An Approach to Safely Evolve Program Families in C

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

Abstract

The C preprocessor is widely used to handle variability and solve portability issues in program families. In this context, developers normally use tools like *GCC* and *Clang*. However, these tools are not variability-aware, i.e., they preprocess the code and consider each family member individually. As a result, even well-known and widely used families, such as *Linux* and *Apache*, contain bad smells and bugs related to variability. To minimize this problem, we propose an approach to safely evolve C program families. We develop a strategy to detect bugs related to variability and define refactorings to remove bad smells in preprocessor directives. Our supporting tool, *Colligens*, implements our strategy to detect bugs and applies our refactorings automatically. By using our approach in 40 program families, we detect 121 bugs related to variability, and developers accepted 78% of the patches we submit. Also, we remove 477 bad smells in 12 C program families without clone code as in previous studies.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

Keywords Program Families; Preprocessors; Bugs; Bad Smells; Refactorings

1. Motivation

Developers often use C to develop infrastructure software like web servers, such as *Apache* and *Cherokee*, and operating systems such as *Linux* and *Android*. Infrastructure software requires variability to run on different platforms. In this context, developers normally use preprocessors to handle variability and portability problems [2]. By using the C preprocessor, developers encompass parts of the source code with preprocessor directives, such as `#ifdef` and `#endif`. In other words, developers deal with a family of programs, which is a set of similar programs whose commonality is so extensive that it is useful to study their common properties before analyzing individual family members [8].

Program families implementing multi-platform infrastructure software, e.g., *Apache* and *Linux*, evolve continually to support new operating systems and recent releases. Also, they require high quality software artifacts to minimize chances of financial losses due to software bugs. For instance, a critical software running on a web server cannot stop serving clients due to a software crash, or an out of memory error. This way, practical studies with the C preprocessor are helpful to understand common problems, detect bugs, provide insights for better tool support, increase software quality, and support evolution of program families.

2. Problem

Developers use the C preprocessor in well-known and widely used program families. It is an effective tool that allows developers to encompass any code fragment with preprocessor directives, even a single token. However, they should be careful when using the C preprocessor to avoid bad smells in preprocessor directives [3]. For instance, *Gnuplot* developers annotate only part of an `if` statement with preprocessor directives as we can see in Listing 1 (Line 3). Notice that the correspondent closing bracket is at Line 10. It is an incomplete annotation, i.e., directives that encompass only parts of C syntactical units [4]. Incomplete annotations are bad smells in preprocessor directives because of their negative impact on code quality, e.g., developers may need more time to reason about the code, to detect where `if` statements end or to analyze whether opening and closing brackets match correctly [2, 5].

Listing 1. Code snippet of *Gnuplot* with bad smells.

```
1. if (*Y_AXIS.label.text) {
2. #ifdef PM3D
3.     if (rot_x <= 90){
4. #endif
5.     double step = (other_end - yaxis_x) / 4;
6.     // Several lines of code..
7. #ifdef PM3D
8.     if (map)
9.         *t = text_angle;
10.    }
11. #endif
12. }
```

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPLASH '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3208-8/14/10.
http://dx.doi.org/10.1145/10.1145/2660252.2660254

Researchers propose refactorings to remove incomplete annotations [4, 9], but they clone code as we can see in Listing 2. It clones the list of statements at Lines 5 and 11. Thus, the proposed refactorings remove incomplete annotations by introducing another bad smell, i.e., code clone [3].

Listing 2. Existing refactoring to remove bad smells.

```
1. if (*Y_AXIS.label.text) {
2. #ifdef PM3D
3.     if (rot_x <= 90){
4.         double step = (other_end - yaxis_x) / 4;
5.         // Several lines of code..
6.         if (map)
7.             *t = text_angle;
8.     }
9. #else
10.    double step = (other_end - yaxis_x) / 4;
11.    // Several lines of code..
12. #endif
13. }
```

Notice that the code snippet presented in Listing 1 contains no bugs. But it contains bad smells that eases the introduction of bugs related to variability [6, 7]. For instance, Listing 3 presents a newer version of the same code. However, developers introduce a bug when we do not define macro `PM3D`. By preprocessing the code of Listing 3 using this configuration, developers close a bracket at Line 8 without open its correspondent bracket at Line 3. Thus, we generate an invalid program that does not compile. But it compiles if we activate `PM3D`. Bugs related to variability are hard to detect because they happen only in specific configurations.

Listing 3. Code snippet of *Gnuplot* with a variability bug.

```
1. if (*Y_AXIS.label.text) {
2. #ifdef PM3D
3.     if (rot_x <= 90){
4. #endif
5.     double step = (other_end - yaxis_x) / 4;
6.     // Several lines of code..
7.     if (map) { *t = text_angle; }
8.     }
9. }
```

In particular, the majority of C development tools are not variability-aware. For instance, *GCC* and *Clang* preprocess the code and consider each family member individually. In academy, there are some variability-aware tools, e.g., *TypeChef* [5], which parses program families code and checks type errors. However, *TypeChef* uses a time-consuming strategy that needs to consider all external dependencies defined through `#include` directives. Despite of a few bug checkers in *TypeChef*, there are no variability-aware tools that focus on detecting different types of semantic bugs.

The strategy that considers each family member individually does not scale due to the high number of possible configurations. On the other hand, strategies that consider all external dependencies need a time-consuming set up to identify and install them. In addition, we have difficulties to install dependencies specific to a particular operating system. For instance, we cannot install *windows.h* in *Linux* since this library is not available for unix-based systems. Thus, without an appropriate tool support, developers introduce bad smells and bugs related to variability [1, 6, 7].

3. Approach

We propose an approach to safely evolve C program families, which supports developers to improve code quality, and detect bugs related to variability. In this context, we propose a catalogue of refactorings to remove bad smells in preprocessor directives, and a strategy to detect different bugs, such as syntax errors, null deferences, memory leaks, resource leaks, and uninitialized variables. Further, we develop a supporting tool named *Colligens*, which implements our strategy to detect bugs and applies our refactorings automatically.

Our refactorings are unidirectional transformation templates, which satisfy specific preconditions in order to minimize chances of introducing behavioural changes. In addition, they are simple and local transformations without global impact. But, we can compose them to perform different transformations. By removing bad smells, we improve code quality in the sense that the refactored code has preprocessor directives encompassing only complete C syntactical units. For instance, Refactoring 1 shows how we remove incomplete annotations in `if` wrappers. In this refactoring, we use an additional variable to keep the statement condition. To avoid compilation errors, we define a precondition that the code is not using identifier `test` in the scope. Notice that our refactoring does not clone code. Refactoring 2 implements runtime variability. Thus, after applying it, we have no directives. We use a local variable `EXPR_RT` to keep the value of macro `EXPR`. By applying Refactoring 1 or 2, we can remove the bad smells of *Gnuplot* presented before.

Refactoring 1. ⟨Remove incomplete if wrappers⟩

```
1. #ifdef expression_1
2.     if (condition_1) {
3. #endif
4.     // Stmts_1
5. #ifdef expression_1
6.     }
7. #endif
```

→

```
1. bool test = TRUE;
2. #ifdef expression_1
3.     test = condition_1;
4. #endif
5. if (test) {
6.     // Stmt_1
7. }
```

(→) variable `test` is not used in this scope.

Refactoring 2. ⟨Remove wrapper with runtime variability⟩

```
1. #ifdef EXPR
2.   if (COND) {
3. #endif
4.     STMTS
5. #ifdef EXPR
6.   }
7. #endif
```

→

```
1. bool EXPR_RT = EXPR;
2. if ((EXPR_RT && COND) || !(EXPR_RT)){
3.     STMTS
4. }
```

(→) variable `EXPR_RT` is not used in the code.

In some cases as depicted in Listing 3, bad smells may ease the introduction of bugs. Thus, we also define a strategy to detect bugs related to variability. To detect syntax errors, we use *TypeChef* to parse all family members, but we use stubs to substitute the external dependencies and avoid the time-consuming initial set up [5]. Figure 1 presents the steps of our strategy as discussed next. In *Step 1*, we exclude the external dependencies defined through `#include` directives and create the stubs. *Step 2* generates a script that calls *TypeChef* for each source file. In *Step 3*, we analyze the syntax errors to identify the ones related to variability. In

the third step, we also get feedback from the actual program families developers to confirm the syntax errors.

To detect semantic bugs, our strategy uses a framework that allows the use of different sampling algorithms and static analysis tools. By sampling configurations and selecting only a few family members to analyze, we can use non-variability-aware tools that exist for several years, e.g., *CppCheck*[1] and *Splint*.[2] Figure 2 shows our strategy to detect semantic bugs. In *Step 1*, we use sampling to select a set of family members. Then, we use a static analysis tool to find bugs in *Step 2*. Notice that we can identify the same bug in different family members. Thus, *Step 3* removes duplicated bugs by analyzing their presence conditions.

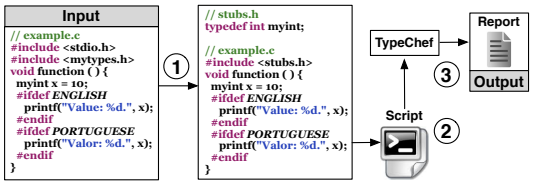

Figure 1. Detecting syntax errors in C program families.

4. Evaluation Methodology

Our goal is to evaluate our approach with respect to its efficiency to remove bad smells and detect bugs related to variability. Thus, we address the following research questions: **Q1.** Can our approach detect bugs related to variability? **Q2.** Which sampling algorithm finds the highest number of bugs? **Q3.** Can our approach detect and remove bad smells in preprocessor directives without clone source code? **Q4.** Does our catalogue of refactorings improve code understanding and maintainability? **Q5.** Why do developers still use preprocessor directives instead of runtime variability?

Planning. We plan to select families of different domains and sizes. To answer question **Q1**, we intend to perform an empirical study. To confirm that we detect real bugs, we will ask developers. In question **Q2**, we plan to compare different sampling algorithms to analyze the ones that detect the highest number of bugs. To answer question **Q3**, we intend to use our tool to detect and remove bad smells. Regarding questions **Q4** and **Q5**, we plan to send questionnaires and do interviews with real developers to find out whether they would use our refactorings. **Threats to validity.** We minimize threats related to construct validity by getting feedback from developers to confirm the bugs detected in our study. Our strategy excludes `#include` directives to eliminate external libraries in order to scale. Notice that we may face false negatives and positives. However, we minimize false positives using developers feedback. To minimize threats related to external validity, we select families of different sizes, ranging from 4.9 thousand to 1.5 million lines of code, and distinct domains, e.g., web servers and databases.

[1] `http://cppcheck.sourceforge.net/`

[2] `http://www.splint.org/`

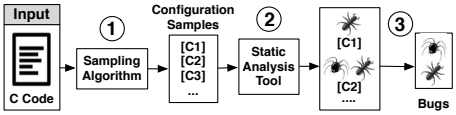

Figure 2. Detecting semantic bugs in C program families.

5. Research Status

Preliminary results. (**Q1**) By analyzing 40 families, we detect 121 bugs. We submit patches to fix bugs and developers accepted 78% of them. The majority of patches rejected were related to invalid configurations. (**Q2**) We implement different sampling algorithms to compare the number of bugs detected, e.g., t-wise. (**Q3**) Regarding our refactorings, we evaluate it by removing 477 bad smells in 12 families without cloning code, and increasing in 0.04% the lines of code and in 2.10% the number of directives. We are currently analyzing whether the order we apply our refactorings impacts the resulting code quality. **Future work.** So far, we have defined a strategy to detect bugs and refactorings to remove bad smells related to variability. We plan to analyze several sampling algorithms to detect bugs. Further, we intend to define a technique that avoids behavioural changes when applying our refactorings like in *SafeRefactor* [10]. Regarding **Q4**, we plan to evaluate the effectiveness of our refactorings by using a questionnaire. We will measure the impact of using incomplete annotations regarding code understandability and maintainability. In addition, we intend to do interviews to ask real program family developers whether they would use our catalogue of refactorings in practice. We will also use interviews to answer question **Q5**.

References

[1] I. Abal, C. Brabrand, and A. Wasowski. 40 variability bugs in the Linux kernel. Technical report, IT Univ. Copenhagen, Denmark, 2014.

[2] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *Trans. on Software Engineering*, 2002.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *ICSM*, 2005.

[5] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of macros and conditional compilation. In *OOPSLA*, 2011.

[6] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating Preprocessor-Based Syntax Errors. In *GPCE*, 2013.

[7] F. Medeiros, M. Ribeiro, R. Gheyi, and B. Fonseca. A catalogue of refactorings to remove incomplete annotations. *Journal of Universal Computer Science*, January 2014.

[8] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 1976.

[9] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. Does the discipline of annotations matter? In *GPCE*, 2013.

[10] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *IEEE Software*, 2010.