Investigating Preprocessor-Based Syntax Errors

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

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

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

Abstract

The C preprocessor is commonly used to implement variability in program families. Despite the widespread usage, some studies indicate that the C preprocessor makes variability implementation difficult and error-prone. However, we still lack studies to investigate preprocessor-based syntax errors and quantify to what extent they occur in practice. In this paper, we define a technique based on a variability-aware parser to find syntax errors in releases and commits of program families. To investigate these errors, we perform an empirical study where we use our technique in 41 program family releases, and more than 51 thousand commits of 8 program families. We find 7 and 20 syntax errors in releases and commits of program families, respectively. They are related not only to incomplete annotations, but also to complete ones. We submit 8 patches to fix errors that developers have not fixed yet, and they accept 75% of them. Our results reveal that the time developers need to fix the errors varies from days to years in family repositories. We detect errors even in releases of well-known and widely used program families, such as *Bash*, *CVS* and *Vim*. We also classify the syntax errors into 6 different categories. This classification may guide developers to avoid them during development.

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

Keywords Program Families, Preprocessors, Syntax Errors

1. Introduction

A program family is a set of programs whose commonality is so extensive that it is advantageous to study their common properties before analyzing individual members [1]. In this context, developers often use the C preprocessor to handle variability and implement these individual members [2]. The C preprocessor is a simple, effective, and language independent tool. However, despite their widespread use in practice, preprocessors suffer of several drawbacks, including no separation of concerns, which obfuscate the code and hampers understanding [3–5].

In particular, preprocessors also ease the introduction of subtle syntax errors [4, 6–8], like when we annotate an opening bracket without its correspondent closing one. Although this claim is pretty reasonable due to the problems that preprocessors may cause, we still lack studies to investigate preprocessor-based syntax errors and quantify to what extent they occur in practice. Notice that categorizing the syntax errors and investigating the way developers introduce them is important to aid developers on minimizing these errors during their development tasks, improving quality and reducing effort.

To formulate the theory that preprocessors cause syntax errors, we define a technique to identify preprocessor-based syntax errors in releases and commits of C program families. Our technique considers a syntax error as an incorrect output of the preprocessing task [9], i.e., it generates an invalid program according to the C grammar. To consider variability during our analysis, we rely on TypeChef, a variability-aware parser that checks all possible configurations of the source code [8].

To evaluate to what extent preprocessor-based syntax errors is a problem in practice, we use our technique to conduct a comprehensive empirical study. In particular, we answer research questions related to the occurrence of syntax errors in releases and commits, whether the errors arise in valid configurations, how developers introduce the syntax errors, the time developers need to fix the errors, the percentage of commits with errors, and if we can classify the syntax errors in type categories.

To answer our research questions, we analyze releases of 41 C program families and more than 51 thousand commits of 8 families. We select these families inspired by previous work [7, 10, 11]. Besides, the majority of families are well-known and used in industrial practice. In this context, however, notice that analyzing many families with thousands of commits seems unfeasible, since it is a time consuming task. Our technique minimizes this problem sufficiently to enable us to analyze several program families while still providing reasonable results.

Our study reveals that preprocessor-based syntax errors are not common in family releases. We roughly conclude the same when considering commits. In particular, we find 33 preprocessor-based syntax errors, out of which only 24 happen in valid configurations. To conclude that 9 errors arise in invalid configurations, we rely on answers—from e-mail and bug reports—of the actual program families developers. Further, we detect that developers introduce syntax errors mainly by changing existing code and adding preprocessor directives, for example, to support a different operating system. Regarding the time that developers need to fix the errors, we detect that it varies from days to years. Moreover, we identify some errors that developers took more than five years to fix, and some errors still not fixed. So, we submit patches with suggestions to fix the errors, and developers accept 6 and reject 2 patches.

Also, we observe that the percentage of commits with syntax errors vary significantly as well. We find files that contain errors only in 0.43% of commits. In contrast, we also find files that contain errors in all commits. Last but not least, we categorize the 24 syntax errors we find into six types of errors. The results reveal that the majority of syntax errors occur because of ill-formed constructions,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2373-4/13/10…$15.00.
http://dx.doi.org/10.1145/2517208.2517221

e.g., an `else` without its correspondent `if` statement, and missing brackets. In summary, the main contributions of this paper are:

- We perform an empirical study using 41 C program families and more than 51 thousands commits to quantify and better understand preprocessor-based syntax errors;
- We classify preprocessor-based syntax errors and study the way developers introduce them;
- We present a technique that makes feasible the task of analyzing the syntax of several program families.

We organize the remainder of this paper as follows. In Section 2, we show a real example of preprocessor-based syntax error that motivates our study. Then, in Section 3, we describe our technique to find preprocessor-based syntax errors. Afterwards, we present the empirical study settings in Section 4, and discuss the results in Section 5. Last, we present the related work in Section 6 and the concluding remarks in Section 7.

2. Motivating Example

Developers often use preprocessors to handle variability in C program families. For instance, *libpng*[1] is a program family implementing the official PNG reference library. Figure 1 presents part of the *libpng* program family related to progressive display style, which is useful to read images from the network. Figure 1 contains a preprocessor macro that implements a progressive display style, i.e., `PNG_READ_INTERLACING_SUPPORTED`. The macro uses the interlacing method, which is responsible for encoding a bitmap image. During the download process, we can already see a copy of the whole image despite the incompleteness. It is useful for transmitting images over slow communication links.

```
1. // Other includes..
2. #include <fenv.h>
3. // Other function definitions..
4. static void progressive_row(png_structp ppIn, png_bytep new_row){
5.    // Code Here..
6.    if (new_row != NULL) {
7.       // Code Here..
8.       if (y >= dp->h)
9.          png_error(pp, "invalid y to progressive row callback");
10.      row = store_image_row(dp->ps, pp, 0, y);
11.#ifdef PNG_READ_INTERLACING_SUPPORTED
12.      if (dp->do_interlace){
13.         // Code Here..
14.      } else
15.         png_progressive_combine_row(pp, row, new_row);
16.   } else if (dp->interlace_type == PNG_INTERLACE_ADAM7)
17.      png_error(pp, "missing row in progressive de-interlacing");
18.#endif
19.}
20.// More function definitions..
```

Figure 1. Code snippet of *libpng* with a syntax error when we do not define macro `PNG_READ_INTERLACING_SUPPORTED`.

Developers of C program families like *libpng* use existing compilers, such as *GCC* and *clang*. However, these compilers do not have a good support to check whether all configurations contain syntax errors. For example, preprocessing Figure 1 without `PNG_READ_INTERLACING_SUPPORTED` generates an invalid program according to the C grammar. It contains a preprocessor-based syntax error since it opens the `if` statement block at line 6, but it does not close at line 16. The error presented in Figure 1 we find in release 1.5.14 of *libpng*, which contains 360 preprocessor macros. If there is no forbidden configuration, we might have 2^{360} possible configurations, where 50% of them contain the preprocessor-based syntax error we discuss here. We report this error by submitting a patch to *libpng* developers, and they accepted and fixed the error. To identify this error, developers have to check each configuration individually to detect this error using the existing compilers. However, it is unfeasible in several cases due to the high number of possible configurations. Developers need a better tool support to detect such kinds of errors. In this context, there are some variability-aware parsers to detect preprocessor-based syntax errors in C program families [8, 12] to help developers.

Previous studies [4, 6–8, 10] refer to syntax errors similar to the one we describe in Figure 1. However, they do not provide a comprehensive study to better understand to what extent these errors happen in practice, if they happen in valid configurations, or even if we can classify syntax errors into type categories. In this paper, we present a technique to find preprocessor-based syntax errors in program families (Section 3) and an empirical study to answer research questions on this topic (Sections 4 and 5).

3. A Technique to Find Preprocessor-Based Syntax Errors

In this section, we present a technique to identify syntax errors in program families. To parse the program families and check all configurations, we use the TypeChef variability-aware parser [8]. Without a variability-aware parser, we need to check each configuration separately, which is unviable for program families with many configurations. To better explain our technique, we refer to Figure 2, and detail its four steps in what follows.

The goal of the first step is to enable us to analyze several program families. In this step, our technique excludes all external libraries from the program family by eliminating `#include` directives. Notice that we still consider the header files of the program families, but exclude the external ones. For example, the C file depicted by Figure 1 includes the *fenv.h* library, which is not available in standard C compilers and it is not part of the program family code. In addition, the program families use specific external dependencies for different operating systems, e.g., we cannot use a package-building mechanism from a *Linux* system to install the external *windows.h* library. Because finding and downloading the correct library version is a manual and time consuming task, considering these external libraries would hind our analysis. In this way, we only focus on the program family code.

By excluding `#include` directives, Step 1 may leave some types and macros undefined in the program family. We generate stubs using C/C++ Development Tooling (CDT) with the default configuration to replace the original types and macros. Then, we create a `stubs.h` file to contain these stubs (Step 2) and now TypeChef is able to parse the source code. We use the CDT parser to generate an Abstract Syntax Tree (AST) for each source code file. Then, we navigate through the AST, get the types and macros that CDT identifies, and add them to the `stubs.h` file. We include this file into the program family source code.

Step 3 generates a shell script that calls TypeChef for each source code file. We built an Eclipse *plug-in* that automates Steps 1-3. Finally, we run the script our technique generates in Step 4. When TypeChef reports an error, we perform a manual check to verify whether the error is related to preprocessors. After fixing the error, we may continue to analyze the program family, i.e., depending on the error, we add a missing bracket, or remove an additional comma, and so on. This way, TypeChef continues to analyze the file. In case of a preprocessor-based syntax error, we create an error report with information like the *problematic configuration*[2] and code snippet with the syntax error.

We use our technique to analyze *Git* and *Mercurial* software repositories. For each set of files of a given commit in the reposi-

[1] `http://www.libpng.org`

[2] By problematic configuration we mean a valid configuration, according to the feature model constraints, that contains a syntax error.

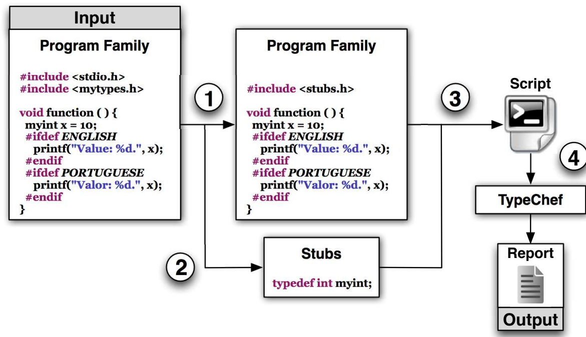

Figure 2. A technique to identify preprocessor-based syntax errors in program families.

tory, we apply the technique to find preprocessor-based syntax errors. In the first commit of a given program family, we analyze all files. In the following commits, we only consider the updated and added files. In this way, we avoid the overhead of analyzing files that have not changed. We use the *Git* and *Mercurial* diff tools to identify the changed files. Figure 3 depicts this process.

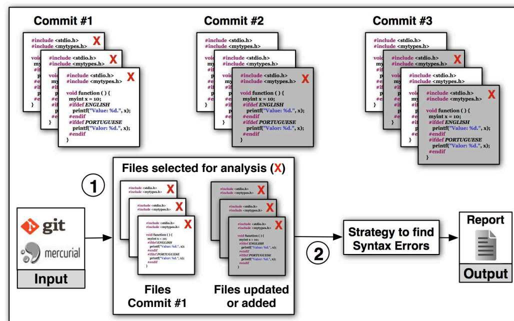

Figure 3. Analyzing software repositories using our technique.

It is important to mention that our technique may generate false positives and negatives. For example, CDT may not identify all types and macros. Additionally, external libraries defining macros may influence the program family code. Finally, our commits retrieval may miss to detect updates to a file that affect other files. Section 5.4 discusses these topics in details.

4. Study settings

In this section, we present the settings of our empirical study to investigate syntax errors. Our study considers 41 C program families and more than 51 thousand commits. To better structure our study, we use the Goal, Question, Metrics approach [13].

4.1 Definition

The goal of this empirical study is to analyze program families for the purpose of evaluation with respect to verifying the presence of preprocessor-based syntax errors in the context of the C language. In particular, this study addresses the following research questions:

- **Question 1.** Do program families releases contain preprocessor-based syntax errors?
- **Question 2.** Do commits to the program families repositories contain preprocessor-based syntax errors?
- **Question 3.** Do preprocessor-based syntax errors arise in valid configurations?
- **Question 4.** How do program families developers introduce the preprocessor-based syntax errors?
- **Question 5.** For how long a preprocessor-based syntax error remains in commits of a particular source file?
- **Question 6.** What is the percentage of commits with preprocessor-based syntax errors for each file?
- **Question 7.** What are the types of preprocessor-based syntax errors we find in practice?

To answer Questions 1 and 2, we count the number of syntax errors in releases and the number of syntax errors in commits for each family we analyze. To answer Question 3, we analyze each syntax error to verify whether it arises in a valid configuration. In this question, we consider feedbacks from the actual developers.

In Question 4, we investigate each syntax error to identify how developers introduce it. For instance, we investigate whether developers introduce the syntax error in a new source file, or in an existing one by altering a function code, and so on. Here we also detect whether developers add or remove preprocessor directives.

Regarding Question 5, we analyze two metrics: Date of Commit that Fixes the syntax Error ($DCFE$) and Date of Commit that Introduces the syntax Error ($DCIE$). Now we can measure the time in-between, the Time to Fix the syntax Error:

$$TFE = DCFE - DCIE$$

To better explain it, we refer to Figure 4 that depicts these metrics. In this context, *developer 2* introduces a syntax error in file `example.c` on June 02, 2013 (commit *#2*). Then, *developer 1* fixes this error on June 10, 2013 (commit *#4*). Thus, $TFE = 8$ days.

To answer Question 6, we measure the Percentage of Commits with syntax Errors (PCE) for each source file. We compute this metric in the following way:

$$PCE\,(file) = \frac{Number\,of\,Commits\,with\,Errors\,(file)}{Total\,Number\,of\,Commits\,(file)}$$

This way, we count the number of updates with errors in a particular file and the total number of commits that changes the specific file. For instance, as we show in Figure 4, we have one update in `example.c` with the syntax error (commit *#2*), and developers update this file four times (commits *#1*, *#2*, *#4* and *#5*). Thus, PCE (`example.c`) $= 1/4 = 25\%$.

Regarding Question 7, we classify all errors. For example, Section 2 illustrates a syntax error in which developers incorrectly annotate an `else if` statement. A similar error appears in other program families. We classify them into a type of error (category).

4.2 Planning

Next, we describe the subjects and the instrumentation of our study.

4.2.1 Subjects Selection

We analyze 41 program families written in C ranging from 2,681 to 1,536,979 lines of code. These families are from different domains, such as operating systems, web servers, text editors, games, and databases. We select these program families inspired by previous work [4, 6, 7]. We also randomly select program families that run on different operating systems and use the C preprocessor from Source Forge.[3] We present the details of each family in Table 1.

4.2.2 Instrumentation

We use the technique presented in Section 3 to investigate syntax errors. We use TypeChef version 0.3.3 to parse all possible config-

[3] `http://sourceforge.net/`

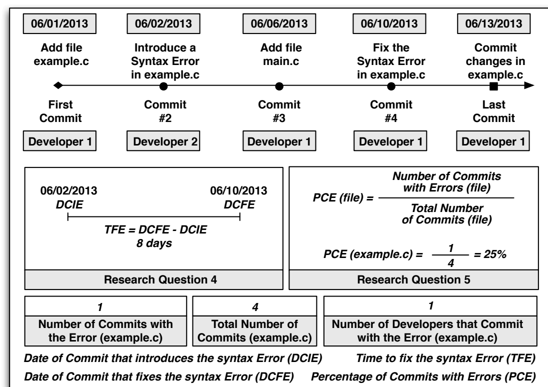

Figure 4. Scenario illustrating commits timeline and how we compute our metrics.

urations and CDT version 8.1.2 to create the stubs. Further, to automatize our technique, we use Eclipse Classic 4.2.2 to implement and run a *plug-in* to analyze the program families. We use Terminal version 2.3 on Mac OS X to run the scripts. We also count the number of lines of code and the number of files of each program family using the Count Lines of Code tool version 1.56, which eliminates blank lines and comments. Finally, we use *Git* version 1.7.12.4 and *Mercurial* version 2.5.4 tools to identify changes in files and get information about program families repositories.

4.3 Operation

We execute the empirical study on a MacBook Pro 2.4GHz dual-core Intel Core i5 8GB, running Mac OS X 10.8 Mountain Lion. As a first part of our analysis, we execute our technique to find syntax errors in releases of all 41 C program families we consider in this study. The analysis of all releases considers 9,064 files and almost 4 Million Lines of Code (MLOC). Then, we investigate syntax errors in commits. However, performing this analysis in all families is a very time consuming task, since the fourth step of our technique is semi-automatic. This way, we decide to analyze the commits only on the families we identify syntax errors in their releases. If some program family does not have *Git* or *Mercurial* repositories, we select another one. To perform this selection, we consider well-known families that several people use, and receive a considerable support from the open source community. During the analysis of the repositories, we consider only the trunk, i.e., we do not analyze the individual branches.

Next, we interpret and discuss the results of this empirical study to investigate preprocessor-based syntax errors.

5. Results and Discussion

In this section, we answer the research questions (Section 5.1), examine the directives that cause the syntax errors (Section 5.2), discuss the patches we submit (Section 5.3), and present the threats to validity (Section 5.4). The artifacts necessary to execute this empirical study are available at the project's web site.[4]

5.1 Research Questions

Next we answer and discuss the research questions.

[4] `http://www.dsc.ufcg.edu.br/~spg/gpce2013/`

5.1.1 Do program families releases contain preprocessor-based syntax errors?

Usually developers make a release available after code reviews and testing activities to minimize errors and improve quality. Nevertheless, our results reveal that preprocessor-based syntax errors still occur in releases. We find 14 syntax errors in 7 program families: *Bash* (2), *CVS* (1), *libpng* (1), *libssh* (4), *Vim* (3), *Xfig* (1), and *XTerm* (2). See more details in Table 1.

Next, we discuss some reasons that may lead to the syntax errors. Firstly, existing C compilers like *GCC* and *clang* are not variability aware. Developers identify syntax errors only when compile the program family using the problematic configuration. So, these errors may be difficult to detect using existing compilers.

Moreover, programs containing preprocessors are difficult to read and understand [3, 4, 6–8]. For example, the error we describe in Section 2 has been fixed immediately after our patch submission. In this case, it seems that the *libpng* developers did not fix the error earlier because they had not identified it during their maintenance tasks. On the other hand, although developers can identify the error earlier, they may decide to fix it later, setting this fixing task as low priority. For example, the syntax error may happen in a not deliverable configuration (consequently not exercised by the compilers), meaning that it is not important at least for now. However, notice that the error can still hamper reading and understanding activities.

5.1.2 Do commits to the program families repositories contain preprocessor-based syntax errors?

To perform the study in repositories, we select four out of seven program families in which we find errors in releases (see Section 5.1.1): *Bash*, *libpng*, *libssh*, *Vim*. We do not select all seven because we do not find the *git* or *mercurial* repositories of three of them. To increase the number of repositories to analyze, we also consider other four program families in which we do not find syntax errors in their releases: *Apache*, *libxml2*, *Dia*, and *Gnuplot*.

We analyze 51,035 commits and identify 27 preprocessor-based syntax errors, out of which 8 syntax errors also belong to releases. We find syntax errors in all repositories we analyze. Table 2 presents the commits results, indicating the number of developers that submitted commits, total number of commits for each program family, date of the first commit, date of the last commit, and total number of syntax errors.

Not surprisingly, here we find more errors than in releases, since the source code in commits is still under development. Nevertheless, our results reveal that preprocessor-based syntax errors are not common in the repositories we analyze. We identify in total 33 distinct preprocessor-based syntax errors in releases and commits as we can see in Figure 5.

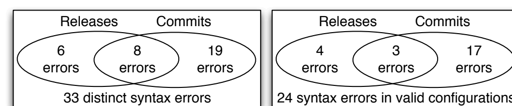

Figure 5. Syntax errors in releases and commits.

5.1.3 Do preprocessor-based syntax errors arise in valid configurations?

In this section, we analyze whether the errors we find in commits and releases happen in valid configurations. To answer this question, we rely on answers of the actual developers of each family we analyze. We get feedback via email and bug track systems.

We find 33 distinct preprocessor-based syntax errors in releases and commits, out of which 24 errors arise in valid configurations (72%) as we can see in Figure 5. In releases, we find 14 syntax

Table 1. Subject Characterization and Number of Syntax Errors in Releases

Family	Version	Application Domain	LOC	Number of Files	Syntax Errors	Errors in Valid Configurations
apache	2.4.3	web server	144,768	362		
atlantis	0.0.2.1	operating system	2,681	103		
bash	4.2	command language interpreter	44,824	138	2	2
bc	1.03	calculator	5,177	27		
berkeley	4.7.25	database system	185,111	580		
bison	2	parser generator	24,325	129		
cherokee	1.2.101	web server	63,109	346		
clamav	0.97.6	antivirus	107,548	377		
cvs	1.11.21	version control system	76,125	236	1	1
dia	0.97.2	diagramming software	28,074	132		
expat	2.1.0	XML library	17,103	54		
flex	2.5.37	lexical analyzer	16,501	41		
fvwm	2.4.15	windows manager	102,301	270		
gawk	3.1.4	GAWK interpreter	43,070	140		
ghostscript	9.05	postscript interpreter	1,536,979	3,230		
gnuchess	5.06	chess player	9,293	37		
gnuplot	4.6.1	plotting tool	79,557	152		
gzip	1.2.4	file compressor	5,809	36		
irssi	0.8.15	IRC client	51,356	308		
kin db	0.5	database system	64,120	119		
libieee	0.2.11	IEEE standards for VHDL library	5,323	27		
libdsmcc	0.6	DVB library	5,453	30		
libpng	1.6.0	PNG library	44,828	61	1	1
libsoup	2.41.1	SOUP library	40,061	178		
libssh	0.5.3	SSH library	28,015	125	4	
libxml2	2.9.0	XML library	234,934	162		
lighttpd	1.4.30	web server	38,847	132		
lua	5.2.1	programming language	14,503	59		
lynx	2.8.7	web browser	80,334	117		
m4	1.4.4	macro expander	10,469	26		
mpsolve	2.2	mathematical software	10,278	41		
mptris	1.9	game	4,988	29		
prc-tools	2.3	C/C++ library for palm OS	14,371	142		
privoxy	3.0.19	proxy server	29,021	67		
sendmail	8.14.6	mail transfer agent	91,288	243		
sqlite	3.7.15.2	database system	94,113	134		
sylpheed	3.3.0	e-mail client	83,528	218		
rcs	5.7	revision control system	11,916	28		
vim	7.3	text editor	288,654	178	3	2
xfig	3.2.4	vector graphics editor	70,493	192	1	1
xterm	2.4.3	terminal emulator	50,830	58	2	2
Total			3,860,078	9,064	14	9

Table 2. General Information about the Program Families Repositories

Project Name	Total Number of Developers	Total Number of Commits	First Commit	Last Commit	Syntax Errors	Errors from valid Configurations
apache	108	24,719	Jul-3-1996	May-3-2013	3	3
bash	2	68	Aug-26-1996	Mar-7-2013	2	2
dia	217	5,397	Jan-3-1997	May-5-2013	2	2
gnuplot	16	7,611	Apr-15-1998	May-6-2013	5	5
libpng	5	2,179	Jul-20-1995	Apr-25-2013	2	2
libssh	26	2,569	Jul-5-2005	Apr-5-2013	7	2
libxml2	169	4,179	Jul-24-1998	May-9-2013	2	2
vim	2	4,313	Jun-13-2004	May-4-2013	4	2
Total		51,035			27	20

errors, out of which 7 errors (50%) happen in valid configurations. We find 27 syntax errors in commits, out of which 20 (74%) arise in valid configurations. Tables 1 and 2 summarize the results.

We identify 9 syntax errors in *invalid configurations*[5] of *libssh* (5), *Vim* (2), and *XTerm* (2). Figure 6 illustrates part of a function of *libssh*. If we do not define macros `HAVE_LIBGCRYPT` and `HAVE_LIBCRYPTO`, our technique identifies a function with no signature in this configuration. However, it is not valid according to the developers. The *libssh* building process checks that these macros are alternative and we must define exactly one of them. Similar problems happen in the *Vim* program family. Some macros (`FEAT_GUI_W32`, `FEAT_GUI_MOTIF`, and `FEAT_GUI_GTK`) are alternative as well and we must define exactly one of them.

```
1. #ifdef (HAVE_LIBGCRYPT)
2.   static void dsa_public_to_string(gcry_sexp_t key, BUFFER *buffer){
3. #elif defined (HAVE_LIBCRYPTO)
4.   static void dsa_public_to_string(DSA *key, BUFFER *buffer){
5. #endif
6.         // Code Here..
7. }
```

Figure 6. Syntax error in *libssh* family in an invalid configuration.

As another example, Figure 7 illustrates a code snippet of *XTerm*. Notice that configuration `__GLIBC__ and USE_ISPTS_FLAG` does not open the `if` statement bracket at line 5, but it closes the bracket at line 14. However, according to the *XTerm* developers, this configuration is invalid. They use `USE_ISPTS_FLAG` to handle macro `ISC` (long obsolete), which predated `__GLIBC__`.

Despite happening in invalid scenarios, we argue that this situation makes the task of understanding and maintaining the code cumbersome. It can confuse developers unaware of particular configuration constraints and lead them to wrongly suppose that there is a syntax error in valid configurations.

5.1.4 How do program families developers introduce the preprocessor-based syntax errors?

In this section, we investigate how developers introduce 20 preprocessor-based syntax errors in commits that happen in valid configurations. We identify four categories:

1. *Changing existing code and adding preprocessor directives:* developers modify existing syntactical units by changing tokens. In addition, they introduce new preprocessor macros, for example, to support different operating systems;
2. *Changing existing code and removing directives:* developers change existing syntactical units by changing tokens. Further, they remove preprocessor macros;
3. *Changing existing code without adding or removing preprocessor directives:* developers modify existing syntactical units by only changing tokens;
4. *Adding completely new code:* developers introduce new code, e.g., adding new syntactical units, functions, and files.

In our study, developers introduce more syntax errors when changing existing code and adding preprocessor directives. In this category, they introduce 7 errors (35%). Further, developers introduce 5 errors (25%) when modifying code without adding or removing preprocessor directives. Finally, they introduce 2 errors (10%) when adding completely new code, and 1 error (5%) when changing code and removing directives. Developers may introduce more errors when changing code because preprocessor directives make the tasks of reading and understanding the source code more difficult [3, 4].

We could not classify 5 errors (25%), since we find these errors in the very first commit available for analysis. So, we miss information. For instance, if developers migrated the program family from one repository to the current one, our analysis does not consider the information with respect to the former repository.

```
1. #ifdef  __GLIBC__
2.     // Code Here..
3. #else
4.     #if defined (USE_ISPTS_FLAG)
5.         if (result) {
6.     #endif
7.     result = ((*pty = open("/dev/ptmx", O_RDWR)) < 0);
8. #endif
9.     // Code Here..
10.#if defined (SVR4) || defined (__SCO__) || defined (USE_ISPTS_FLAG)
11.    if (!result)
12.        strcpy(ttydev, ptsname(*pty));
13.    #ifdef USE_ISPTS_FLAG
14.        }
15.    #endif
16.    // Code Here..
17.#endif
18.    // Code Here..
```

Figure 7. Syntax error in an invalid configuration of *XTerm*.

5.1.5 For how long a preprocessor-based syntax error remains in commits of a particular source file?

In this section, we investigate the time required to fix errors in valid configurations. In our study, the time developers need to fix the errors varies from days to years. For example, developers fixed the *Vim* error in file `ex_cmds2.c` in a few days after introducing it. In contrast, developers took more than 5 years to fix the error in `parser.c` of *Gnuplot*. Table 3 depicts each syntax error we find during the analysis of commits, indicating the file name that contains the error, the date of the first commit containing the error, the date of commit that fixes the error, and time to fix the error.

There are some reasons why developers may take a long time to fix these errors. As described before, this might happen because developers do not identify the errors. They may not use variability-aware parsers or may have difficulties when reading and understanding preprocessor-based code. On the other hand, even if they find an error, they might take some time to fix it, since the error may arise in not exercised or deliverable configurations, leading developers to set lower priority to these errors when compared to semantic ones, for example.

In our study, we do not find a correlation between the file size—see Column "*Lines of Code (LOC)*"—and the time developers need to fix the errors (we remove errors developers have not fixed). There are some syntax errors in files with thousands lines of code fixed in two days, such as the file `xpath.c` in *libxml2*. On the other hand, we also find the opposite: errors in smaller files like `parser.c` (468 LOC) in *Gnuplot* fixed after more than 5 years.

Finally, we do not find a correlation between the time to fix errors and the number of developers that commit a file with syntax error (we also remove errors developers have not fixed). Table 3 indicates the number of developers that commit a file with syntax error (see column "*Developers*"). For instance, 13 different developers committed 77 times the file `mod_include.c` containing a syntax error in *Apache*. They took almost a year to fix the error. As another example, we also find an error in `os_unix.c` (*Vim*) that still needs fixing, in spite of 131 commits in 9 years. Only two developers committed this file. Nevertheless, it is important to note that our study cannot conclude if the developers were aware of these syntax errors before fixing them.

[5] By invalid configuration we mean that it is not valid according to the feature model constraints.

Table 3. Results of the Analysis of Commits per Syntax Error (only in valid configurations)

Project Name	File Name	Lines of Code (LOC)	Date of Commit that Introduce the Error	Date of Commit that Fix the Error	Time to Fix the Error (TFE) in days	Total Number of Commits	Number of Commits with the Error	Number of Developers that Commit with the Error	Percentage of Commits with Errors (PCE)	Directive Type
apache	ssl_util_ssl.c	365	Jun-28-2001	Apr-2-2002	278	62	15	4	24.19%	Complete
apache	ab.c	751	Abr-24-2000	May-16-2000	22	222	4	1	1.80%	Complete
apache	mod_include.c	2105	Oct-16-2000	Set-10-2001	329	353	77	13	21.81%	Complete
bash	getcppsyms.c	401	Aug-26-1996	Dec-23-1996	119	1	1	1	100%	Complete
bash	execute_cmd.c	2578	Jul-27-2004	Not Fixed	-	23	12	2	52.17%	Incomplete
dia	app_procs.c	429	Sep-3-2001	Nov-1-2001	59	232	1	1	0.43%	Complete
dia	preferences.c	645	Sep-3-2001	Sep-23-2002	385	100	10	5	10%	Incomplete
gnuplot	plot.c	450	Apr-15-1998	Sep-22-1998	160	180	5	1	2.78%	Incomplete
gnuplot	util.c	505	Jun-2-1999	Jun-9-1999	7	123	2	1	1.63%	Incomplete
gnuplot	parse.c	468	Apr-15-1998	Jul-22-2003	1924	89	27	3	30.34%	Complete
gnuplot	graph3d.c	2211	Oct-21-2002	Jan-7-2003	78	293	3	2	1.02%	Incomplete
gnuplot	datafile.c	3662	Feb-23-2008	Apr-13-2009	414	268	34	1	12.69%	Incomplete
libpng	pngtest.c	1198	Mar-14-2001	May-14-2001	0	1204	8	1	0.66%	Incomplete
libpng	pngtrans.c	186	May-16-1997	Jan-30-1998	259	738	4	1	0.54%	Incomplete
libssh	server.c	910	Jul-5-2005	Jul-5-2005	0	173	1	1	0.58%	Complete
libssh	channels.c	715	Jun-12-2008	Jun-16-2008	4	229	1	1	0.44%	Complete
libxml2	xmlregexp.c	2790	Apr-20-2002	Sep-17-2002	150	98	3	1	3.06%	Complete
libxml2	xpath.c	7659	Apr-18-2004	Apr-20-2004	2	370	2	1	0.54%	Complete
vim	ex_cmds2.c	3088	Jan-19-2010	Jan-19-2010	0	99	2	1	2.02%	Incomplete
vim	os_unix.c	4520	Jun-13-2004	Not Fixed	-	131	131	2	100%	Incomplete

5.1.6 What is the percentage of commits with preprocessor-based syntax errors for each file?

In this section, we analyze the percentage of commits with preprocessor-based syntax errors for each file. We only consider errors in valid configurations. As can be seen, the percentage of commits with errors also varies significantly, from 0.43% to 100% (see column "*Percentage of Commits with Errors*" (PCE) in Table 3). For instance, *Dia* developers committed the `app_procs.c` file 232 times but only once with the syntax error ($PCE = 0.43\%$).

We also identify two syntax errors that developers have not fixed, e.g., in `os_unix.c` (*Vim*), and `execute_cmd.c` (*Bash*). Regarding the syntax error in *Vim*, we find it in all commits (100%) with the specific file, totaling 131 commits. It may arise only in configurations that users do not use in practice. It is probably the reason why developers did not fix it yet. Regarding the syntax error in *Bash*, we find it in 57 commits (52.17%). However, this syntax error may not be affecting the users of *Bash* as well, since it is in the repository since 2004. This way, *Bash* and *Vim* developers may not be worried about these syntax errors despite the fact that they arise in valid configurations.

We do not observe correlation between PCE and LOC, TFE, or the number of developers (again, we remove the errors not fixed yet). Nevertheless, our results suggest a tendency that with few commits, developers fix the errors.

5.1.7 What are the types of preprocessor-based syntax errors we find in practice?

In this section, we categorize the preprocessor-based syntax errors we find in valid configurations in our study. We classify them in 6 types: missing/additional array separator, ill-formed construction, missing bracket, missing logical operator, missing parenthesis, and missing semicolon. Our results reveal that the types of errors ill-formed construction and missing bracket are the most common ones. We present the number of errors of each type in Table 4.

Table 4. Syntax Errors for each type.

Type of error	Number of Occurrences	Percentage
Array Separator	2	8.33%
Ill-Formed Construction	7	29.16%
Missing Bracket	6	25%
Missing Logical Operator	2	8.33%
Missing Parentheses	4	16.66%
Missing Semicolon	3	12.5%

We find two errors in the missing/additional array separator type. For example, Figure 8 illustrates a syntax error of the *Gnuplot* family. This code snippet generates an invalid C program when we have `EAM_OBJECTS and !WITH_IMAGE`. In this configuration, we have two array separators, one at line 3 and another at line 9.

```
1. df_bin_default_columns default_style_cols[LAST_PLOT_STYLE + 1] = {
2.     // other elements here
3.     {HISTOGRAMS, 1, 0},
4. #ifdef WITH_IMAGE
5.     {IMAGE, 1, 2},
6.     {RGBIMAGE, 3, 2}
7. #endif
8. #ifdef EAM_OBJECTS
9.     , {CIRCLES, 2, 1}
10.#endif
11.};
```

Figure 8. Code snippet of *Gnuplot* with a syntax error when we define `EAM_OBJECTS and !WITH_IMAGE`.

We also find some syntax errors related to ill-formed construction type. The code snippet in Figure 9 generates an invalid C program when we have `START_RSH_WITH_POPEN_RW and !SHUTDOWN_SERVER` in *CVS*. In this configuration, we introduce an `else if` without its corresponding `if`.

Regarding the missing bracket type, we present an example of *libpng* in Section 2. In this example, we show a directive that causes a syntax error, i.e., a missing bracket, when we do not define `PNG_READ_INTERLACING_SUPPORTED`. We find 6 errors of this type. Figure 10 depicts an error we find in *Vim*, where we classify as missing logical operator type. In this example, if we define `WIN32`, an error arises, since there is a missing logical operator at line 4.

```
1. // Code here..
2. #ifdef (SHUTDOWN_SERVER)
3.     if (current_parsed_root->method != server_method)
4. #endif
5. #ifndef (NO_SOCKET_TO_FD)
6.     {
7.         if (S_ISSOCK (s.st_mode))
8.             shutdown (fileno (bc->fp), 0);
9.     }
10. #endif
11. #ifdef (START_RSH_WITH_POPEN_RW)
12.     else if (pclose (bc->fp) == EOF){
13.         error (1, errno, "closing connection to %s");
14.         closefp = 0;
15.     }
16. #endif
17. // Code continues here..
```

Figure 9. Code snippet of *CVS* with a syntax error when we define `START_RSH_WITH_POPEN_RW and !SHUTDOWN_SERVER`.

```
1. // More code here..
2. int fd_tmp = mch_open(filename, O_RDONLY
3. #ifdef WIN32
4.     O_BINARY | O_NOINHERIT
5. #endif
6.     , 0);
7. // Code continues here..
```

Figure 10. Code snippet of *Vim* with a syntax error at line 4 when we define `WIN32`.

Next, we present in Figure 11 a syntax error in *Apache* of the missing opening parentheses type. In this example, there is a syntax error when we define `SSL_EXPERIMENTAL_PROXY`. There is a missing opening parentheses at the `if` statement condition.

```
1. #ifdef SSL_EXPERIMENTAL_PROXY
2. // More code here..
3. if (apr_dir_open(&dir, pathname, sp)) != APR_SUCCESS) {
4.     apr_pool_destroy(sp);
5.     return FALSE;
6. }
7. // Code continues here..
8. #endif
```

Figure 11. Code snippet of *Apache* with a syntax error at line 3 when we define `SSL_EXPERIMENTAL_PROXY`.

Finally, Figure 12 presents another syntax error in *Apache* of the missing semicolon type. We find a syntax error at line 8 in configurations defining `NOT_ASCII`. In this case, developers do not include a semicolon at the end of line 8.

```
1. // More code here..
2. #ifdef NOT_ASCII
3.     status = ap_xlate_open(&to_ascii, "ISO8859-1", cntxt);
4.     if (status) {
5.         fprintf(stderr, "ap_xlate_open(to ASCII)->%d\n", status);
6.         exit(1);
7.     }
8.     status = ap_xlate_open(&from_ascii, "ISO8859-1", cntxt)
9.     if (status) {
10.        fprintf(stderr, "ap_xlate_open(from ASCII)->%d\n", status);
11.        exit(1);
12.    }
13. #endif
14. // Code continues here..
```

Figure 12. Code snippet of *Apache* with a syntax error at line 8 when we define `NOT_ASCII`.

Notice that making developers aware of these types might be useful to avoid these errors. For example, when defining optional array elements, they can pay more attention to either not add unnecessary or miss separators. So, they may avoid the code constructions related to these types.

5.2 Verifying the Type of the Preprocessor Directives that causes the Syntax Errors

The C preprocessor is expressive enough so that we can encompass any code snippet. Developers can annotate part or a complete syntactical unit using preprocessor directives. For example, the `if` statement in Figure 11 represents a complete annotation, since the preprocessor completely encompasses the statement. As another example, Figure 10 presents the `#ifdef` directive that separates the parameters of the `mch_open` function call. In this case, it is an incomplete annotation [6].

In our study, we find 24 distinct errors in valid configurations in commits and releases (we find 3 common errors in both analyses). We observe that 10 syntax errors (41.67%) are related to complete annotations, and 14 to incomplete ones (58.33%). Table 3 details these results considering commits and valid configurations. Regarding only the commits results, 10 errors (50%) happen in incomplete annotations. On the other hand, we have 7 errors in releases, out of which 6 (85.71%) happen in incomplete annotations.

In this context, related approaches [3, 4, 6, 7] suggest that the use of incomplete annotations may be more error prone. Although this hypothesis seems reasonable due to difficulties of reading and understanding incomplete annotations, we find that 41.67% of the syntax errors occur in complete annotations.

5.3 Submitting Patches to Fix the Syntax Errors

We submit 8 patches—for each syntax error not fixed—to 6 families: *Bash* (1), *CVS* (1), *libpng* (1), *libssh* (2), *Vim* (2), and *Xfig* (1). In each patch, we also suggest how to fix the error. To the best of our knowledge, *Xfig* and *Vim* do not use bug track systems, so we submit patches to these families via email. Regarding patches to the other 4 families, we submit them via bug track systems.

We consider that developers accept a patch when they mention that it is an error by email, or keep the patch open after updating information like its priority. On the other hand, we consider that developers reject the patch when they mention it is not an error by email, or update this information on the patch. Thus, developers accepted 6 out of 8 patches. We present information about the patches we submit in Table 5, illustrating the family name, the file name with the error, and the patches status and priority.

Two out of six patches accepted have been set as low priority because the errors happen in invalid configurations. Figure 6 illustrates one of them. To fix it, we suggest to add an `#else` directive followed by the `#error` directive, making the source code explicit regarding the definition of exactly one of the macros. Next, we quote one of the *libssh* developers in response to our suggestion:

> "*Yes, we could add an error in this case. But the configure step takes care of making sure either libcrypto or libgcrypt is available.*"

Notice that the *libssh* developers accepted our patch even occurring in invalid configurations. Therefore, it seems that it is worthwhile to change the source code so it becomes more readable and understandable regarding configuration constraints. The second patch of *libssh* is in the same file (`keys.c`), and it is very similar to the one we present in Figure 6.

We submit a patch to *libpng* and developers fixed the error immediately after our patch submission (see Section 2). Regarding two patches we submit to *Bash* and *CVS*, developers accepted and the patches are still with the open status and normal priority. Developers accepted another patch to *Vim* as well, but they set no priority explicitly since we submit patches to *Vim* via email.

Vim developers rejected one patch by just arguing that it arises in an invalid configuration. Developers rejected a patch we submit to the *Xfig* program family as well. In this case, developers mention they do not use (at least for now) the erroneous macro we identify. According to the following quotation, it seems that the macro will be used when they decide to distribute the *Xfig* manual in Japanese. So, we still count this as an error, since it may arise in the future.

"It is not used now as Japanese PDF manual is not distributed with xfig, and I think you can simply ignore it."

Table 5. Patches we submit to Program Families.

Family	File	Configuration	Status	Priority
bash	execute_cmd.c	valid	open	normal
cvs	buffer.c	valid	open	normal
libpng	pngvalid.c	valid	fixed	normal
libssh	keys.c	invalid	new	low
libssh	keys.c	invalid	new	low
vim	os_unix.c	valid	open	-
vim	if_mzsch.c	invalid	not a bug	-
xfig	w_cmdpanel.c	valid	not a bug	-

5.4 Threats to Validity

Construct Validity. It refers to whether the preprocessor-based syntax errors we find are indeed errors in valid configurations. We minimize this threat by getting feedback from the actual developers. They accepted 6 out of the 8 syntax errors we report.

Internal Validity. Our technique excludes `#include` directives to eliminate external libraries in order to scale. However, notice that we may face false negatives due to the exclusion of these `#include` directives, which makes our technique unsound. In some cases, the external libraries can introduce additional code through macro definitions that may cause preprocessor-based syntax errors into the family source code. In this context, our technique may miss some syntax errors. Moreover, our technique may yield false positives due to types and macros that the CDT parser does not identify, i.e., these types and macros may not be included in our `stubs.h` file. So, we add the type or macro manually, which is an error-prone task. Still, in our study, our technique found 33 syntax errors in 26.83% of the program families we analyze. Moreover, we detect that 24 out of 33 syntax errors arise in valid configurations.

Our technique analyzes only updated and added files in software repositories from the second to the last commit, as described in Section 3. However, this approach may lead to false negatives. For instance, developers may update a macro definition in a file *A*, which leads to errors in a different file *B*. In our approach, because only *A* has been modified, we only analyze *A*. However, later, if developers modify *B*, our technique may catch the syntax error. Further, we may miss some syntax errors during the analysis of the repositories since we analyze only the trunk, i.e., branches may contain syntax errors as well.

Finally, the last step of our technique is semi-automatic, which is an error prone activity. However, it is important to be semi-automatic. For example, due to several C standards such as ANSI C, C99, and C11, TypeChef might not parse some C constructions, arising false positives easily recognizable by humans.

External Validity. We analyze 41 releases of different domains, sizes, and different number of developers. Moreover, we analyze more than 51 thousands commits of 8 families from small to mid sizes. We select well-known and active C families used in industrial practice. The families communities exist for years and seem very active: there are commits in 2013. In this way, we alleviate this threat. However, the small number of errors we identify makes it hard to apply inference statistics. Thus, the results are initial measurements and we should not use them to any direct comparison.

6. Related Work

Analysis of C Preprocessor Usage. Some approaches studied the way developers use the C preprocessor in practice. Liebig et al. [7] analyzed 40 program families, and suggested that developers can introduce subtle syntax errors, for example, by annotating a closing bracket but not the opening one. They define this kind of annotation as *undisciplined*. According to their study, undisciplined annotations correspond to 15.6% of the total number of annotations. Undisciplined annotations are similar to incomplete annotations [6, 14, 15] we use in this paper. In our work, we found 24 syntax errors related to undisciplined and disciplined annotations.

Baxter and Mehlich proposed DMS, a source-code transformation tool for C and C++ [16]. In a more recent work, these authors used DMS and emphasized the problem of using unstructured annotations [3], similar to incomplete annotations as well. Further, the authors presented an example with a syntax error related to the missing bracket error type we discuss here. In our work, we perform an empirical study investigating the presence of syntax errors in program families different from their approach.

Ernst et al. [4] presented an empirical study on how the C preprocessor is used in practice. They analyzed 26 packages comprising 1.4 MLOC. They found that most C preprocessor usage follows simple patterns. It also discussed about the undisciplined use of the C preprocessor and its problems, such as that it makes the program more difficult to understand. However, it focused mainly on macro definitions using `#define` directives. In this sense, our work complements the analysis of using the C preprocessor and presents findings about problems related to syntax errors in practice.

Others approaches also complemented these studies providing more information about the preprocessor usage. In a previous work, Ribeiro et al. [11] analyzed how often methods with preprocessor directives contain feature dependencies. Liebig et al. [10] proposed and collected some metrics to analyze the feature code scattering and tangling when using preprocessor directives. They analyzed 40 families implemented in C. However, none of them investigated the presence of syntax errors in C program families.

Variability-Aware Parsers. There are some strategies to parse C code with preprocessor directives. Some approaches [6, 17, 18] applied the strategy of preprocessing or modifying the code before parsing it. However, this strategy is not interesting to analyze variability since we lose information about the preprocessor directives.

Kästner et al. [8] proposed a variability-aware parser, i.e., a parser that analyze all possible configurations of a C program at once. In addition, it performs type checking analysis [19, 20]. In our work, we use TypeChef to identify errors in C program families.

Gazzillo and Grimm [12] proposed a variability-aware parser called SuperC. This parser is faster than TypeChef, but it does not perform type checking analysis. Since SuperC does not recognize some C constructions of different standards, we did not use it in our work. It does not parse some families that we use in this study.

Extracting Variability Information. Others proposed techniques to extract variability information from C program families. Some researches considered the Linux kernel in their studies and analyzed its source code files, Kconfig files, and Makefiles [21–23]. Other researches analyzed the rapid evolution of the Linux configurations. The number of features had doubled in the period analyzed [24]. She et al. [25] analyzed different operating systems, such as FreeBSD and eCos. In our work, we decide to contact the developers of the program families to check configuration constraints. This way, we avoid the effort of gathering information about configuration constraints for each family.

Tartler et al. [26] revealed the presence of zombie configurations, i.e., macros that cannot be either enabled or disabled at all, in the Linux kernel. Besides, others researches found several inconsistencies in the Linux kernel by analyzing source files, Kconfig and makefiles [27, 28]. In our work, we focused only on syntax errors in source code files and their presence in valid configurations. To the best of our knowledge, there is no existing work that investigated the impact of syntax errors in many C program families.

7. Concluding Remarks

In this paper, we presented an empirical study to investigate preprocessor-based syntax errors in C program families. Firstly, we defined a technique to identify syntax errors. Then, we analyzed 41 C program families and more than 51 thousands commits to answer our research questions. In summary, we found 24 distinct syntax errors (7 syntax errors in releases and 20 errors in commits). Moreover, we detected 9 errors that arise only in invalid configurations.

The results showed that preprocessor-based syntax errors are not common in practice. Furthermore, the results revealed that developers introduce syntax errors mainly by modifying existing code and adding preprocessor directives, e.g., adding new directives to support other operating systems. We detected that the time developers need to fix syntax errors varies from days to years. In this context, we could not find correlation with the LOC of the file that contains the error, or even with the number of developers modifying that particular file. The percentage of commits with syntax errors varies as well. We found files in which 0.43% of the commits with them contain the syntax error, but also files that contain the syntax error in all commits. Regarding the types of errors, we identified that we can categorize the syntax errors we found in 6 different types. We presented them using real code snippets of the program families we consider in this study. Finally, our empirical study presented findings that may be helpful to actual program families developers minimize the problem of syntax errors in practice.

Acknowledgments

We gratefully thank Christian Kästner for helpful comments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq grants 573964/2008-4 and 480160/2011-2.

References

[1] D. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. 2, pp. 1–9, 1976.

[2] H. Spencer, "Ifdef considered harmful, or portability experience with C news," in *USENIX Annual Technical Conference*, pp. 185–197, 1992.

[3] I. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *Proceedings of the Working Conference on Reverse Engineering*, WCRE '01, pp. 281–290, IEEE Computer Society, 2001.

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

[5] C. Kästner and S. Apel, "Virtual separation of concerns – a second chance for preprocessors," *Journal of Object Technology (JOT)*, vol. 8, no. 6, 2009.

[6] A. Garrido and R. Johnson, "Analyzing multiple configurations of a C program," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pp. 379–388, IEEE Computer Society, 2005.

[7] 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 '11, pp. 191–202, ACM, 2011.

[8] 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 26th ACM SIGPLAN Object-Oriented Programming Systems Languages and Applications*, OOPSLA '11, ACM, 2011.

[9] IEEE, "Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[10] 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 32nd International Conference on Software Engineering*, ICSE '10, pp. 105–114, ACM, 2010.

[11] 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 10th Generative Programming and Component Engineering*, GPCE '11, pp. 23–32, ACM, 2011.

[12] P. Gazzillo and R. Grimm, "SuperC: parsing all of C by taming the preprocessor," in *Proceedings of the 33rd Programming Language Design and Implementation*, PLDI '12, pp. 323–334, ACM, 2012.

[13] V. Basili, G. Caldiera, and D. H. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, Wiley, 1994.

[14] A. Garrido and R. Johnson, "Challenges of refactoring C programs," in *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '02, pp. 6–14, 2002.

[15] A. Garrido and R. Johnson, "Refactoring C with conditional compilation," in *Proceedings of the 18th Automated Software Engineering*, ASE '03, pp. 323–326, IEEE Computer Society, 2003.

[16] I. Baxter, "Design maintenance systems," *Communication of the ACM*, vol. 35, no. 4, pp. 73–89, 1992.

[17] S. Somé and T. Lethbridge, "Parsing minimization when extracting information from code in the presence of conditional," in *Proceedings of the International Workshop on Program Comprehension*, IWPC '98, pp. 118–125, 1998.

[18] Y. Padioleau, "Parsing C/C++ code without pre-processing," in *Compiler Construction*, vol. 5501 of *Lecture Notes in Computer Science*, pp. 109–125, Springer Berlin Heidelberg, 2009.

[19] A. Kenner, C. Kästner, S. Haase, and T. Leich, "Typechef: toward type checking #ifdef variability in C," in *Proceedings of the 2nd Feature-Oriented Software Development*, FOSD '10, pp. 25–32, ACM, 2010.

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

[21] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat, "Efficient extraction and analysis of preprocessor-based variability," in *Proceedings of the 9th Generative Programming and Component Engineering*, GPCE '10, pp. 23–32, ACM, 2010.

[22] N. Andersen, K. Czarnecki, S. She, and A. Wasowski, "Efficient synthesis of feature models," in *Proceedings of the 16th Software Product-Line Conference*, SPLC '12, pp. 106–115, ACM, 2012.

[23] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A robust approach for variability extraction from the linux build system," in *Proceedings of the 16th Software Product-Line Conference*, SPLC '12, pp. 21–30, ACM, 2012.

[24] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the linux kernel variability model," in *Proceedings of the 14th Software Product-Line Conference*, SPLC '10, pp. 136–150, Springer-Verlag, 2010.

[25] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 461–470, ACM, 2011.

[26] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Dead or alive: finding zombie features in the linux kernel," in *Proceedings of the 1st Feature-Oriented Software Development*, FOSD '09, pp. 81–86, 2009.

[27] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem," in *Proceedings of the 6th Computer Systems*, pp. 47–60, ACM, 2011.

[28] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann, "Revealing and repairing configuration inconsistencies in large-scale system software," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 5, pp. 531–551, 2012.