An Empirical Study on Configuration-Related Issues: Investigating Undeclared and Unused Identifiers

Flávio Medeiros
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
flaviomedeiros@copin.ufcg.edu.br

Iran Rodrigues
Federal University of Alagoas
Maceió, Alagoas, Brazil
irgj@ic.ufal.br

Márcio Ribeiro
Federal University of Alagoas
Maceió, Alagoas, Brazil
marcio@ic.ufal.br

Leopoldo Teixeira
Federal University of Pernambuco
Recife, Pernambuco, Brazil
lmt@cin.ufpe.br

Rohit Gheyi
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
rohit@dsc.ufcg.edu.br

Abstract

The variability of configurable systems may lead to configuration-related issues (i.e., faults and warnings) that appear only when we select certain configuration options. Previous studies found that issues related to configurability are harder to detect than issues that appear in all configurations, because variability increases the complexity. However, little effort has been put into understanding configuration-related faults (e.g., undeclared functions and variables) and warnings (e.g., unused functions and variables). To better understand the peculiarities of configuration-related undeclared/unused variables and functions, in this paper we perform an empirical study of 15 systems to answer research questions related to how developers introduce these issues, the number of configuration options involved, and the time that these issues remain in source files. To make the analysis of several projects feasible, we propose a strategy that minimizes the initial setup problems of variability-aware tools. We detect and confirm 2 undeclared variables, 14 undeclared functions, 16 unused variables, and 7 unused functions related to configurability. We submit 30 patches to fix issues not fixed by developers. Our findings support the effectiveness of sampling (i.e., analysis of only a subset of valid configurations) because most issues involve two or less configuration options. Nevertheless, by analyzing the version history of the projects, we observe that a number of issues remain in the code for several years. Furthermore, the corpus of undeclared/unused variables and functions gathered is a valuable source to study these issues, compare sampling algorithms, and test and improve variability-aware tools.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

Keywords Configurable Systems, Faults and Warnings

1. Introduction

Many software systems provide configuration options to tailor the system to different target platforms and application scenarios. The tailored systems differ in terms of features [16], which encapsulate platform-specific code, and optional functionalities. Developers often implement features in C with conditional compilation, through the C preprocessor (i.e., `#ifdef` and `#endif` directives). Feature implementations often cross-cut each other and share program elements such as variables, types, and functions, raising feature dependencies [32, 33]. In real-world software systems with a number of features, such dependencies become complex and may lead to issues that appear only when we select certain combinations of features, i.e., configuration-related faults and warnings.

We found in a previous study that more than 74% of developers believe that configuration-related issues are harder to detect and more critical than issues that appear in all configurations [28], because the variability increases the complexity of configuration-related issues by causing, for example, undesirable feature interactions [1]. However, little effort has been put into understanding configuration-related faults (e.g., undeclared functions and variables) and warnings (e.g., unused variables and functions). Configurable systems are prone to these issues because some of the generated configurations might try to use, for example, variables and functions previously removed during compilation due to the selection of certain features [18]. The corpus of configuration-related issues previously studied focuses only on issues detected by using sampling algorithms (e.g., *t-wise, and statement-coverage*) that check only a subset of valid configurations [15, 31, 37], and by analyzing software repositories [1, 11]. Sampling does not check the entire configuration space and misses issues in configurations not selected by the sampling algorithm. Analyses of software repositories also miss configuration-related issues, because they can only detect issues that developers already fixed. Unfixed issues may camouflage other problems, distract developers, lead to bug reports, and impact software quality.

To ground research on this topic and better understand configuration-related undeclared/unused variables and functions, we perform an empirical study of 15 popular open-source systems written in C, which are statically configurable with the C preprocessor, such as *Bash*, *Gzip*, and *Libssh*. We answer research questions related to the way developers introduce configuration-related issues, the number of configuration options involved, and how long these issues re-

main in source files. Answering these questions is important to quantify and study these issues, understand their peculiarities, and support tool developers, so they can provide means to minimize or even avoid configuration-related issues.

To detect configuration-related issues in a systematic way and go beyond sampling and analysis of repositories, in this paper we consider variability-aware analysis [17, 39] to check the entire configuration space. To detect issues that span multiple files, we perform global analysis (instead of a per-file analysis), and take the header files into account. However, to scale our study and minimize setup problems of variability-aware tools, we propose a strategy that only considers the header files of the target platform. We instantiate our strategy with the *TypeChef* [17] variability-aware parser and we target headers of the Linux platform.

We detect 16 configuration-related faults (2 undeclared variables and 14 undeclared functions) and 23 warnings related to configurability. Warnings include 16 unused variables and 7 unused functions, which do not cause compilation errors, but are still considered by developers through several bug reports.[1] The results reveal that configuration-related issues remain in the code for several years, and others are still not fixed. We report that more than 87% of the configuration-related issues involve two or less configuration options, which support the effectiveness of sampling algorithms, such as *pair-wise* [6, 9], confirming previous findings [1, 11, 27]. Developers introduce 73% of the configuration-related issues by adding new code, such as new source files and functions. This result differs from configuration-related syntax errors, i.e., developers introduce more syntax errors when changing existing code [27]. Furthermore, our results do not support the claim that configuration-related issues occur more frequently in source files with high numbers of preprocessor conditional directives.

In summary, the main contributions of this paper are:

- An empirical study of 15 C software systems to quantify and better understand undeclared/unused variables and functions related to configurability;
- Results showing differences regarding the way developers introduce configuration-related undeclared/unused variables and functions when compared to configuration-related syntax errors;
- Findings that support the effectiveness of sampling analysis;
- A corpus of undeclared/unused variables and functions that can be used by researchers and practitioners to study configuration-related issues, compare sampling algorithms, and test and improve variability-aware tools;
- A strategy that makes feasible the task of analyzing issues related to configurability in several software systems.

We organize the remainder of this paper as follows. In Section 2, we show a real example of configuration-related undeclared function that motivates our study. Then, in Section 3, we describe our strategy to find configuration-related issues. Afterwards, we present the empirical study settings in Section 4, and discuss the results in Section 5. In Section 6, we discuss the implications of the results. Last, we present the related work in Section 7, and the concluding remarks in Section 8.

2. Motivating Example

Developers often use conditional compilation to implement features in several real-world and popular systems [23, 35]. For instance, Figure 1 depicts an excerpt of the C source code of the

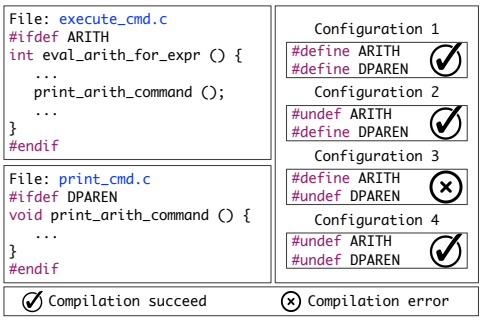

Figure 1. An undeclared function in the *Bash* project that occurs when `ARITH` is enabled and `DPAREN` is disabled.

Bash[2] project, related to executing arithmetic in commands. The arithmetic feature is optional and is included only when we enable the configuration option `ARITH`. This code snippet also contains a configuration option to use the *Korn Shell* evaluation pattern controlled by the configuration option `DPAREN`. We can generate four different configurations from this code snippet: (1) both configuration options enabled; (2) only `DPAREN` enabled; (3) only `ARITH` enabled; and (4) both options disabled.

Most analysis tools for C code, such as *gcc* and *clang*, operate on preprocessed code, i.e., one configuration at a time. By compiling the code snippet of Figure 1 with `ARITH` enabled and `DPAREN` disabled, we get a compilation error. File ***execute_cmd.c*** uses function `eval_arith_for_expr`, which is not declared in ***print_cmd.c*** when `DPAREN` is disabled. Because traditional C compilers check only one configuration at a time, they do not show warning or error messages when one compiles the code depicted in Figure 1 considering the remaining configurations. This is an example of a configuration-related undeclared function exposed only under some combinations of configuration options [11, 21, 41]. Unfortunately, the space of possible combinations is exponential in the worst case, and it is usually too large to explore exhaustively.

Previous work [1, 10–12] studied configuration-related issues similar to the one we discuss by analyzing software repositories [1, 12], and using a number of sampling algorithms [15, 30, 31, 36]. Such studies focus on issues that developers have already fixed in software repositories, and do not check all configurations of the source code (i.e., sampling checks only a subset of valid configurations), potentially missing configuration-related issues. In addition, a number of previous studies perform a per-file analysis [13, 27, 36], which do not detect issues that span multiple files. This specific issue in *Bash*, for example, spans multiple files. Having two different files makes the task of detecting and fixing the issues harder, specially in case we have two or more developers maintaining these source files.

To detect configuration-related issues in a systematic way, we can use variability-aware tools capable of checking all configurations of the source code. However, when using these tools, there is a time-consuming setup that hinders us from scaling the analysis for several software systems. Therefore, we propose a strategy to minimize this scalability problem (Section 3), and report an empirical study of 15 popular open-source systems (Section 4) to better understand configuration-related issues.

3. Detecting Configuration-Related Issues

In this section, we present our strategy to detect configuration-related issues in software systems, explaining it using constructs of the C language. Our strategy parses the system source code (C files

[1] `https://bugzilla.gnome.org/show_bug.cgi?id=461011`, 167715, and 401580, as discussed in our previous study [33].

[2] `http://www.gnu.org/software/bash/`

only) without preprocessing and generates an Abstract Syntax Tree (AST) for each source file. We create a data structure with global information about variables, functions, and preprocessor macros defined in all source files to check dependencies and uses. This data structure also maintains information regarding which functions, and variables are defined and used in each system configuration, allowing us to detect issues related to configurability [19]. Figure 2 explains the three steps of our strategy, detailed in what follows.

The goal of *Step 1* is to enable us to analyze several software systems. Variability-aware analysis tools can identify certain classes of faults (mostly syntax and type issues) by covering the entire configuration space. A common difficulty in setting up these tools is that many configuration options are related to platform-specific definitions and libraries. Hence, our strategy preprocesses the included header files and generates platform-specific versions of these files. Despite focusing only on one platform at a time, the strategy enables us to analyze several software systems in such a platform. To generate platform-specific headers, the strategy removes the preprocessor conditional directives (such as `#ifdef` and `#endif`) of the header files, according to the characteristics of a specific platform. For instance, Figure 3 presents how we generate platform-specific headers for *Linux* using *gcc*. After preprocessing the source code, the C preprocessor removes the preprocessor conditional directives associated with the `WIN32` configuration option, and resolves the includes. Thus, our strategy considers only one configuration of each header file. To instantiate our strategy for different platforms, one needs to generate platform-specific header files for each different target platform. However, notice that we do not preprocess the C files. For those files, we consider the entire configuration space, as we explain in what follows.

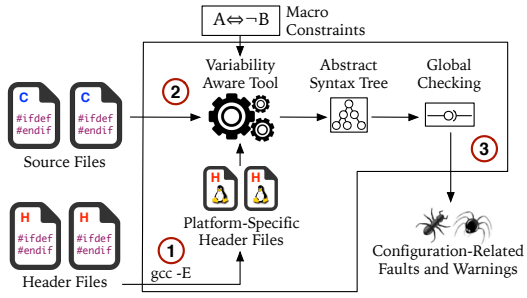

Figure 2. Strategy to detect configuration-related issues.

In *Step 2*, we use a variability-aware tool to parse the source code (C files) and generate an AST for each source file. When parsing each source file, the tool uses the platform-specific header files generated in the first step. Since we do not preprocess the source files, they still contain preprocessor conditional directives. Therefore, the resulting AST has variable nodes to represent the optional and alternative code blocks. Figure 4 depicts a simplified AST enhanced with variability information from the code excerpt of Figure 1. During this step, our strategy may receive any known constraints to eliminate invalid configurations (e.g., configuration options `A` and `B` are mutually exclusive). We pass this information to the variability-aware tool, which then ignores the invalid configurations. Unfortunately, the majority of C open-source projects do not have such constraints information defined explicitly.

Step 3 uses the abstract syntax trees of the source files to detect the issues. Notice that we consider the abstract syntax trees of all source files, which allow us to detect configuration-related issues that span multiple files. Similar to safe composition approaches [7, 38], we check simultaneously for all configurations, if the required definitions (variables, functions, and preprocessor

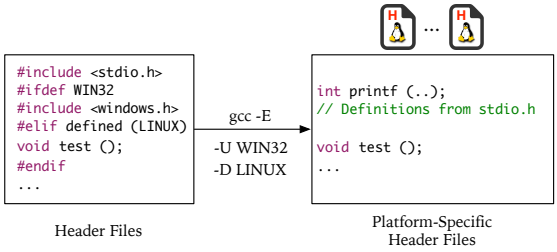

Figure 3. Generating platform-specific headers for *Linux*.

macros) are being provided. However, we are also able to capture warnings such as unused variables and functions. For instance, we can see in Figure 4 that `ARITH` requires a function definition (`print_arith_command`) provided by `DPAREN`, as discussed in Section 2. For this reason, *Bash* has an undeclared function when we enable `ARITH` and disable `DPAREN`. The latter provides the function `print_arith_command`, and no other source file provides this required function definition for this specific configuration. At this point, we have the following variability-aware checkers implemented: undeclared variables, unused variables, undeclared functions, and unused functions. Nonetheless, we can extend our infrastructure to add other checkers, such as checking for return types, and fields in structure declarations.

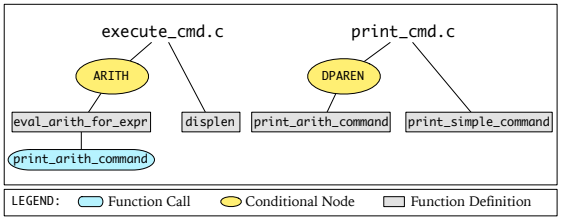

Figure 4. Simplified AST of the code excerpt of Figure 1.

4. Study Settings

In this section, we present the settings of the empirical study we perform to better understand configuration-related issues. To perform the study, we instantiate our strategy to detect issues using the well-known *gcc* compiler, *TypeChef* [17], a variability-aware parser widely used in previous studies [25–27, 40], and the *Linux* operating system to generate platform-specific header files. We choose *Linux* because it provides simple and effective packaging tools to identify and install the software system dependencies.

In particular, this empirical study addresses the following research questions:

- **RQ1.** What are the frequencies of undeclared variables, unused variables, undeclared functions, and unused functions?
- **RQ2.** Do configuration-related issues involve multiple configuration options?
- **RQ3.** Do configuration-related issues span multiple files?
- **RQ4.** How do developers introduce configuration-related issues?
- **RQ5.** For how long do configuration-related issues remain in source files?
- **RQ6.** Do configuration-related issues occur more frequently in source files with many configuration options?

Before answering the research questions, we consider feedback from the actual system's developers to confirm each configuration-related issue. So, all numbers we report here do not include false positives. We also receive feedback regarding configuration option constraints and we use this information to avoid checking invalid configurations in *Bash*, *Libssh*, and *Privoxy*. To answer **RQ1**, we execute our four checkers (i.e., undeclared function, unused function, undeclared variables, and unused variables) and count their frequencies. Regarding **RQ2**, we count the number of configuration options involved in each configuration-related issue. To answer **RQ3**, we identify the issues that span multiple source files. In **RQ4**, we analyze each issue to verify how developers introduced them by using the source file history in the software repository. Regarding **RQ5**, we analyze the dates that developers introduced and fixed the issues to measure the time in-between. To answer **RQ6**, we count the number of preprocessor conditional directives of each source file with at least one issue and compare with the average number of preprocessor conditional directives.

4.1 Subject Selection

We analyze 15 subject systems written in C ranging from 4,988 to 44,828 lines of code. These systems are from different domains, such as revision control systems, programming languages, and games. Furthermore, we consider mature systems with many developers as well as small systems with few developers. We select these subject systems inspired by previous work [8, 10, 24, 27]. We present the details of each subject system in Table 1. For the subject systems with *git* software repository available, we also consider the commits history of the source files, as we present in Table 2.

Table 1. Subject characterization and number of issues.

Family	Version	Application Domain	LOC	Files	Issues
Bash	4.2	Language interpreter	44,824	138	20
Bc	1.03	Calculator	5,177	27	
Expat	2.1.0	XML library	17,103	54	
Flex	2.5.37	Lexical analyzer	16,501	41	
Gnuchess	5.06	Chess player	9,293	37	1
Gzip	1.2.4	File compressor	5,809	36	3
Libdsmcc	0.6	DVB library	5,453	30	
Libpng	1.6.0	PNG library	44,828	61	9
Libsoup	2.41.1	SOUP library	40,061	178	
Libssh	0.5.3	SSH library	28,015	125	2
Lua	5.2.1	Language Interpreter	14,503	59	2
M4	1.4.4	Macro expander	10,469	26	1
Mptris	1.9	Game	4,988	29	
Privoxy	3.0.19	Proxy server	29,021	67	1
Rcs	5.7	Revision control system	11,916	28	
Total			**287,961**	**936**	**39**

4.2 Instrumentation

We use the strategy presented in Section 3 to investigate configuration-related issues. We use *TypeChef* version 0.3.3 to parse all configurations of the source code. Furthermore, we also count the number of lines of code, and the number of files of each subject system using the Count Lines of Code (*CLOC*) tool version 1.56, which eliminates blank lines and comments. Finally, we use *Git* version 1.7.12.4 to identify changes in source files, and to get information about the project's repositories.

5. Results and Discussion

In this section, we discuss the results of our empirical study and answer the research questions. Table 1 presents the number of configuration-related issues we detect in each subject system. Notice that we confirm all issues so that the numbers we report do not include false positives. All results are available at the companion web site.[3] We answer the research questions in what follows.

Table 2. General information about the software repositories

Project	Developers	Commits	First Commit	Last Commit
Bash	2	109	Aug 26, 1996	May 19, 2015
Expat	13	47	Jan 12, 1970	Jun 22, 2014
Flex	5	1,607	Nov 8, 1987	Mar 21, 2012
Gnuchess	1	236	Oct 8, 2001	Jan 25, 2011
Gzip	13	464	Jan 21, 1993	Mar 16, 2015
Libpng	5	2,195	Jul 20, 1995	Mar 26, 2015
Libsoup	186	2,225	Dec 5, 2000	May 27, 2015
Libssh	26	3,206	Jul 5, 2005	May 8, 2015
Lua	7	85	Sep 30, 2010	Fev 19, 2014
M4	13	986	Feb 17, 2000	Dec 12, 2014
RCS	5	915	Nov 18, 1989	Jun 5, 2012
Total		**12,075**		

5.1 What are the frequencies of undeclared variables, unused variables, undeclared functions, and unused functions?

We analyze 15 subject systems. We find 14 undeclared functions; 7 unused functions; 2 undeclared variables; and 23 unused variables, as presented in Figure 5. Overall, we detect 39 configuration-related issues. Because of variability, more than 74% of developers believe that configuration-related issues are more difficult to detect than issues that appear in all configurations [28]. Nevertheless, during the analysis, we also detect issues that occur in mandatory code, i.e., issues that appear in all configurations. Yet, because we focus on configuration-related issues, we remove numbers related to mandatory code from our statistics. Figure 6 presents an example of undeclared variable. This code excerpt is part of the *Libpng* project, and it fails to compile when we enable `SPLT` and disable `POINTER`. As we can see, developers declare variable `p` at line 4 only when `POINTER` is enabled. The problem is that they use this variable at lines 8 and 9, in which configuration option `POINTER` is disabled, causing a compilation error.

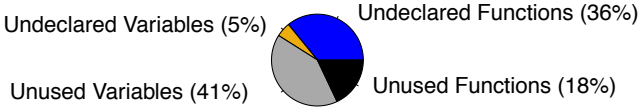

Figure 5. Kinds of configuration-related issues.

We also find unused variables and functions. Traditional C compilers raise warnings like unused variables and functions when developers set specific command line parameters. Still, we are able to find several unused variables and functions related to configurability. As these warnings do not cause compilation errors, developers might neglect them, even in mandatory code. Figure 7 presents a code excerpt with an unused variable in *Libssh*. In this code excerpt, variable `strong` is not used when we disable `LIBCRYPTO` and enable `LIBCRYPT`. The warning disappears when the opposite configuration selection happens. Although unused variable is a simple warning, some developers still care about them, by raising bug reports and providing patches to fix them. Indeed, we find bug reports and patches to fix unused variables and functions, such as the one to fix the *Libssh* warning[4] presented in Figure 7.

[3] `http://www.dsc.ufcg.edu.br/~spg/gpce2015`

[4] `http://www.dsc.ufcg.edu.br/~spg/gpce2015/libssh.html`

Overall, we conclude that the configuration-related issues we focus on this paper are not so common in the repositories we study. Still, it seems they are more common than configuration-related syntax errors. In a previous work, we analyze 41 software families but find only 24 syntax errors in valid configurations [27].

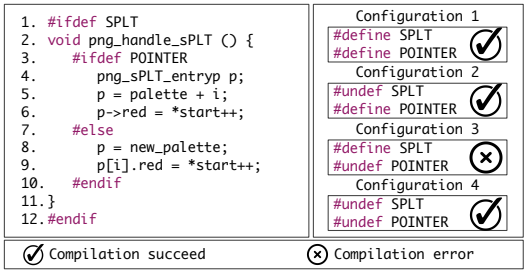

Figure 6. An undeclared variable in the *Libpng* project that occurs when `SPLT` is enabled and `POINTER` is disabled.

5.2 Do configuration-related issues involve multiple configuration options?

We find that the majority of configuration-related issues (more than 87%) involve two or less configuration options. Table 3 details the number of configuration options involved in the issues. For example, we find 16 issues involving only one configuration option. We also find 18 issues depending on two options, 3 issues involving three configuration options, and only 2 issues when setting four or more configuration options.

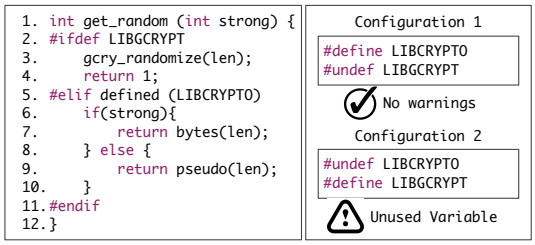

Figure 7. An unused variable in the *Libssh* project that occurs when `LIBCRYPTO` is disabled and `LIBCRYPT` is enabled.

We observe similar results when comparing to the investigation of configuration-related syntax errors we report in our previous study [27]. Furthermore, studies that detect configuration-related issues by analyzing software repositories and by using sampling analysis also find similar results [1, 11]. Because we use a different technique, i.e., variability-aware analysis, our empirical study provides more evidence that configuration-related issues involving more than two configuration options are not common in C open-source systems. Our findings also support the effectiveness of sampling algorithms, as the majority of the issues do not involve high numbers of configuration options.

5.3 Do configuration-related issues span multiple files?

Previous studies perform per-file instead of global analysis [27, 36]. Per-file analysis can only detect issues that do not span multiple files. Global analysis considers information across all source files instead of considering each file separately. In our empirical study, however, the results reveal that the majority of configuration-related issues occur in single files. Other 13 issues (33%) span multiple source files.

Table 3. Configuration options involved in issues.

Some configuration options enabled	5
a	4
$a \wedge b$	1
Some configuration options disabled	20
$!a$	12
$!a \wedge !b$	7
$!a \wedge !b \wedge !c$	1
Some options enabled and some disabled	14
$a \wedge !b$	9
$a \vee !b$	1
$a \wedge !b \wedge !c$	2
$a \wedge !b \wedge !c \wedge !d$	1
$a \wedge b \wedge c \wedge d \wedge e \wedge f \wedge !g$	1

5.4 How do developers introduce configuration-related issues?

We investigate how developers introduce the issues we find in our empirical study. Our goal here is to identify whether developers introduce more issues when implementing new functionalities or fixing other bugs in the source code. According to the results, developers introduce more issues (73%) when introducing new functionalities, such as a new source file, or adding a new function. In contrast to configuration-related syntax errors, the results are the opposite: developers introduce the majority of syntax errors when fixing existing code [27]. We now present the results in the following order: undeclared functions, undeclared variables, unused functions, and unused variables.

Developers introduce configuration-related undeclared functions in three different cases: **(I)** adding a call to existing functions without checking the preprocessor conditional directives that encompass such function definitions; **(II)** adding a call to a function without including the header file with the function definition; and **(III)** changing a function definition without modifying the corresponding function calls. Figure 8 illustrates these three cases with small code excerpts. We find that developers introduce 79% of the undeclared functions with case **(I)**: *Bash* (1), *Gnuchess* (1), *Gzip* (2), *Libpng* (6), and *Privoxy* (1); for case **(II)**, we have one (7%) undeclared function: *Lua* (1); and 14% of the undeclared functions follow case **(III)**: *Libssh* (1), and *Lua* (1).

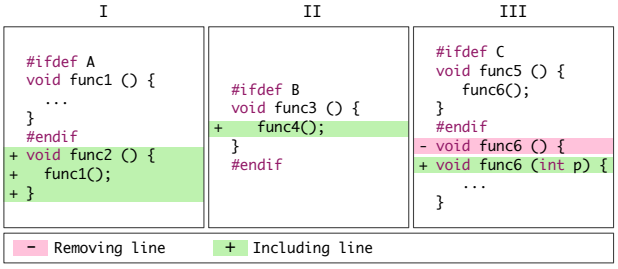

Figure 8. Introducing configuration-related undeclared functions.

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

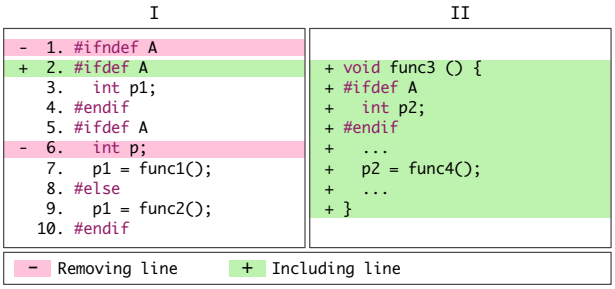

Figure 9. Introducing configuration-related undeclared variables.

Developers introduce unused functions in two cases: **(I)** conditionally defining a function and calling it in code encompassed with different preprocessor conditional directives; and **(II)** removing a call to a conditionally defined function, and adding another call to a mandatory function. Figure 10 depicts these two cases. We find that 86% of unused functions follow case **(I)**: *Bash* (4), *Libpng* (1), and *M4* (1); and 14% follow case **(II)**: *Libpng* (1).

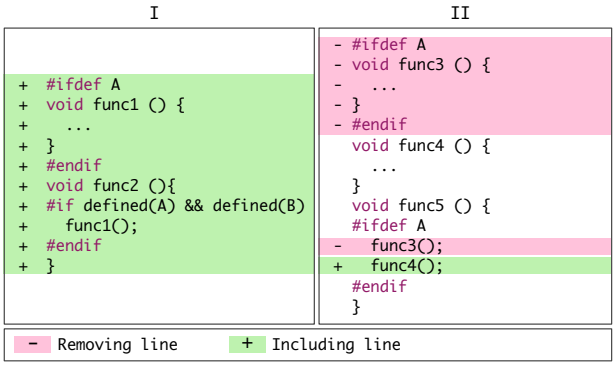

Figure 10. Introducing configuration-related unused functions.

Regarding the unused variables we find in our study, developers introduce them following three cases: **(I)** adding a new variable to an optional code without using such variable; **(II)** adding a new variable to mandatory code and using this variable only in optional code; and **(III)** moving the uses of a variable to optional code. Figure 11 depicts these three cases. Case **(II)** is the most common (50%): *Bash* (8); six unused variables (38%) follow case **(I)**: *Bash* (6); and 12% follow case **(III)**: *Bash* (1), and *Libssh* (1).

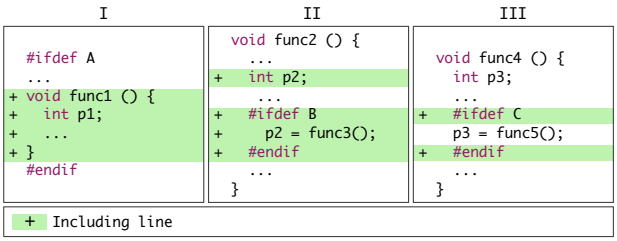

Figure 11. Introducing configuration-related unused variables.

We also analyze whether developers introduce issues by changing mandatory or optional code. Our results reveal that the number of issues developers introduce when working on optional code (55%) or in mandatory code (45%) is fairly similar.

5.5 For how long do configuration-related issues remain in source files?

In this section, we analyze the time that developers take to fix configuration-related issues. Our results show that the time varies from days to years. For example, developers fix an issue of the *Libssh* system (`keyfiles.c`) after 69 days. In contrast, the issue of *Bash* we discuss in Section 2 remains in the source code since July 2004. The *Bash* developers accept our patch to fix this issue. Table 4 depicts the time developers take to fix some issues we find in our empirical study. Notice that we only list issues we know exactly when developers introduce them, and issues already fixed.

Table 4. Time to fix configuration-related issues.

Family	File	Kind	Days to Fix
Gzip	deflate.c	undeclared function	6,678
Gzip	util.c	undeclared function	5,983
Libpng	iccfrompng.c	undeclared function	1,289
Libpng	iccfrompng.c	undeclared function	1,289
Libpng	iccfrompng.c	undeclared function	1,289
Libpng	iccfrompng.c	undeclared function	1,289
Libpng	pngpixel.c	undeclared function	1,289
Libpng	pngpixel.c	undeclared function	1,289
Libpng	pngrutil.c	undeclared variable	530
Libssh	keyfiles.c	undeclared function	69
Libssh	dh.c	unused variable	268
Lua	loadlib_rel.c	undeclared function	748
Lua	loadlib_rel.c	undeclared function	999

Developers may take a long time to fix issues due to different reasons. First, the configuration-related issues may be difficult to detect because of variability [28]. Second, developers might have problems to understand code they are not familiar with, possibly written by another developer [28]. Third, in case the issues arise in not exercised or deliverable configurations, developers tend to rank the fixing task as lower priority [27].

5.6 Do configuration-related issues occur more frequently in source files with many configuration options?

To answer this question, we compute the average of preprocessor conditional directives regarding all source files, i.e., we count the number of preprocessor conditional directives of all source files and divide by the number of files. We thus have the average for each project. Then, for each file that contains at least one issue, we compare the number of directives with the average.

Figure 12 illustrates the averages (and standard deviation) of directives for each system. We represent the averages by using filled circles. In contrast, we represent the number of directives of each file with at least one issue using open circles. We denote the standard deviation by using vertical lines. Our results do not support the claim that configuration-related issues occur more frequently in source files with high numbers of preprocessor conditional directives. We can see in Figure 12 that the numbers of conditional directives of some files with at least one issue are close to the average. In particular, only 9 out of 29 files (31%) with at least one issue contain more conditional directives than the average increased by the standard deviation.

5.7 Submitting Patches to Fix Configuration-Related Issues

We submitted 30 patches—for each issue not fixed—to 3 families: *Bash* (20), *Libpng* (6), and *Libssh* (4). We submitted these patches using bug tracking systems and via email directly to the main developer of the system. We consider that developers accept a patch when they mention that it is a problem by email, or keep the patch open after updating information, such as priority. Conversely, we

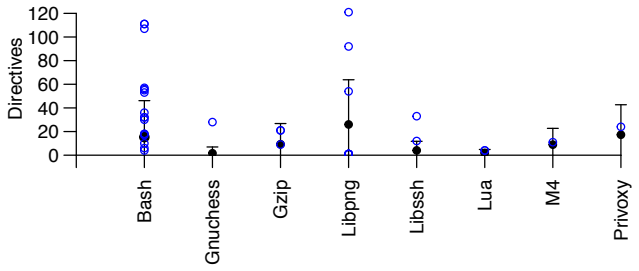

Figure 12. Average numbers of conditional directives.

consider that developers reject the patch when they mention it is not a problem by email, or update this information on the patch. Thus, developers accepted 7 patches, rejected 4 patches, ignored 15 patches, and we did not receive feedback regarding the 4 patches we submit to *Libssh*. Notice that we do not consider these 4 issues of *Libssh* in our statistics. We present information about the patches we submit in Table 5, which does not include the 15 patches ignored by the *Bash* developers.

We submitted 20 patches to *Bash* and developers accepted only one. Four issues do not happen in practice (i.e., they are false positives), as the build system avoids the specific configurations they appear in. In addition, one particular developer confirmed but ignored the 15 patches we report to fix unused variables:

> "*I don't care about unused variables too much; the compiler gets rid of them. So, they have no cost.*"

Despite having no performance cost, unused variables and functions slightly pollute the code, which might explain other developers caring about them. For instance, we find a single patch to *Gnuchess* that fixes 19 unused variables.[5]

Regarding the patches we submit to *Libpng*, developers accepted all 6 patches, and they have already fixed the issues in the software repository.[6]

Table 5. Patches we submit to software systems.

Family	File	Accept	Status	Variable / Function
Bash	execute_cmd.c	valid	open	`arith_cmd` undeclared
Bash	bashline.c	invalid	closed	`add_history` undeclared
Bash	flags.c	invalid	closed	`init_hist` undeclared
Bash	jobs.c	invalid	closed	`imp_sigchld` undeclared
Bash	strerror.c	invalid	closed	`strerror` undeclared
Libpng	iccfrompng.c	valid	fixed	`init_io` undeclared
Libpng	iccfrompng.c	valid	fixed	`get_iCCP` undeclared
Libpng	iccfrompng.c	valid	fixed	`read_info` undeclared
Libpng	iccfrompng.c	valid	fixed	`destroy` undeclared
Libpng	pngpixel.c	valid	fixed	`get_depth` undeclared
Libpng	pngpixel.c	valid	fixed	`get_type` undeclared
Libssh	sftp.c	-	open	`sftp_read` undeclared
Libssh	main.c	-	open	`sftp_open` undeclared
Libssh	torture_rand.c	-	open	`ssh_pthread` undeclared
Libssh	chmodtest.c	-	open	`sftp_new` undeclared

(-) We did not receive feedback regarding 4 issues of *Libssh*, so we do not consider them in our statistics.

[5] We list a number of patches (from different projects and developers) to fix warnings, such as unused variables and functions in the project's website: `http://www.dsc.ufcg.edu.br/~spg/gpce2015/warnings.html`.

[6] See the patch submitted to fix six undeclared functions in the *Libpng* code: `http://www.dsc.ufcg.edu.br/~spg/gpce2015/libpng.html`.

5.8 Threats to Validity

Checking whether the configuration-related issues appear in valid configurations or represent false positives threatens **construct validity**. To minimize this threat, we perform two tasks: (i) for the systems we know configuration option constraints in advance, we set *TypeChef* to take them into account and consequently avoid analyzing invalid configurations (e.g., *Bash*, *Libssh*, and *Privoxy*); and (ii) ask the actual developers to confirm each issue not fixed in the software repository. Unfortunately, most projects do not provide constraints information explicitly.

We analyze the issues manually, which is a time-consuming and error-prone activity. This threatens **internal validity**. Nevertheless, because we get feedback from the actual developers and confirm the issues we report, we minimize this threat. Furthermore, we do not consider build-system information, which is inherently difficult to analyze automatically by using make files. In this way, our study does not consider that certain files are not compiled in all configurations (i.e., depending on certain configuration options). Thus, our strategy may miss to detect configuration-related issues that are not explicitly surrounded by preprocessor conditional directives.

To scale our analysis, our strategy considers only one configuration of header files. We use *gcc* and generate only header files for the *Linux* platform. However, notice that we may face false negatives due to this limitation, which threatens **external validity**. In this context, our strategy may miss some configuration-related issues that occur only for other platforms, such as *Windows* and *Mac OS*. Still, in our study, we find 39 configuration-related issues, and we confirm them either by checking if developers fixed them in software repositories or by getting feedback from developers. Also, we analyze subject systems of different domains, sizes, and different number of developers. We select well-known and active C software systems used in industrial practice. Their communities exist for years and they are in constant development. Therefore, we alleviate this threat.

6. Implications

This section presents some implications that our results bring to practice. First, we find evidence that configuration-related issues remain in source files for several years, while issues that appear in all configurations are normally fixed within a few days. Thus, it seems that variability makes the detection of even simple bugs—such as undeclared variables—more difficult.

Second, we find some differences regarding distinct kinds of configuration-related issues. Developers normally introduce configuration-related syntax errors when changing code [27]. On the other hand, developers frequently introduce undeclared/unused variables and functions when adding new code. Hence, instead of using only one particular technique to detect both configuration-related syntax errors and other issues, our results support the claim that we need different strategies and tools to properly catch them. For instance, the use of lightweight tools that check for syntax errors on the fly, and more time-consuming analysis with global information to detect other issues (including undeclared variables and unused functions) only when introducing new source files or before submitting new code versions to project repositories.

Previous studies perform analysis of software repositories and sampling analysis to detect configuration-related issues and report that most configuration-related issues do not involve several configuration options. However, as discussed, they miss issues. Thus, as we perform variability-aware analysis, which might minimize the number of missed configuration-related issues, the findings of our study increases evidence that configuration-related issues involve, in most cases, one or two configuration options. So, the third implication we present is that our findings regarding the num-

Table 6. Configuration-related issues detected in our empirical study.

Project	File	Kind	Fix/New	Optional/Mandatory	Macros	Single/Multiple	Directives	LOC
bash	vi_mode.c	unused variable	fix	optional	1	single	36	2071
bash	macro.c	unused variable	new	optional	1	single	4	271
bash	display.c	unused variable	fix	optional	1	single	53	2688
bash	array.c	unused variable	new	optional	1	single	6	1085
bash	bashline.c	unused variable	new	optional	1	single	57	3611
bash	braces.c	unused variable	fix	optional	1	single	18	680
bash	error.c	unused variable	new	optional	2	single	30	462
bash	execute_cmd.c	unused variable	new	optional	1	single	111	4028
bash	finfo.c	unused variable	new	mandatory	1	single	10	569
bash	finfo.c	unused variable	new	mandatory	1	single	10	569
bash	general.c	unused variable	new	mandatory	1	single	17	903
bash	malloc.c	unused variable	new	optional	3	single	55	1107
bash	pcomplete.c	unused variable	new	mandatory	1	single	32	1462
bash	shell.c	unused variable	fix	mandatory	1	single	18	208
bash	watch.c	unused variable	new	mandatory	1	single	4	150
bash	execute_cmd.c	undeclared function	fix	optional	2	multiple	111	4028
bash	variables.c	unused function	new	optional	2	multiple	107	4793
bash	pcomplete.c	unused function	-	-	2	multiple	32	1546
bash	bashline.c	unused function	new	optional	2	multiple	56	3704
bash	array.c	unused function	new	-	4	multiple	6	1130
gnuchess	getopt.c	undeclared function	new	optional	1	single	28	1067
gzip	deflate.c	undeclared function	new	optional	1	single	21	763
gzip	util.c	undeclared function	new	-	2	single	9	462
gzip	deflate.c	undeclared variable	new	-	1	single	21	763
libpng	iccfrompng.c	undeclared function	new	mandatory	2	multiple	1	185
libpng	iccfrompng.c	undeclared function	new	mandatory	2	multiple	1	185
libpng	iccfrompng.c	undeclared function	new	mandatory	3	multiple	1	185
libpng	iccfrompng.c	undeclared function	new	mandatory	1	multiple	1	185
libpng	pngpixel.c	undeclared function	new	mandatory	2	multiple	1	371
libpng	pngpixel.c	undeclared function	new	mandatory	2	multiple	1	371
libpng	pngrutil.c	undeclared variable	new	optional	2	single	92	4476
libpng	pngvalid.c	unused function	fix	optional	1	single	121	10140
libpng	pngget.c	unused function	new	optional	2	single	54	1177
libssh	dh.c	unused variable	fix	optional	2	single	33	629
libssh	keyfiles.c	undeclared function	fix	optional	2	multiple	12	1028
lua	loadlib_rel.c	undeclared function	new	mandatory	3	single	4	704
lua	loadlib_rel.c	undeclared function	fix	optional	7	single	4	704
m4	input.c	unused function	fix	optional	2	multiple	11	886
privoxy	filter.c	undeclared function	-	-	2	single	24	2451

(-) We do not find the necessary information to answer the research question, e.g., in case we detect an issue in the first commit available for analysis, so, we miss information regarding how developers introduce the issue. Developers introduce issues by adding **new** code, and by modifying existing code (**fix**), see column *"Fix/New"*. They introduce issues by adding / modifying **mandatory** or **optional** code, as we can see in column *"Optional/Mandatory"*. Column *"Macros"* depicts the number of macros involved in each issue, which may appear in **single** files or span **multiple** files (column *"Single/Multiple"*). We present the number of directives and lines of code for each file with at least one issue in *"Directives"* and *"LOC"*.

ber of macros involved in issues support the effectiveness of sampling analysis. For instance, the *pair-wise* sampling algorithm [31] checks all combinations of two configuration options, and would detect all issues involving one or two configuration options. Overall, *pair-wise* can detect more than 87% of the issues we find in our study, but there are also configuration-related issues involving more than two configuration options that require more complex sampling algorithms, such as *statement-coverage* [36], and *three-wise* [15].

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

7. Related Work

Several researchers studied the way in which developers use the C preprocessor, performing empirical studies with open-source systems written in C that are statically configurable with the C preprocessor [5, 8, 24]. Hunsen et al. [14] performed a study to understand how the C preprocessor is used in open-source and industrial systems. In a previous study [28], we interviewed 40 developers and performed a survey with 202 developers to understand why the C

preprocessor is still widely used in practice despite the strong criticism the preprocessor receives in academia. All of these studies discussed the C preprocessor and its problems, such as configuration-related faults, inconsistencies, and code quality.

Other studies analyzed software repositories by looking at faults already fixed by developers to understand the characteristics of configuration-related faults [1, 27]. In particular, researchers analyzed configuration-related faults in dynamic configurable systems [11, 12, 21]. Iago et al. [1] analyzed the *Linux Kernel* software repository to study configuration-related faults. Tartler et al. [37] also performed studies to find configuration-related faults in the *Linux kernel* using sampling. In addition, there are several studies proposing tools to find faults and dead code, such as *Undertaker* [36], and *Splint* [22].

Previous studies considered combinatorial interaction testing to check different combinations of configuration options and prioritize test cases [21]. Nie et al. [29] performed a survey with combinatorial testing approaches. Several researchers used the *t-wise* sampling algorithm to cover all t configuration option combinations [15, 31]. Other researchers proposed the *statement-coverage* [36] sampling algorithm, and Iago et al. [1] suggested the *one-disabled* algorithm.

Kästner et al. [17] developed a variability-aware parser, which analyzes all possible configurations of a C program at the same time. In addition, it performs type checking [18] and data-flow analysis [25]. Gazzillo and Grimm [13] developed a similar parser. Difficulties in setting up these tools and narrow classes of detectable faults limit their applicability. In addition, variability-aware analysis tools work at the preprocessor level, which hinders the reuse of existing tools, such as *gcc* and *clang*. Our strategy minimizes these problems by reducing the initial setup problems of variability-aware analysis tools.

Another strategy adopted by a number of studies was to develop variability-aware type systems [2, 7, 18, 34, 38], by proposing new languages or language extensions, together with a soundness proof for the underlying type system. Thaker et al. presented techniques for verifying type safety properties of feature-oriented product lines using SAT solvers [38]. Delaware et al. formalized this work, proposing the Lightweight Feature Java (LFJ), an extension of Lightweight Java with features [7], which inferred type checking constraints. Apel et al. proposed the Feature Featherweight Java (FFJ) [2], performing analysis using SAT solvers to check if all possible configurations of a product line are well-typed. Schaefer et al. proposed a compositional type system for delta-oriented product lines implemented using Java [34]. Using annotations, Kästner et al. proposed the Color Featherweight Java (CFJ) calculus, and implemented checks for full Java in the Colored Integrated Development Environment (CIDE) tool. Different from these studies, we do not propose new languages, language extensions or formalize a type system for C with preprocessor directives. Here, instead of focusing on a particular feature-oriented language, which could limit the applicability of our strategy to real-world systems, we try to leverage the way that developers often implement configurable systems. This way, we perform an empirical study that goes beyond previous studies on variability-aware type systems. However, we cannot claim that our strategy can find all of the configuration-related issues.

Apel et al. also proposed a language-independent reference checking algorithm for product lines [3], evaluated using small product lines written in Java and C. They extend feature structure trees with references, to have language-independent model of the program. The strategy presented in this paper can also be applied to configurable systems written in different languages, although we only evaluated systems using the C preprocessor.

Some studies compared sampling and variability-aware strategies. Apel et al. [4] developed a model checking tool for product lines and used it to compare sampling and variability-aware strategies with regard to verification performance and the ability to find defects. Liebig et al. [25] performed studies to detect the strengths and weaknesses of variability-aware and sampling analyses. Kolesnikov et al. [20] compared variability-aware, feature-based, and product-based type checking. In our study, we performed complimentary studies regarding understanding configuration-related issues, and our findings support the effectiveness of sampling analysis.

8. Concluding Remarks

In this paper, we presented an empirical study to investigate and better understand preprocessor-related issues in C. We defined a strategy to identify issues that minimizes the setting up problems of variability-aware tools and allows us to analyze several systems. We analyzed 15 subject systems to answer our research questions. In particular, we answered questions related to how developers introduce configuration-related issues, number of configuration options involved in each issue, and time issues remain in source files. In summary, we found 39 distinct configuration-related issues, including 14 undeclared functions, 2 undeclared variables, 7 unused functions, and 23 unused variables that appear only in some configurations of the source code.

The results revealed that developers took several years to fix some configuration-related issues, and others are still not fixed. The majority of configuration-related issues (87%) are detected when we enable or disable one or two configuration options, which support the effectiveness of sampling algorithms, such as *pair-wise*. In addition, we found differences regarding the way developers introduce configuration-related undeclared/unused variables and functions when compared to configuration-related syntax errors. They introduce the majority of undeclared/unused functions and variables when adding new source files and functions. In contrast, developers introduce most syntax errors when modifying existing code [27]. Our empirical study presented findings that may be helpful to understand configuration-related issues, support tool developers, minimize these issues in practice, and improve software quality.

Acknowledgement

This work has been partially supported by CNPq (Universal 460883/2014-3) and the project DEVASSES, funded by the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no PIRSES-GA-2013-612569.

References

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the Linux Kernel: A qualitative analysis. In *Proceedings of the International Conference on Automated Software Engineering*, ASE. IEEE/ACM, 2014.

[2] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Journal of Automated Software Engineering*, 17:251–300, September 2010.

[3] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Language-independent reference checking in software product lines. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 65–71, 2010. ISBN 978-1-4503-0208-1.

[4] S. Apel, A. v. Rhein, P. Wendler, A. Grösslinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the International Conference on Software Engineering*, ICSE. IEEE, 2013.

[5] I. Baxter. Design maintenance systems. *Communication of the ACM*, 35(4):73–89, 1992.

[6] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pairwise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, 2006.

[7] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: a machine-checked model of safe composition. In *Proceedings of the Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2009.

[8] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28: 1146–1170, 2002.

[9] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and R. Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 2008.

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

[11] B. J. Garvin and M. B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceeding of the International Symposium on Software Reliability Engineering*, ISSRE, 2011.

[12] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In *Proceedings of the Workshop on Assurances for Self-adaptive Systems*, ASAS. ACM, 2011.

[13] P. Gazzillo and R. Grimm. SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the programming language design and implementation*, PLDI. ACM, 2012.

[14] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Lebenich, M. Becker, and S. Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Journal of Empirical Software Engineering*, 2015.

[15] M. F. Johansen, O. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the International Software Product Line Conference*, SPLC, 2012.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, November 1990.

[17] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and 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*, OOPSLA. ACM, 2011.

[18] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14:1–14:39, July 2012.

[19] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2012.

[20] S. Kolesnikov, A. von Rhein, C. Hunsen, and S. Apel. A comparison of product-based, feature-based, and family-based type checking. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*, GPCE. ACM, 2013.

[21] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, June 2004.

[22] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Conference on USENIX Security Symposium*, SSYM. USENIX Association, 2001.

[23] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the International Conference on Software Engineering*, ICSE. ACM, 2010.

[24] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the 10th Aspect-Oriented Software Development*, AOSD. ACM, 2011.

[25] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 81–91. ACM, 2013.

[26] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware refactoring in the wild. In *Proceedings of the International Conference on Software Engineering*, ICSE. IEEE, 2015.

[27] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*, GPCE. ACM, 2013.

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

[29] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011.

[30] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 196–210. Springer Berlin Heidelberg, 2010.

[31] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST, 2010.

[32] M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand, and S. Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proceedings of the Generative Programming and Component Engineering*, GPCE. ACM, 2011.

[33] M. Ribeiro, P. Borba, and C. Kästner. Feature maintenance with emergent interfaces. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2014.

[34] I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, 2011.

[35] H. Spencer and G. Collyer. Ifdef considered harmful, or portability experience with C news. In *USENIX Annual Technical Conference*, pages 185–197, 1992.

[36] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the Workshop on Programming Languages and Operating Systems*, PLOS, 2011.

[37] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *2014 USENIX Annual Technical Conference*, USENIX, 2014.

[38] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 95–104, 2007.

[39] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014.

[40] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. Presence-condition simplification in highly configurable systems. In *Proceedings of the International Conference on Software Engineering*, ICSE. IEEE, 2015.

[41] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA. ACM, 2004.