Information and Software Technology 78 (2016) 27–52

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Assessing fine-grained feature dependencies

Iran Rodrigues[a,*], Márcio Ribeiro[a], Flávio Medeiros[c], Paulo Borba[b], Baldoino Fonseca[a], Rohit Gheyi[c]

[a] Federal University of Alagoas, Maceió, Brazil
[b] Federal University of Pernambuco, Recife, Brazil
[c] Federal University of Campina Grande,Campina Grande, Brazil

ARTICLE INFO

Article history:
Received 9 August 2015
Revised 18 May 2016
Accepted 23 May 2016
Available online 24 May 2016

Keywords:
Preprocessor
Software family
Feature dependency

ABSTRACT

Context: Maintaining software 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.

Objective: This work assesses to what extent feature dependencies exist in actual software families, answering research questions regarding the occurrence of intraprocedural, global, and interprocedural dependencies and their characteristics.

Method: We perform an empirical study covering 40 software families of different domains and sizes. We use a variability-aware parser to analyze families source code while retaining all variability information.

Results: Intraprocedural and interprocedural feature dependencies are common in the families we analyze: more than half of functions with preprocessor directives have intraprocedural dependencies, while over a quarter of all functions have interprocedural dependencies. The median depth of interprocedural dependencies is 9.

Conclusion: Given these dependencies are rather common, there is a need for tools and techniques to raise developers awareness in order to minimize or avoid problems when maintaining code in the presence of such dependencies. Problems regarding interprocedural dependencies with high depths might be harder to detect and fix.

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 [4–7]. 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.

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 might be harder to detect.

* Corresponding author.
E-mail addresses: irgj@ic.ufal.br, iran@iranrodrigues.com.br (I. Rodrigues), marcio@ic.ufal.br (M. Ribeiro), flaviomedeiros@copin.ufcg.edu.br (F. Medeiros), phmb@cin.ufpe.br (P. Borba), baldoino@ic.ufal.br (B. Fonseca), rohit@dsc.ufcg.edu.br (R. Gheyi).

http://dx.doi.org/10.1016/j.infsof.2016.05.006
0950-5849/

Despite important, we still lack a study that takes other kinds of feature dependencies into account.

Therefore, to minimize these lack and better understand feature dependencies, in this work we perform an empirical study to assess to what extent feature dependencies occur in practice, identifying their characteristics and frequency. We also compare some of our results with results from previous work [8].

Before executing this study, 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 configuration-related bugs happen in such families and better prepare our study. After finding examples of bugs related to feature dependencies, 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 global dependencies? How often do program families contain interprocedural dependencies? How often do dependencies of different directions occur in practice? What is the dependency depth distribution for interprocedural dependencies? How the results of the current study compare with the previous ones? Answering these questions is important to better understand feature dependencies and assess their occurrence in practice.

To answer our research questions, our study covers 40 C program families of different domains and sizes. We select these families inspired by previous work [6–11]. 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 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 ± standard deviation", our results show that 51.44% ± 17.77% of functions with preprocessor directives have intraprocedural dependencies, 11.90% ± 12.20% of the functions which use global variables have global dependencies, while 25.98% ± 19.99% of all functions have interprocedural dependencies.

In summary, the main contributions of this paper are:

- data on feature dependency that reveal to what extent they are common in practice, complementing previous work by considering new types of dependencies;
- a strategy to compute feature dependencies based on the TypeChef variability parser.

We organize the remainder of this paper as follows. In Section 2 we introduce the concept of feature dependency. Next, in Section 3 we show motivating examples that illustrate variability bugs from industrial systems. Then, we present the empirical study settings in Section 4. After, in Section 5 we present and discuss the results. Later, in Section 6 we present some consequences of our work. In Section 7 we discuss the related work and in Section 8 we present the final considerations of this work.

This paper is an extension of our previous work [8] on feature dependency analysis. In this work we bring more evidence regarding bugs related to intraprocedural feature dependencies. Compared to the previous study, we analyze new families and use a different tool, improving its external validity. Moreover, we now take global and interprocedural dependencies into account, presenting bugs related to such types of dependencies and computing data regarding their presence in a set of industrial software families.

```
1.  #ifdef A
2.    int x;
3.  #endif
4.  …
5.  #ifdef B
6.    x++;
7.  #endif
```

Fig. 1. Example of a feature dependency regarding variable `x`.

2. Feature dependency

A program family consists of a set of programs that share a common core but also have distinguishing functionalities. These commonalities and variabilities are often modeled as features, each representing increments in functionality to the program. Each feature provides a potential configuration option, so developers can generate different programs tailored for specific tasks or platforms. When we consider program families written in C, developers often use the C preprocessor (*cpp*) to implement variability in those systems [4–7].

The C preprocessor allows the use of conditional compilation directives such as `#if` or `#ifdef` along with a *macro expression* to surround feature-specific fragments of code. Macro expressions might contain one or more macros as a boolean formula, as in `#if defined(A) && defined(B)`, which might refer to specific configuration options. The minimum subset of features in which a fragment of code is included in the conditional compilation is called *presence condition* [13]. Developers can use preprocessor directives to wrap from entire structures such as functions to part of a statement such as a single variable, allowing variability in different levels of granularity. This flexibility also allows code from a single feature to be scattered all over the program.

Often features communicate and collaborate with each other, so their implementations might share program elements and data. When different features refer to the same program element, such as a variable, we have a feature dependency. Following the classification proposed by Apel et al. [14], such feature dependencies we consider in this paper are operational feature interactions, since a feature pass data to another one.

To better explain this concept, we refer to the code snippet in Fig. 1. In the figure, the definition of variable `x` (see line 2) is inside an `#ifdef` block, associated to the macro expression `A` (see line 1). In practice, not all macros in a macro expression correspond to actual features in a broader sense. However, since feature models are not always available, and for the sake of simplicity, in this work we consider that each macro in a macro expression refers to a different feature. That said, we consider that the definition of `x` is in a *code fragment* of feature `A`. Likewise, `x` is later incremented (see line 6) in a *code fragment* of feature B (see line 5). This means that the definition of `x` will be included in the compilation if and only

[1] https://gcc.gnu.org/bugzilla/.
[2] https://bugzilla.gnome.org/.
[3] https://bugzilla.kernel.org/.

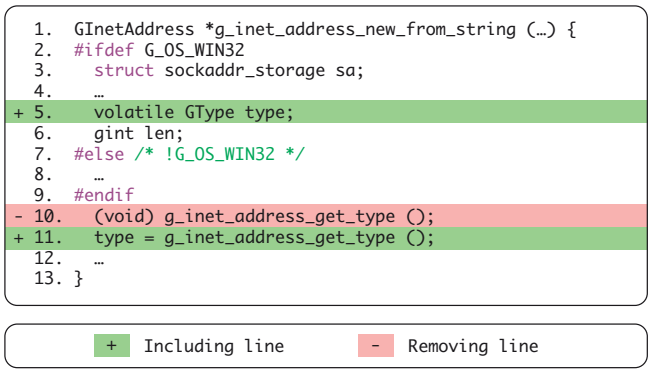


Fig. 2. Adding an intraprocedural dependency, causing a bug in *GLib*.

if we define A. Similarly, the increment of the same variable will be included if and only if we define B. In other words, the presence condition of statement `int x` is A, whereas the presence condition of statement `x++` is B. Since the use of `x` in a code fragment of B depends on its definition in a code fragment of A, we can say that feature B depends on feature A.

In this paper we focus on three types of feature dependencies: *intraprocedural, global*, and *interprocedural* dependencies. We explore them in the next section.

3. 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 both faults and errors, in which a fault is an incorrect instruction in the software code, due to a developer mistake, and an error is an incorrect program state, due to a fault [13].

A variability bug is a fault or error that happens in some, but not all, feature configurations of a software family [13]. A category of variability bugs is related to feature dependencies. Due to this dependency, a maintenance task in one feature might break another one [8]. This might happen since there is no mutual agreement [17] between the developers.

To better illustrate that feature dependencies might cause problems, in this section 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 3.1). Then, we present a variability bug related to a global dependency (Section 3.2). Next, we present a variability bug regarding an interprocedural dependency (Section 3.3). Finally, we summarize our findings on this topic (Section 3.4).

3.1. 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 local variable defined in a feature and used in another one.

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*. Intraprocedural dependencies can only occur in functions containing preprocessor directives, since there should be at least one `#ifdef` (or equivalent) directive surrounding a dependent variable inside such a function.

Fig. 2 presents a code snippet from *GLib*,[4] 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.[5] 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, `type` (see line 5), assigning the function return value to such a variable (by removing line 10 and adding line 11).

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`. Such macro expressions, consisting of one or few macros, are fairly common in the program families of our study, as we shall see in Table 2. 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.

Fig. 3 shows a new modification in the code, in order to fix this variability bug.[6] To do so, the developer relocated[7] 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 Fig. 2, we have an example of maintenance point at line 5 and an impact point at line 11.

[4] https://developer.gnome.org/glib/.
[5] https://git.gnome.org/browse/glib/.
[6] https://bugzilla.gnome.org/show_bug.cgi?id=580750.
[7] https://goo.gl/YFPD4F.

```
 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
```

Fig. 3. Removing the dependency to fix the bug in *GLib*.

M → O

```
1.  int x;
2.    …
3.  void f() {
4.  #ifdef A
5.    x++;
6.  #endif
7.  }
```

O → M

```
1.  #ifdef A
2.  int x;
3.  #endif
4.  …
5.  void f() {
6.    x++;
7.  }
```

Fig. 4. Global dependencies of different directions.

3.2. 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.

Notice that, unlike intraprocedural dependencies, we can have global dependencies even in functions that do not have preprocessor directives. This can happen since we have a global variable declared in an optional feature and used within such a function in a mandatory feature. Fig. 4 illustrates two functions with global dependencies. While the function in the left-hand side of the figure contains a mandatory-to-optional dependency and a preprocessor directive (`#ifdef`), the right-hand side shows a function with an optional-to-mandatory dependency, but without any directives.

Fig. 5 presents a code snippet of the *libxml2*[8] software family, a XML parser written in C. The figure depicts a modification made to the code, committed to a *Git* repository.[9] 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 bug[10] 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. Fig. 6 presents another modification to the code, aiming to solve this variability bug. In the figure, the developer included [11] the same conditional directive of the variable definition to its use (by adding lines 10 and 13).

3.3. 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 a 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*.

For instance, Fig. 7 depicts two situations in which we have interprocedural dependencies regarding functions `f` and `g`. In the left-hand side of the figure we have a mandatory-to-optional dependency, with a maintenance point at line 3 in a mandatory feature (where `f` passes `x` to `g`), and an impact point at line 8 in an optional feature (where `g` uses `x`). In the right-hand side of the figure the situation is just the opposite, as we have an optional-to-mandatory dependency, given the maintenance point at line 4 is now in an optional feature, while the impact point at line 9 is in a mandatory feature. This way, just as might happen with global dependencies, we can also have functions without preprocessor directives involved in interprocedural feature dependencies.

Fig. 8 presents a code snippet from *Lustre*,[12] a parallel distributed file system for high-performance cluster computing. The figure depicts a modification made to the code, committed to a Git repository.[13] 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 struct of type `nameidata`. Developers reported a null pointer dereference bug[14] 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.

To solve this problem, the program now checks[15] if `nd` is null, right before accessing `nd->flags` (see Fig. 9, line 12). Despite

[8] http://xmlsoft.org/.
[9] https://git.gnome.org/browse/libxml2/.
[10] https://bugzilla.gnome.org/show_bug.cgi?id=611806.
[11] https://goo.gl/gACFs6.
[12] http://www.lustre.org.
[13] http://git.whamcloud.com/.
[14] https://jira.hpdd.intel.com/browse/LU-3483.
[15] http://review.whamcloud.com/#/c/6715/5/lustre/llite/dcache.c,cm.

```
 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

Fig. 5. 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 defined(HTML) || defined(VALID)
 11.    if (xmlout)
 12.      saveOpts |= XML_SAVE_AS_XML;
+13.  #endif
 14.   …
 15. }
```

+ Including line

Fig. 6. Removing the dependency to fix the bug in *libxml2*.

M → O

```
1.  void f() {
2.    int x = 0;
3.    g(x);
4.  }
5.  …
6.  void g(int x) {
7.  #ifdef A
8.    x++;
9.  #endif
10. }
```

O → M

```
1.  void f() {
2.    int x = 0;
3.  #ifdef A
4.    g(x);
5.  #endif
6.  }
7.  …
8.  void g(int x) {
9.    x++;
10. }
```

Fig. 7. Interprocedural dependencies of different directions.

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. Fig. 10 shows such an indirect call in line 26. In this case, the field `d_op` of struct `dentry` corresponds to the struct `ll_d_ops` (Fig. 11). Therefore, `dentry->d_op->d_revalidate` points to the function `ll_revalidate_nd` (see line 2 in Fig. 11). 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 Fig. 8, line 5), this problem would occur in every implementation, probably being more noticeable.

As this call is in a mandatory section of code and we access the parameter `nd` in an optional feature (see Fig. 8,

```
 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

Fig. 8. Adding an interprocedural dependency 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

Fig. 9. Fixing the possible null pointer dereference in *Lustre*.

```
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. }
```

Fig. 10. Code snippet from *Linux kernel*.

```
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.  };
```

Fig. 11. Code snippet from *Lustre*.

line 5), we have an mandatory-to-optional interprocedural dependency. The presence condition of `nd` at first was `LOOKUP_RCU` (see Fig. 8, line 4), but further modifications changed the presence condition to `(!HAVE_IOP_ATOMIC_OPEN && !HAVE_DCACHE_LOCK)`(see Fig. 9, 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 Fig. 10, line 26, and an impact point in Fig. 8, 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 Fig. 10, the null problematic value comes from 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 maximum depth is four, as the null value (Fig. 10, 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.

3.4. 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 [18,19]. Even if a simple bug could be easily fixed, when developers have to deal with a great number of them, the combined effort to perform all those tasks would raise significantly.

Variability bugs—including the ones regarding feature dependencies—are even more difficult to deal with, since they occur only in specific configurations, possibly delaying their detection and subsequent fix. Developers themselves consider variability bugs easier to introduce, harder to fix, and more critical than other bugs [20]. Moreover, global and interprocedural dependencies are particularly problematic, as different features might share data from a variable between different files. The bug in Section 3.3 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. Although these motivating examples demonstrate how feature dependencies might lead to variability bugs, we cannot state that a dependency generates bugs by itself. Instead, dependencies might become a problem when developers are not aware of them. This way, it is necessary to learn more about such dependencies in order to avoid or minimize potential problems when maintaining software families. Note that these are code dependencies. Although we expect them to often correspond to dependencies specified in feature models, there is no guarantees that the code dependencies actually match the feature model dependencies. When they do not, we have the so called unsafe composition [21]. To assess how often these dependencies occur in practice, and what are their characteristics, we present next an empirical study to answer research questions on this topic (Section 4).

4. Study settings

In this section we present the settings of our study to investigate feature dependencies on program families. Our study covers 40 C industrial program families, varying from different domains and sizes. We select these families inspired by previous work [6–11]. To structure our research, we use the Goal, Question, and Metrics [22] approach.

4.1. Goal, question, and metrics

With this empirical study we aim to investigate to what extent feature dependencies occur in C program families, and learn more about the characteristics of such dependencies. We also want to replicate part of a previous work [8], now using a more appropriate tool to analyze C program families in a variability-aware manner.

We define the goals, questions, and metrics of this study as follows:

Goal 1 To assess the occurrence of different types of feature dependencies in C program families

- **Question 1.1** How often do program families contain intraprocedural dependencies?
  - **Metric 1.1.1** Number of functions with preprocessor directives (FDi)
  - **Metric 1.1.2** Number of functions with intraprocedural dependencies (FIntra)
  - **Metric 1.1.3** Number of functions with intraprocedural dependencies among the functions with preprocessor directives (FIntra / FDi)
- **Question 1.2** How often do program families contain global dependencies?
  - **Metric 1.2.1** Number of functions which use global variables (FGRef)
  - **Metric 1.2.2** Number of functions with global dependencies (FGlobal)
  - **Metric 1.2.3** Number of functions with global dependenciesamong the functions which use global variables (FGlobal / FGRef)
- **Question 1.3** How often do program families contain interprocedural dependencies?
  - **Metric 1.3.1** Number of functions with maintenance points regarding interprocedural dependencies (FM)
  - **Metric 1.3.2** Number of functions with impact points regarding interprocedural dependencies (FI)
  - **Metric 1.3.3** Number of functions with either maintenance or impact points regarding interprocedural dependencies (FInter)

Goal 2 To investigate further characteristics of feature dependencies

- **Question 2.1** How often do dependencies of different directions (mandatory-to-optional, optional-to-mandatory, and optional-to-optional) occur?
  - **Metric 2.1.1** Number of mandatory-to-optional dependencies (M → O)
  - **Metric 2.1.2** Number of optional-to-mandatory dependencies (O → M)
  - **Metric 2.1.3** Number of optional-to-optional dependencies (O → O)
- **Question 2.2** What is the dependency depth distribution for interprocedural dependencies?
  - **Metric 2.2.1** Dependency depth (DD)

Goal 3 To compare results on intraprocedural feature dependency detection using *TypeChef* with previous work based on *srcML*

- **Question 3.1** How the results of the current study compare with the previous ones?
  - **Metric 3.1.1** Number of functions with intraprocedural dependencies in the previous study
  - **Metric 3.1.2** Number of functions with intraprocedural dependencies in the current study

With **Goal 1** we intend to verify how often different types of feature dependencies (intraprocedural, global, and interprocedural) exists in program families.

In the next three questions we express feature dependency occurrence as the number of functions with feature dependencies, to be consistent with our previous work [8]. We also summarize these results by reporting the mean, standard deviation, median and interquartile range (IQR). This way, we can better estimate how often families contain dependencies and what is the dispersion of each metric among the families.

To answer **Question 1.1**, 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 1.2**, 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 5.7.

To answer **Question 1.3**, 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.

With **Goal 2** we aim to better understand the characteristics of feature dependencies, in terms of directions and depths (in case of interprocedural dependencies).

To answer **Question 2.1**, we classify every dependency based on its direction (mandatory-to-optional, optional-to-mandatory, or optional-to-optional). Then we count the occurrences of dependencies for each direction, grouping by type of dependency. We use this metric in order to verify if some direction is particularly common for some type of feature dependency.

To answer **Question 2.2**, we create a call graph [23], 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 [24] 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 7.1). Considering that every interprocedural dependency may have many different paths, we present this data as a distribution, so one can easily see the relative likelihood for random dependency to have a given depth. Again, we summarize these results by presenting the mean, standard deviation, median and interquartile range (IQR). This way, we can better estimate the depth of interprocedural dependencies and its dispersion among the families.

Finally, with **Goal 3** we plan to establish a comparative analysis between the results of our current study and the results of our previous study on intraprocedural dependencies.

To answer **Question 3.1**, we use the data we gather in Question 1.1 and compare it to the results of our previous study on feature dependencies, where we focused on intraprocedural dependencies using a different tool, which do not properly support non-disciplined annotations. By doing so, we intend to verify if previous results still hold.

To better explain the metrics we compute, we refer to the code snippet in Fig. 12. We extract this code snippet from *libssh*,[16] 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*,[17] a compression library; `HAVE_LIBGCRYPT`, related to *libgcrypt*,[18] a cryptographic library; and `DEBUG_CRYPTO`, 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.

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

[16] http://www.libssh.org/.

[17] http://www.zlib.net/.

[18] http://www.gnu.org/software/libgcrypt/.

```
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_CRYPTO
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. }
```

Fig. 12. Code snippet from *libssh*.

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.

4.2. Subject selection

We analyze 40 program families written in C, ranging from 2, 681 to 288, 654 lines of code. Although we do not systematically target diversity, we have some in our set of families [25] since our selection covers different domains, such as web servers, database systems, text editors, and programming languages. We select these subject systems inspired by previous work [6–11]. Although we want to compare our results with results from previous work [8], our set of families is not exactly the same. In part because previous work included some Java families, for instance, and we focus on C families in this study. Another reason is that we want to analyze new families and discover whether previous results still apply to them. Moreover, although our set of families does not guarantee high representativeness [25], we include some well-known and mature program families used in industrial practice. We present more details on each family in Table 2.

4.3. 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. *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* [26] to create ASTs represented in the

Table 1
Metrics summary for the code snippet in Fig. 12. We do not consider Goal 3 and its question and metrics, due to this being a illustrative example.

Goal	Question	Metric	Value
Goal 1 (Dependencies occurrence)	Question 1.1 (Intraprocedural)	Metric 1.1.1 (FDi)	4 (80%)
		Metric 1.1.2 (FIntra)	1 (20%)
		Metric 1.1.2 (FIntra / FDi)	1 (25%)
	Question 1.2 (Global)	Metric 1.2.1 (FGRef)	1 (20%)
		Metric 1.2.2 (FGlobal)	1 (20%)
		Metric 1.2.3 (FGlobal / FGRef)	1 (100%)
	Question 1.3 (Interprocedural)	Metric 1.3.1 (FM)	2 (40%)
		Metric 1.3.2 (FI)	3 (60%)
		Metric 1.3.3 (FIntra)	4 (80%)
Goal 2 (Further characteristics)	Question 2.1 (Direction)	Metric 2.1.1 (M → O)	7 (87.5%)
		Metric 2.1.2 (O → M)	0 (0%)
		Metric 2.1.3 (O → O)	1 (12.5%)
	Question 2.2 (Depth)	Metric 2.2.1 (DD)	1.33 ± 0.47

Table 2
Subject characterization.

Family	Version	Application domain	LOC	Functions	Files	TD
apache	2.4.3	Web server	144,768	3910	362	1.51 ± 1.17
atlantis	0.0.2.1	Operating system	2681	117	103	1.25 ± 0.52
bash	2.01	Command language interpreter	44,824	1647	138	1.80 ± 1.10
bc	1.03	Calculator	5177	166	27	1.24 ± 0.43
berkeley db	4.7.25	Database system	185,111	3468	580	1.39 ± 0.82
bison	2.0	Parser generator	24,325	684	129	1.70 ± 1.06
cherokee	1.2.101	Web server	63,109	1838	346	1.57 ± 1.08
clamav	0.97.6	Antivirus	107,548	2072	377	1.68 ± 1.40
cvs	1.11.21	Revision control system	76,125	1122	236	1.45 ± 0.87
dia	0.96.1	Diagramming software	28,074	814	132	1.06 ± 0.51
expat	2.1.0	XML library	17,103	543	54	1.70 ± 1.01
flex	2.5.35	Lexical analyzer	16,501	277	41	1.76 ± 1.27
fvwm	2.4.15	Window manager	102,301	2141	270	1.27 ± 0.76
gawk	3.1.4	GAWK interpreter	43,070	745	140	1.86 ± 1.17
gnuchess	5.06	Chess engine	9293	217	37	1.86 ± 1.14
gnuplot	4.6.1	Plotting tool	79,557	1861	152	1.90 ± 1.49
gzip	1.2.4	File compressor	5809	114	36	1.46 ± 0.71
irssi	0.8.15	IRC client	51,356	2853	308	1.20 ± 0.64
kin	0.5	Database system	64,120	1248	119	1.31 ± 0.79
libdsmcc	0.6	DVB library	5453	100	30	1.39 ± 0.57
libieee	0.2.11	IEEE standards for VHDL library	5323	197	27	2.30 ± 1.98
libpng	1.0.60	PNG library	44,828	476	61	3.16 ± 1.55
libsoup	2.41.1	HTTP library	40,061	1475	178	1.01 ± 0.10
libssh	0.5.3	SSH library	28,015	943	125	1.64 ± 0.94
libxml2	2.9.0	XML library	234,934	6009	162	2.11 ± 1.51
lighttpd	1.4.30	Web server	38,847	1004	132	1.34 ± 0.90
lua	5.2.1	Programming language	14,503	837	59	1.80 ± 1.71
lynx	2.8.7	Web browser	80,334	1448	117	1.92 ± 1.21
m4	1.4.4	Macro expander	10,469	216	26	2.21 ± 1.41
mpsolve	2.2	Mathematical software	10,278	411	41	1.27 ± 0.45
mptris	1.9	Game	4988	99	29	1.73 ± 1.00
prc-tools	2.3	C/C++ library for palm OS	14,371	369	142	1.19 ± 0.49
privoxy	3.0.19	Proxy server	29,021	478	67	1.75 ± 1.03
rcs	5.7	Revision control system	11,916	299	28	1.96 ± 1.27
sendmail	8.14.6	Mail transfer agent	91,288	861	243	1.76 ± 1.08
sqlite	3.7.15.3	Database system	94,113	2612	134	1.72 ± 0.99
sylpheed	3.3.0	E-mail client	83,528	2597	218	1.34 ± 1.36
vim	7.3	Text editor	288,654	5600	178	2.31 ± 1.42
xfig	3.2.3	Vector graphics editor	70,493	1689	192	2.15 ± 2.20
xterm	2.9.1	Terminal emulator	50,830	989	58	2.05 ± 1.72

LOC: Number of lines of code; **TD**: Average number of macros per macro expression (tangling degree).

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.

For instance, Fig. 13 presents a code snippet containing non-disciplined annotations. In the figure, the statement beginning at line 2 spans through lines 4 or 6, depending on macro `A`. So the full statement can be either `int x = 2 + 3` or `int x = 2 + 4`. If we use *srcML* to generate an AST from this code, we would get an ill-formed XML, as shown in Fig. 14. The XML depicted in this figure could not be parsed, since some of its tags do not nest correctly. More specifically, in the code snippet we have an `#ifdef` clause (see Fig. 13, line 3) followed by an `#else` clause (see Fig. 13, line 5). After *srcML* translates such directives to XML tags, we get a `<cpp:ifdef>` tag (see Fig. 14, lines 19–23) and a misaligned `<cpp:else>` tag (see Fig. 14, lines 30–33), positioned after the closing tag `</decl_stmt>` (see Fig. 14, line 29).

Since *TypeChef* supports such non-disciplined annotations without outputting invalid results, we believe that it is a better

```
1.  void f(){
2.    int x = 2
3.  #ifdef A
4.      + 3;
5.  #else
6.      + 4;
7.  #endif
8.  }
```

Fig. 13. Code snippet containing non-disciplined annotations. Notice that `#ifdef` and `#else` directives surround only part of the statement.

```
1.  <function>
2.    <type>
3.      <name>f</name>
4.    </type>
5.    <name>test</name>
6.    <parameter_list>()</parameter_list>
7.    <block>
8.      {
9.      <decl_stmt>
10.       <decl>
11.         <type>
12.           <name>int</name>
13.         </type>
14.         <name>x</name>
15.         <init>
16.           =
17.           <expr>
18.             2
19.             <cpp:ifdef>
20.               #
21.               <cpp:directive>ifdef</cpp:directive>
22.               <name>A</name>
23.             </cpp:ifdef>
24.             + 3
25.           </expr>
26.         </init>
27.       </decl>
28.       ;
29.     </decl_stmt>
30.     <cpp:else>
31.       #
32.       <cpp:directive>else</cpp:directive>
33.     </cpp:else>
34.     <expr_stmt>
35.       <expr>+ 4</expr>
36.       ;
37.     </expr_stmt>
38.     <cpp:endif>
39.       #
40.       <cpp:directive>endif</cpp:directive>
41.     </cpp:endif>
42.     }
43.   </block>
44. </function>
```

Fig. 14. Ill-formed XML as generated by *srcML* due to a code with non-disciplined annotations. Notice that `cpp:` tags do not nest accordingly.

solution for this task. Apart from *TypeChef* and *srcML*, other tools can generate ASTs from C source code files. *Clang*'s APIs[19] and *Eclipse CDT Parser*,[20] for instance, can parse C code and extract an AST from it. However, such tools do not perform variability-aware analysis. Thus, we would have to define every possible combination of macros to cover all the configuration space, rendering our analysis unfeasible. *SuperC* [27], on the other hand, could be an alternative, since it preserves variability information in the AST. Nevertheless, since our previous studies were focused on error detection [7,28], we have been using *TypeChef* also due to its type checking capabilities, which *SuperC* lacks.

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.

4.4. 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.

Algorithm 1 General algorithm for dependency search

```
ASTS ← set of all abstract syntax trees of a program family
FUNCTIONS ← set of all function definitions within the current AST
VARIABLES ← set of all variables used within the current function
CALLS ← set of all function calls to the current function
USES(v) ← function that returns the set of all uses of the variable v
DEFINITIONS(v) ← function that returns the set of all definitions of the variable v
IS_LOCAL(v) ← function that returns true, if the variable v is a local variable; false, otherwise
IS_GLOBAL(v) ← function that returns true, if the variable v is a global variable; false, otherwise
IS_PARAMETER(v) ← function that returns true, if the variable v is a function parameter; false, otherwise
PC(s) ← 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) ≠ PC(call) then
8:               – There is an interprocedural dependency
9:             end if
10:          end for
11:        else                    ▷ The variable is either local or global
12:          for each definition in DEFINITIONS(variable) do
13:            if PC(definition) ≠ 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
```

In the algorithm we traverse all ASTs for a program family, each corresponding to a source file. For each AST, we look for function definitions, or *FunctionDef* nodes. For every function, we get all variables being used within the function, represented by *Id* nodes (excluding those non-related to variables, belonging to functions or constants, for instance). For each variable, we search for all of its uses within the function, or all *Id* nodes with the same name. Then, we check the variable scope. If it is a function parameter (that is, there is an *Id* node inside a *ParameterDeclarationD* node, for instance), we get all function calls (*FunctionCall* nodes) in all ASTs, comparing the presence condition of each function call with the presence condition of the variable use. Every time such

[19] http://clang.llvm.org.
[20] https://eclipse.org/cdt/.

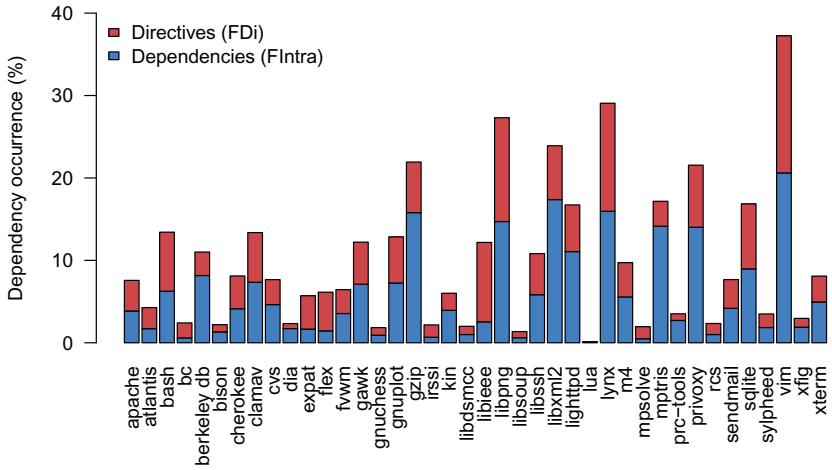


Fig. 15. Percentage of functions with preprocessor directives and intraprocedural dependencies. Notice that, for most families, more than half of functions with directives also have dependencies.

presence conditions differ, we have an interprocedural dependency. Now, if the variable is not a function parameter, it must be either a local or global variable. So, we get all variable definitions (declarations, assignments, and increments/decrements, including various different nodes) inside the current function, comparing their presence condition with the presence condition of the variable use. Once more, every time a definition and a posterior use have different presence conditions, we have a dependency. This time, this dependency can be intraprocedural or global, depending on the variable scope (that is, if its declaration is inside or outside the function). Obviously, we do not consider cases where the presence condition of a variable definition and of its use results in a contradiction when combined. For instance, if a variable has a presence condition `A` for its definition and `!A` for its use, we would not have a feature dependency.

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.

5. Results and discussion

In this section, we answer the research questions based on the results of our empirical study and present the threats to validity. All results are available at the companion web site.[21]

5.1. Question 1.1: 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)*. Fig. 15 shows a bar chart with *FDi* and *FIntra* for all families, expressed as a percentage of the total of functions we analyze. Notice that the bars in the chart are superimposed, so the height of *FDi* also includes the height for *FIntra*. This way, we can also get an idea on how many of the functions with directives have intraprocedural dependencies, by comparing both bars for a given family. According to the chart, both metrics differ considerably depending on the family. 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 ± standard deviation", our results show that 10.09% ± 8.74% of the functions of all families have preprocessor directives within. However, due to the great variation and the heterogeneity of data distribution for this and the next metrics, we also opt to report results using a robust measure of central tendency and statistical dispersion. The median for *FDi* in all families is 7.67%, while its interquantile range (IQR) is 10.56%.

Regarding *FIntra*, while *mpsolve* has no intraprocedural dependencies, this metric reaches 17.04% on *libxml2*, and 18.77% on *Vim*. Considering all families, there are intraprocedural dependencies in 5.79% ± 5.53% of their functions. The median for *FIntra* over all families is 4.03%, while its IQR is 5.94%, characterizing a high dispersion once more.

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. So, by dividing the total of functions with intraprocedural dependencies by the total of functions with preprocessor directives (*FIntra/FDi*) we have a better estimate of the dependency occurrence, 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. That is, the probability of finding intraprocedural dependencies randomly picking a function, for instance, is greater if we consider only the functions with preprocessor directives than if we consider all functions for a program family. Our data shows that 51.44% ± 17.77% of the functions with directives also have intraprocedural dependencies. The median for *FIntra/FDi* is 54.87% and its IQR is 15.11%. Therefore, intraprocedural dependencies are rather common in the product families we analyze.

Table A.1 shows all the values for FDi, FIntra, and FDi/FIntra for all the families we analyze.

5.2. Question 1.2: 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*

[21] http://www.iranrodrigues.com.br/ist2016.

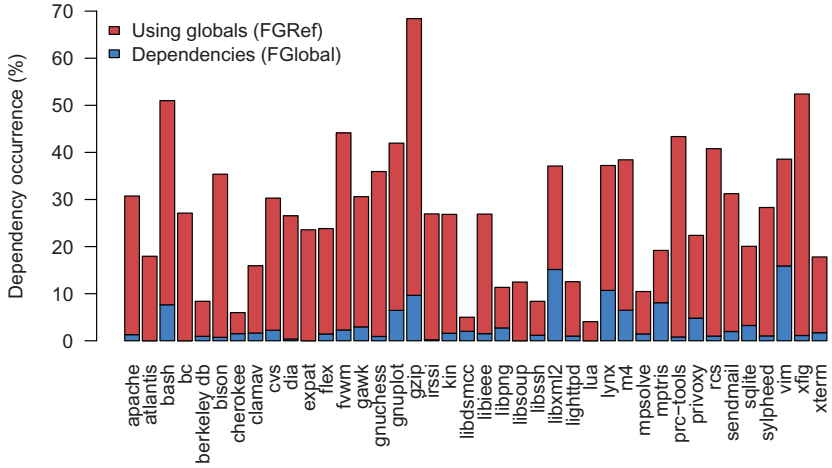


Fig. 16. Percentage of functions referencing global variables and with global dependencies. Notice that the occurrence of global dependencies is relatively low.

global dependencies (FGlobal). Fig. 16 shows a bar chart with superimposed bars representing *FGRef* and *FGlobal*. Notice 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% ± 14.42% of the functions do reference global variables. Median and IQR for *FGRef* are, respectively, 26.93% and 19.81%.

The number of functions with global dependencies (*FGlobal*) also vary across the families we analyze. According to the chart, five of the families do not have any global dependencies: *Atlantis, bc, Expat, libsoup*, and *Lua*. Families with the highest values for *FGlobal* include *Vim* (15.89%) and *libxml2* (15.14%). Considering all families, 3.09% ± 3.95% of the functions have global dependencies. Median and IQR are 1.52% and 2.08%, respectively. However, these percentages relate to the total of functions. 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 5.1, since global dependencies might occur even in functions without such directives.

Now, if we consider only functions which reference global variables, we can better 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. We find this value by diving the number of functions with global dependencies by the number of functions referencing global variables (*FGlobal*/*FGRef*). Our results show that 11.90% ± 12.20% of the functions which refer to global variables also have global dependencies. Median and IQR for the *FGlobal*/*FGRef* ratio are 7.65% and 13.19%, revealing the high dispersion of data. Moreover, 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 5.7.

Moreover, we cannot neglect such dependencies, because depending on the family, the total of global dependencies may be reasonably higher. Besides, Section 3.2 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.

Table A.2 shows the values for *FGRef, FGlobal* and *FGRef/FGlobal* for all families.

5.3. Question 1.3: 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. Fig. 17 shows a bar chart with grouped bars representing *FM* and *FI*, and dots over the bars representing *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% ± 16.62% 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. Median and IQR for *FM* are, respectively, 12.81% and 23.56%, denoting its high variation.

In most families we analyze, *FI* is lower than *FM*, with some exceptions. *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% ± 10.59% of the functions references dependent variables in interprocedural dependencies. Median for *FI* is 8.98%, while its IQR is 11.57%. This means that *FI* varies slightly less than *FM* in the families we analyze.

According to Fig. 17, 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 families, the average number of functions with interprocedural dependencies is 25.98% ± 19.99%. Median and IQR for *FInter* are 20.49% and 30.85%, showing that this metric has the highest variation among all others regarding dependency occurrence.

All these metrics consider all the functions for each family. Once more, we cannot restrict them to consider only the functions

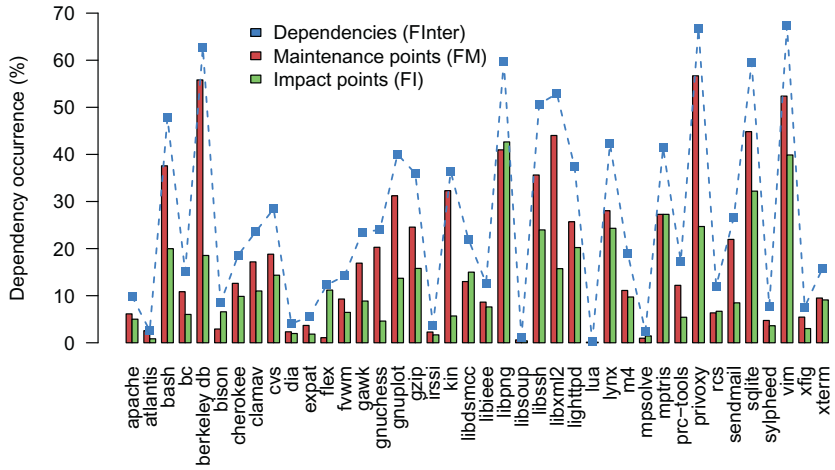
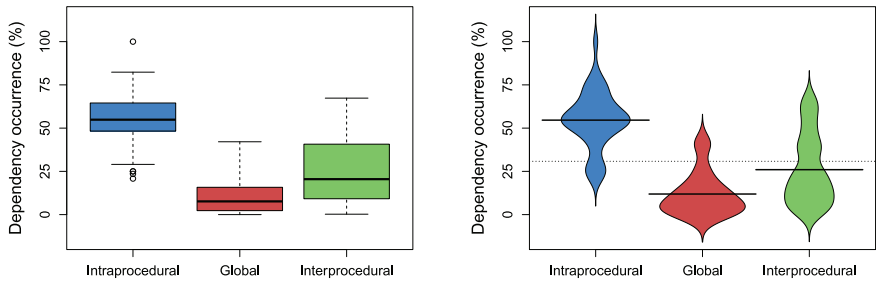


Fig. 17. Percentage of functions with maintenance points, impact points, and interprocedural dependencies. Notice the high variation of *FInter*.



(a) Adjusted boxplots of different types of feature dependency.

(b) Beanplots of different types of feature dependency.

Fig. 18. Dependency occurrence distributions by type.

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. 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 program families (see Section 3.3), making maintenance tasks in the presence of such dependencies rather risky.

Table A.3 shows the values for *FM, FI*, and *FInter* for all the families we analyze.

To better compare intraprocedural, global, and interprocedural dependency occurrence in all families, we plot data regarding *FIntra/FDi, FGlobal/FGRef*, and *FInter* using both an adjusted boxplot [29] (Fig. 18a) and a beanplot [30] (Fig. 18b) chart. While the former clearly shows the values for the first and third quartiles (the bottom and the top of the box, respectively) and the median (the band inside the box), the latter shows variation in between the values, alongside the average for each bean (the line inside the bean) and the overall average (the dotted line). Both charts show that intraprocedural dependency is the most common type of dependency, which has also an almost-symmetrical distribution. Interprocedural dependency follows intraprocedural as the second most common type. Besides, interprocedural dependency has the greatest variation, with the highest boxplot. Global dependency is relatively uncommon, as its 75th percentile is 15.58%, meaning that in 75% of the families its occurrence is less than or equal to 15.58%.

5.4. Question 2.1: 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 → O)*, the *number of optional-to-mandatory dependencies (O → M)*, and the *number of optional-to-optional dependencies (O → O)*.

Fig. 19 shows the distribution of dependency directions according to their types using a beanplot chart. 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 → O), followed by optional-to-optional (O → O) dependencies. Only few are optional-to-mandatory (O → 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 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

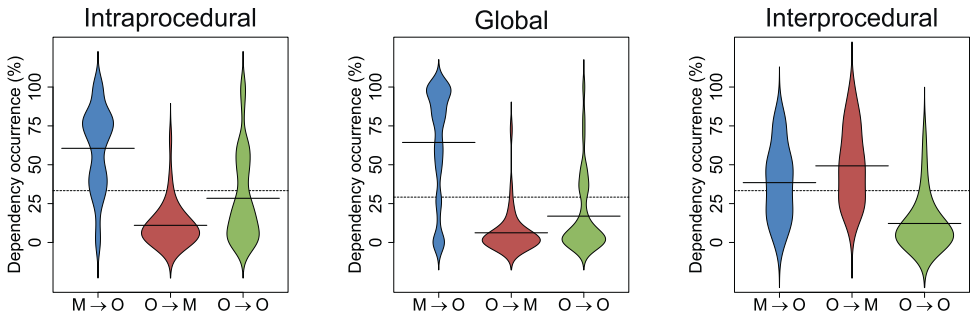


Fig. 19. Dependency directions distribution by type.

```
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;
```

Fig. 20. Code snippet from *libssh*.

better explain this, we refer to the code snippet in Fig. 20. 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 optional 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].

The distribution of interprocedural dependencies is very different (see Fig. 19). 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.

5.5. Question 2.2: what is the dependency depth distribution for interprocedural dependencies?

Fig. 21 shows an adjusted beanplot summarizing the dependency depths for program families. According to the figure, *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*, *libxml2*, and *xterm* have relatively high values for their dependency depths. The highest outlier for *Berkeley DB*, for instance, is 23, meaning that this family shares data across 23 functions before reaching an impact point of a particular interprocedural dependency. Notice that the distribution of dependency depths for *Lua* has even higher values, with depths as high as 29. However, our results for *Lua* are still indefinite. This particular family has a very 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.

Fig. 23 depicts individual histograms of dependency depths for each family we analyze, showing more detail on dependency depths distributions. 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. Thus, although most of interprocedural dependencies have a depth of one, there is no single pattern that fits into all the families we analyze.

Fig. 22 shows a adjusted boxplot summarizing all the depths for the interprocedural dependencies of all the families we analyze. According to the chart, the 25th and 75th percentile are, respectively, 6 and 12. Thus, half of the paths to interprocedural dependencies have depths between 6 and 12. However, the other half of paths have depths between 1 and 6, or above 12, up to 29, meaning that this metric also varies greatly. Considering all families, the dependency depth is 8.69 ± 4.11. Median and IQR are, respectively, 9 and 6.

High values for dependency depths may hinder developer work when maintaining such chained 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 becomes 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.

Table A.4 summarizes the dependency depths for all interprocedural dependencies we collect.

5.6. Question 3.1: how the results of the current study compare with the previous ones?

To answer this question, we compare the number of intraprocedural feature dependencies we obtain in this work with

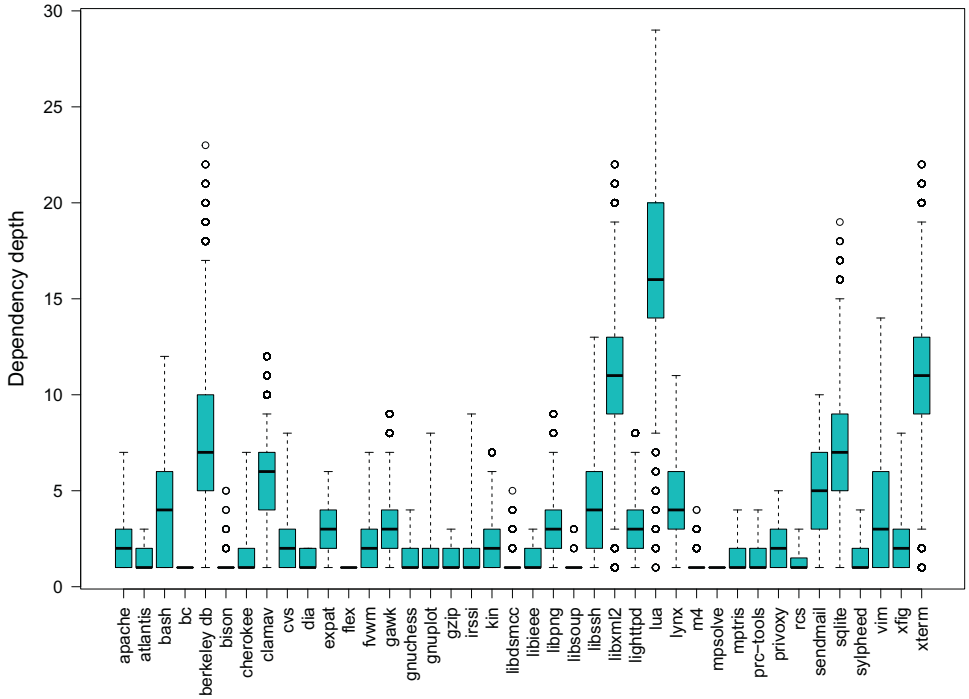


Fig. 21. Dependency depth distribution in the program families.

the number of such dependencies reported in our previous work [8].

Our current study has 17 families in common with the previous study, so we can establish a direct comparison among these families: *Apache, Berkeley DB, Cherokee, ClamAV, Dia, gnuplot, Irssi, libxml2, lighttpd, Lynx, Privoxy, Sendmail, SQLite, Sylpheed, Vim, Xfig*, and *xterm*.

Both studies also use similar metrics to gather information regarding intraprocedural dependencies. For instance, both studies present intraprocedural dependency occurrence as the ratio of the number of functions with intraprocedural dependencies to the number of functions with preprocessor directives. This way, we now compare the values of these metrics across the two studies.

Fig. 24b shows a beanplot featuring the distribution of the number of functions with preprocessor directives on each family. The left side of the bean shows our current results, while the right side of the bean shows results from previous work. Although the overall shape of the bean is nearly symmetrical, notice that its left side stretches upwards beyond its right side, meaning that in our current study we have more functions with dependencies. This is because our previous study did not account for functions with non-disciplined annotations, due to limitations of *srcML*. Now that we consider such functions, we obtain a slightly higher number of functions with preprocessor directives. Considering the 17 families both studies have in common, our previous study reports preprocessor directives in 12.25% ± 7.63% of the functions, with a median of 10.64% and an IQR of 9.16%. Our current results show that there are preprocessor directives in 13.23% ± 10.00% of such families, with values for median and IQR of 11.01% and 9.28%, respectively.

Fig. 24a shows a beanplot with distribution of the number of functions with intraprocedural dependencies on each family. Once more, the left side of the bean shows results from our current study, and the right side shows previous work results. Since we now consider functions with non-disciplined annotations, one might expect that we catch a higher number of dependencies in our current study. However, the right side of the bean presents slightly higher values, meaning that our previous study reports more dependencies than our current study. This is because the notion of intraprocedural dependency in our previous study also includes dependencies regarding function parameters. For instance, Fig. 25 shows a function from *Vim* which our previous study considers as having an intraprocedural dependency. Notice that variable `filename` is a function parameter (see line 1) in a mandatory feature and is used in an optional feature (see line 4). In our current study, we do not consider this an intraprocedural dependency, but an interprocedural dependency, considering that there is a call to this function somewhere else in the code. Thus, considering that the notion of intraprocedural dependency is stricter in our current study, we find less intraprocedural dependencies this time. Considering only the common families, previous study reports intraprocedural dependencies in 9.12% ± 6.63% of functions, with a median of 7.66% and an IQR of 6.64%. Our current study reports intraprocedural dependencies in 7.51% ± 5.78% of functions. Median and IQR are, respectively, 6.23% and 6.98%.

Finally, when we consider the number of functions with intraprocedural dependencies among the functions with preprocessor directives, by dividing these values, the difference between both studies increases. Fig. 24c shows a beanplot with the distribution of this relative intraprocedural dependency occurrence on each family. The asymmetry is now more noticeable, with the right side of the bean featuring higher values. Since previous study considers more dependencies and less directives, consequently this ratio would be higher. Among the common families in both studies, previous work report 69.83% ± 16.33% of functions with intraprocedural dependencies over functions with preprocessor directives. Median and IQR for this metric are 71.52% and 17.25%, respectively. Our current results for these families show that 55.21% ± 11.89% of functions with directives have intraprocedural dependencies, with a median of 52.75% and an IQR of 14.11%. Since our results on in-

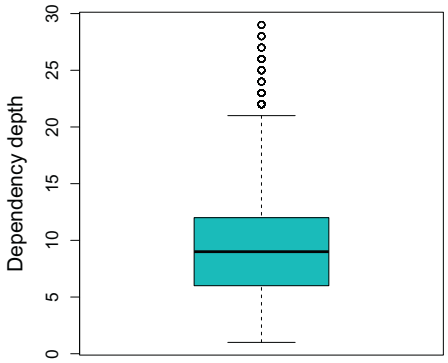
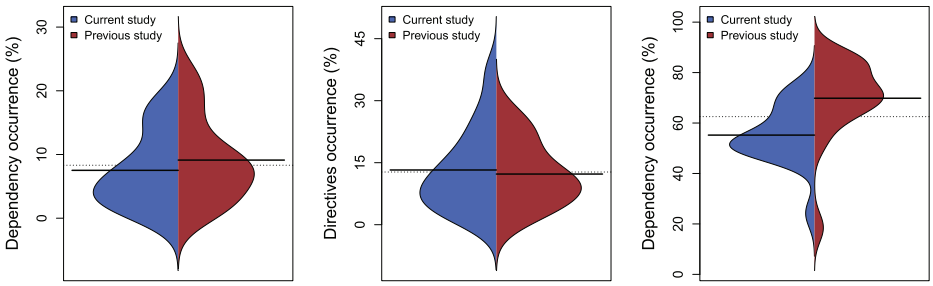


Fig. 22. Dependency depths summary considering all program families.

traprocedural dependencies in this subset of families are close to those regarding all the 40 families (see Section 5.1), we conclude that the results of the previous work would also apply to our entire set, improving its external validity.

Fig. 26 shows a bar chart featuring this last metric for each family. Notice that the bars regarding previous study results have consistently higher values than the corresponding ones from current study, with few exceptions. For instance, *Sendmail* has more intraprocedural dependencies in our current study. This is due to its high number of non-disciplined annotations [11], which prevent our previous study to detect dependencies like the one in Fig. 27. In this figure there is a dependency regarding variable `sigerr`. This variable is defined in a mandatory feature (see line 2), then initialized in an optional feature (see line 7) and later used in a mandatory feature once more (see line 10). As the preprocessor

Fig. 23. Dependency depth distribution in the program families.



(a) Percentage of functions with intraprocedural dependencies in both studies.

(b) Percentage of functions with preprocessor directives in both studies.

(c) Percentage of functions with intraprocedural dependencies among the functions with preprocessor directives in both studies.

Fig. 24. Comparison on metrics regarding intraprocedural dependencies.

```
1.  void workshop_show_file(char *filename) {
2.  #ifdef WSDEBUG_TRACE
3.    …
4.    wstrace(…, filename);
5.  #endif
6.    …
7.  }
```

Fig. 25. Code snippet from *Vim* showing a dependency classified as intraprocedural by previous study [8]. In this work we do not consider such dependency intraprocedural, due to the dependent variable being a function parameter.

directives `#if` and `#else` annotates only part of an `if` block (see lines 4-8), this function could not be properly parsed by *srcML*.

In summary, putting aside the limitations of *srcML* in our previous study and the stricter definition of intraprocedural dependencies from our current study (which excludes function parameters), we may realize that intraprocedural dependencies are still frequent in the subset of families common to both studies. Although our current study reports a lower number of intraprocedural dependencies in such families, both studies agree on the fact they are present in the majority of the functions with preprocessor directives. Moreover, our tool built based on *TypeChef* proved to be more

```
1.  static void * mi_signal_thread(…) {
2.    int sigerr;
3.    …
4.  #if defined(SOLARIS) || defined(__svr5__)
5.    if ((sig = sigwait(&set)) < 0)
6.  #else
7.    if ((sigerr = sigwait(&set, &sig)) != 0)
8.  #endif
9.    {
10.     if (sigerr <= 0)
11.       …
12.   }
13.   …
14. }
```

Fig. 27. Code snippet from *Sendmail* featuring a false negative of previous study [8].

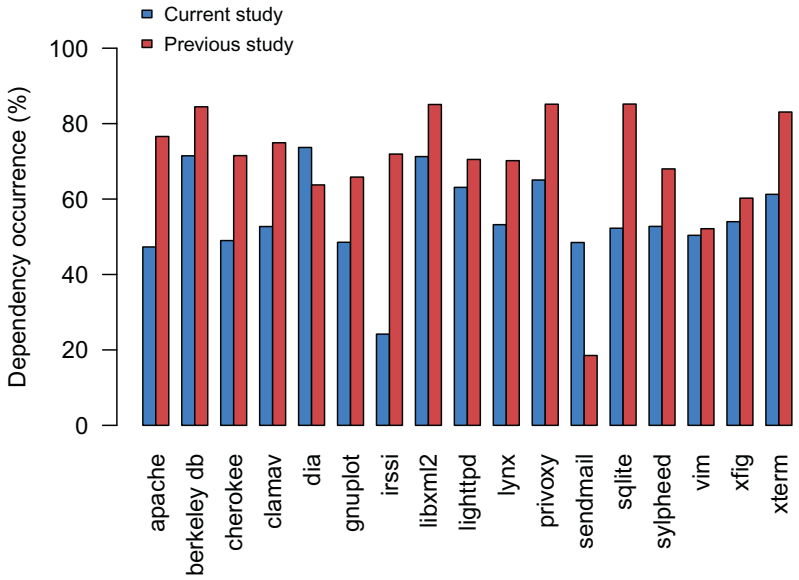


Fig. 26. Comparison of results from this study with results from previous study [8]. Here we consider the percentage of functions with intraprocedural dependencies among the total of functions with preprocessor directives.

reliable than previous one built over *srcML*, in the sense it correctly reports dependencies that are false negatives of previous work.

5.7. Threats to validity

Now, we present the threats to validity. To structure this section, we follow the Wohlin et. al. validity system [31].

Construct validity. We do not have access to the specification of valid configurations (the feature model) of the families we analyze. This raises two issues: first, we cannot know whether a macro in a macro expression refers to an actual feature. So, we might consider a dependency between different fragments of the same feature as a feature dependency because such fragments use different macros. Second, we cannot ensure that all the dependencies we find in our study arise in valid configurations.

Since we consider that macros in a macro expression refer to actual features, the existence of a negation within the macro expression might also pose a threat. For instance, a macro expression `#if !defined(A)` does not map into an `!A` feature. In other words, the negation of a feature is not a different feature. However, such dependencies between macro expressions still could cause problems, as any other feature dependency. For this reason, we still consider them in this work.

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 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 the 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.

Also, we rely on a previous technique [7] to restrict the analysis to program families code only, excluding external libraries, by removing their `#include` directives. In this approach, we still keep the header files of the program families, but exclude the external ones. Considering that external libraries might be platform-specific, resolving such dependencies would be a manual and time-consuming task, hindering our analysis. To prevent most of the syntax errors that the suppression of these libraries would cause, we generate stubs using *C/C++ Development Tooling* (*CDT*) to replace all needed macros and types, placing these stubs in a separate header file. This process is semi-automatic, because *CDT* might not identify all types and macros, so we add them manually whenever possible. Therefore, sometimes we are not able to resolve all missing resources and *TypeChef* cannot successfully generate the AST for some files. We present the rates of successful generation of ASTs for all families in Table A.5. 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 cannot 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 and interquartile range found in some results, for instance, the number of functions with global dependency, evidence their high variation. For that reason, we should not use these results in direct comparison among different families.

6. Consequences

In this section, we present some of the consequences of our results. First, we present evidence that global dependencies are far less common than intraprocedural and interprocedural feature dependencies. Thus, any effort in developing tools and techniques to properly capture feature dependencies might focus firstly on intraprocedural and interprocedural types.

Second, our results might also benefit users of existing tools, such as *Emergo* [2], a dataflow analysis tool that infers interfaces on demand (see Section 7.1). This tool performs feature-sensitive analyzes that rely on a call depth parameter to detect interprocedural dependencies. Its performance depends on user input of such parameter. Low values might prevent the detection of some dependencies, while high values might make analyzes slow. Since we provide the dependency depth distribution, users can better set up such a tool, improving its performance.

Previous work [13,28,32] presented some variability bugs and issues found in program families. A significant amount of these bugs involves the presence of feature dependencies like the ones we present in our study. So, the third consequence we present is that we expect our results could guide the development of tools to go beyond feature dependency detection, but also catch existing bugs and issues that might arise in certain configurations due to feature dependencies. For instance, the tool could use pattern-matching to find potential variability issues due to feature dependencies in a software repository. Such patterns might include, for instance, variables declared in an optional feature but used in a mandatory feature. Assuming that such optional feature is not included in the compilation, we would face an undeclared variable error.

Fourth, in addition to searching for existing bugs regarding feature dependencies, we also believe our results might favor the development of tools to avoid the introduction of such bugs. This way, the same patterns that could be used to find current bugs

could also be useful to warn about a variability bug as soon as the developers adds it. In this sense, the tool could run as an IDE plug-in, for instance, in an interactive manner checking the code as the developer types it.

7. Related work

In this section, we present the related work regarding feature dependencies (Section 7.1), variability bugs (Section 7.2), and C preprocessor usage (Section 7.3).

7.1. 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 [26] to generate abstract syntax trees from source code and collect data regarding intraprocedural dependencies in such families. They found that 65.92% ± 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 [33], 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 maximum depth 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* [34] 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. [35] 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. [36] 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) [37], and Feature Oriented Programming (FOP) [38], 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 feature dependencies and bring evidence that these dependencies might induce to problems.

Apel et al. [14] conducted an exploratory study on the nature of feature interactions in feature-oriented systems. In their work, they proposed a classification of feature interactions according to their order and visibility. Then, they presented preliminary data regarding feature interaction occurrence in four real-world systems: *Linux, BusyBox, GCC*, and *Apache*. Our definition of feature dependency in our work corresponds to an internal, operational feature interaction, since we refer to the sharing of program elements and data among features.

7.2. 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 syntax errors in all possible configurations. Their results showed that such errors are not common in practice. Later, Medeiros et al. [28] presented an empirical study on other types of configuration-related issues. They analyzed 15 popular open-source families using *TypeChef* and found 39 issues regarding undeclared and unused variables and functions. Of this total, approximately 82% relates to the presence of feature dependencies. In our work, we also use *TypeChef* to perform variability-aware parsing. Likewise, we present some variability bugs regarding feature dependencies as motivating examples, but we do not focus on the identification of such errors. Instead, we focus on identifying and quantifying feature dependencies in software families.

Abal et al. [13] 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. Once more, while we bring some motivation examples with bugs regarding feature dependencies, the catalog of such bugs is not an objective of our work. It is important to emphasize that these studies present a relatively small number of variability bugs. However, they consider only bugs that remain after commits. Both studies are missing bugs detected during builds or tests, because they got fixed before committing the code. This way, we have no estimate on how many variability bugs show up during development and get readily fixed.

Melo et al. [32] conducted a quantitative analysis of variability warnings in Linux. By analyzing more than 20,000 valid configurations on both a stable version and an in-development version of Linux, they classified a total of 400,000 compilation warnings. Most common warnings in stable and in-development versions of Linux were due to unused function and unused variable, respectively. Although a feature dependency might trigger an unused variable warning, if we define a feature that declares a variable, but do not define the feature that uses such a variable, they do not investigate the cause of those warnings. Likewise, in our study we detect and classify feature dependencies, but we do not infer what sort of problems they might cause.

7.3. C preprocessor usage

Several studies analyzed the usage of variability mechanisms of C preprocessor, *cpp*. Medeiros et al. [20] conducted an interview study regarding how practitioners perceive the C preprocessor. They interviewed 40 developers, cross-validating their results

with data from a survey with 202 developers, repository mining, and previous studies. Their results show that the C preprocessor is still widely used to solve portability and variability problems. Besides, developers consider variability bugs due to preprocessor misuse easier to introduce, harder to fix, and more critical than other bugs. In our work we present actual variability bugs regarding the presence of feature dependencies.

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 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. [39] 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 dependency occurrence.

Similary, Queiroz et al. [40] 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.

8. Concluding remarks

This work presents an empirical study to better understand the occurrence of fine-grained feature dependencies in C program families.

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

Then, we perform an empirical study of 40 C software families to answer our research questions. Our results show that feature dependencies are fairly common in the families we analyze, except for global dependencies. We detect intraprocedural dependencies in 51.44% ± 17.77% of the functions containing preprocessor directives. We discovered global dependencies in only 12.14% ± 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% ± 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 show that the most common dependency direction is mandatory-to-optional, occurring in 54.47% ± 31.08% of all dependencies. This means that developers are more likely to face a dependency when maintaining mandatory code. We also find that the dependency depth distribution for interprocedural dependencies varies considerably, depending on the family we analyze. In our results, the median interprocedural dependency depth is 9, meaning that functions could share data of a variable 9 times, over different functions, until its value reaches a dependency. When a bug involves such a deep interprocedural dependency, it might become harder to track and fix. Finally, we confirm previous work [8] results on intraprocedural feature dependencies, since they are present in most of the functions containing preprocessor directives, although in this work we present more precise results. Furthermore, we conclude that *TypeChef* is a more appropriate tool to deal with software families containing non-disciplined annotations, compared to *srcML*. This empirical study presents results that complement previous work on feature dependencies, and may be helpful for developers to understand how different types of dependencies occur in practice. Furthermore, our study can possibly guide the implementation of tools and techniques to assist the developer to prevent problems maintaining software families in the presence of feature dependencies.

Acknowledgments

This work has been supported by CNPq 460883/2014-3, 573964/2008-4 (INES), 477943/2013-6, 306610/2013-2, and APQ-1037-1.03/08, CAPES 175956, project DEVASSES (European Union Seventh Framework Programme, agreement PIRSES-GA-2013-612569).

Appendix

Table A.1
Intraprocedural dependencies in the program families. Notice that for most families (25 out of 40), at least half of functions with preprocessor directives also contain intraprocedural dependencies.

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

(*continued on next page*)

Table A.1 (*continued*)

Family	Version	Application domain	FDi	FIntra	FIntra/FDi	NoF
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

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

Table A.2
Global dependencies in the program families. Despite not so common in general, this type of dependency happens quite often in few families, such as *Vim* and *libxml2*.

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

(*continued on next page*)

Table A.2 (*continued*)

Family	Version	Application domain	FGRef	FGlobal	FGlobal/FGRef	NoF
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
sylpheed	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

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

Table A.3
Interprocedural dependencies in the program families. This type of dependency has the greatest variation among 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

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.

Table A.4
Interprocedural dependency depths in the program families. Notice that values for *Lua* are only a lower bound.

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

(*continued on next page*)

Table A.4 (*continued*)

Family	Version	Application domain	Max.	Avg.	St. Dev.
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

Table A.5
Successful AST generation rate in the program families. Most of the families are completely parsed.

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%

(*continued on next page*)

Table A.5 (*continued*)

Family	Version	Application domain	Successful AST generation rate
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%

References

[1] M. Cataldo, A. Mockus, J. Roberts, J. Herbsleb, Software dependencies, work dependencies, and their impact on failures, IEEE Trans. Softw. Eng. 35 (6) (2009) 864–878.

[2] M. Ribeiro, T. Tolêdo, J. Winther, C. Brabrand, P. Borba, 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, in: AOSD Companion, ACM, 2012, pp. 23–26.

[3] M. Ribeiro, P. Borba, C. Kästner, Feature maintenance with emergent interfaces, in: Proceedings of the 36th International Conference on Software Engineering, in: ICSE, ACM, 2014, pp. 989–1000.

[4] H. Spencer, G. Collyer, #ifdef considered harmful, or portability experience with C News, in: USENIX Summer Technical Conference, 1992, pp. 185–197.

[5] J.M. Favre, The CPP Paradox, 9th European Workshop on Software Maintenance, 1995.

[6] J. Liebig, S. Apel, C. Lengauer, C. Kästner, M. Schulze, An analysis of the variability in forty preprocessor-based software product lines, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, in: ICSE, ACM, 2010, pp. 105–114.

[7] F. Medeiros, M. Ribeiro, R. Gheyi, Investigating preprocessor-based syntax errors, in: Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences, in: GPCE, ACM, 2013, pp. 75–84.

[8] M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand, S. Soares, On the impact of feature dependencies when maintaining preprocessor-based software product lines, in: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering, in: GPCE, ACM, 2011, pp. 23–32.

[9] M.D. Ernst, G.J. Badros, D. Notkin, An empirical analysis of c preprocessor use, IEEE Trans. Softw. Eng. 28 (12) (2002) 1146–1170.

[10] A. Garrido, R. Johnson, Analyzing multiple configurations of a C program, in: Proceedings of the International Conference on Software Maintenance, in: ICSM, IEEE, 2005.

[11] J. Liebig, C. Kästner, S. Apel, 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, in: AOSD, ACM, 2011, pp. 191–202.

[12] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, T. Berger, 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, in: OOPSLA, ACM, 2011.

[13] I. Abal, C. Brabrand, A. Wasowski, 42 Variability bugs in the linux kernel: A qualitative analysis, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, in: ASE, ACM, 2014, pp. 421–432.

[14] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, B. Garvin, Exploring feature interactions in the wild: The new feature-interaction challenge, in: Proceedings of the 5th International Workshop on Feature-Oriented Software Development, in: FOSD '13, ACM, 2013, pp. 1–8.

[15] M. Krone, G. Snelting, On the inference of configuration structures from source code, in: Proceedings of the 16th International Conference on Software Engineering, in: ICSE, IEEE Computer Society Press, 1994, pp. 49–57.

[16] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, G. Saake, Do background colors improve program comprehension in the #ifdef hell? Empir. Softw. Eng. 18 (4) (2013) 699–745.

[17] W. Wulf, M. Shaw, Global variable considered harmful, SIGPLAN Not. 8 (2) (1973) 28–34.

[18] J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug? in: Proceedings of the 28th International Conference on Software Engineering, in: ICSE, ACM, 2006, pp. 361–370.

[19] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, C. Weiss, What makes a good bug report? IEEE Trans. Softw. Eng. 36 (5) (2010) 618–643.

[20] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, R. Gheyi, The love/hate relationship with the c preprocessor: an interview study, in: Proceedings of the European Conference on Object-Oriented Programming, in: ECOOP, 2015.

[21] S. Thaker, D. Batory, D. Kitchin, W. Cook, Safe composition of product lines, in: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, 2007, pp. 95–104.

[22] V.R. Basili, G. Caldiera, H.D. Rombach, The goal question metric approach, Encyclopedia of Software Engineering, Wiley, 1994.

[23] B. Ryder, Constructing the call graph of a program, IEEE Trans. Softw. Eng. SE-5 (3) (1979) 216–226.

[24] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.

[25] M. Nagappan, T. Zimmermann, C. Bird, Diversity in software engineering research, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2013, ACM, 2013, pp. 466–476.

[26] J.I. Maletic, M. Collard, H. Kagdi, Leveraging XML technologies in developing program analysis tools, in: Proceedings of 4th International Workshop on Adoption-Centric Software Engineering, in: ACSE, 2004, pp. 80–85.

[27] P. Gazzillo, R. Grimm, Superc: Parsing all of c by taming the preprocessor, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI '12, ACM, 2012, pp. 323–334.

[28] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, R. Gheyi, An empirical study on configuration-related issues: investigating undeclared and unused identifiers, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, in: GPCE 2015, ACM, 2015, pp. 35–44.

[29] E. Vandervieren, M. Hubert, An adjusted boxplot for skewed distributions, Proceedings in Computational Statistics COMPSTAT 2004 (2004) 1933–1940.

[30] P. Kampstra, Beanplot: a boxplot alternative for visual comparison of distributions, J. Stat. Softw., Code Snippets 28 (1) (2008) 1–9.

[31] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.

[32] J. Melo, E. Flesborg, C. Brabrand, A. Wąsowski, A quantitative analysis of variability warnings in linux, in: Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, in: VaMoS '16, ACM, 2016, pp. 3–8.

[33] A. Kenner, C. Kästner, S. Haase, T. Leich, Typechef: Toward type checking #ifdef variability in c, in: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, in: FOSD, ACM, 2010, pp. 25–32.

[34] M. Ribeiro, H. Pacheco, L. Teixeira, P. Borba, Emergent feature modularization, in: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, in: OOPSLA, ACM, 2010, pp. 11–18.

[35] F. Queiroz, M. Ribeiro, S. Soares, P. Borba, 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, in: LA-WASP, 2012.

[36] B. Cafeo, F. Dantas, A. Gurgel, E. Guimaraes, E. Cirilo, A. Garcia, C. Lucena, Analysing the impact of feature dependency implementation on product line stability: an exploratory study, in: 2012 26th Brazilian Symposium on Software Engineering (SBES), 2012, pp. 141–150.

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (Eds.), ECOOP'97 — Object-Oriented Programming, Lecture Notes in Computer Science, 1241, Springer Berlin Heidelberg, 1997, pp. 220–242.

[38] C. Prehofer, Feature-oriented programming: a fresh look at objects, in: M. Akşit, S. Matsuoka (Eds.), ECOOP'97 — Object-Oriented Programming, Lecture Notes in Computer Science, 1241, Springer Berlin Heidelberg, 1997, pp. 419–443.
[39] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Lebenich, M. Becker, S. Apel, Preprocessor-based variability in open-source and industrial software systems: an empirical study, J. Empir. Softw. Eng. (2015).
[40] R. Queiroz, L. Passos, M. Valente, C. Hunsen, S. Apel, K. Czarnecki, The shape of feature code: an analysis of twenty c-preprocessor-based systems, Softw. Syst. Model. (2015) 1–20.