Safely Evolving Configurable Systems

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

Abstract

Developers use configuration options to tailor systems to different platforms. This configurability leads to exponential configuration spaces and traditional tools (e.g., *Gcc*) check only one configuration at a time. As a result, developers introduce configuration-related issues (i.e., bad smells and faults) that appear only when we select certain configuration options. By interviewing 40 developers and performing a survey with 202 developers, we found that configuration-related issues are harder to detect and more critical than issues that appear in all configurations. We propose a strategy to detect configuration-related issues and an approach to improve code quality (i.e., to remove bad smells in preprocessor directives using a catalogue of refactorings). We found 131 faults and 500 bad smells in 40 real configurable systems, including *Apache* and *Libssh*, ranging from 2.6 KLOC to 1.5 MLOC.

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

Keywords Configuration-Related Issues, Software Variability

1. Research Problem and Motivation

A number of software systems provide configuration options to tailor the system to different platforms. The inherent variability of configurable systems leads to configuration spaces of exponential sizes. It is commonly infeasible to explore the entire configuration space exhaustively. As a result, developers introduce configuration-related issues (i.e., bad smells and faults) when evolving configurable systems. For instance, Figure 1a depicts a bad smell in *Gnuplot*. Developers split up parts of an `if` statement with conditional directives, as we can see at lines 2 and 7. To better understand this problem, we perform interviews and a survey. We found that more than 80% of developers agree that this type of directive impacts code understanding, maintainability and error proneness [6]. Furthermore, many configuration-related faults appear in this type of conditional directive [3]. When evolving the code snippet of Figure 1a, developers introduce a configuration-related fault, as we present in Figure 1b. We expose this syntax fault only when we disable `PM3D`. Despite being easy to spot, traditional C compilers, such as *Gcc* and *Clang*, report no warnings or error messages when one compiles this code snippet without disabling `PM3D`. Thus, the existing tool support is not adequate to detect configuration-related issues. In this context, we found that configuration-related faults remain in the code for several years [3, 7], and more than 74% of developers believe that such issues are harder to detect and more critical than issues that appear in all configurations [6].

```
1. #ifdef PM3D
2. if (rot_x <= 90){
3. #endif
4.   ...
5. #ifdef PM3D
6.   if (map) *t = text_angle;
7. }
8. #endif     (a)
```

```
1. #ifdef PM3D
2. if (rot_x <= 90){
3. #endif
4.   ...
5.   if (map) *t = text_angle;
6. }
              (b)
```

Figure 1. (a) Bad smell in a preprocessor conditional directive of *Gnuplot*; and (b) Configuration-related fault in the *Gnuplot* project.

2. Background

The C preprocessor has received strong criticisms in research studies. In particular, many researchers and practitioners discussed the problems of splitting up parts of statements with preprocessor conditional directives [1, 3, 6]. By interviewing 40 developers (using *grounded theory*) and performing a survey with 202 developers, we found evidence that this type of conditional directive is a bad small regarding preprocessor usage [6]. Despite the existence of refactorings to remove these bad smells, they clone the source code [8], i.e., impacting code quality.

Configuration-related issues appear only in some configurations. Thus, more 74% of developers agree that they are harder to detect than issues that appear in all configurations [6]. It is normally infeasible to check each system configuration separately. To minimize the problems of checking the entire configuration space, researchers proposed variability-aware tools that consider all configurations simultaneously [2], and applied sampling analysis to select only a subset of valid configurations [5]. Variability-aware tools require a time-consuming initial setup [3, 7], and there is a lack of studies comparing the efficiency of sampling algorithms. To fill this gap, we compared 10 sampling algorithms using a set of 135 known faults [5]. We identified algorithms that provide an useful balance between the number of configurations selected for analysis and the number of faults detected. For instance, the *LSA* algorithm as presented in Figure 2. The effectiveness of sampling for detecting configuration-related issues depends significantly on how samples are selected.

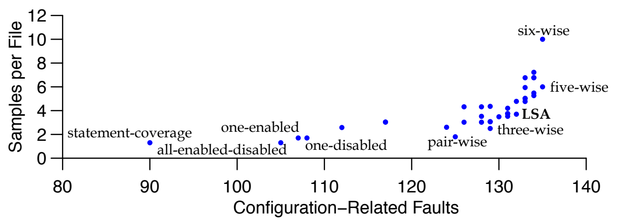

Figure 2. Comparing sampling algorithms.

3. Approach

We found in a recent study that developers face configuration-related issues in practice [6]. Hence, we propose an approach to safely evolve configurable systems that supports developers to improve code quality, and to detect configuration-related issues. In this context, we propose a strategy to detect configuration-related issues, including bad smells, undeclared functions and variables, syntax errors, null pointer deferences, memory leaks, resource leaks, and uninitialized variables. Furthermore, we propose a catalogue of refactorings to remove bad smells in preprocessor directives. Our tool *Colligens* implements our strategy and applies our refactorings automatically.

Our strategy to detect configuration-related issues supports sampling and variability-aware analysis (see Figure 3). Notice that it defines two alternative paths: one that applies sampling (*Steps 1* and *2*) and another for variability-aware analysis (*Steps 3* and *4*). To perform sampling, our strategy needs a sampling algorithm [5] (e.g., *LSA*) and a static analysis tool (e.g., *Cppcheck*).[1] Moreover, the strategy receives as input the code of the project, the constraints among configuration options and the build-system information. In *Step 1*, it uses the sampling algorithm to select a subset of valid configurations, and *Step* 2 applies the analysis tool. To perform variability-aware analysis, we create stubs (*Step 3*) to substitute the external dependencies and to avoid the time-consuming initial setup of variability-aware tools, such as *TypeChef* [2]. The stubs contain all type definitions, which allow our strategy to ignore the external dependencies defined through `#include` directives, making the strategy scalable. In *Step 4*, we apply the variability-aware tool the strategy receives as input. Last, *Step 5* reports the configuration-related issues detected by the strategy.

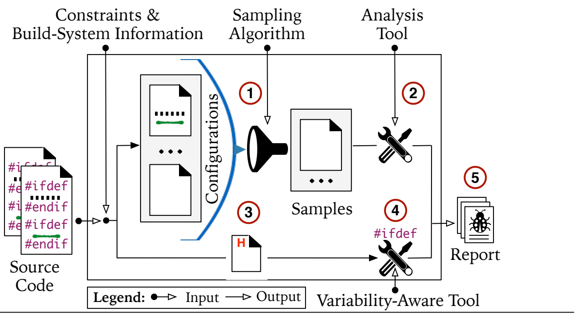

Figure 3. Strategy to detect configuration-related issues.

Our refactorings are unidirectional transformation templates that satisfy specific preconditions. The refactorings are simple and local transformations without global impact. We can compose refactorings to perform more coarse-grained transformations. By removing bad smells, we improve code quality in the sense that the refactored code contains no preprocessor directives that split up parts of statements. For instance, Refactoring 1 shows how we remove conditional directives that split up parts of `if` statements. In this refactoring, we use an additional local variable and define a precondition to avoid compilation errors. By applying Refactoring 1, we can remove the bad smell presented in Figure 1a.

4. Results and Contributions

We evaluate our refactorings by applying our catalogue in 12 software systems [4]. We detect and remove 477 occurrences of preprocessor conditional directives that split parts of statements (i.e., bad smells). Our catalogue of refactorings does not introduce code clone, and increase the lines of code and number of conditional directives in 0.04% and 2.1% respectively. The results reveal that we

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

Refactoring 1. ⟨Remove incomplete if wrappers⟩

```
#ifdef expression_1
   if (condition_1) {
#endif
   // Stmts_1
#ifdef expression_1
   }
#endif
```

```
bool test = TRUE;
#ifdef expression_1
   test = condition_1;
#endif
if (test) {
    // Stmts_1
}
```

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

can remove bad smell in preprocessor directives without cloning the source code. In addition, we detect other kinds of bad smells: 7 unused functions and 16 unused variables (see Figure 4).

We also performed an additional empirical study using 40 software systems. We instantiate our strategy to detect configuration-related faults using the following tools: *TypeChef*, a variability-aware tool; and *Cppcheck*, a static analysis tool used by developers of many open-source systems. We detect 24 syntax faults, 16 type faults (e.g., undeclared functions and variables), and 91 semantic faults (e.g., memory and resource leaks), totalling 131 configuration-related faults in 24 out of 40 software systems ranging from 2.6 KLOC to 1.5 MLOC. Figure 4 presents these faults. We confirmed all these faults by getting feedback from the actual developers or by finding the fixes in the software repositories.

Unused Function	7	Bash (4), Libpng (1), M4 (1), Libpng (1)
Unused Variable	16	Bash (15), Libssh (1)
Split up parts of statements with directives	477	Apache (178), Bc (6), Dia (31), Expat (31), Flex (16), Fvwm (61), Ghostscript (87), Gnuchess (2), Gzip (19), Lighttpd (23), Lua (6), Mptris(17)
Syntax Fault	24	Apache (3), Bash (2), Cvs (1), Dia (2), Gnuplot (5), Libpng (3), Libssh (2), Vim (4), Xfig (1), Xterm (1)
Undeclared Variable	2	Gzip (1), Libpng (1)
Undeclared Function	14	Bash (1), Gnuchess (1), Gzip (2), Libpng (6), Lua (2), Libssh (1), Privoxy (1)
Uninitialized Variable	31	Apache (4), Bash (3), Cherokee (4), Dia (3), Gawk (4), Libsoup (1), Libssh (2), Lua (2), Mpsolve (1), Sqlite (5), Vim (1), Xterm (1)
Memory Leak	27	Apache (2), Cherokee (3), Gawk (1), Fvwm (3), Kindb (5), Gnuplot (5), Libssh (4), Vim (4)
Resource Leak	5	Cvs (1), Gnuplot (2), Kindb (2), Libxml (1), Lighttpd (1), Sylpheed (1)
Deference of Null Pointer	26	Apache (6), Bash (3), Cherokee (4), M4 (4), Privoxy (2), Sylpheed (3), Vim (2), Xterm (2)

Figure 4. Configuration-related issues (bad smells and faults).

References

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

[2] 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 OOPSLA*. ACM, 2011.

[3] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of GPCE*. ACM, 2013.

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

[5] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. Reasoning about sampling algorithms for configurable systems. Technical Report TR-15-001, Federal University of Campina Grande, 2015.

[6] 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 ECOOP*, 2015.

[7] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In *Proceedings of GPCE*. ACM, 2015.

[8] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. Does the discipline of preprocessor annotations matter? A controlled experiment. In *Proceedings of GPCE*, 2013.