different paths with a depth of three ($i \rightarrow h \rightarrow g \rightarrow f$ and $j \rightarrow h \rightarrow g \rightarrow f$). In this example, the average depth is $(1 + 2 + 3 + 3)/4 = 2.25$.

The average depth we present in Table 5 varies depending on the family. For instance, *gnuplot* has an average depth of 1.57 ± 0.87 , which is a relatively low value, considering its maximum depth is 9. *xterm*, on the other side, has an average depth of 11.11 ± 3.16 , while its maximum depth is 22. Figure 15 shows individual histograms of dependency depths for each family we analyze. In such charts, the vertical dashed line indicates the average depth in the family. While most of families have the majority of interprocedural dependencies with a depth of one, some histograms look like a bell-shaped curve. Take *libxml2* as an example: most of its dependencies have depths above 10, while few have a depth of one.

We conclude that, although most of interprocedural dependencies have a depth of one, there is no single pattern that fits into all the families we analyze. Depending on the family, the developer may face dependencies of higher depths more or less often. High values for dependency depths may hinder developer work when maintaining such chained

Figure 14 – Different depths for a interprocedural dependency.

```

1. void f(int x) {
2.     #ifdef A
3.         x++;
4.     #endif
5. }
6.
7. void g(int x) {
8.     f(x);
9. }
10.
11. void h(int x) {
12.     g(x);
13. }
14.
15. void i(int x) {
16.     h(x);
17. }
18.
19. void j(int x) {
20.     h(x);
21. }

```

Source: Author's own elaboration.

functions, especially when the developer is unaware of the existence of those dependencies. In these cases, modifying a variable do not impact only the current function, but all the functions that use that variable from that point on. Moreover, the greater the depth, the harder it is for the developer detect the dependency, therefore, it is easier to introduce a bug. Introducing a bug in such way may hamper its posterior correction, since it may be difficult to trace it back to the source of the problem.

3.2.6 Threats to validity

Now, we present the threats to validity. To structure this section, we follow the Cook and Campbell validity system (26).

Construct Validity. We do not have access to the specification of valid configurations of the families we analyze. Thus, we cannot ensure that all the dependencies we find in our study arise in valid configurations.

Internal Validity. Our analysis of global dependencies is not exhaustive. We do not consider all possible maintenance points on global variables that may occur inside functions. This is because we cannot determine if a global variable reference (an impact point) in a function takes place before or after a particular assignment (a maintenance point) inside another function. As we do not track the dataflow of global variables across functions, we limit to consider only the variable definition as a possible maintenance point. Thus, we have a lower bound. The real number of global dependencies might be higher.

Also, our results for dependency depths for *Lua* are not complete. While the depth computation for every other family occurs without hassle, our tool cannot finish it for *Lua*. We attribute this problem to the high number of chained functions in *Lua*, as virtually all of its functions share a parameter regarding the state of Lua interpreter. Our tool

Table 5 – Interprocedural dependency depths in the program families.

Family	Version	Application Domain	Max.	Avg.	St. Dev.
apache	2.4.3	web server	7	2.48	1.42
atlantis	0.0.2.1	operating system	3	1.49	0.60
bash	2.01	command language interpreter	13	4.14	2.78
bc	1.03	calculator	1	1.00	0.00
berkeley db	4.7.25	database system	23	7.29	3.72
bison	2.0	parser generator	5	1.10	0.37
cherokee	1.2.101	web server	7	1.56	0.83
clamav	0.97.6	antivirus	12	5.51	2.37
cvs	1.11.21	revision control system	8	2.32	1.38
dia	0.96.1	diagramming software	2	1.26	0.44
expat	2.1.0	XML library	6	3.13	1.45
flex	2.5.35	lexical analyzer	1	1.00	0.00
fvwm	2.4.15	window manager	7	2.34	1.35
gawk	3.1.4	GAWK interpreter	9	3.09	1.68
gnuchess	5.06	chess engine	4	1.28	0.49
gnuplot	4.6.1	plotting tool	9	1.57	0.87
gzip	1.2.4	file compressor	3	1.41	0.64
irssi	0.8.15	IRC client	9	1.87	1.59
kin	0.5	database system	7	2.10	1.34
libdsmcc	0.6	DVB library	5	1.38	0.76
libieee	0.2.11	IEEE standards for VHDL library	3	1.73	0.83
libpng	1.0.60	PNG library	9	3.06	1.44
libsoup	2.41.1	HTTP library	3	1.29	0.64
libssh	0.5.3	SSH library	13	4.30	2.78
libxml2	2.9.0	XML library	23	11.03	3.32
lighttpd	1.4.30	web server	8	2.94	1.57
lua	5.2.1	programming language	29	23.40	11.20
lynx	2.8.7	web browser	11	4.44	1.83
m4	1.4.4	macro expander	4	1.31	0.64
mpsolve	2.2	mathematical software	1	1.00	0.00
mptris	1.9	game	4	1.41	0.68
prc-tools	2.3	C/C++ library for palm OS	4	1.60	0.75
privoxy	3.0.19	proxy server	6	2.05	0.94
rcs	5.7	revision control system	3	1.40	0.73
sendmail	8.14.6	mail transfer agent	10	4.83	2.54
sqlite	3.7.15.3	database system	19	7.34	2.94
sylpheed	3.3.0	e-mail client	4	1.48	0.59
vim	7.3	text editor	14	4.01	3.04
xfig	3.2.3	vector graphics editor	8	2.48	1.45
xterm	2.9.1	terminal emulator	22	11.11	3.16

Source: Author's own elaboration.

completes this task in a few minutes for most of the families. With *Lua*, on the other hand, our tool spends a week running without finishing its computations. To alleviate this threat, we limit the call graph size when computing depths for *Lua*, in order to get a lower bound, at least. Even though limited, dependency depths for *Lua* are the greatest among families we analyze.

Another point is that our tool does not know anything about the compiler linking process. Therefore, we cannot determine if a particular function accesses a given resource (such as a global variable or another function) in another file. Thus, we consider that every function can access any global variable and function simply by referencing it. This can be problematic if some program defines global variables or functions with duplicate names across its files, since our tool is unable to link these resources properly.

Our analysis depends on the *TypeChef* C parser, which generates an Abstract Syntax Tree (AST) for each source code file we provide. The resulting AST is not always

completely equivalent to the original code, that is, *TypeChef* may refactor a code before generating the AST. This is necessary since *TypeChef* cannot directly map some non-disciplined annotations to individual AST elements. We also find that *TypeChef* do not handle well `#ifdef` blocks that contain one or more `#elif` clauses and no `#else`, producing nodes with incorrect presence conditions. This is a minor problem, since we find that in all families we analyze, such situation only occur in 0.43% of annotations. Therefore, we may face false positives and false negatives in our results, since we analyze the ASTs, not the original files. Although, due to the low occurrence of such divergences, they could not significantly affect the results, thereby alleviating this threat.

Also, we rely on a previous technique (7) to restrict the analysis to program families code only, to make the analysis feasible, by excluding external libraries. In this approach, we remove all `#include` directives from the program family code. To prevent possible syntax errors caused by the suppression of these libraries, all needed macros and types are recreated in a separate header file. This process is semi-automatic, and hence, error-prone.

Finally, *TypeChef* cannot successfully generate the AST for all families files. Some files cause errors during the AST generation. We cannot determine if these errors are due to syntax errors in source code, a faulty header file, or if *TypeChef* simply cannot handle some C constructs. We present the rates of successful generation of ASTs for all families in Table 6. Our data show that *TypeChef* successfully parses 97.70% of all source code files we select in the study, which is an acceptable ratio. After a manual inspection of the files that *TypeChef* rejects, we conclude that they could not substantially change the results, thus alleviating this threat.

External Validity. In our study we analyze 40 C program families from different sizes and domains. These families are well-known in the industry. Their communities are very active, despite some of them exist for many years. Nevertheless, our results might not hold to other families, as some of them have very distinctive results. The high standard deviation values found in some metrics, for instance, the number of functions with global dependency, evidence the high variability in the results. For that reason, we should not use these results in direct comparison among different families.

Table 6 – Successful AST generation rate in the program families.

Family	Version	Application Domain	Successful AST Generation Rate
apache	2.4.3	web server	98.81%
atlantis	0.0.2.1	operating system	97.78%
bash	2.01	command language interpreter	100.00%
bc	1.03	calculator	100.00%
berkeley db	4.7.25	database system	99.78%
bison	2.0	parser generator	100.00%
cherokee	1.2.101	web server	96.55%
clamav	0.97.6	antivirus	96.15%
cvs	1.11.21	revision control system	84.85%
dia	0.96.1	diagramming software	93.85%
expat	2.1.0	XML library	100.00%
flex	2.5.35	lexical analyzer	100.00%
fvwm	2.4.15	window manager	100.00%
gawk	3.1.4	GAWK interpreter	100.00%
gnuchess	5.06	chess engine	100.00%
gnuplot	4.6.1	plotting tool	98.57%
gzip	1.2.4	file compressor	100.00%
irssi	0.8.15	IRC client	100.00%
kin	0.5	database system	100.00%
libdsmcc	0.6	DVB library	100.00%
libieee	0.2.11	IEEE standards for VHDL library	100.00%
libpng	1.0.60	PNG library	100.00%
libsoup	2.41.1	HTTP library	86.41%
libssh	0.5.3	SSH library	98.86%
libxml2	2.9.0	XML library	93.62%
lighttpd	1.4.30	web server	98.89%
lua	5.2.1	programming language	100.00%
lynx	2.8.7	web browser	96.49%
m4	1.4.4	macro expander	100.00%
mpsolve	2.2	mathematical software	100.00%
mptris	1.9	game	100.00%
prc-tools	2.3	C/C++ library for palm OS	100.00%
privoxy	3.0.19	proxy server	100.00%
rcs	5.7	revision control system	100.00%
sendmail	8.14.6	mail transfer agent	99.41%
sqlite	3.7.15.3	database system	98.28%
sylpheed	3.3.0	e-mail client	98.90%
vim	7.3	text editor	93.33%
xfig	3.2.3	vector graphics editor	100.00%
xterm	2.9.1	terminal emulator	100.00%

Source: Author's own elaboration.

Figure 15 – Dependency depth distribution in the program families.



Source: Author's own elaboration.

4 FEATURE DEPENDENCIES CAUSING VARIABILITY BUGS

In this chapter we present an empirical study to better understand how feature dependencies might lead to variability bugs. After studying the occurrence of different types of feature dependencies in practice (see Chapter 3), we now focus on finding actual variability bugs regarding feature dependencies. In addition to counting bugs occurrence, we aim to understand how developers introduce such bugs, in order to possibly avoid them in the future. Here, these bugs include undeclared variables, undeclared functions, unused variables, and unused functions. We restrict our study to these variability bugs because they are likely to happen due to an overlooked feature dependency (see Section 2.1 and Section 2.2). Even simple problems like unused variables and unused functions, that we can consider as less severe issues, receive attention from developers seeking to solve them, as they pollute the code and might raise warnings during compilation. For instance, a single patch to *GNU Chess* fixes 19 unused variables.¹

As an example of bug we intend to study here, Figure 16 depicts an excerpt of the C source code of the *Bash*² project, related to executing arithmetic in commands. The arithmetic feature is optional and is included only when we enable the macro `ARITH`. This code snippet also contains a configuration option to use the *Korn Shell* evaluation pattern controlled by the macro `DPAREN`. We can generate four different configurations from this code snippet: (1) both macros enabled; (2) only `ARITH` enabled; (3) only `DPAREN` enabled; and (4) both macros disabled.

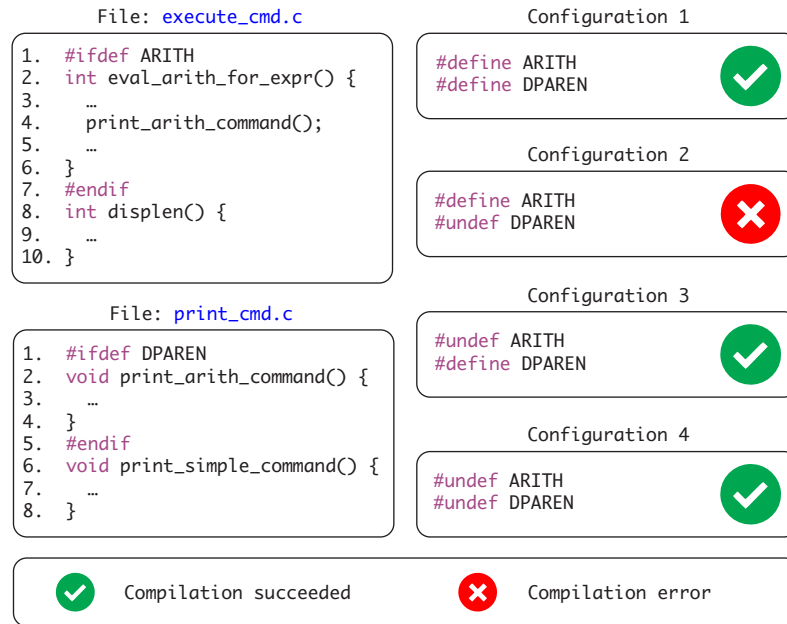
Most analysis tools for C code, such as *GCC*, and *clang*, operate on preprocessed code, that is, one configuration at a time. By compiling the code snippet of Figure 16 with `ARITH` enabled and `DPAREN` disabled, we get a compilation error. File `execute_cmd.c` uses function `eval_arith_for_expr`, which `print_cmd.c` does not declare when we disable `DPAREN`. Thus, we have a feature dependency regarding such a function, since the presence condition of its definition (`DPAREN`) differs from its use (`ARITH`). Because traditional C compilers check only one configuration at a time, they do not show warning or error messages when one compiles the same code depicted in Figure 16 considering the remaining configurations. This is an example of a variability bug exposed only under some combination of configuration options (27, 28, 29). Unfortunately, the space of possible combinations is exponential in the worst case, and it is usually too large to explore exhaustively.

Previous work (10, 17, 29, 30) study variability bugs similar to the one we discuss by analyzing software repositories (17, 30), and using a number of sampling algorithms (31, 32, 33, 34). Such studies focus on bugs that developers have already fixed

¹ <https://goo.gl/1hfm6m>

² <http://www.gnu.org/software/bash/>

Figure 16 – A variability bug in *Bash* that occurs when we enable ARITH and disable DPAREN.



Source: Author's own elaboration.

in software repositories, and do not check all configurations of the source code (that is, sampling checks only a subset of valid configurations), potentially missing bugs. All these studies, however, do not relate the occurrence of variability bugs to the presence of feature dependencies. In addition, a number of previous studies perform per-file analysis (7, 31, 35), which do not detect bugs that span multiple files. This specific variability bug in *Bash*, for example, spans multiple files. Having two different files makes the task of detecting and fixing the bug harder, specially in case we have two or more developers maintaining these files.

To detect variability bugs in a systematic way, we can use variability-aware tools capable of checking all configurations of the source code. However, when using these tools, there is a time-consuming setup that hinders us from scaling the analysis for several software systems. Therefore, in this chapter we propose a strategy to minimize this scalability problem (Section 4.1). Then, we report an empirical study of 15 popular open-source systems (Section 4.2) to better understand variability bugs that feature dependencies might cause. Finally, we present and discuss the results (Section 4.3).

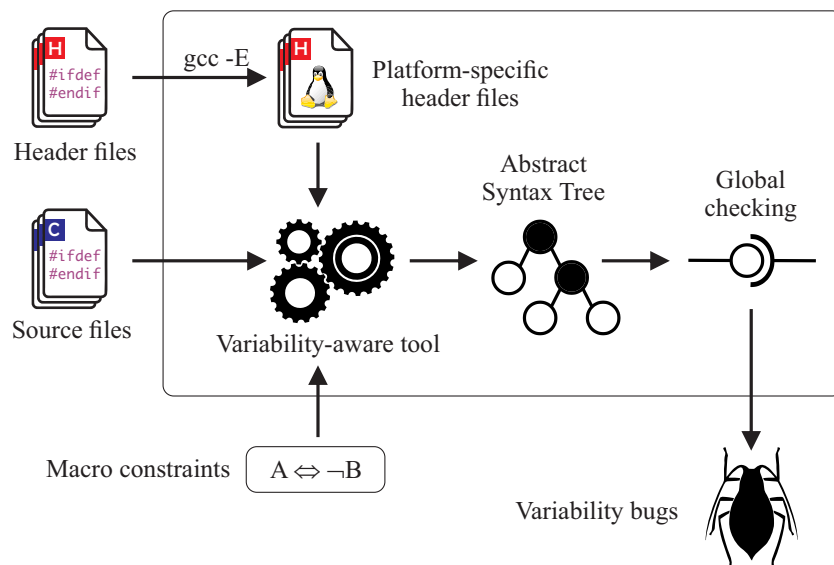
4.1 Strategy to detect variability bugs

In this section, we present our strategy to detect variability bugs regarding feature dependencies in software systems, explaining it using constructs of the C language. Our strategy parses the system source code (C files only) without preprocessing and generates an Abstract Syntax Tree (AST) for each source file. We create a data structure with global

information about variables, functions, and preprocessor macros defined in all source files to check dependencies. This data structure also maintains information regarding which functions, and variables each system configuration defines, allowing us to detect variability bugs (36). Figure 17 illustrates the three steps of our strategy, detailed in what follows.

The goal of *Step 1* is to enable us to analyze several software systems. Variability-aware analysis tools can identify certain classes of bugs (mostly syntax, and type) by covering the entire configuration space. A common difficulty in setting up these tools is that many configuration options are related to platform-specific definitions and libraries. Hence, our strategy preprocesses the included header files and generates platform-specific versions of these files. Despite focusing only on one platform at a time, the strategy enables us to analyze several software systems in such a platform. To generate platform-specific headers, the strategy removes the preprocessor conditional directives (such as `#ifdef` and `#endif`) of the header files, according to the characteristics of a specific platform. For instance, Figure 18 presents how we generate platform-specific headers for *Linux* using *GCC*. After preprocessing the source code, the C preprocessor removes the preprocessor conditional directives associated with the *WIN32* configuration option, and resolves the includes. Thus, our strategy considers only one configuration of each header file. Notice that by considering only *Linux* headers, we might miss other platform-specific variability bugs. To instantiate our strategy for different platforms, one needs to generate platform-specific header files for each different target platform. However, notice that we do not preprocess the C files. For those files, we consider the entire configuration space, as we explain in what follows.

Figure 17 – Strategy to detect variability bugs.

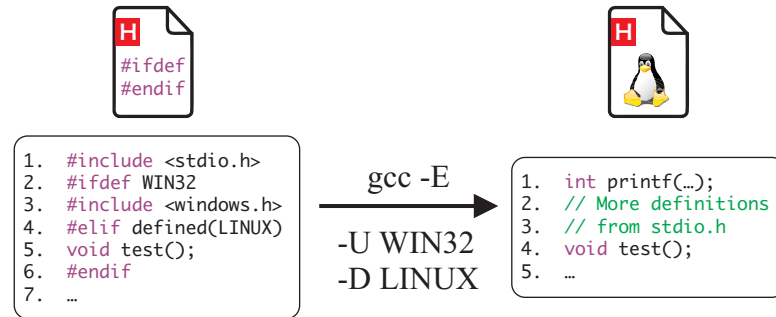


Source: Author's own elaboration.

In *Step 2*, we use a variability-aware tool to parse the source code (C files) and generate

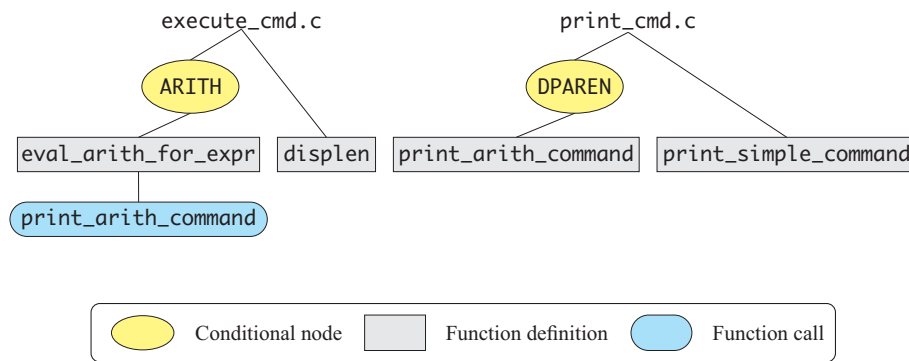
an AST for each source file. When parsing each source file, the tool uses the platform-specific header files generated in the first step. Since we do not preprocess the source files, they still contain preprocessor conditional directives. Therefore, the resulting AST has variable nodes to represent the optional and alternative code blocks. Figure 19 depicts a simplified AST enhanced with variability information from the code excerpt of Figure 16. During this step, our strategy may receive any known constraints to eliminate invalid configurations (for instance, macros A and B are mutually exclusive). We pass this information to the variability-aware tool, which then ignores the invalid configurations. Unfortunately, the majority of C open-source projects do not have such constraints information defined explicitly.

Figure 18 – Generating platform-specific headers for *Linux*.



Source: Author's own elaboration.

Figure 19 – Simplified AST of the code excerpt of Figure 16.



Source: Author's own elaboration.

Step 3 uses the abstract syntax trees of the source files to detect the variability bugs. Notice that we consider the abstract syntax trees of all source files, which allow us to detect variability bugs that span multiple files. Similar to safe composition approaches (37, 38), we check, for each configuration, if the required definitions (variables and functions) are being provided. However, we are also able to capture other issues such as unused variables and functions. For instance, we can see in Figure 19 that **ARITH** requires a function definition (**print_arith_command**) provided by **DPAREN**, as discussed in the beginning of

this chapter. For this reason, *Bash* has a variability bug that arises when we enable `ARITH` and disable `DPAREN`. `DPAREN` provides the function `print_arith_command`, and no other source file provides this required function definition for this specific configuration. At this point, we have the following variability-aware checkers implemented: undeclared variables, unused variables, undeclared functions, and unused functions. Nonetheless, we can extend our infrastructure to add other checkers, such as checking for return types, and fields in struct declarations.

4.2 Study settings

In this section, we present the settings of the empirical study we perform to better understand variability bugs that feature dependencies might cause. To perform the study, we instantiate our strategy to detect variability bugs using the well-known *GCC* compiler, *TypeChef* (12), a variability-aware parser widely used in previous studies (7, 39, 40, 41), and the *Linux* operating system to generate platform-specific header files. We choose *Linux* because it provides simple and effective packaging tools to identify and install the software system dependencies.

In particular, this study addresses the following research questions:

- **Question 1:** what are the frequencies of undeclared variables, unused variables, undeclared functions, and unused functions?
- **Question 2:** how do developers introduce variability bugs related to feature dependencies?

Before answering the research questions, we consider feedback from the actual systems developers to confirm each variability bug. So, all numbers we report here do not include false positives. To answer **Question 1**, we execute our four checkers (undeclared functions, unused functions, undeclared variables, and unused variables) and count their frequencies. Regarding **Question 2**, we analyze each variability bug to verify how developers introduced them by using the source file history in the software repository.

4.2.1 Subject selection

We analyze 15 subject systems written in C ranging from 4,988 to 44,828 lines of code. These systems are from different domains, such as revision control systems, programming languages, and games. Furthermore, we consider mature systems with many developers as well as small systems with few developers. We select these subject systems inspired by previous work (6, 7, 8, 9, 10, 11). This selection is a subset of the 40 families we analyze in our previous study (see Chapter 3). We reduce the number of software families we analyze in this second empirical study because our strategy (see Section 4.1) now

demands a greater effort to set up software families when compared to our approach in the first study. We present the details of each subject system in Table 7. For the subject systems with *git* software repository available, we also consider the commits history of the source files.

4.2.2 Instrumentation

We use the strategy presented in Section 4.1 to investigate variability bugs. We use *TypeChef* version 0.3.5 to parse all configurations of the source code. Furthermore, we also count the number of lines of code, and the number of files of each subject system using the Count Lines of Code (*CLOC*) tool version 1.56, which eliminates blank lines and comments. Finally, we use *Git* version 1.7.12.4 to identify changes in source files.

Table 7 – Subject characterization and number of variability bugs.

Family	Version	Application Domain	LOC	Files	Bugs
bash	4.2	command language interpreter	44,824	138	14
bc	1.03	calculator	5,177	27	
expat	2.1.0	XML library	17,103	54	
flex	2.5.37	lexical analyzer	16,501	41	
gnuchess	5.06	chess engine	9,293	37	1
gzip	1.2.4	file compressor	5,809	36	3
libdsmcc	0.6	DVB library	5,453	30	
libpng	1.6.0	PNG library	44,828	61	9
libsoup	2.41.1	SOUP library	40,061	178	
libssh	0.5.3	SSH library	28,015	125	2
lua	5.2.1	programming language	14,503	59	1
m4	1.4.4	macro expander	10,469	26	1
mptris	1.9	game	4,988	29	
privoxy	3.0.19	proxy server	29,021	67	1
rcs	5.7	revision control system	11,916	28	
Total			287,961	936	32

Note: **LOC**: Number of lines of code.

Source: Author’s own elaboration.

4.3 Results and discussion

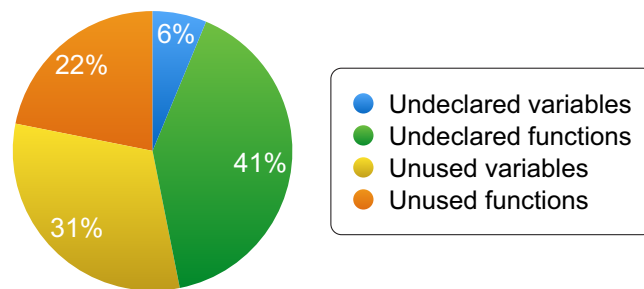
In this section, we discuss the results of our empirical study and answer the research questions. Table 8 presents the variability bugs we detect in each subject system. Notice that we confirm all variability bugs so that the numbers we report do not include false positives. We provide all results at the companion web site.³ We answer the research questions in what follows.

³ <<http://www.iranrodrigues.com.br/masterthesis>>

4.3.1 Question 1: What are the frequencies of undeclared variables, unused variables, undeclared functions, and unused functions?

We analyze 15 subject systems. We find 13 undeclared functions; 7 unused functions; 2 undeclared variables; and 10 unused variables. Figure 20 illustrates these numbers. Overall, we detect 32 variability bugs (14). Because of variability, more than 74% of developers believe that variability bugs issues are more difficult to detect than bugs that appear in all configurations (42). Nevertheless, during the analysis, we also detect bugs that occur in mandatory code, that is, issues that appear in all configurations. Yet, because we focus on variability bugs, we remove numbers related to mandatory code from our statistics. Figure 21 presents an example of undeclared variable. This code excerpt is part of the *libpng* project, and it fails to compile when we enable SPLT and disable POINTER. As we can see, developers declare variable `p` at line 6 only when they enable POINTER. The problem is that they use this variable at lines 13 and 14, in which macro POINTER is disabled, creating an intraprocedural dependency and causing a compilation error.

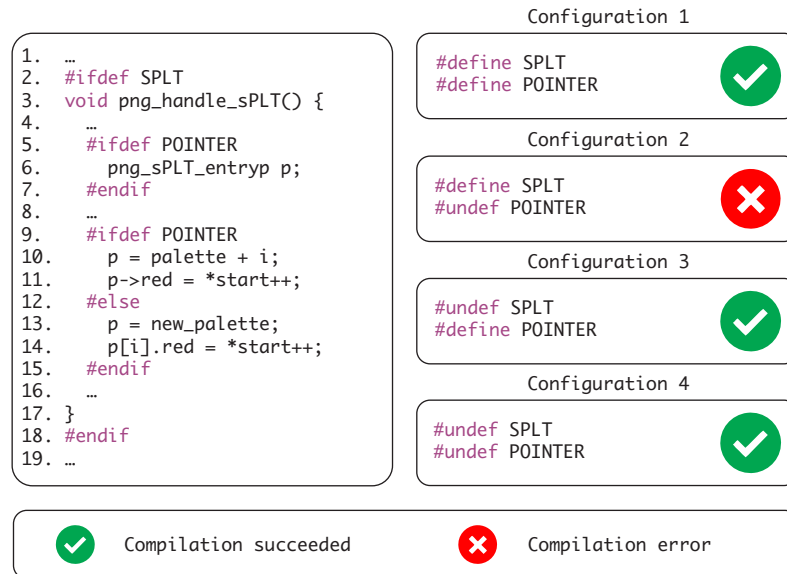
Figure 20 – Frequencies of each variability bug we focus.



Source: Author's own elaboration.

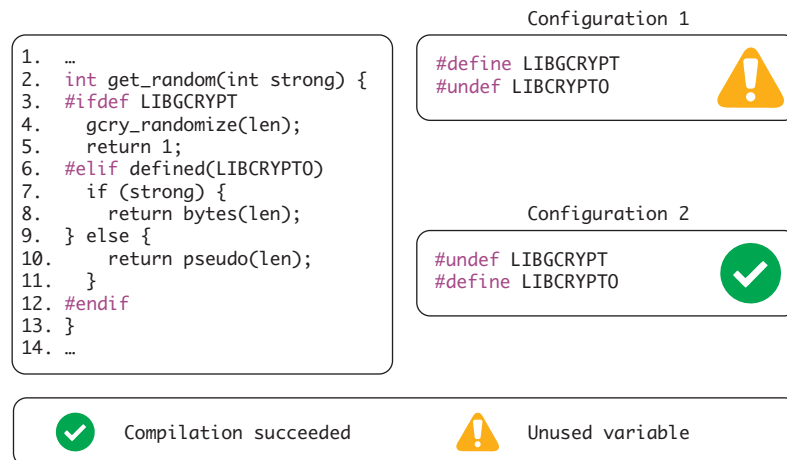
We also find unused variables and functions. Traditional C compilers raise warnings like unused variables and functions when developers set specific command line parameters. Still, we are able to find several unused variables and functions related to feature dependencies. As these warnings do not cause compilation errors, developers might neglect them, even in mandatory code. Figure 22 presents a code excerpt with an unused variable in the *libssh* project. In this code excerpt, variable `strong` is not used when we disable LIBCRYPTO and enable LIBCRYPT. The warning disappears when the opposite configuration selection happens. Notice that for each call to function `get_random` with other presence condition than `!LIBCRYPT && LIBCRYPTO`, we have an interprocedural dependency. Although unused variable is a simple warning, some developers still care about them, by raising bug reports and providing patches to fix them. Indeed, we find bug reports and patches to fix unused variables and functions, such as the one to fix the

Figure 21 – An undeclared variable in the *Libpng* project that occurs when SPLT is enabled and POINTER is disabled.



Source: Author's own elaboration.

Figure 22 – An unused variable in the *libssh* project that occurs when LIBCRYPTO is disabled and LIBCRYPT is enabled.



Source: Author's own elaboration.

libssh warning⁴ presented in Figure 22 and others.⁵

Overall, we conclude that the kinds of variability bugs we focus on this work are not so common in the repositories we study. Still, it seems they are more common than variability bugs regarding syntax errors. A previous work analyzed 41 software families but found only 24 syntax errors in valid configurations (7).

⁴ <https://goo.gl/3Y78M4>

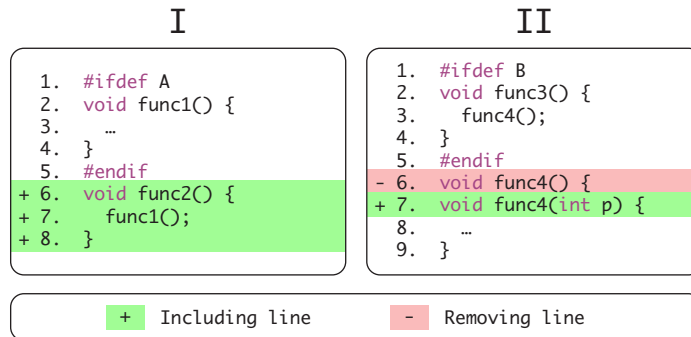
⁵ https://bugzilla.gnome.org/show_bug.cgi?id=461011, 167715, and 401580

4.3.2 Question 2: How do developers introduce variability bugs related to feature dependencies?

We investigate how developers introduce the variability bugs we find in our empirical study. Our goal here is to identify whether developers introduce more bugs when implementing new functionalities or fixing bugs in the source code. According to the results, developers introduce more variability bugs (73%) when introducing new functionalities, such as a new source file, or adding a new function. We now present the results in the following order: undeclared functions, undeclared variables, unused functions, and unused variables.

Developers introduce undeclared functions in two different cases: **(I)** adding a call to existing functions without checking the preprocessor conditional directives that encompass such function definitions, thus creating a new dependency; and **(II)** changing a function definition without modifying the corresponding function calls, due to an existing dependency. Figure 23 illustrates these two cases with small code excerpts. We find that developers introduce 85% of the undeclared functions with case **(I)**: *Bash* (1), *GNU Chess* (1), *gzip* (2), *Libpng* (6), and *Privoxy* (1); and 15% of the undeclared functions follow case **(II)**: *libssh* (1), and *Lua* (1).

Figure 23 – Introducing undeclared functions.



Source: Author's own elaboration.

Figure 24 presents the two cases we detect for undeclared variables. In case **(I)**, developers try to eliminate a shadowed declaration of variable `p1` at line 6. However, they change the conditional directive at line 1, creating a dependency and raising an undeclared variable at line 9. Developers introduce another undeclared variable following case **(II)**, that is, they introduce a new source file that defines variable `p2` conditionally, but uses it in mandatory code, adding a new dependency. We find only one issue for each case: **(I)** in *libpng*, and **(II)** in *gzip*.

Developers introduce unused functions in two cases: **(I)** conditionally defining a function and calling it in code encompassed with different preprocessor conditional directives, ending up in a feature dependency; and **(II)** removing a call to a conditionally defined function, and adding another call to a mandatory function, thus creating a new depen-

Figure 24 – Introducing undeclared variables.

I	II
<pre> - 1. #ifndef A + 2. #ifdef A 3. int p1; 4. #endif 5. #ifdef A - 6. int p1; 7. p1 = func1(); 8. #else 9. p1 = func2(); 10. #endif </pre>	<pre> + 1. void func3() { + 2. #ifdef A + 3. int p2; + 4. #endif + 5. ... + 6. p2 = func4(); + 7. ... + 8. } </pre>
<div> <div>+</div> Including line <div>-</div> Removing line </div>	

Source: Author's own elaboration.

dependency. Figure 25 depicts these two cases. We find that 86% of unused functions follow case **(I)**: *Bash* (4), *libpng* (1), and *m4* (1); and 14% follow case **(II)**: *libpng* (1).

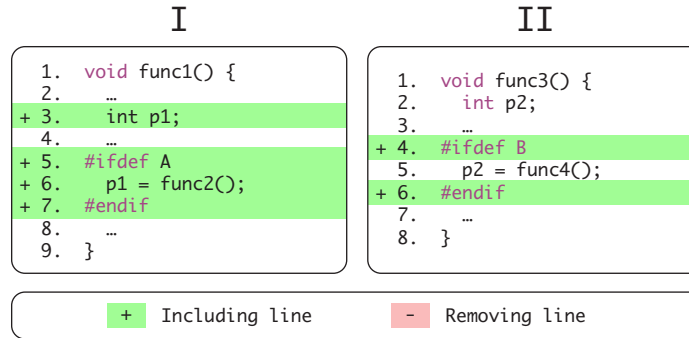
Figure 25 – Introducing unused functions.

I	II
<pre> + 1. #ifdef A + 2. void func1() { + 3. ... + 4. } + 5. #endif + 6. void func2() { + 7. #if defined(A) && defined(B) + 8. func1(); + 9. #endif + 10. } </pre>	<pre> - 1. #ifdef A - 2. void func3() { - 3. ... - 4. } - 5. #endif 6. void func4() { 7. ... 8. } 9. void func5() { 10. #ifdef A - 11. func3(); + 12. func4(); 13. #endif 14. } </pre>
<div> <div>+</div> Including line <div>-</div> Removing line </div>	

Source: Author's own elaboration.

Regarding the unused variables we find in our study, developers introduce them following two cases: **(I)** adding a new variable to mandatory code and using this variable only in optional code; and **(II)** moving the uses of a variable to optional code. In both cases, developers add new dependencies in the process. Figure 26 depicts these two cases. Case **(I)** is the most common (80%): *Bash* (8); and 20% follow case **(II)**: *Bash* (1), and *libssh* (1).

We also analyze whether developers introduce variability bugs by changing mandatory or optional code. Our results reveal that developers introduce most of the bugs we detect (63%) when working on optional code, when compared to bugs introduced in mandatory code (37%).

Figure 26 – Introducing unused variables.

Source: Author's own elaboration.

4.3.3 Summary

The corpus of dependency-related variability bugs gathered in our study is a valuable source to study variability bugs, compare sampling algorithms, and test and improve variability-aware tools. Besides the bugs themselves, this corpus also includes ways in which developers introduce them in practice. These different ways can be explored for developing techniques to detect such bugs as soon as their introduction, through pattern-matching, for instance. Another possibility is that these cases can provide guidance for sampling algorithms, indicating which configurations to test. Thus, developers of bug finder tools can use our results to provide support for detecting variability bugs and consequently minimize them in practice, improving software quality.

4.3.4 Threats to validity

Now, we present the threats to validity. To structure this section, we again follow the Cook and Campbell validity system (26).

Construct Validity. We have to check whether the variability bugs we find appear in valid configurations or represent false positives. To minimize this threat, we perform two tasks: (i) for the systems we know configuration option constraints in advance, we set *TypeChef* to take them into account and consequently avoid analyzing invalid configurations (e.g., *Bash*, *libssh*, and *Privoxy*); and (ii) ask the actual developers to confirm each variability bug not fixed in the software repository. Unfortunately, most projects do not provide constraints information explicitly.

Internal Validity. We analyze the issues manually, which is a time-consuming and error-prone activity. Nevertheless, because we get feedback from the actual developers and confirm the variability bugs we report, we minimize this threat.

External Validity. To scale our analysis, our strategy considers only one configuration of header files. We use *GCC* and generate only header files for the *Linux* platform. However, notice that we may face false negatives due to this limitation. In this context,

our strategy may miss some variability bugs that occur only for other platforms, such as *Windows* and *Mac OS*. Still, in our study, we find 32 variability bugs regarding feature dependencies, and we confirm them either by checking if developers fixed them in software repositories or by getting feedback from developers. Also, we analyze subject systems of different domains, sizes, and different number of developers. We select well-known and active C software systems used in industrial practice. Their communities exist for years and they are in constant development. Therefore, we alleviate this threat.

Table 8 – Variability bugs detected in our empirical study.

Project	File	Kind	Fix/New	Optional/Mandatory
bash	macro.c	unused variable	new	optional
bash	display.c	unused variable	fix	optional
bash	error.c	unused variable	new	optional
bash	execute_cmd.c	unused variable	new	optional
bash	finfo.c	unused variable	new	mandatory
bash	general.c	unused variable	new	mandatory
bash	malloc.c	unused variable	new	optional
bash	shell.c	unused variable	fix	mandatory
bash	watch.c	unused variable	new	mandatory
bash	execute_cmd.c	undeclared function	fix	optional
bash	variables.c	undeclared function	new	optional
bash	pcomplete.c	unused function	-	-
bash	bashline.c	unused function	new	optional
bash	array.c	unused function	new	-
gnuchess	getopt.c	undeclared function	new	optional
gzip	deflate.c	undeclared function	new	optional
gzip	util.c	undeclared function	new	-
gzip	deflate.c	undeclared variable	new	-
libpng	iccfrompng.c	undeclared function	new	mandatory
libpng	iccfrompng.c	undeclared function	new	mandatory
libpng	iccfrompng.c	undeclared function	new	mandatory
libpng	iccfrompng.c	undeclared function	new	mandatory
libpng	pngpixel.c	undeclared function	new	mandatory
libpng	pngpixel.c	undeclared function	new	mandatory
libpng	pngutil.c	undeclared variable	new	optional
libpng	pngvalid.c	unused function	fix	optional
libpng	pngget.c	unused function	new	optional
libssh	dh.c	unused variable	fix	optional
libssh	keyfiles.c	undeclared function	fix	optional
lua	loadlib_rel.c	undeclared function	fix	optional
m4	input.c	unused function	fix	optional
privoxy	filter.c	undeclared function	-	-

Notes: (-) We do not find the necessary information to answer the research question, for instance, in case we detect a bug in the first commit available for analysis, so, we miss information regarding how developers introduce the variability bug. Developers introduce bugs by adding **new** code, and by modifying existing code (**fix**), see column “*Fix/New*”. They introduce bugs by adding / modifying **mandatory** or **optional** code, as we can see in column “*Optional/Mandatory*”.

Source: Author’s own elaboration.

5 RELATED WORK

In this chapter, we present the related work regarding feature dependencies (Section 5.1), variability bugs (Section 5.2), and C preprocessor usage (Section 5.3).

5.1 Feature dependencies

Prior studies investigated the occurrence of feature dependencies in preprocessor-based families. Ribeiro et al. presented an empirical study on the impact of feature dependencies during maintenance of software families (8). This study comprised 43 families written in C and Java. They developed a tool based on *srcML* (24) to generate abstract syntax trees from source code and collect data regarding intraprocedural dependencies in such families. They found that $65.92\% \pm 18.54\%$ of methods contain intraprocedural dependencies. In our study, we focus on families implemented in C, in a total of 40 software families. Instead of *srcML*, our tool uses *TypeChef* (43), which is a more robust solution as it can handle code containing undisciplined annotations, which *srcML* cannot (11). We extend their study by collecting data regarding three types of dependencies: besides intraprocedural, we consider global and interprocedural. In addition, we also classify the dependencies according to their direction, and identify the dependency depth distribution of the interprocedural dependencies.

In another study, Ribeiro et al. (2) presented *Emergo*, a tool capable of inferring interfaces from dataflow analysis on demand. *Emergo* uses *emergent interfaces* (44) to raise awareness of intraprocedural and interprocedural feature dependencies during the maintenance of configurable systems. Later, Ribeiro et al. (3) conducted an experiment that showed that the awareness of feature dependencies decreases the effort and reduces errors on maintenance tasks. In our work, we present the depth distribution for interprocedural dependencies found on some software families. This information can possibly benefit *Emergo* and similar tools, as the call depth is a required parameter in the computing of emergent interfaces. A low depth value may prevent the detection of some feature dependencies, while a high depth value may cause performance issues. Thus, we complement their work by providing the depth distribution, which might be helpful to better set tools like *Emergo*.

Queiroz et. al. (45) analyzed the correlation between software complexity and feature dependencies in 45 preprocessor-based software families and product lines. Moreover, their study also pointed which preprocessor directives (such as `#ifdef` or `#elif`) are responsible for the largest number of dependencies. While they classify dependencies by preprocessor directive, we perform a classification by direction. Besides, their work comprised only intraprocedural dependencies, whilst our work also includes global and

interprocedural dependencies.

Cafeo et al. (46) conducted a comparative study of three programming techniques to implement feature dependencies and their impact on SPL development. Authors analyzed 15 releases of three SPLs in Java, comparing conditional compilation, Aspect Oriented Programming (AOP) (47), and Feature Oriented Programming (FOP) (48), assessing their contribution to instabilities caused by feature dependencies in such SPLs. In our work, we analyzed families written in C. All of them use conditional compilation to implement variability, by using `#ifdef` and other preprocessor directives. Although we do not assess the stability of such families, we quantify the occurrence of variability bugs regarding feature dependencies.

5.2 Variability bugs

Some studies indicated that the indistinct use of C preprocessor may degrade the understandability of the code, hampering its maintenance, and ultimately leading to the introduction of errors (4, 5, 15). Recently, researchers investigated the occurrence of errors that variability might induce. Medeiros et al. (7) conducted an exhaustive search for syntax errors regarding preprocessor usage on 41 product family releases and over 51 thousand commits of 8 program families. They built a tool based on *TypeChef* to parse the code and check for errors in all possible configurations. Their results showed that such errors are not common in practice. In our work, we also use *TypeChef* to perform variability-aware parsing, but we do not focus on the identification of syntax errors. Instead, we focus on variability bugs such as undeclared variables and functions, and unused variables and functions. Besides, in their work, they consider all syntax errors regarding preprocessor directives, not only the ones related to feature dependencies.

Abal et al. (17) performed a qualitative study of 42 variability bugs found on the *Linux* kernel. As well as syntax errors, their study also includes semantic errors. They collected such bugs from bug-fixing commits to the *Linux* kernel repository. These bugs include 30 *feature-interaction bugs*, bugs that arise as a result of feature interactions. In our work, we do not analyze syntax or semantic variability bugs, just undeclared variables and functions, and unused variables and functions. Also, while they search for bugs already fixed, our analysis also includes actual unfixed bugs.

Garvin and Cohen (29) investigated several bug reports regarding configuration-related faults in two configurable systems: *GCC* and *Firefox*. They classified those faults according to the feature selection in which they arise, such as faults caused by wrong features being enabled or when a feature violates another one. They found that only three out of 28 variability bugs were due to feature interactions. In our study, we classify the variability bugs we find according to the type of issue they cause, not according to how their features interact. Besides, we consider a much larger selection of systems. In addition,

we do not restrict our corpus of variability bugs to already fixed ones, since some of bugs we find were still unreported.

5.3 C preprocessor usage

Several studies analyzed the usage of variability mechanisms of C preprocessor, *cpp*. Ernst et al. (9) presented an empirical study concerning C preprocessor usage. They analyzed 26 C software families, collecting data regarding the occurrence of preprocessor directives and macro usage. They also measured to what extent macro dependences occur in the analyzed families, that is, the dependence of a line of code on a macro. In our work, we analyze a broader set of C software families, although some families are common to both studies. We also present data regarding preprocessor directives occurrence, but we focus on the number of functions which contain such directives. Besides, the feature dependencies we analyze do not relate to their concept of macro dependence. While they basically count the number of lines of code depending on a macro, we go beyond by taking into account the sharing of a variable across different features.

Liebig et al. (6) analyzed 40 product families implemented in C to gather information regarding feature code scattering and tangling in the use of preprocessor directives. Later, Liebig et al. (11) analyzed the discipline of preprocessor annotations in those families. In both studies they developed a tool using *srcML* to perform their analysis. Likewise, in our first study we also analyze 40 product families, although not exactly the same families. However, we focus on the analysis of the feature dependencies in such families.

Hunsen et al. (49) performed a study to understand how the C preprocessor is used in open-source and industrial systems. Their study answers questions regarding general use and size of *cpp*-annotated code, and the scattering, tangling, and nesting of preprocessor directives. They analyzed 33 software families, including open-source and proprietary software, relying on *srcML* to generate ASTs from source code. In our work we do not focus on understand how developers use *cpp*. Instead, we aim to understand feature dependencies and how developers introduce variability bugs related to feature dependencies.

Similarly, Queiroz et. al. (50) conducted an analysis of 20 well-known C preprocessor-based systems from different domains, gathering statistics regarding scattering, tangling, and nesting depth of preprocessor annotations. We do not consider such statistics in our work, since we focus on feature dependencies and their implications.

6 CONCLUDING REMARKS

This work presents two empirical studies to better understand the occurrence of fine-grained feature dependencies in C program families and their implications.

Firstly, we present three scenarios to illustrate that different types of feature dependency might cause problems.

Next, we perform a first empirical study of 40 C software families to answer our research questions. Our results show that feature dependencies are fairly common in practice, except for global dependencies. We find intraprocedural dependencies in $51.44\% \pm 17.77\%$ of the functions containing preprocessor directives. We find global dependencies in only $12.14\% \pm 10.46\%$ of the functions which use global variables. Despite being more problematic, this type of dependency is less common. Regarding interprocedural dependencies, we find them in $25.98\% \pm 19.99\%$ of the functions. This data is concerning, since interprocedural dependencies are at least as problematic as global dependencies, as both can spread through different files, being easier to miss them. Our results also show that the most common dependency direction is mandatory-to-optional, occurring in $54.47\% \pm 31.08\%$ of all dependencies. This means that developers are more likely to face a dependency when maintaining mandatory code. Finally, we find that the dependency depth distribution for interprocedural dependencies varies considerably, depending on the family we analyze. In our results, the average depth ranges from 1 to 23.40 ± 11.20 .

Then, we conduct a second empirical study of 15 C software families to assess the problems feature dependencies might cause. To do so, we define a strategy to identify variability bugs that minimizes the setting up problems of variability-aware tools and allows us to analyze several systems. This second study answers questions related to how often software families have variability bugs regarding feature dependencies, and how developers introduce such bugs. In summary, we find 32 distinct variability bugs related to the presence of feature dependencies, including 13 undeclared functions, 2 undeclared variables, 7 unused functions, and 10 unused variables that appear only in some configurations of the source code.

Both empirical studies presents findings that may be helpful to understand feature dependencies and variability bugs, develop tools and techniques to minimize such problems, and improve software quality.

6.1 Review of the contributions

This work makes the following contributions:

- Data on feature dependency that reveal to what extent they are common in practice, complementing previous work;

- A tool to compute code level feature dependencies based on *TypeChef* variability-aware parser;
- A corpus of variability bugs that researchers can use to study variability bugs and test and improve variability-aware tools;
- Findings showing how developers introduce variability bugs when maintaining software families;
- A strategy that makes feasible the task of analyzing variability bugs in several software families.

6.2 Limitations

Our work has some limitations:

- The notion of feature dependency we consider takes only intraprocedural, global, and interprocedural into consideration;
- Our analysis captures only purely syntactic dependencies, by navigating throughout the AST looking for variables definitions and uses, as well as functions definitions and calls. Another technique suitable to this task is the dataflow analysis;
- The tool we create to compute dependencies do not capture all possible existing feature dependencies, specially the ones regarding global variables, due to reasons explained in Section 3.2.6;
- We restrict our variability bugs analysis to four kinds of issues: undeclared variables and functions, and unused variables and functions, when such issues involve feature dependencies. Still, we can extend our infrastructure to add more checkers.

6.3 Future work

In particular, we intend to complement this work with the following:

- In our first empirical study, we analyze 97.70% of all source files. We intend to enhance our technique so we can analyze all files and provide more accurate results;
- Another point to improve is the global dependency detection, particularly when functions use global variables after some other function modifies them, which our strategy currently neglects. We intend to use a dataflow analysis approach, so we can have a more precise global dependency identification;

- In our second empirical study, we analyze 15 software families, compared with 40 of the previous study. This reduction is due to the time required to prepare each family for analysis, including setting up the environment, managing package dependencies, and so on. As a future work, we intend to use our second study strategy in all 40 families we analyze previously, since we expect to find even more variability bugs in families that were left out;
- Our variability bugs analysis includes only four kinds of issues: undeclared variables, undeclared functions, unused variables, and unused functions. We intend to add more checkers to our strategy, so we can detect a wider set of variability bugs.

BIBLIOGRAPHY

- 1 CATALDO, M. et al. Software dependencies, work dependencies, and their impact on failures. *Software Engineering, IEEE Transactions on*, v. 35, n. 6, p. 864–878, Nov 2009.
- 2 RIBEIRO, M. et al. Emergo: A tool for improving maintainability of preprocessor-based product lines. In: *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development Companion*. [S.l.]: ACM, 2012. (AOSD Companion), p. 23–26.
- 3 RIBEIRO, M.; BORBA, P.; KÄSTNER, C. Feature maintenance with emergent interfaces. In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.]: ACM, 2014. (ICSE), p. 989–1000.
- 4 SPENCER, H.; COLLYER, G. `#ifdef` considered harmful, or portability experience with C News. In: *USENIX Summer Technical Conference*. [S.l.: s.n.], 1992. p. 185–197.
- 5 FAVRE, J. M. The CPP Paradox. In: *9th European Workshop on Software Maintenance*. [S.l.: s.n.], 1995.
- 6 LIEBIG, J. et al. An analysis of the variability in forty preprocessor-based software product lines. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. [S.l.]: ACM, 2010. (ICSE), p. 105–114.
- 7 MEDEIROS, F.; RIBEIRO, M.; GHEYI, R. Investigating preprocessor-based syntax errors. In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. [S.l.]: ACM, 2013. (GPCE), p. 75–84.
- 8 RIBEIRO, M. et al. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In: *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. [S.l.]: ACM, 2011. (GPCE), p. 23–32.
- 9 ERNST, M. D.; BADROS, G. J.; NOTKIN, D. An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 28, n. 12, p. 1146–1170, dez. 2002.
- 10 GARRIDO, A.; JOHNSON, R. Analyzing multiple configurations of a C program. In: *Proceedings of the International Conference on Software Maintenance*. [S.l.]: IEEE, 2005. (ICSM).
- 11 LIEBIG, J.; KÄSTNER, C.; APEL, S. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In: *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. [S.l.]: ACM, 2011. (AOSD), p. 191–202.
- 12 KÄSTNER, C. et al. Variability-aware parsing in the presence of lexical macros and conditional compilation. In: *Proceedings of the ACM SIGPLAN Object-oriented programming systems languages and applications*. [S.l.]: ACM, 2011. (OOPSLA).
- 13 THÜM, T. et al. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, ACM, v. 47, n. 1, p. 6:1–6:45, jun. 2014.

- 14 MEDEIROS, F. et al. An empirical study on configuration-related type issues. In: *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences*. [S.l.]: ACM, 2015. (GPCE). To appear.
- 15 KRONE, M.; SNETLING, G. On the inference of configuration structures from source code. In: *Proceedings of the 16th International Conference on Software Engineering*. [S.l.]: IEEE Computer Society Press, 1994. (ICSE), p. 49–57.
- 16 FEIGENSPAN, J. et al. Do background colors improve program comprehension in the `#ifdef` hell? *Empirical Software Engineering*, Springer US, v. 18, n. 4, p. 699–745, 2013.
- 17 ABAL, I.; BRABRAND, C.; WASOWSKI, A. 42 variability bugs in the linux kernel: A qualitative analysis. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. [S.l.]: ACM, 2014. (ASE), p. 421–432.
- 18 WULF, W.; SHAW, M. Global variable considered harmful. *SIGPLAN Not.*, ACM, v. 8, n. 2, p. 28–34, fev. 1973.
- 19 ANVIK, J.; HIEW, L.; MURPHY, G. C. Who should fix this bug? In: *Proceedings of the 28th International Conference on Software Engineering*. [S.l.]: ACM, 2006. (ICSE), p. 361–370.
- 20 ZIMMERMANN, T. et al. What makes a good bug report? *Software Engineering, IEEE Transactions on*, v. 36, n. 5, p. 618–643, Sept 2010.
- 21 BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: *Encyclopedia of Software Engineering*. [S.l.]: Wiley, 1994.
- 22 RYDER, B. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, SE-5, n. 3, p. 216–226, May 1979.
- 23 TARJAN, R. Depth-first search and linear graph algorithms. *SIAM journal on computing*, SIAM, v. 1, n. 2, p. 146–160, 1972.
- 24 MALETIC, J. I.; COLLARD, M.; KAGDI, H. Leveraging xml technologies in developing program analysis tools. In: *in Proceedings of 4th International Workshop on Adoption-Centric Software Engineering*. [S.l.: s.n.], 2004. (ACSE), p. 80–85.
- 25 KAMPSTRA, P. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, v. 28, n. 1, p. 1–9, 11 2008.
- 26 COOK, T. D.; CAMPBELL, D. T. *Quasi-experimentation: Design & analysis issues for field settings*. [S.l.]: Houghton Mifflin Boston, 1979.
- 27 YILMAZ, C.; COHEN, M. B.; PORTER, A. Covering arrays for efficient fault characterization in complex configuration spaces. In: *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*. [S.l.]: ACM, 2004. (ISSTA).
- 28 KUHN, D. R.; WALLACE, D. R.; GALLO JR., A. M. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, IEEE Press, v. 30, n. 6, p. 418–421, jun. 2004.

- 29 GARVIN, B. J.; COHEN, M. B. Feature interaction faults revisited: An exploratory study. In: *Proceeding of the International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2011. (ISSRE).
- 30 GARVIN, B. J.; COHEN, M. B.; DWYER, M. B. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In: *Proceedings of the Workshop on Assurances for Self-adaptive Systems*. [S.l.]: ACM, 2011. (ASAS).
- 31 TARTLER, R. et al. Configuration coverage in the analysis of large-scale system software. In: *Proceedings of the Workshop on Programming Languages and Operating Systems*. [S.l.: s.n.], 2011. (PLOS).
- 32 OSTER, S.; MARKERT, F.; RITTER, P. Automated incremental pairwise testing of software product lines. In: BOSCH, J.; LEE, J. (Ed.). *Software Product Lines: Going Beyond*. [S.l.]: Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6287). p. 196–210.
- 33 PERROUIN, G. et al. Automated and scalable t-wise test case generation strategies for software product lines. In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. [S.l.: s.n.], 2010. (ICST).
- 34 JOHANSEN, M. F.; HAUGEN, O.; FLEUREY, F. An algorithm for generating t-wise covering arrays from large feature models. In: *Proceedings of the International Software Product Line Conference*. [S.l.: s.n.], 2012. (SPLC).
- 35 GAZZILLO, P.; GRIMM, R. SuperC: parsing all of C by taming the preprocessor. In: *Proceedings of the programming language design and implementation*. [S.l.]: ACM, 2012. (PLDI).
- 36 KÄSTNER, C.; OSTERMANN, K.; ERDWEG, S. A variability-aware module system. In: *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. [S.l.]: ACM, 2012.
- 37 THAKER, S. et al. Safe Composition of Product Lines. In: *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*. [S.l.: s.n.], 2007. p. 95–104.
- 38 DELAWARE, B.; COOK, W.; BATORY, D. Fitting the Pieces Together: a machine-checked model of safe composition. In: *Proceedings of the Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2009.
- 39 LIEBIG, J. et al. Scalable analysis of variable software. In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. [S.l.]: ACM, 2013. (ESEC/FSE), p. 81–91.
- 40 LIEBIG, J. et al. Morpheus: Variability-aware refactoring in the wild. In: *Proceedings of the International Conference on Software Engineering*. [S.l.]: IEEE, 2015. (ICSE).
- 41 RHEIN, A. von et al. Presence-condition simplification in highly configurable systems. In: *Proceedings of the International Conference on Software Engineering*. [S.l.]: IEEE, 2015. (ICSE).

- 42 MEDEIROS, F. et al. The love/hate relationship with the c preprocessor: An interview study. In: *Proceedings of the European Conference on Object-Oriented Programming*. [S.l.: s.n.], 2015. (ECOOP).
- 43 KENNER, A. et al. Typechef: Toward type checking `#ifdef` variability in c. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. [S.l.: ACM, 2010. (FOSD), p. 25–32.
- 44 RIBEIRO, M. et al. Emergent feature modularization. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. [S.l.: ACM, 2010. (OOPSLA), p. 11–18.
- 45 QUEIROZ, F. et al. Towards a better understanding of feature dependencies in preprocessor-based systems. In: *Proceedings of the 6th Latin American Workshop on Aspect-Oriented Software Development: Advanced Modularization Techniques*. [S.l.: s.n.], 2012. (LA-WASP).
- 46 CAFEIO, B. et al. Analysing the impact of feature dependency implementation on product line stability: An exploratory study. In: *Software Engineering (SBES), 2012 26th Brazilian Symposium on*. [S.l.: s.n.], 2012. p. 141–150.
- 47 KICZALES, G. et al. Aspect-oriented programming. In: AKŞIT, M.; MATSUOKA, S. (Ed.). *ECOOP'97 — Object-Oriented Programming*. [S.l.: Springer Berlin Heidelberg, 1997, (Lecture Notes in Computer Science, v. 1241). p. 220–242.
- 48 PREHOFER, C. Feature-oriented programming: A fresh look at objects. In: AKŞIT, M.; MATSUOKA, S. (Ed.). *ECOOP'97 — Object-Oriented Programming*. [S.l.: Springer Berlin Heidelberg, 1997, (Lecture Notes in Computer Science, v. 1241). p. 419–443.
- 49 HUNSEN, C. et al. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Journal of Empirical Software Engineering*, 2015.
- 50 QUEIROZ, R. et al. The shape of feature code: an analysis of twenty c-preprocessor-based systems. *Software & Systems Modeling*, Springer Berlin Heidelberg, p. 1–20, 2015.