

# Static Program Analysis In Presence Of Multiple Configurations

by

Farnaz Behrang

A thesis submitted to the Graduate Faculty of  
Auburn University  
in partial fulfillment of the  
requirements for the Degree of  
Master of Science

Auburn, Alabama  
August 2, 2014

Keywords: Refactoring, Preprocessor, Static Analysis

Copyright 2014 by Farnaz Behrang

Approved by

Munawar Hafiz, Chair, Assistant Professor of Computer Science  
Jeffrey L. Overbey, Assistant Professor of Computer Science  
David Umphress, Professor of Computer Science

## Abstract

C programs make heavy use of the C preprocessor, which makes them highly configurable. C programming IDEs ignore multiple configurations of C preprocessor because of its complexity. However, program analyses and transformations have to consider all possible macro configurations; otherwise, they will be incorrect. This problem is exacerbated in the context of program transformations: it is impossible to implement any non-trivial program transformation correctly without supporting sophisticated static analysis.

We modified OpenRefactory/C, a framework for building correct and complex program transformations for C, to support configuration-aware static analysis and program transformations. We modified the program representation to include conditional directives and extended the static program analysis to be aware of multiple configurations. For evaluation, we used the Extract Function refactoring implemented in OpenRefactory/C.

## Acknowledgments

I would like to thank Dr. Munawar Hafiz and Dr. Jeffrey Overbey for their guidance and support throughout this work.

## Table of Contents

Abstract . . . . .	ii
Acknowledgments . . . . .	iii
List of Figures . . . . .	vi
List of Tables . . . . .	viii
1 Introduction . . . . .	1
1.1 Configuration-aware Analysis . . . . .	2
1.2 Impact of Multiple Configurations on Refactoring and Static Analysis . . . . .	2
1.3 Thesis Statement . . . . .	4
1.4 Thesis Outline . . . . .	4
2 OpenRefactory/C . . . . .	6
3 Preprocessor Modifications to support Multiple Configurations . . . . .	8
3.1 Introduction . . . . .	8
3.2 Handling Macros . . . . .	8
3.3 Handling Conditional Directives . . . . .	12
3.3.1 Calculating guarding conditions . . . . .	14
3.3.2 Running Example . . . . .	15
3.4 Conclusions . . . . .	16
4 Program Representation that Supports Multiple Configuration Preprocessor . . . . .	17
4.1 Introduction . . . . .	17
4.2 Abstract Syntax Tree . . . . .	19
4.2.1 Running Example . . . . .	19
4.3 Conclusion . . . . .	21
5 Multiple configuration aware static analysis . . . . .	22

5.1	Control Flow Analysis . . . . .	22
5.1.1	Introduction . . . . .	22
5.1.2	Running Example . . . . .	23
5.1.3	Control Entry and Control Exit . . . . .	25
5.1.4	Calculating Successors and Predecessors . . . . .	25
5.2	Data Flow Analysis . . . . .	27
5.2.1	Running Example . . . . .	28
5.3	Preprocessor Dependence Preservation Analysis . . . . .	28
5.4	Conclusion . . . . .	30
6	Evaluation . . . . .	31
6.1	General Testing Approach . . . . .	31
6.2	A Problem . . . . .	32
6.3	Test cases . . . . .	34
6.3.1	Designing Test cases . . . . .	34
6.3.2	Extract Function Refactoring . . . . .	34
6.3.3	Evaluating Test cases . . . . .	36
6.3.4	Discussion . . . . .	38
7	Related Work . . . . .	40
8	Conclusions and Future Works . . . . .	42
	Bibliography . . . . .	43

## List of Figures

1.1	An example to apply Extract Function refactoring . . . . .	3
1.2	Code segment from Zlib . . . . .	4
3.1	An example of a macro with multiple definitions . . . . .	9
3.2	Use of a macro with multiple definition . . . . .	9
3.3	Another example of a macro with multiple definitions . . . . .	10
3.4	An example of incompatible conditions . . . . .	13
3.5	An example of conditions causing infinite loop . . . . .	14
3.6	A segment of code extracted from Zlib . . . . .	15
4.1	An example of breaking the complete C constructs . . . . .	18
4.2	Examples extracted from GNU Coreutils 8.21 . . . . .	18
4.3	Grammar to include C preprocessor directives . . . . .	19
4.4	An example extracted from gmp-4.3.2 . . . . .	20
4.5	Partial AST generated for the example in figure 4.4 . . . . .	20
5.1	A code snippet extracted from GNU Coreutils 8.21 . . . . .	24
5.2	CFG for figure 5.1 . . . . .	24

5.3	Strategy to calculate non jump exits of directive condition node . . . . .	25
5.4	Strategy to calculate predecessors of directive condition node . . . . .	26
5.5	Strategy to calculate successors of directive condition node . . . . .	26
5.6	Strategy to calculate successors for every item of a conditional directive block item list . . . . .	27
5.7	Strategy to calculate predecessors for every item of a conditional directive block item list . . . . .	27
5.8	A code snippet extracted from gmp-4.3.2 . . . . .	28
5.9	An example of moving macro to undefined region . . . . .	29
6.1	An example extracted from chmod.c . . . . .	33
6.2	Expansion of macro S_IRWXU . . . . .	33
6.3	Test coverage results for MoveLocalVariable micro refactoring using EclEmma .	37
6.4	Part of Add Brace micro refactoring code . . . . .	38

## List of Tables

3.1	Guarding conditions . . . . .	16
4.1	Nodes of AST . . . . .	21
5.1	Reaching definition information for code in figure 5.8 . . . . .	28
6.1	Failure and cluster statistics for C projects . . . . .	32
6.2	Test coverage results for Extract Function micro refactorings using EclEmma . .	37



## Chapter 1

### Introduction

Refactorings are known to improve the design and structure of the code, making it more maintainable and easier to understand and modify. Refactoring tools are included in modern IDEs such as Eclipse and Visual Studio to change the source code in a way that preserves behavior.

Extensive work has been done to provide refactoring tools in IDEs that support object-oriented programming languages like Java and C#. However, in spite of its popularity, the same is not true for C. Existing refactoring tools for C only support primitive refactorings without any sophisticated analysis. Gligoric et al. [14] applied Eclipse's refactorings for Java and C to a number of open source codes; 1.4% of the Java test cases failed, while 7.5% of the C tests failed.

There are many factors that make C difficult to refactor correctly. One of the main factors is heavy use of C preprocessor [7]. While using preprocessor directives makes the C code more flexible, its lack of structure adds complexity. Since preprocessor directives do not conform to the C grammar, code must be preprocessed first before it can be parsed. It is impossible to apply refactorings on preprocessed version of C code since the un-preprocessed version cannot be recovered later. Most IDEs including Eclipse CDT provide program analysis and refactoring support based on a single preprocessor configuration, which only considers one branch of conditional directives. Supporting multiple preprocessor configurations will enable the refactoring tool to analyze and transform the un-preprocessed source code considering all possible configurations.

It is impossible to implement refactorings correctly without any sophisticated static analysis. Preprocessor directives and multiple configurations makes the analysis more complicated since all possible macro configurations need to be considered.

## 1.1 Configuration-aware Analysis

Configuration-aware analyses is a class of analyses that analyze code with respect to all possible macro configurations. This analyses recently has received attention among researchers in different contexts including type checking, data-flow analysis, and model checking. The idea behind all of them is to analyze the similar code among variants only once to make the analysis more efficient and applicable to large-scale systems. However, the details are different.

The most similar work to ours is due to Liebig et al. [25]. They implemented variability-aware type checking and liveness analysis to compare the performance of these variability-aware analysis with the performance of corresponding sampling heuristics (single configuration, pair-wise, and code coverage). They found that the performance of variability-aware analysis scales to large scale systems such as the Linux kernel. However, they did not apply their analysis in any application areas other than empirical studies. Refactoring tools are practical application of configuration-aware analysis which present challenges specific to this context. We used OpenRefactory/C as a practical infrastructure to implement configuration-aware analysis which is a real necessity to perform certain refactorings correctly.

## 1.2 Impact of Multiple Configurations on Refactoring and Static Analysis

The existing refactoring tools in C ignore multiple preprocessor configurations since it adds more complexity to build correct and complex refactoring tools. Instead, they consider single configuration which only provides partial information. Therefore, the resulted refactoring and static analysis is inaccurate.

Figure 1.1 shows a sample code. If we try to extract "`x = 0;`" using an extract function refactoring that only supports a single configuration, the result will be the program in figure 1.2 on the left. The conditional directive has been evaluated, and since macro "M" is not defined, the `#if` branch is ignored. However, supporting multiple preprocessor configurations will result in the code on the right, which considers both branches of `#if` directive.

<pre>int main(){   #ifdef M   int x;   #else   unsigned int x;   #endif   x = 0;   printf("x is %d", x); }</pre>	<pre>unsigned int foo(unsigned int x) {   x = 0;   return x; }</pre>	<pre>#ifdef M int foo(int x) {   x = 0;   return x; } #else unsigned int foo(unsigned int x) {   x = 0;   return x; } #endif</pre>
(a) before refactoring	(b) single configuration	(c) multiple configuration

Figure 1.1: An example to apply Extract Function refactoring

Reaching definition analysis and definition-use chains are used to identify which parameters to pass to a method and which value to return from an extracted method.

Consider the code segment in figure 1.2, extracted from the Zlib data compression library, v1.2.5, trees.c, lines 970 to 991. Variable "`opt_lenb`" is defined on line 970 and used on lines 976 and 990. We want to extract line 970 using extract function refactoring. If only `#ifdef` branches are taken, ignoring `#else` branches, there would not be any uses of "`opt_lenb`" after extraction. Hence, the extracted function won't return any value. However, considering all the branches there would be uses of "`opt_lenb`" which requires the extracted method to return this variable.

Since libraries might be used under different system configurations, no prediction can be made about taking a particular branch by conditional directives. Therefore, the program analysis and refactorings must consider all feasible possibilities, otherwise they will be incorrect.

```

970 opt_lenb = static_lenb = stored_len + 5;
    ...
973 #ifdef FORCE_STORED
974   if (buf != (char*)0) { /* force stored block */
975 #else
976   if (stored_len+4 <= opt_lenb && buf != (char*)0) {
977       /* 4: two words for the lengths */
978 #endif
979 #ifdef FORCE_STATIC
980   } else if (static_lenb >= 0) { /* force static trees */
981 #else
982   } else if (s->strategy == Z_FIXED || static_lenb == opt_lenb) {
983 #endif

```

Figure 1.2: Code segment from Zlib

### 1.3 Thesis Statement

Correct program transformations on C programs depend on correctly accounting for multiple configurations. This involves creating a program representation that supports multiple configurations, augmenting program analyses, and introducing new program analyses to include preprocessor directives. The program transformations should be tested for correctness on real programs in addition to synthetic test cases.

### 1.4 Thesis Outline

The outline of this thesis is:

- Chapter 2 describes the frame work we worked on. OpenRefactory/C is an infrastructure for building correct and complex refactorings and other source-level program transformations for C.
- Chapter 3 discusses handling of macros and conditional directives in presence of multiple configuration
- Chapter 4 discusses the program representation that allows multiple configuration support

- Chapter 5 discusses the multiple configuration, preprocessor-aware static analysis
- Chapter 6 discusses the evaluation of multiple configuration support using Extract Function Refactoring
- Chapter 7 discusses the related work
- Chapter 8 discusses the future work and conclusion

## Chapter 2

### OpenRefactory/C

OpenRefactory/C [17] is a C-specific infrastructure for building refactorings and similar source-level transformations, while correctly handling all of the complexities of C. It is written in Java and supports different front ends including Eclipse plug-in, Vim plug-in, command line, and web demo. Twelve program transformations including both refactorings and behavior enhancing transformations [16] has also been implemented in this framework. OpenRefactory/C supports different static analysis:

- Name binding Analysis: Name binding information is stored in the abstract syntax tree (AST). Any AST node representing a name reference can be queried to return the corresponding declaration node.
- Type Analysis: Type information is computed as a query to the AST using name binding information.
- Control flow Analysis : Control flow information is computed dynamically querying AST nodes using improved version of Morgenthaler's [27] algorithm.
- Alias Analysis : OpenRefactory/C supports an inclusion-based (Andersen-style) alias analysis for local variables.
- Data flow Analysis : OpenRefactory/C stores intraprocedural reaching definitions and definition-use relationships in the AST.
- Dependence Analysis : OpenRefactory/C supports scalar data dependence analysis for local variables, as well as computation of control dependences. Control and data

dependences check behavior preservation of transformations that move code within a procedure.

OpenRefactory/C's C parser and rewritable AST are generated using Ludwig [29], which was also used to generate the syntactic manipulation infrastructure in Photran. OpenRefactory/C uses mutable ASTs as its primary program representation. Source code manipulation is performed by modifying the AST and later traversing the tree to output the revised source code (or create a patch file). All AST nodes implement a common interface that includes methods to perform tree traversals using the Visitor pattern, find nodes by type, determine preprocessor constructs affixed to nodes, and of course, manipulate the source code associated with a node. Like the majority of refactoring tools, OpenRefactory/C uses AST for both analysis of the code and transformation.

One of the main research goals for OpenRefactory/C is to support analysis and transformation of code containing C preprocessor directives, with correct results under all feasible preprocessor configurations. These changes require intricate modifications to every part of the infrastructure: the preprocessor, the lexical analyzer, the parser, the AST, every static analysis, and every transformation.

The following chapters of this thesis will discuss about the modifications that we made to every part in order to support configuration-aware static analysis (control flow and data flow analysis) and configuration-aware transformations.

## Chapter 3

### Preprocessor Modifications to support Multiple Configurations

#### 3.1 Introduction

There are two ways to construct an AST from source code containing preprocessor directives. One way would be to preprocess the code, parse it and construct the AST. Using this approach, branches of conditional directives except one will be lost. Since we want to consider all feasible preprocessor configurations, we need to pseudo-preprocess [9] the code. When a pseudo-preprocessor is used, the structure of AST is similar to the code after preprocessing has been applied while it still contains all feasible preprocessor configurations of the code.

OpenRefactory/C follows the pseudo-preprocessing model described by Garrido [9]. It includes a lexical analyzer and a parser for generating ASTs. There is a customized version of C preprocessor that feeds the lexical analyzer.

In this chapter we discuss about the modifications made in OpenRefactory/C's preprocessor to enable multiple configuration support. Since during pseudo-preprocessing all the feasible branches are considered simultaneously, macros may have multiple expansions. In the rest of the chapter, we will discuss about determining feasible configurations and calculating the guarding condition for every point of the source code.

#### 3.2 Handling Macros

Considering all feasible preprocessor configurations, macros may have more than one definition under different branches of conditional directives. Figure 3.1 shows an example



of a macro with different definitions in each branch extracted from `/usr/include/i386-linux-gnu/bits/stdio.h`.

```
#ifndef __extern_inline
# define __STDIO_INLINE inline
#else
# define __STDIO_INLINE __extern_inline
#endif
```

Figure 3.1: An example of a macro with multiple definitions

To make multiple definitions of a macro name distinguishable, we have to keep track of guarding conditions associated with each macro definition. Later, when we expand a macro call, every possible expansion is guarded by some guarding conditions. The guarding condition for the first definition of macro `__STDIO_INLINE` is `!defined (__extern_inline)` and for the second definition is `defined (__extern_inline)`. Figure 3.2 shows a use of this macro before and after expansion, in the same file. For this simple example, only keeping track of the immediate level of guarding conditions is sufficient. However, this is not always true.

<pre>__STDIO_INLINE int fgetc_unlocked (FILE *__fp) {     return _IO_getc_unlocked (__fp); }</pre>	<pre>#ifndef __extern_inline inline #else __extern_inline #endif int fgetc_unlocked (FILE *__fp) {     return _IO_getc_unlocked (__fp); }</pre>
(a) Before expansion	(b) After expansion

Figure 3.2: Use of a macro with multiple definition

During preprocessing when we encounter a macro, we are not aware if there will be more macros with the same name. More importantly, we are not aware of the sufficient level of conditions we have to consider to make multiple definitions of that macro distinguishable. Hence, we have to keep track of all the conditions. However, later when we do the expansion, we will determine the sufficient level of guarding conditions we need for each macro to make it distinguishable from other macros with the same name.

Figure 3.3 shows a segment of code extracted from `/usr/include/i386-linux-gnu/sys/cdefs.h`, lines 380 to 410. Function like macro `__REDIRECT_LDBL` is defined on lines 393 and 406. For this example, if we just keep track of the enclosing guarding conditions, it would be lines 382 and 405 which is similar for both definitions. In this case, we have to consider another level of guarding conditions as well to make two definitions distinguishable during expansion.

```

380 #if defined __LONG_DOUBLE_MATH_OPTIONAL && defined __NO_LONG_DOUBLE_MATH
381 # define __LDBL_COMPAT 1
382 # ifdef __REDIRECT
...
393 # define __REDIRECT_LDBL(name, proto, alias) \
394 __LDBL_REDIR1 (name, proto, __nldbl_##alias)
...
397 # endif
398 #endif
399 #if !defined __LDBL_COMPAT || !defined __REDIRECT
...
405 # ifdef __REDIRECT
406 # define __REDIRECT_LDBL(name, proto, alias) __REDIRECT (name, proto, alias)
...
409 # endif
410 #endif

```

Figure 3.3: Another example of a macro with multiple definitions

Thus, we wrote an algorithm to determine the sufficient level of guarding conditions for every macro that we expand with more than one definition. Generally, two cases may happen:

- The enclosing guarding condition for the macro definition is either **#elif** or **#else** directive. In this case, we have to find the nearest if directive. (steps 1-2)
- The enclosing guarding condition for the macro definition is either **#if** or **#ifdef** or **#ifndef** directive:
  - *The enclosing guarding condition is not the same for both definitions.* We keep the condition, but still find the nearest if directive for both definitions since the **#if** directive of one of the definitions might be in the **#elif** or **#else** branches

of the other definitions. If they are the same, then keep these conditions as well.  
(steps 3-16)

- *The enclosing guarding condition is the same for both definitions.* We keep the conditions until the conditions are not the same any more (the example showed in figure 3.3 ). If we have reached an **#elif** or **#else** directive, we find the nearest **#if** directive. (steps 17-24)

In the following algorithm, the main data structure used is stack since we need the immediate level of guarding conditions first. C1 is the stack which contains the guarding conditions associated with the first macro and C2 contains the guarding conditions of the second macro. Stack C is the final conditions that should be considered for the first macro. This algorithm is just showed to find the conditions for two macro definitions with the same name. However, it can easily be used to determine conditions if multiple definitions exist.

```
1: if C1.top is elif directive OR C1.top is else directive then
2:   C = FINDNEARESTIFDIRECTIVE(C1, C)
3: else
4:   if pop(C1).top != pop(C2).top then
5:     push(C, pop(C1))
6:     if C1 is not empty then
7:       temp = FINDNEARESTIFDIRECTIVE(C1, temp)
8:       if C2.top is elif directive OR C2.top is else directive then
9:         while C2 is not empty AND C2.top is not if directive do
10:          pop(C2)
11:        end while
12:      end if
13:    if temp is not empty AND temp.top == C2.top then
14:      push(C, temp)
15:    end if
```

```

16:     end if
17: else
18:     while C1 is not empty AND C2 is not empty AND C1.top == C2.top do
19:         push(C, pop(C1))
20:         C2.pop
21:     end while
22:     C = FINDNEARESTIFDIRECTIVE(C1, C)
23: end if
24: end if
25:
26: function FINDNEARESTIFDIRECTIVE(C1, C)
27:     while C1 is not empty AND C1.top is not if directive do
28:         push(C, pop(C1))
29:     end while
30:     push(C, pop(C1))
31:     return C
32: end function

```

### 3.3 Handling Conditional Directives

Garrido [9] claims in her thesis that "A refactoring in the presence of CPP conditionals is correct if and only if it is correct for all possible system configurations". In other words, each possible configuration should be analyzed and refactored together. One approach is to analyze and refactor each configuration separately, but this approach does not scale to large scale systems in the presence of exponential configurations like the Linux Kernel which provides more than 10,000 configurable features, which can be combined almost arbitrarily [34]. The approach we are using is to process all configurations simultaneously [12]. It only

constructs a single AST by analyzing the similar code between configurations only once. Liebig et al. [25] showed this approach works even in the case of the Linux kernel.

In order to consider all feasible configurations of a program, we consider every branch to be true without evaluating the conditions. Except these cases that we consider them as infeasible:

- Conditions that are always false such as "0" or "!1".

```
351 #ifndef _LIBC
352 #define _IO_stdin ((_IO_FILE*)(&_IO_2_1_stdin_))
353 #define _IO_stdout ((_IO_FILE*)(&_IO_2_1_stdout_))
354 #define _IO_stderr ((_IO_FILE*)(&_IO_2_1_stderr_))
355 #else
356 extern _IO_FILE *_IO_stdin attribute_hidden;
357 extern _IO_FILE *_IO_stdout attribute_hidden;
358 extern _IO_FILE *_IO_stderr attribute_hidden;
359 #endif
```

Figure 3.4: An example of incompatible conditions

- Conditions that do not comply with our requirements such as "defined\_--cplusplus" or "defined\_--GNU--" which equals to "defined\_--GNU--&&defined\_--cplusplus".
- There are some conditions that cannot be true at the same time. Garrido [9] called these conditions incompatible. Consider figure 3.4 as an example of incompatible conditions, extracted from /usr/include/libio.h, lines 351 to 359. If you first take the **#ifndef** branch and then the **#else** branch, `_IO_stdin` in the **#else** branch is considered as a macro and it is expanded which it unexpected.
- There are some conditions that considering them to be always true creates an infinite loop. They mostly happen in the case of including header files. Figure 3.5 shows segment of codes from /usr/include/features.h and /usr/include/i386-linux-gnu/sys/cdefs.h. As you can see, features.h includes cdefs.h in the case macro `_SYS_CDEFS_H` is not defined and cdefs.h also includes features.h. If we consider the guarding conditions of these two include files always true, there will be an infinite loop. Therefore, although

we do not evaluate the conditions, we need to keep track of them to determine if some conditions should not be taken.

When we reach a conditional directive during preprocessing, we are aware of the macros that has been defined so far. Considering the defined macros at that point and the directive condition, we create a logical expression and then determine if the expression could always be false. If so, we won't take the branch anymore. For the example in figure 3.4, the first time that we include features.h, macro `_SYS_CDEFS_H` has not been defined yet, so we include cdefs.h but next time since we had this macro defined in cdefs.h, we won't take the branch anymore and skip it.

<pre>#ifndef __ASSEMBLER__ # ifndef _SYS_CDEFS_H #  include &lt;sys/cdefs.h&gt; # endif</pre>	<pre>#define      _SYS_CDEFS_H      1  /* We are almost always included from features.h. */ #ifndef _FEATURES_H # include &lt;features.h&gt; #endif</pre>
(a) features.h	(b) cdefs.h

Figure 3.5: An example of conditions causing infinite loop

### 3.3.1 Calculating guarding conditions

Every point in the source code has a guarding condition. These guarding conditions are used in the static analysis where we need to annotate the control-flow graph edges. In this chapter, we only talk about calculating the conditions and discuss it in more details later in chapter 5.

Each part of a source code has a set of conditional directives. The guarding condition of that part is the conjunction of conditions that is calculated for each conditional directive. The condition for a conditional directive is calculated as follows:

- When conditional directive is **#if** constant-expression, the condition is the constant-expression.

- When conditional directive is either **#ifdef** identifier or **#ifndef** identifier, the conditions are `defined(identifier)` and `!defined(identifier)` respectively.
- When conditional directive is **#elif** constant-expression, the condition is the conjunction of constant-expression and negate of all previous constant-expressions and identifiers.
- When conditional directive is **#else**, the condition is the negate of all previous constant-expressions and identifiers.

We keep track of the conditions as an attribute for AST nodes, so that we could easily retrieve them later. Since we probably won't need to be aware of conditions for all the AST nodes, we decided to store them just for identifiers in order to save time and space. However, in the case that we need them for all the nodes later it could be easily set for them as well.

### 3.3.2 Running Example

Figure 3.6 shows a segment of code extracted from Zlib-1.2.5, `crc32.c`, lines 38 to 50. Table 3.1 shows the guarding condition for some lines of the program.

```

38: #   if (UINT_MAX == 0xffffffffUL)
39:     typedef unsigned int u4;
40: #   else
41:     if (ULONG_MAX == 0xffffffffUL)
42:       typedef unsigned long u4;
43:     else
44:       if (USHRT_MAX == 0xffffffffUL)
45:         typedef unsigned short u4;
46:       else
47:         undef BYFOUR      /* can't find a four-byte integer type! */
48:       endif
49:     endif
50: #   endif

```

Figure 3.6: A segment of code extracted from Zlib

Line#	Guarding Condition
39	<code>(UINT_MAX == 0xffffffffUL)</code>
42	<code>!(UINT_MAX == 0xffffffffUL)&amp;&amp;(ULONG_MAX == 0xffffffffUL)</code>
45	<code>!(UINT_MAX == 0xffffffffUL)&amp;&amp;!(ULONG_MAX == 0xffffffffUL)&amp;&amp;(USHRT_MAX == 0xffffffffUL)</code>
47	<code>!(UINT_MAX == 0xffffffffUL)&amp;&amp;!(ULONG_MAX == 0xffffffffUL)&amp;&amp;!(USHRT_MAX == 0xffffffffUL)</code>

Table 3.1: Guarding conditions

### 3.4 Conclusions

This chapter discussed about modifications made to OpenRefactory/C’s preprocessor in presence of multiple configurations. We talked about handling macros with multiple definitions, determining feasible conditional directives and calculating the guarding conditions. Now the result is a token stream that should be parsed to generate the AST. The modifications of the parser is currently under development and it is beyond the scope of this thesis. In the next chapter we will describe the modifications we made to generate AST.



## Chapter 4

### Program Representation that Supports Multiple Configuration Preprocessor

As mentioned in chapter 2, OpenRefractory/C uses rewritable ASTs to represent the source code. ASTs are constructed during parsing where nodes are created to represent the grammar productions. Before parsing a C program, the program has to be preprocessed first by the C preprocessor. Then, the resulting token stream is parsed. However, the grammar of the C parser does not include preprocessor constructs such as macros, **#include** directives, and conditional directives (like **#if**). Thus, we must extend the C grammar to include the preprocessor constructs. This way, the generated AST has some additional nodes that represent the information of preprocessor directives. In this chapter, we talk about the modifications we made to the grammar to generate an AST that reflects C preprocessor information.

#### 4.1 Introduction

The preprocessor constructs are not necessarily enclosed by statements, they might appear anywhere in the code while breaking the complete C constructs. Liebig et al. [24] gives a description that "Disciplined annotations are annotations on certain syntactic code structures, such as entire functions and statements, whereas we call annotations of individual tokens or brackets that do not align with underlying code structure undisciplined annotations". Figure 4.1 shows an example of a code structure that has undisciplined annotations. The code segment is extracted from GNU Coreutils version 8.21, printf.c. The syntax of labeled-statement in C is case constant-expression : statement but in this example the constant-expression is enclosed by an **#if** directive which breaks a complete C construct.

```

switch (*f)
{
#ifdef __GLIBC__ == 2 && 2 <= __GLIBC_MINOR__ || 3 <= __GLIBC__
    case 'I':
#endif
...

```

Figure 4.1: An example of breaking the complete C constructs

Making the grammar flexible enough to be able to parse all these cases will result in a large and ambiguous grammar. Garrido [9] describes a technique that is called completing the conditional to handle the incomplete C constructs. First, she tokenizes the input and recognizes incomplete preprocessor conditionals. Then, she moves them forward or backward as necessary to complete the conditionals. Although the process is reasonable, it is heuristic-based and it may become complicated in the presence of nested conditionals.

The OpenRefactory/C's approach is to extend the C grammar, based on Section 6.10 of the ISO C99 standard [18], in order to include productions of preprocessor directives in the grammar. However, the grammar only allows preprocessor directives in between complete C constructs including statement, declaration, and function definition.

Figure 4.2 shows two examples extracted from GNU Coreutils 8.21, `stty.c`. In figure 4.2 (a), the statement is surrounded by a conditional directive while on figure 4.2 (b), the conditional directive appears in between the statement. In the case that a conditional directive appears at an unexpected location, a new parsing algorithm [13] that is currently under development in OpenRefactory/C will be able to handle it.

<pre> #ifdef VEOF == VMIN     if ((mode-&gt;c_lflag &amp; ICANON) == 0         &amp;&amp; (STREQ (control_info[i].name, "eof")                STREQ (control_info[i].name, "eol")))         continue; #endif </pre>	<pre> #ifdef VEOF == VMIN     if ((mode-&gt;c_lflag &amp; ICANON) == 0) #endif         wrapf ("min = %lu; time = %lu;",             (unsigned long int) mode-&gt;c_cc[VMIN],             (unsigned long int) mode-&gt;c_cc[VTIME]); </pre>
(a)	(b)

Figure 4.2: Examples extracted from GNU Coreutils 8.21

## 4.2 Abstract Syntax Tree

OpenRefactory/C's internal program representation is a rewritable AST generated by Ludwig [29]. Figure 4.3 shows the simplified version of the extended grammar productions to include C preprocessor directives in the AST. The grammar follows the Ludwig's grammar notation, a variation of Backus-Naur Form (BNF), where nonterminals are denoted by names in  $\langle \text{angle-brackets} \rangle$ , while terminals are denoted by names in SMALL-CAPS. Different alternatives are separated by '|' character and '?' indicates the optional nonterminals. Directives such as **#define** are omitted from the tree since they do not necessarily fit into the syntactic structure.

```
 $\langle \text{directive-block-item-list} \rangle ::= \langle \text{directive-block-item} \rangle |$   
 $\langle \text{directive-block-item-list} \rangle \langle \text{directive-block-item} \rangle$   
  
 $\langle \text{directive-block-item} \rangle ::= \langle \text{declaration} \rangle$   
|  $\langle \text{statement} \rangle$   
|  $\langle \text{function-definition} \rangle$   
|  $\langle \text{directive} \rangle$   
  
 $\langle \text{macro-condition-identifier} \rangle ::= \text{IDENTIFIER}$   
  
 $\langle \text{if-directive} \rangle ::= \text{\#ifdef} \langle \text{macro-condition-identifier} \rangle \langle \text{directive-block-item-list} \rangle?$   
|  $\text{\#ifndef} \langle \text{macro-condition-identifier} \rangle \langle \text{directive-block-item-list} \rangle?$   
|  $\text{\#if} \langle \text{constant-expression} \rangle \text{NEWLINE} \langle \text{directive-block-item-list} \rangle?$   
  
 $\langle \text{else-directive} \rangle ::= \text{\#else} \langle \text{directive-block-item-list} \rangle?$   
|  $\text{\#elif} \langle \text{constant-expression} \rangle \text{NEWLINE} \langle \text{directive-block-item-list} \rangle? \langle \text{else-directive} \rangle?$   
  
 $\langle \text{directive} \rangle ::= \langle \text{if-directive} \rangle \langle \text{else-directive} \rangle? \text{\#endif}$ 
```

Figure 4.3: Grammar to include C preprocessor directives

### 4.2.1 Running Example

Figure 4.4 shows an example extracted from gmp-4.3.2, gen.c, lines 350 to 356. We used the above mentioned grammar to generate the AST in OpenRefactory/C. Figure 4.5 shows

the AST for the example in figure 4.4. We only show the part of AST that is related to preprocessor directives. Every node represents part of the code. Since some nodes were big, we only represent the node type and the method that is used to get that node, in the AST. The rest of the information is listed in table 4.1.

```

350: #if HAVE_STRTOL
351:     n = strtoul (argv[0], (char **) NULL, 10);
352: #elif HAVE_STRTOL
353:     n = (unsigned long int) strtol (argv[0], (char **) NULL, 10);
354: #else
355:     n = (unsigned long int) atoi (argv[0]);
356: #endif

```

Figure 4.4: An example extracted from gmp-4.3.2

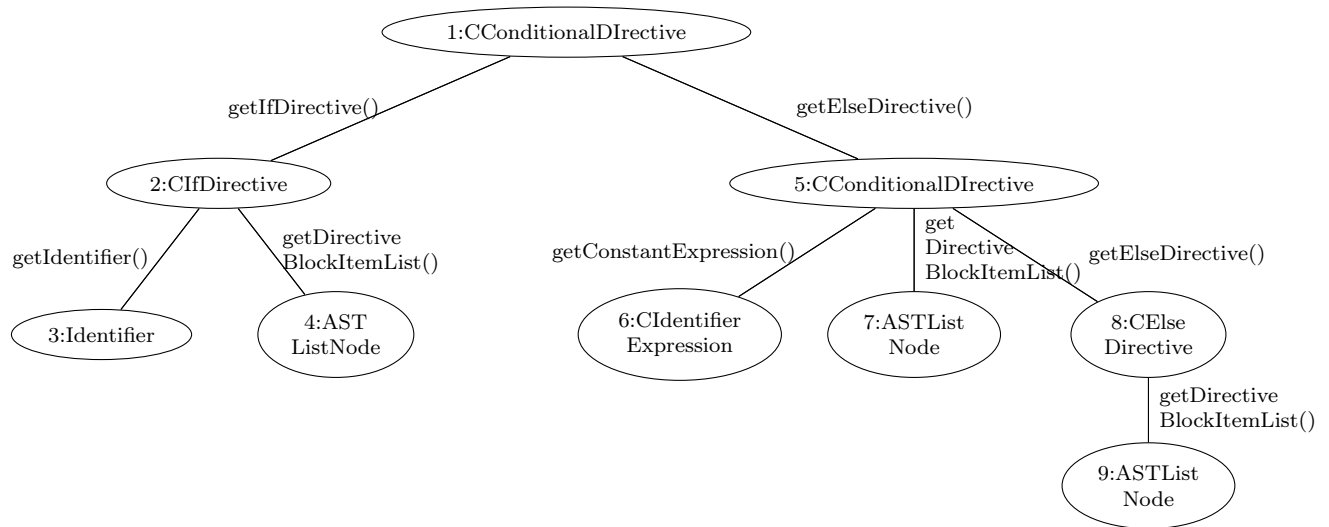


Figure 4.5: Partial AST generated for the example in figure 4.4

Node#	Node Type	Node
1	CConditionalDirective	<code>#if HAVE_STRTOUL n = strtoul (argv[0], (char **) NULL, 10); #elif HAVE_STRTOL n = (unsigned long int) strtol (argv[0], (char **) NULL, 10); #else n = (unsigned long int) atoi (argv[0]); #endif</code>
2	CIfDirective	<code>#if HAVE_STRTOUL n = strtoul (argv[0], (char **) NULL, 10);</code>
3	Identifier	<code>HAVE_STRTOUL</code>
4	ASTListNode	<code>n = strtoul (argv[0], (char **) NULL, 10);</code>
5	CConditionalDirective	<code>#elif HAVE_STRTOL n = (unsigned long int) strtol (argv[0], (char **) NULL, 10); #else n = (unsigned long int) atoi (argv[0]); #endif</code>
6	CIdentifierExpression	<code>HAVE_STRTOL</code>
7	ASTListNode	<code>n = (unsigned long int) strtol (argv[0], (char **) NULL, 10);</code>
8	CElseDirective	<code>#else n = (unsigned long int) atoi (argv[0]);</code>
9	ASTListNode	<code>n = (unsigned long int) atoi (argv[0]);</code>

Table 4.1: Nodes of AST

### 4.3 Conclusion

This chapter described the modifications we made in AST in order to include the C preprocessor directives. We made these changes in AST since we want to be aware of multiple configuration information during static analysis and refactorings. Now that we modified AST with additional preprocessor nodes, in the next chapter we could look into extending the current static analysis in OpenRefactory/C to be preprocessor aware.

## Chapter 5

### Multiple configuration aware static analysis

In the previous chapters, we discussed about integration of multiple configuration support to OpenRefactory/C. We changed the syntactic structure and also the AST. Correspondingly, the program analysis methods need to be modified. In chapter 1, we discussed about the impact of multiple configuration on static analysis and refactorings. In this chapter, we will talk about supporting static analysis for multiple configurations.

#### 5.1 Control Flow Analysis

It is possible to derive control flow information directly from a program's AST. Control flow information abstracts the actual behavior of the program which is explicitly represented by control flow graph (CFG). CFG includes all possible execution paths of a program and it is also used for most data flow analysis. Each node of a program has a set of control predecessors from which control may flow and a set of control successors to which control may flow [27]. To create a CFG for a program, we need to compute successors and predecessors of every node of the program. In this section, we discuss about computing successors and predecessors in presence of preprocessor directives and multiple configurations.

##### 5.1.1 Introduction

In chapter 4 we modified the AST to represent preprocessor information by adding additional nodes. Now, we need to determine how to represent each additional node in the CFG and how to calculate the successors and predecessors. Two additional nodes that we are considering when constructing CFG is `CConditionalDirective` and `CMacroConditionIdentifier`.

`CConditionalDirective` node represents every conditional directive in the AST. It might have two children, `CIfDirective` and `CElseDirective`. Note that as shown in the grammar in chapter 4, `#elif` directive is treated as else if structure in regular conditional statement to make the analysis easier.

`CMacroConditionIdentifier` as shown in the grammar, is an identifier that is the condition in `#ifdef` or `#ifndef` directives. Although identifier itself is not among flow nodes, we created `CMacroConditionIdentifier` to be able to treat `#ifdef` and `#ifndef` directives the same as `#if` and `#elif` directives. This way the analysis are simpler and more consistent. From this point, when we use `#if` directive it represents `#ifdef` and `#ifndef` directives as well unless otherwise mentioned.

Generally, we deal with conditional directives like regular conditional statements. In other words, when constructing a CFG, we express configurations with conditional branches. As we mentioned in chapter 3, we store the presence conditions of every AST node as an attribute. Thus, we can annotate the branch edges of CFG as either true or false. However, there is one difference when dealing with conditional directives. In regular conditional statements, a single item which is a single AST node comes after `if(expression)`, while in a conditional directive a block of items comes after `#if constant-expression` which contains multiple AST nodes. In the next section we look into an example of constructing a CFG for a program that contains preprocessor directives.

### 5.1.2 Running Example

Figure 5.1 shows a code snippet extracted from GNU Coreutils 8.21, `timeout.c`, lines 349 to 363 and figure 5.2 shows the corresponding CFG where nodes are line numbers, T stands for true and F stands for false.

As shown in the CFG, `#if` directive on line 349 might have two successors. One is on line 350 when we take the true branch, and the other one is `#elif` directive on line 353. On

line 353, there is another conditional branch. If we take the true branch, its successor is on line 356, if not, its successor is on line 360 in the **#else** directive.

```
349: #if HAVE_PRCTL && defined PR_SET_DUMPABLE
350:   if (prctl (PR_SET_DUMPABLE, 0) == 0)
351:     return true;
352:
353: #elif HAVE_SETRLIMIT && defined RLIMIT_CORE
354:   /* Note this doesn't disable processing by a filter in
355:    /proc/sys/kernel/core_pattern on Linux. */
356:   if (setrlimit (RLIMIT_CORE, &(struct rlimit) {0,0}) == 0)
357:     return true;
358:
359: #else
360:   return false;
361: #endif
362:
363:   error (0, errno, _("warning: disabling core dumps failed"));
```

Figure 5.1: A code snippet extracted from GNU Coreutils 8.21

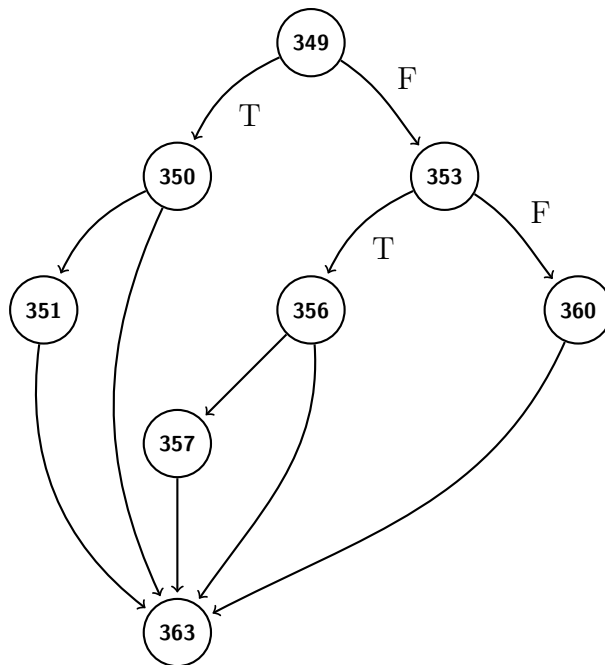


Figure 5.2: CFG for figure 5.1

In the following sections we introduce some concepts that we are using during calculation of successors and predecessors.



### 5.1.3 Control Entry and Control Exit

In calculating the successors and predecessors of a flow node, we use the concepts of control entry and control exit [27]. Control enters the CFG through control entry and exits through control exit. For instance, control entry of a selection statement like **if(E) S** is **E**. Control exits might be more than one, true and false exits. Using logical operators such as **||** and **&&** in control expressions make multiple exits to happen. For instance, for expression **A && B**, there is a true exit of **B** and false exits of **A** and **B**. Control exits could also be either jump or nonjump exit. Jump exits include return, break, continue, and goto exits while nonjump exits are only regular control exits like **E** in **if(E) S** which is also a false exit.

For conditional directives, we consider control entry to be the control entry of constant expression in **#if** or **#elif** directives, or **CMacroConditionIdentifier** in **#ifdef** or **#ifndef** directives. For calculating control exits, first we use the strategy in figure 5.3 to calculate nonjump exits, then we calculate jump exits by just finding return, break, continue, and goto exits. Now that we have information of control entry and exit in CFG, in the next section we look into strategies of computing successors and predecessors in presence of preprocessor directives.

- 1: - Get the corresponding directive block item list
- 2: - Add control exits of last block item to list of exits
- 3: **if** directive is **#if then**
- 4:     - Add exits of constant expression to list of exits
- 5: **else**
- 6:     - Add **CMacroConditionIdentifier** to list of exits
- 7: **end if**
- 8: - repeat the same procedure for **#elif** and **#else** directives
- 9: - Return list of exits

Figure 5.3: Strategy to calculate non jump exits of directive condition node

### 5.1.4 Calculating Successors and Predecessors

In order to calculate successors and predecessors in presence of preprocessor directives, we add two methods to their current implementation in OpenRefactory/C. One is to find

successors/predecessors of directive condition nodes and the other one is calculating successors/predecessors for every item that belongs to a conditional directive block item list.

Figure 5.4 and 5.5 show the strategy used to find successors of directive condition node that is either `constantExpression` or `MacroConditionIdentifier` in `#if` or `#elif` directives.

- 1: **if** the condition is for `#if` directive **then**
- 2:     - Return the predecessor of `#if` directive as part of another block item list
- 3: **else**
- 4:     - find the the previous `#if` or `#elif` branch
- 5:     - Return false exits of that branch
- 6: **end if**

Figure 5.4: Strategy to calculate predecessors of directive condition node

- 1: **if** the directive block item list is not empty **then**
- 2:     - Add control entry flow node of first item in the list to successors
- 3: **end if**
- 4: **if** the directive has either `#elif` or `#else` branches **then**
- 5:     **if** the directive has `#elif` branch **then**
- 6:         - Add control entry flow node of `#elif` constant expression to successors
- 7:     **else**
- 8:         **if** the `#else` directive block item list is not empty **then**
- 9:             - Add control entry flow node of first item in the list to successors
- 10:         **else**
- 11:             - Find the nearest `#if` directive
- 12:             - Add successors of `#if` directive to the list of successors
- 13:         **end if**
- 14:     **end if**
- 15: **else**
- 16:     - Find the nearest `#if` directive
- 17:     - Add successors of `#if` directive to the list of successors
- 18: **end if**
- 19: - Return list of successors

Figure 5.5: Strategy to calculate successors of directive condition node

Figure 5.6 and 5.7 show the strategy used to compute successors and predecessors for every item that belongs to a conditional directive block item list (conditional directive could be either `#if`, `#elif`, or `#else` directives).

```

1: - Get the corresponding directive block item list
2: if statement is not the last item in the list then
3:   - Return control entry flow node of the next item in the list
4: else
5:   - Find the nearest #if directive
6:   - Return the successor of condition node
7: end if

```

Figure 5.6: Strategy to calculate successors for every item of a conditional directive block item list

```

1: - Get the corresponding directive block item list
2: if statement is not the first item in the list then
3:   - Return none jump control exit flow nodes of the previous item in the list
4: else
5:   if statement is in #if part of block item list then
6:     - Return true exits of #if directive condition
7:   else
8:     - Return false exits of #if directive condition
9:   end if
10: end if

```

Figure 5.7: Strategy to calculate predecessors for every item of a conditional directive block item list

## 5.2 Data Flow Analysis

As we mentioned in chapter 2, OpenReactory/C stores intraprocedural reaching definitions and definition-use relationships in the AST. These analysis are classic data flow analysis that are used to compute the definitions and uses for every point of a program. Since successor and predecessor information is needed to calculate data flow information, CFG is used for data flow analysis. For instance, reaching definition uses two equations, IN and OUT, to determine if a definition reaches a specific point of a program. Let B be a point of program and P its predecessors,  $IN(B) = \bigcup_{p \in P} OUT(p)$  and  $OUT(B) = Gen(B) \cup (IN(B) - Kill(B))$ . Gen(B) are definitions within B that reach the end of B and Kill(B) are definitions that never reach the end of B due to redefinitions of variables in B. As we changed the CFG to be configuration aware, now the result of data flow analysis is also affected by those modifications.

### 5.2.1 Running Example

Figure 5.8 shows a segment of code extracted from gmp-4.3.2, memory.c, lines 39 to 44. Table 5.1 shows reaching definition information of the code. The format of entry in the table is (variable name, line number, condition). As it is shown in the table, the definitions of req\_size and size is only reached if DEBUG is selected.

```

39: void *ret;
40: #ifdef DEBUG
41:     size_t req_size = size;
42:     size += 2 * BYTES_PER_MP_LIMB;
43: #endif
44: ret = malloc (size);

```

Figure 5.8: A code snippet extracted from gmp-4.3.2

Line#	Gen	Kill	In	Out
39	(ret,39,)			(ret,39,)
40			(ret,39,)	(ret,39,)
41	(req_size,41,DEBUG)		(ret,39,)	(ret,39,) (req_size,41,DEBUG)
42	(size,42,DEBUG)		(ret,39,) (req_size,41,DEBUG)	(ret,39,) (req_size,41,DEBUG) (size,42,DEBUG)
44	(ret,44,)	(ret,39,)	(ret,39,) (req_size,41,DEBUG) (size,42,DEBUG)	(ret,44,) (req_size,41,DEBUG) (size,42,DEBUG)

Table 5.1: Reaching definition information for code in figure 5.8

### 5.3 Preprocessor Dependence Preservation Analysis

The baseline analyses mentioned in the previous sections are not sufficient to ensure the correctness of certain kinds of refactorings—notably, those that reorder, remove, or copy code within a file. Of the five most commonly used refactorings [28], three of them—Extract Method, Extract Local, and Move—perform code movement and, therefore, must respect preprocessor dependences. Figure 5.9 shows an example extracted from GNU Coreutils, who.c. Applying Extract Function refactoring on line 352 will move it above line 341 where

the macro `DEV_DIR_WITH_TRAILING_SLASH` is undefined. Compiling the code after refactoring will result in a compiler error.

```
341: #define DEV_DIR_WITH_TRAILING_SLASH "/dev/"
342: #define DEV_DIR_LEN (sizeof (DEV_DIR_WITH_TRAILING_SLASH) - 1)
    ...
351: if ( ! IS_ABSOLUTE_FILE_NAME (utmp_ent->ut_line))
352:     p = stpcpy (p, DEV_DIR_WITH_TRAILING_SLASH);
```

Figure 5.9: An example of moving macro to undefined region

Another analysis, preprocessor dependence preservation analysis, has also been implemented and integrated with refactorings in `OpenRefactory/C` to ensure macro correctness in addition to baseline correctness. It uses a program representation called preprocessor dependence graph, or PPDG, that models the relationships between macro definitions and invocations, as well as the relationships between conditional compilation directives and the code affected by them. A refactoring tool can simply build a PPDG prior to refactoring, then build another PPDG after refactoring, and compare the two. The differences between the two can be used to determine whether the transformation will cause any changes in preprocessor behavior.

We integrated the preprocessor dependence preservation analysis with three refactorings, Move External Declaration, Move Statement, and Extract Function implemented in `OpenRefactory/C`. We automatically applied these refactorings on a large number of targets in GNU Coreutils version 8.21 and identified that on average, 10% of the refactoring test cases were blocked by the preprocessor dependence preservation analysis and errors would be introduced if the preprocessor dependence were not considered.

Therefore, another analyses in addition to baseline analyses might also be needed to ensure that refactorings—particularly those that move, delete, or copy code—operate correctly on code containing lexical macros and conditional compilation.

## 5.4 Conclusion

In this chapter, we looked into modifications that we made to control flow and data flow analyses in order to include preprocessor information. We also introduced another type of analysis that has been implemented in `OpenRefactory/C` to ensure macro correctness. In the next chapter, we evaluate our multiple configuration aware analysis using Extract Function refactoring.

## Chapter 6

### Evaluation

To evaluate our multiple configuration aware analysis, we used Extract Function refactoring implemented in OpenRefactory/C. This refactoring moves a sequence of statements into a new function, replacing the original statements with a call to that function. We selected this refactoring since it is particularly important to refactoring tools. Martin Fowler [8] identified this refactoring as a tipping point for refactoring tools, claiming that if a refactoring tool could implement Extract Function refactoring well, it could implement many other refactorings.

#### 6.1 General Testing Approach

The general testing approach [14] that we use in OpenRefactory/C consists of three main steps: (1) finding all the program elements where a given refactoring can be applied, running the refactoring on each of the elements, and recording the failures with associated message; (2) splitting the failures into clusters based on the recorded messages; and (3) manually analyzing the clusters to identify bugs. The first step is entirely automated; step 2 is semi-automated and step 3 is manual.

In order to collect failures, three inputs are needed: the refactoring under test, a C project containing the program on which the refactoring will be applied, and a threshold that determines the maximum number of times to apply the refactoring on the project. Based on the inputs, a set of refactoring tasks that a given refactoring can be applied are calculated. Then, the properties for each refactoring task are configured such as providing a new name for a new function. Before applying the refactoring, it is checked whether proceeding with the refactoring could change the program behavior. If not, the refactoring is performed. The

result after applying the refactoring is a refactored project which is compiled. If there are any compiler errors, the failures with all the error messages are recorded.

The second step clusters the failures where the goal is to reduce the number of failures that should be inspected to detect bugs. Abstract messages that are regular expressions are computed from the concrete messages that were recorded with the failures. After the messages are abstracted, the clustering splits the failures into groups that have the same refactoring name and the same abstract messages.

In the third step, the clusters are manually inspected to find bugs.

We used this approach to test the C refactorings in Eclipse CDT on three real software projects, GMP [15], libpng [22], and zlib [35]. Table 6.1 shows the results where we found 53 bugs and reported them to Eclipse Bugzilla.

Refactoring	# Refact. Tasks	# Failures	Compiler Error					Exception				
			# Failures	# Clusters	Min Cluster	Max Cluster	# Bugs	# Failures	# Clusters	Min Cluster	Max Cluster	# Bugs
Rename	10652	598	252	3	1	173	2	346	2	3	343	0
Extract Method	23330	1782	1453	34	1	435	21	329	8	1	56	3
Extract Local	12026	1323	754	11	3	412	7	569	4	6	263	3
Extract Constant	23796	860	142	3	1	84	2	718	3	125	426	2
Toggle Function	2068	863	9	2	1	8	1	854	5	23	409	2
$\Sigma$	71872	5426	2610	53			<b>33</b>	2816	22			<b>10</b>
7.5%	7.5%											

Table 6.1: Failure and cluster statistics for C projects

## 6.2 A Problem

As mentioned in chapter 2, when we consider all possible macro configurations, it is possible that conditional directives appear in the middle of a complete C constructs. The current parser in OpenRefactory/C is not able to handle those cases. At the first place,



we planned to manually change every C file in GNU Coreutils which is our testing target. However, for some C constructs the number of macros appearing in one construct is so high that is actually impossible to manually change them for many of the C files. Figure 6.1 shows one of the examples in `chmod.c` where the definition for macro `CHMOD_MODE_BITS` consists of multiple macros in which some of them have multiple expansions and some others are again expanded to macros with multiple expansions. For instance, figure 6.2 shows the definition of `S_IRWXU` where it consists of three other macros. Subfigures (b), (c), and (d) show multiple definitions of every one of those macros under different guarding conditions. Again, for every macro one of the definitions is a macro which may be expanded to one or more macros.

```

return ((old_mode ^ new_mode) & CHMOD_MODE_BITS) != 0;
(a) cp-hash.c

```

```

#define CHMOD_MODE_BITS \
    (S_ISUID | S_ISGID | S_ISVTX
 | S_IRWXU | S_IRWXG | S_IRWXO)
(b) CHMOD_MODE_BITS
definitions

```

Figure 6.1: An example extracted from `chmod.c`

```

#if !S_IRWXU
# define S_IRWXU \
(S_IRUSR | S_IWUSR
 | S_IXUSR)
#endif
(a) S_IRWXU def-
inition

```

```

#if !S_IRUSR && S_IREAD
# define S_IRUSR S_IREAD
#endif
#if !S_IRUSR
# define S_IRUSR 00400
#endif
(b) S_IRUSR definition

```

```

#if !S_IWUSR && S_IWRITE
# define S_IWUSR S_IWRITE
#endif
#if !S_IWUSR
# define S_IWUSR 00200
#endif
(c) S_IWUSR definition

```

```

#if !S_IXUSR && S_IEXEC
# define S_IXUSR S_IEXEC
#endif
#if !S_IXUSR
# define S_IXUSR 00100
#endif
(d) S_IXUSR definition

```

Figure 6.2: Expansion of macro `S_IRWXU`

Since it seems very difficult to manually change 108 C files of our testing target, GNU Coreutils, we decided to write unit tests in Java using JUnit tests. In the rest of this chapter, we discuss about how we designed and evaluated our test cases.

## 6.3 Test cases

### 6.3.1 Designing Test cases

Extract Function refactoring implementation in OpenRefactory/C composed of ten micro refactorings where some of them are directly using either control flow or data flow analysis. In the remaining of this chapter we look into each of the micro refactorings and determine how each step has been implemented, which analysis it is used and how they are affected by the changes we made. We designed our test cases for each micro refactoring based on these changes.

### 6.3.2 Extract Function Refactoring

Extract Function refactoring composes of ten micro refactoring as follows:

- *Replace Statement Sequence with Compound Statement (AddBrace)* : This micro refactoring puts curly braces around the statements to be extracted. However, this is not always legal. We have to validate the statement selection. One of the criteria that we consider is that the selection must be a single entry, single exit region in the control flow graph. Therefore, this micro refactoring should be tested specifically for the changes we made on the control flow graph.
- *Move Local Variable Declaration into Compound Statement (MoveLocal)* : This step finds every local variable used in the compound statement and checks if the variable has a declaration out of the compound statement while it is only used inside the compound statement. If this is true for any variable, it moves the variable declaration inside the compound statement. To determine the uses of each definition, this micro refactoring uses definition-use chains from data flow analysis. Thus, we have to make sure that we consider all the uses under all the configurations when calculating every variable uses.

- *Move Initial Assignment into Declaration (MoveInitial)* : For every declaration that is moved inside the compound statement in the previous step, this micro refactoring determines if there is any initial assignment that could be moved to the declaration. To ensure whether this movement is legal or not, we use dependence analysis where it uses the control flow and data flow information.
- *Introduce Pointer (IntroPointer)* : For each local variable used in the compound statement, if it is aliased or there is any uses of it out of the compound statement, we introduce a pointer to that variable and replace every uses of that variable inside the compound statement with the introduced pointer. This steps again uses the information from definition-use chains in order to determine if any variable is used out of the selected area.
- *Remove Superfluous Parentheses* : This step just removes the superfluous parentheses introduced in the previous step and it is not affected.
- *Add Empty Function* : This micro refactoring creates an empty function with appropriate arguments. This step does not use any information from either control or data flow analysis. However, when we determine the appropriate arguments we have to ensure that we consider all the feasible configurations.
- *Move Compound Statement into Function Definition (MoveCompound)* : This micro refactoring moves the statements inside the compound statement into the empty function added in the previous step. This step uses preprocessor dependence preservation analysis mentioned end of the previous chapter. We previously tested this analysis on real C code.
- *Replace Pointer Parameter with Function Return (ReplacePointer)* : If we only introduced one pointer-typed parameter, which is not a pointer to a struct, union, or an array and it is always used indirectly (as either pointer or an address, not just the

variable itself) we replace the pointer with a function return. This step also has not been affected by the changes we made.

- *Rename* : If in the previous step we replaced any pointer parameter with function return, we rename the variable name to its original name, before introducing the pointer. This refactoring, in this context, is not affected by the changes.
- *Remove Unused Parameter (RemoveParameter)* : If any function argument is dominated by an assignment inside the function body, that parameter is no longer needed. This is also determined by using dependence analysis.

We wrote on average 10 new test cases (excluding their previous tests) for every micro refactoring that our changes have any impact on it. Then, we ran the Extract Function refactoring for every test case as well to make sure we are not breaking anything.

### 6.3.3 Evaluating Test cases

In order to evaluate our test cases, we used EclEmma [6], version 2.3.1, a free Java code coverage tool. We ran this tool for every micro refactoring that is affected. EclEmma uses a set of different counters including instructions, branches, lines, methods, and complexity to calculate coverage metrics.

- *Instructions* : It provides information about the amount of code that has been executed.
- *Branches* : It counts the total number of if and switch statements in a method and determines the number of executed or missed branches.
- *Lines* : For all class files that have been compiled with debug information, coverage information for individual lines can be calculated.
- *Methods* : Each non-abstract method contains at least one instruction. A method is considered as executed when at least one instruction has been executed.

- *Cyclomatic complexity* : It is the minimum number of paths that can, in (linear) combination, generate all possible paths through a method. Thus the complexity value can serve as an indication for the number of unit test cases to fully cover a certain piece of software. It also calculates cyclomatic complexity [26] for each non-abstract method and summarizes complexity for classes, packages and groups.

Figure 6.3 shows the generated result for MoveLocalVariable refactoring using EclEmma.






Counter	Coverage	Covered	Missed	Total
Instructions	 89.4 %	1,019	121	1,140
Branches	 73.0 %	92	34	126
Lines	 92.3 %	179	15	194
Methods	 92.3 %	12	1	13
Complexity	 53.9 %	41	35	76

Figure 6.3: Test coverage results for MoveLocalVariable micro refactoring using EclEmma

We generated the results for all other micro refactorings. The numbers are shown as percentages. Table 6.2 shows the results where the average instruction coverage for six micro refactorings is 85.95%, the branch, line, methods, and complexity coverages are 77.8%, 85.2%, 88.6%, and 47.8% respectively.

Micro Refactoring Name	Instruction	Branch	Line	Methods	Complexity
AddBrace	87.2	76.1	78.4	92.3	54.7
MoveLocal	89.4	73	92.3	92.3	53.9
MoveInitial	78.5	79.3	72.3	94.1	34.1
IntroPointer	78.8	73.2	78.2	85	46.5
MoveCompound	88.4	77	94.8	93.1	46
RemoveParameter	93.4	70.6	95.2	75	52
Average	85.95	77.8	85.2	88.6	47.8

Table 6.2: Test coverage results for Extract Function micro refactorings using EclEmma

### 6.3.4 Discussion

As mentioned earlier Extract Function refactoring is implemented by integrating multiple micro refactorings. The integration requires using encapsulation in Java where several getters and setters methods are needed so that micro refactorings can communicate with each other. These methods are not tested in individual micro refactorings, they are tested as part of Extract Function itself. Percentages are affected by these methods and become lower.

Cyclomatic complexity is highly dependent on number of branches and decision points. In the case that two different branches are slightly different, we did not write separate test cases for each branch. Therefore, the cyclomatic complexity percentage is lower. Figure 6.4 shows such a code from Add Brace micro refactoring.

```
if (parent instanceof CIfStatement) {
    statement = (ICStatement)firstBlockItem;
} else if (parent instanceof CSwitchStatement) {
    CSwitchStatement switchStatement = firstBlockItem.findNearestAncestor(CSwitchStatement.class);
    statement = switchStatement.getStatement();
} else if (parent instanceof CWhileStatement) {
    CWhileStatement whileStatement = firstBlockItem.findNearestAncestor(CWhileStatement.class);
    statement = whileStatement.getStatement();
} else if (parent instanceof CDoStatement) {
    CDoStatement doStatement = firstBlockItem.findNearestAncestor(CDoStatement.class);
    statement = doStatement.getStatement();
} else if (parent instanceof CForStatement) {
    CForStatement forStatement = firstBlockItem.findNearestAncestor(CForStatement.class);
    statement = forStatement.getStatement();
} else if (parent instanceof CCaseStatement) {
    CCaseStatement caseStatement = firstBlockItem.findNearestAncestor(CCaseStatement.class);
    statement = caseStatement.getStatement();
} else if (parent instanceof CLabeledStatement) {
    CLabeledStatement labelStatement = firstBlockItem.findNearestAncestor(CLabeledStatement.class);
    statement = labelStatement.getStatement();
} else if (parent instanceof CDefaultStatement) {
    CDefaultStatement defaultStatement = firstBlockItem.findNearestAncestor(CDefaultStatement.class);
    statement = defaultStatement.getStatement();
}
```

Figure 6.4: Part of Add Brace micro refactoring code

In order to increase coverage percentage we could write more test cases to cover all the branches and statements. However, functional testing is mostly used to test refactorings rather than just unit testing. The part of code mentioned in the above example is assumed to

return correct result when writing test cases. Therefore, increasing the coverage percentage may not have a substantial impact on refactoring correctness.

## Chapter 7

### Related Work

The importance of preprocessor concerns in program analyses (and transformations) is confirmed in an empirical study by Ernst et al. [7]. Several authors have attempted to understand how commonly preprocessors are used in C code. These studies rely on heuristics to approximate the actual usage [1, 7, 10, 23, 30]. The problems of using **#ifdef** directives have also been noted [33].

One of the earliest papers on handling the C preprocessor in a software maintenance tools is due to Platoff et al. [31]. Badros and Notkin [3] describe an infrastructure for handling the C preprocessor in static analysis tools, which invokes user-defined callbacks when interesting preprocessing events occur. Their infrastructure is based on GCC's preprocessor, so it only handles a single configuration, although they provide access to unprocessed **#ifdef** regions as unstructured text, and they provide access to the parser stack in order to aid analysis of this text.

The first comprehensive work on handling the C preprocessor in a refactoring tool is due to Garrido [9, 10, 11]; this was discussed previously. Notably, Garrido's work identifies rules for refactoring code containing macros and file inclusions, it recognizes the need to guard symbol table entries with the preprocessor configurations under which they are valid, and it addresses the difficulties of refactoring codes where conditional compilation directives do not enclose complete syntactic constructs.

In recent years, researchers have proposed and evaluated variability-aware analysis in different domains where they do not rely on heuristics and do not impose any restrictions on the location of preprocessor constructs.



Product-line-aware type checkers [21, 2, 20, 19] are able to determine if all potential variants of a product line that is written in C and implements variability with **#ifdef** directives of the C preprocessor are well-typed.

Brabrand et al. [5] demonstrated the benefits of using feature-sensitive data flow analysis against the brute force approach that generates all the variants of a software product line. They proposed several algorithms to adapt an existing standard intraprocedural data flow analysis into a feature-sensitive data flow analysis by extending the representation of control flow graphs with an inclusion condition associated with each node, describing under which configurations the node is present in the graph. However, their approach is limited as it gets Java programs as the input and by using a tool to annotate them with conditional directives like **#if**. Another limitation of this approach is that they make the conditional directive annotation disciplined [24] which is not true for real C programs.

Bodden et al. [4] proposed an approach to automatically convert traditional static analysis to software product line analysis. However, their approach is only limited to software product lines that are expressed with CIDE, an extension of the Eclipse IDE and they are not currently able to handle C preprocessor features (**#ifdef** constructs). Moreover, their tool is limited to analyses expressed in the IFDS [32] framework.

As mentioned earlier, the most similar work to ours is by Liebig et al. [25]. They implemented variability-aware type checking and liveness analysis to compare the performance of these variability-aware analysis with the performance of corresponding sampling heuristics (single configuration, pair-wise, and code coverage). They found that the performance of variability-aware analysis scales to large scale systems such as the Linux kernel. However, they did not apply their analysis in any application areas other than empirical studies. Refactoring tools are practical application of configuration-aware analysis which present challenges specific to this context. We used OpenRefactory/C as a practical infrastructure to implement configuration-aware analysis which is a real necessity to perform certain refactorings correctly.

## Chapter 8

### Conclusions and Future Works

In this thesis, we discussed handling multiple configurations in C programs in order to build more accurate program transformations for refactoring. The C language libraries are highly configurable and use C preprocessor directives heavily. Program analysis and transformations have to consider all possible macro configurations; otherwise they will be incorrect.

We modified the C preprocessor in OpenRefactory/C to handle macros with multiple definitions and conditional directives in presence of multiple configurations. We also included the preprocessor directives in our program representation so that we could extend the static analysis to account for multiple configurations. We modified the CFG to express configurations with conditional branches and annotating the branch edges of CFG as either true or false.

For evaluation, we modified Extract Function refactoring and tested it against our changes using JUnit tests. In future, we will test our code on real C programs including GNU Coreutils. Moreover, a new set of refactorings that needs preprocessor directive support could be developed in OpenRefactory/C. For instance, a Rename Macro refactoring that renames a macro or an Extract Macro refactoring that extracts parts of code to a macro. More information will be available on <https://sites.google.com/site/masterthesisresults/>.

## Bibliography

- [1] B. Adams, W. De Meuter, H. Tromp, and A. Hassan. Can we refactor conditional compilation into aspects? In *AOSD '09*, pages 243–254, New York, NY, USA, 2009. ACM.
- [2] S. Apel, C. Kästner, A. Grolinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engg.*, 17(3):251–300, Sept. 2010.
- [3] G. Badros and D. Notkin. A framework for preprocessor-aware c source code analyses. *Softw. Pract. Exper.*, 30(8):907–924, July 2000.
- [4] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. Spllift: Statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 355–364, New York, NY, USA, 2013. ACM.
- [5] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 13–24, New York, NY, USA, 2012. ACM.
- [6] EclEmma home page. <http://www.eclEmma.org/>.
- [7] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Engg.*, 28(12):1146–1170, Dec. 2002.
- [8] M. Fowler. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, Jun 1999.
- [9] A. Garrido. *Program refactoring in the presence of preprocessor directives*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005.
- [10] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *ICSM '05*, pages 379–388, 2005.
- [11] A. Garrido and R. Johnson. Embracing the C preprocessor during refactoring. *J. Softw. Evol. and Proc.*, June. 2013.
- [12] A. Garrido and R. E. Johnson. Refactoring c with conditional compilation. In *ASE*, pages 323–326. IEEE Computer Society, 2003.

- [13] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. PLDI '12, pages 323–334, New York, NY, 2012. ACM.
- [14] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In *ECOOP 2013*, volume 7920 of *Lecture Notes in Computer Science*, pages 629–653. Springer Berlin Heidelberg, 2013.
- [15] GMP home page. <http://gmplib.org>.
- [16] M. Hafiz. *Security On Demand*. PhD thesis, University of Illinois Urbana-Champaign, 2010.
- [17] M. Hafiz, J. Overbey, F. Behrang, and J. Hall. OpenRefactory/C: An infrastructure for building correct and complex C transformations. In *WRT '13*, 2013.
- [18] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. Dec 1999.
- [19] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14:1–14:39, July 2012.
- [20] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *OOPSLA '12*, pages 773–792, New York, NY, 2012. ACM.
- [21] A. Kenner, C. Kästner, S. Haase, and T. Leich. Typechef: Toward type checking #ifdef variability in c. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 25–32, New York, NY, USA, 2010. ACM.
- [22] libpng home page. <http://www.libpng.org/pub/png/libpng.html>.
- [23] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. ICSE '10, pages 105–114, New York, NY, USA, 2010. ACM.
- [24] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. AOSD '11, 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 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 81–91, New York, NY, USA, 2013. ACM.
- [26] T. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.
- [27] J. D. Morgenthaler. Static analysis for a software transformation tool. Technical report, 1997.
- [28] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE '09*, 2009.

- [29] J. Overbey and R. Johnson. Generating rewritable abstract syntax trees. In *Software Language Engineering: First International Conference, SLE 2008. Revised Selected Papers*. Springer-Verlag, 2009.
- [30] Y. Padioleau. Parsing C/C++ code without pre-processing. In *ETAPS '09, CC '09*, pages 109–125, Berlin, Heidelberg, 2009. Springer-Verlag.
- [31] M. Platoff, M. Wagner, and J. Camaratta. An integrated program representation and toolkit for the maintenance of c programs. In *ICSM '91*. IEEE, 1991.
- [32] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.
- [33] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer 1992 Technical Conference*, pages 185–198, Berkeley, CA, USA, June 1992. Usenix Association.
- [34] 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 Sixth Conference on Computer Systems*, EuroSys '11, pages 47–60, New York, NY, USA, 2011. ACM.
- [35] zlib home page. <http://www.zlib.net>.