Analyzing Multiple Configurations of a C Program

Alejandra Garrido and Ralph Johnson
University of Illinois at Urbana-Champaign
garrido@cs.uiuc.edu, johnson@cs.uiuc.edu

Abstract

Preprocessor conditionals are heavily used in C programs since they allow the source code to be configured for different platforms or capabilities. However, preprocessor conditionals, as well as other preprocessor directives, are not part of the C language. They need to be evaluated and removed, and so a single configuration selected, before parsing can take place. Most analysis and program understanding tools run on this preprocessed version of the code so their results are based on a single configuration.

This paper describes the approach of CRefactory, a refactoring tool for C programs. A refactoring tool cannot consider only a single configuration: changing the code for one configuration may break the rest of the code. CRefactory analyses the program for all possible configurations simultaneously. CRefactory also preserves preprocessor directives and integrates them in the internal representations. The paper also presents metrics from two case studies to show that CRefactory's program representation is practical.

1. Introduction

The C preprocessor (`Cpp`) adds many useful features to the C language, such as the ability to configure a program for different platforms by way of preprocessor conditionals. A configuration can be defined as the initial value of macros (a.k.a. configuration variables) that Cpp receives as input to preprocess a program. With this, a configuration determines which *single* branch of each preprocessor conditional will be present in the output of Cpp. Others consider this output of Cpp, the preprocessed code, to be a configuration. The definitions are isomorphic but we generally refer to the first.

Most projects written in C are highly configurable. For example, Flex [4] has less than 20K lines of code among 21 files, but has 5 configuration variables that make up a space of 25 possible configurations. The Linux kernel (version 2.6.7) has about 1,672 configuration variables with binary value. The number of possible configurations is huge.

The virtue of Cpp conditionals is lost once a program is preprocessed. Since the syntax of Cpp directives is different than the syntax of the C language, they need to be evaluated and removed before further processing [14]. This is satisfactory when the goal is to compile the program to be able to execute it: Cpp directives need to be removed and the program needs to be targeted to a single platform. But when the goal is to analyze or understand a program, selecting a single configuration loses information.

Although the problem has been recognized and partially approached, it has remained unsolved. The section on Related Work describes these approaches, from which the one in DMS [9] is the best approximation to handle multiple configurations.

The problem cannot be ignored for a refactoring tool. Refactoring tools have become popular for object-oriented languages like Smalltalk [16] and Java [2, 5]. They let programmers interactively and incrementally improve the structure of large programs without introducing errors. We are building a refactoring tool for C: *CRefactory*, and a major obstacle has been dealing with Cpp [12]. If refactorings are applied on the preprocessed version of a program, it may not be possible to recover the un-preprocessed version with Cpp directives and macro calls. Moreover, changing the code once it has been targeted to a specific configuration isolates that code from all the rest: if the changed code is merged back, the source code for other configurations may not compile anymore or the behavior may be altered.

Therefore, it is not acceptable to have the refactoring tool work on a single configuration of a program, nor to lose any Cpp directives. A refactoring tool must ensure program behavior is preserved for all possible configurations. CRefactory solves the problem: it preserves Cpp directives and represents the program for all possible configurations simultaneously.

Our previous paper ([13]) gives an overview of the kind of problems we had to face to be able to handle conditional directives. This paper extends our previous paper with details of our solution and results of its applicability. Section 2 shows how we integrate Cpp conditionals in the C grammar and in the program representations that CRefac-

Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)
1063-6773/05 $20.00 © 2005 **IEEE**

tory creates. It also provides a short description of how CRefactory handles other Cpp directives: `#define` (used to define macros) and `#include` (for file inclusion) in relation to conditionals. Detailed discussion of these two directives will appear elsewhere. Section 3 describes the internal transformation CRefactory performs on Cpp conditionals so they can be parsed, and how it later pretty prints the abstract syntax tree so the transformation on conditionals remains transparent. Section 4 shows how the program representations are finally used during refactoring. Section 5 shows measurements of CRefactory's program representation on two case studies: rm [1] and Flex [4]. Section 6 lists related work and Section 7 presents conclusions and future work.

2. Including multiple configurations in a program's representation

CRefactory preserves all possible configurations of a program. It does not use Cpp and does not remove any Cpp directives. However, some processing is needed before parsing takes place, to tokenize the source code [18] and gather information about Cpp directives. We call this step *pseudo-preprocessing* [13]. CRefactory has a pseudo-preprocessor called P-Cpp that outputs a tokenized version of the input source code plus some representations of Cpp directives described in Section 3.

The C grammar used by CRefactory allows Cpp directives at the level of five syntactic constructs, which are listed in Table 1. Although `#define` and `#include` directives may potentially break these constructs, we have not found any examples in any of our test cases (open-source packages like the Linux Kernel, the GNU library, q-mail, Flex, make). However, conditional directives often break these constructs. As explained below, CRefactory does not restrict the position of conditional directives but internally manipulates them to comply with the grammar. P-Cpp also expands macro calls so that the program can be parsed [12]. For both, conditional directive manipulation and macro expansion, P-Cpp tags the tokens accordingly so the changes can be reversed by the pretty-printer.

Table 1. Syntactic constructs allowed in between Cpp directives

Statement
Declaration
Structure field
Enumerator value
Array initializer value

CRefactory builds two main data structures to represent C programs: the symbol table and the abstract syntax tree (AST) [13]. Both data structures integrate Cpp directives with C entities (variables, functions, etc.). For example, the symbol table includes entries for macro definitions. Moreover, the symbol table allows a symbol to have different definitions that depend on the configuration, even allowing a symbol to have both a macro definition and a definition as a variable or some other C entity. The AST represents Cpp directives as nodes and macro calls as node labels. This representation allows programs to be analyzed and transformed while preserving the *un-preprocessed* source code.

Cpp directives also cause problems during refactoring [12, 13]. Therefore, refactoring preconditions in CRefactory check for instances of Cpp directives invalidating the refactoring. Transformations are applied by manipulating the AST. Finally, the AST is pretty-printed by reversing the internal manipulations that P-Cpp does on the source code (moving conditionals and expanding macro calls), so the source code appears just as the developer wrote it except for the refactorings he applied. Exact pretty-printing is hard to achieve but makes a refactoring tool much more usable.

2.1. Conditional directives

Conditional compilation directives define separate code branches, which are included or excluded from the final compilation unit depending on the value of conditions evaluated by Cpp [14]. Figure 1 shows a piece of code extracted from Make-3.80 which contains some conditional directives.

```
#ifndef alloca
#if __STDC__
typedef void *pointer;
#else
typedef char *pointer;
#endif
```

Figure 1. Example of conditional directives

A conditional directive is one of the following: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` or `#endif`. The `#if`, `#ifdef` and `#ifndef` directives start a *Cpp conditional* construct, creating its first branch. The `#elif` and `#else` directives create additional branches on the Cpp conditional and the `#endif` ends the construct. The `#if` and `#elif` tokens are followed by a constant expression in terms of macro definitions. The lines "`#ifdef` *id*" and "`#ifndef` *id*" are abbreviations of "`#if defined` *id*" and "`#if !(defined` *id*`)`" respectively. The source text inside a branch may include other Cpp directives. Consequently, Cpp conditionals can also be nested.

We can say that every token in the source code is guarded by a condition that depends on the conditional directives that surround it. P-Cpp associates tokens and symbol table entries with the condition that guards them. For example, the condition associated with the token `void` in the third line of Figure 1 is $(\neg defined(alloca) \wedge \_\_STDC\_\_)$, which in CRefactory's representation translates to:

$$AndCondition(\\ NotCondition(DefinedCondition(alloca)),\\ Condition(\_\_STDC\_\_))$$

Figure 2 shows the entry for symbol `pointer` in the symbol table (the conditions have been abbreviated).

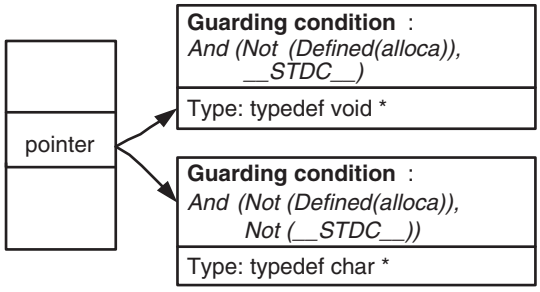

Figure 2. Enhanced symbol table

One way to represent all possible configurations of a program is to compute all possible combinations of configuration variables, parse the program in multiple passes, one for each combination, and create a different AST for each combination. We could apply this solution with Flex but we cannot possibly apply it for the more than 2.5M possible configurations of the Linux kernel. In the average case, this solution would be complex and expensive, and it would also produce a large amount of duplication in the ASTs. Our approach, instead, is to parse the program in a *single pass* and produce a *single* AST that includes all possible configurations [13]. This solution creates a compact representation and we think it can handle programs as large as the Linux kernel (although we have not tested the whole kernel at this time, only parts).

The problem with conditional directives is that they usually break statements and other C constructs as shown in Figure 3. We can also think of it as the branches created by a Cpp conditional being incomplete, i.e., they do not contain complete syntactical units. DMS solves this problem by restricting conditional directives to appear at certain places in the grammar, and manually modifying the code that does not comply [9]. This solution is simpler but not scalable to large, open-source projects.

CRefactory does not restrict the places where conditional directives may appear, but automatically and transparently

```
#ifdef VMS
    if (dep->changed && strchr (name, ':') != 0)
#else
    if (dep->changed && *name == '/')
#endif
  {
      freerule (rule, lastrule);
      …
  }
```

Figure 3. Incomplete conditional branches

transforms them, turning each branch of a Cpp conditional into a *complete syntactical unit*, so it complies with the grammar. This transformation is explained in detail in Section 3.

A macro can have different definitions in different configurations [13]. Moreover, a symbol can be a macro in one configuration and a C entity in another. When P-Cpp finds a call to a macro with multiple definitions (and therefore with more than one possible expansion), or a symbol that can be either a macro call or a reference to a C entity, P-Cpp expands it to a preprocessor conditional with one branch for each possible macro expansion or C symbol. This preprocessor conditional introduced by P-Cpp may break the statement or declaration that contains it. Therefore, those conditionals introduced by P-Cpp will also need to be completed for parsing to work, and they turn out to be the hardest to complete. The reason is that there can be more than one macro call in the same statement and the preprocessor conditionals in the macro expansions need to be combined. For example, in the case of 3 macro calls in the same statement, each macro with 2 possible definitions, the 3 preprocessor conditionals in the macro expansions need to be combined to create 8 conditional branches. In our testing so far we have not seen more than 3 macro calls that expand to a Cpp conditional in the same statement. Section 5 shows the growth of our case studies after the completion of Cpp conditionals.

In the case of file inclusion, when a given file is included more than once in a compilation unit, Cpp preprocesses it again every time, since different conditional directive branches may be selected in subsequent inclusions if macro definitions changed. This is because Cpp selects a *single* branch of each Cpp conditional. Conversely, P-Cpp processes *all* branches of a conditional in a single pass. As a result, P-Cpp does not need to process a file more than once, but it can reuse the previously generated representation. The only exception to this is when an included file uses macros defined previously in the compilation unit, and those macro definitions changed since the previous time the file was included. Although we can handle this situation, we have never encountered it in our testing.

3. Completing preprocessor conditionals

Preprocessor conditionals are completed in two passes through the source code. In the first pass, P-Cpp tokenizes the input and recognizes incomplete Cpp conditionals, creating descriptors that contain information on how to complete them. In the second pass, incomplete conditionals are completed by moving and copying tokens as stated by the descriptors created in the first pass. After this second pass, all Cpp conditionals are complete syntactical units, i.e., they can be integrated in the C grammar and the source code can now be parsed.

3.1. First pass of P-Cpp

A Cpp conditional is considered *complete* when its branches enclose a whole syntactic construct, or a whole list of them, from the constructs that appeared in Table 1.

P-Cpp needs to be able to recognize the beginning and end of each of the syntactic constructs in Table 1 to recognize if a conditional is incomplete. In the first pass of P-Cpp, while it tokenizes the input, it keeps track of the tokens that mark the beginning or the end of any of the constructs in the table, by maintaining a state stack (as a pushdown automata). The following description names the states used to represent each construct.

Simple statement or declaration While scanning a simple statement or declaration, the top of the state stack is **In-construct**. The appearance of a ';' marks the end of the construct and so P-Cpp replaces **In-construct** by **End-of-construct** at the top of the stack. To prevent confusing the use of ';' inside the expressions of a `for` statement (where conditional directives are not allowed), the state **In-for** is at the top of the state stack while scanning the expressions of a `for` statement. This state ignores ';' characters.

Composite statement When state **In-construct** is at the top of the state stack, the appearance of a '{' makes P-Cpp push state **In-composite-statement**. Inside a composite statement, the states **In-construct** or **End-of-construct** will be pushed on top of the state stack to represent inner statements or declarations.

Structure The keyword 'struct' makes P-Cpp push state **In-Struct**. Inside a struct definition, the states **In-construct** or **End-of-construct** will be pushed on top of the state stack to mark the beginning and end of a field declaration.

Enumerator The keyword 'enum' makes P-Cpp push state **In-Enum**. While in this state, the scanning of an enumerator value is represented by pushing **In-Enum-Elem** or **End-of-Enum-Elem** on top.

Array initializer When the state **In-construct** is at the top of the state stack, an '=' followed by a '{' mark the beginning of an array initializer. P-Cpp pushes state **In-initializer** in this case. The scanning of an initializer value is represented by pushing **In-Init-Value** or **End-of-Init-Value** on top of the state stack.

With this representation of states, a Cpp conditional is complete if and only if its branches start and end when the top of the state stack is **End-of-construct**. Conversely, if a Cpp conditional *starts* while **In-construct** is at the top of the stack, the Cpp conditional is set to have a *bad start*. Moreover, if a Cpp conditional *ends* when the top of the state stack is **In-construct**, the Cpp conditional has a *bad ending*. The pseudo-code for the first pass of P-Cpp appears in Figure 5 in the Appendix.

When P-Cpp encounters conditional directives in its first pass, it constructs *Cpp_Conditional_Descriptor*s from them. These descriptors form a tree that represents the nesting of conditionals. Moreover, they contain enough information to fix incomplete conditionals in the second pass. Some of the information in a Cpp_Conditional_Descriptor appears in Table 2.

Table 2. Data in a Cpp_Conditional_Descriptor

startPosition	The position where the Cpp conditional starts in the current file
endPosition	The position where it ends
badStart	True if the Cpp conditionals has a bad start
badEnding	True if it has a bad ending
startPosShouldBe	The position where the conditional should start to be complete
endPosShouldBe	The position where it should end to be complete
branches	A sequence of *Cpp_Conditional_Branch_Descriptors*

When P-Cpp pushes a state in the state stack, it sets the state's starting position to be the current source code position. This value is used to set the field $startPosShouldBe$ of conditionals that start in the middle of the construct that the top state represents.

When the end of a Cpp conditional is found in the middle of a construct, its descriptor is added to a list of conditionals breaking the current construct in the state at the top of the stack (this list is called $condsWBadEnding$ in the pseudo-code). Later on, when P-Cpp finds the end of the current construct, it sets the value $endPosShouldBe$ of all descriptors in the list $condsWBadEnding$ to be the current position.

3.2. Second pass of P-Cpp

Once the tree of Cpp_Conditional_Descriptors has been created, all incomplete conditionals should be fixed by moving and copying tokens as dictated by their descriptors. Figures 6 and 7 in the Appendix have the pseudo-code for this step.

First, if the conditional has a bad start, P-Cpp calculates the tokens that complete the start of the conditional. Second, P-Cpp checks if the next conditional is also incomplete and if it breaks the same construct as the current one (the positions where they should start and end match). In this case, the next conditional is completed in each branch of the current conditional, creating this way all four combinations. Following conditionals are checked recursively for intersections with the previous ones.

If the next conditional does not break the same construct, and if the current conditional has a bad ending, P-Cpp calculates the tokens that complete the end of the conditional. Here it is also possible that the next conditional, when being complete, will become the child of the current one. Figure 4 shows an example where the first conditional breaks the while statement and the second conditional breaks the assignment statement inside the body of the while. In this case, the next conditional is completed inside the stream of tokens that complete the end of the current conditional.

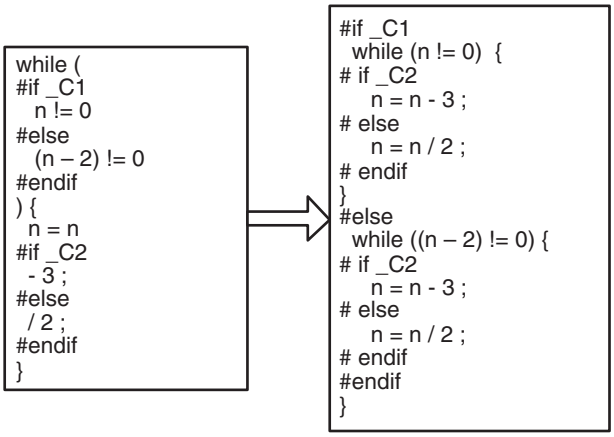

Figure 4. The second conditional becomes child of the first

After the tokens that complete the start and the end of each incomplete conditional have been calculated, P-Cpp actually moves and copies those tokens, while it labels them, in each branch of the conditional. Figure 7 in the Appendix shows the pseudo-code for how P-Cpp completes each branch. The basic idea is that the tokens that complete the start of the conditional are *moved* to the beginning of the first branch and *copied* to the beginning of the other branches. If the conditional has a bad ending, the tokens that complete the end of the conditional are *moved* to the end of the last branch and *copied* to the end of the other branches. Tokens are labelled accordingly so this manipulation can be reversed as described in the next section.

3.3. Pretty-printing of Cpp conditionals

The pretty-printer visits the nodes in the AST and prints the leaf nodes according to the tokens they represent and the tokens' labels. Figure 8 in the Appendix shows the pseudo-code for pretty-printing a leaf node that does not come from macro expansion.

The pretty-printer uses two queues: $ifsQueue$ stores the start directive of Cpp conditionals with bad start. These start directives go later in the output, after the tokens that have been *moved forward* to complete the first branch. The other queue is the $movedBackQueue$, and this one is for nodes representing tokens that have been *moved backwards* to complete the end of a conditional. The nodes in the $movedBackQueue$ are printed after the #endif of the current conditional. Nodes that represent *copied* tokens are not printed.

4. Applying refactorings

The program representation used by CRefactory could also be used by a program analysis or understanding tool like Genoa [10], Columbus [11] or Visual C# IntelliSense [6] to provide information about the multiple configurations of a program. We use this program representation for refactoring. This section shows two examples that illustrate the power of the representation to analyze the preconditions of refactorings and apply the transformations.

4.1 Delete unreferenced variable

After a program has been changed a number of times, some variables may become unused. However, programmers may resist to delete their definitions if they are unsure the variable is used under some particular configuration.

Input values: variable *V*.

Preconditions: Opdyke [15] describes the precondition of this refactoring as: $referencesTo(V) = \emptyset$. When multiple configurations of a program are considered, V can have more than one definition. CRefactory allows deleting those definitions of V that have no references, even when other definitions of V are used in other configurations. In terms of CRefactory's analysis functions, the precondition of this

refactoring is expressed as:

$$\exists D_i \in all\_var\_definitions(V) : uses(V, D_i) = \emptyset$$

The function $all\_var\_definitions(V)$ looks up the entry for V in the symbol table and returns all the definitions of V as a variable (note that V could be defined as a macro in some configuration, so the macro definition would not be returned by this function). The function $uses(V, D_i)$ returns the set of uses of definition D_i of V, again directly looking it up in the symbol table. So as long as one definition of V has no uses, the refactoring can proceed.

Mechanics: If V has more than one definition under different configurations, the refactoring must check which of those definitions have no references and can be safely removed. The following pseudo-code describes the mechanics of this refactoring:

1. $conds$:= { $guarding\_condition(D_i)$:
   $D_i \in all\_var\_definitions(V)$
   $\wedge\ uses(V, D_i) = \emptyset$ }
2. Search AST for nodes representing a definition or declaration of V. Upon a match:
   if ($guarding\_condition(node) \in conds$)
   delete($node$)

Step 1 creates a set $conds$ with the conditions that guard the definitions of V that have no uses. This result is again directly returned by the symbol table, which attaches to each definition the condition in which it applies.

This refactoring may leave a Cpp conditional branch empty, if the deleted definition or declaration of V was the only thing in the branch. We leave the decision to the user to apply the refactoring "Delete empty branches of Cpp conditional", which, given a Cpp conditional, removes the branches that are left empty.

4.2 Move variable into structure

A variable defined outside any structure is moved so that it becomes a field of a structure. This refactoring is useful when creating a structure out of global variables. The next steps are to add a pointer reference to this structure and to pass the pointer as argument to the functions, thus reducing the use of global variables in the program.

Input values: variable V and structure declaration S.

Preconditions: the preconditions of this refactoring are divided in two cases: 1. There is a single definition of V and 2. There are multiple definitions of V.

Case 1. $single\_definition(V) = true$. In the preconditions given below, $guarding\_cond(D_S) = TrueCondition$ means that the definition D_S is not inside a Cpp conditional.

$$\begin{aligned}
&single_definition(S) && (1)\\
&\quad \wedge (guarding_cond(D_S) = guarding_cond(D_V) && (2)\\
&\qquad \vee\ guarding_cond(D_S) = TrueCondition) && (3)\\
&\quad \wedge V \notin fields(D_S) && (4)\\
&\quad \wedge scope(D_S) \supseteq scope(D_V) && (5)\\
&\quad \wedge (\forall u_i \in uses(V) : \exists SVar : refers(type(SVar), S)\\
&\qquad \wedge guarding_cond(SVar) = guarding_cond(u_i)\\
&\qquad \wedge scope(SVar) \supseteq scope(D_V)) && (6)
\end{aligned}$$

where D_V is the single definition of V and D_S is the single definition of S.

The expression numbered (6) in the above equation means that at each use of V, there is a way to refer to an instance of S, i.e., the type of $SVar$ is '`struct` S', '`struct` S `*`', or similar.

Case 2. $single\_definition(V) = false$. The refactoring is allowed if:

$$\begin{aligned}
&(single_definition(S)\\
&\quad \wedge guarding_condition(D_S) = TrueCondition\\
&\quad \wedge (\forall D_V \in all_var_definitions(V) : (4) \wedge (5) \wedge (6)))\\
&\vee (single_definition(S) = false\\
&\quad \wedge \#(all_var_defs(V)) = \#(all_struct_defs(S))\\
&\quad \wedge (\forall D_V \in all_var_defs(V) : \exists D_S \in all_struct_defs(S) :\\
&\qquad guarding_cond(D_S) = guarding_cond(D_V)\\
&\qquad \wedge (4) \wedge (5) \wedge (6)))
\end{aligned}$$

That is, either there is a single definition of S and for each definition of V, the sub-equations numbered (4), (5) and (6) in Case 1 hold, or there are as many definitions of V as there are of S, and for each pair of definitions with the same condition, the sub-equations (4), (5) and (6) hold.

Mechanics: the mechanics of Case 1 serve as a base case, that is, when there is a single definition of V and S. Case 2 applies the mechanics of Case 1 for each appropriate combination of variable-structure definitions.

Case 1. Given a definition of V, D_V, and a definition of S, D_S:

1. AST_S := {AST in scope of D_S}.
2. Search AST_S for the node N_{D_S} representing D_S.
3. Search AST_S for the node N_{D_V} representing D_V.
   if ($guarding\_cond(D_S) = guarding\_cond(D_V)$)
   4. Move N_{D_V} to the right-most leaf in N_{D_S}.

else if $(guarding\_cond(D_S) = TrueCondition)$
5.Move N_{D_V} to the right-most leaf in N_{D_S} and surround it with a Cpp conditional with only one branch for condition $guarding\_cond(D_V)$.
6. Search AST_S for uses of D_V and replace each match by a subtree representing $(temp\_Svar).V$.

Step 1 creates a set with all trees in the scope of D_S. This set may have more than one element if D_S is global, because CRefactory builds one AST for each file that composes the program. If D_S is local, CRefactory maintains indexes to the trees so that it can directly reach the sub-tree for that local scope.

In Step 6, $temp\_Svar$ is a template that represents the way to refer, directly or indirectly, to an instance of S, i.e., it may have the form 'var' or '(*ptr)' or similar.

Case 2.

if $(single\_definition(S))$
for each $D_V \in all\_var\_definitions(V)$
1. Apply Case 1 with D_V and the single definition of S.
else
for each $D_V \in all\_var\_definitions(V)$ {
2. Find $D_S \in all\_struct\_defs(S)$ such that $guarding\_cond(D_S) = guarding\_cond(D_V)$.
3. Apply Case 1 with D_V and D_S. }

5. Case studies

The previous section shows some of the analysis functions that CRefactory uses to check the preconditions of refactorings. This section provides a quantitative analysis of the program representations built when two programs were loaded in CRefactory: rm, the 'remove file' function in GNU core utilities [1], and Flex, the lexical analyzer generator [4]. These metrics help us to show that, in practice, conditional completion does not appear to produce an exponential growth of the representation of programs.

CRefactory is implemented in VisualWorks Smalltalk^{TM}. The refactoring engine mimics the design of the Smalltalk Refactoring Browser [16]. To load a program, CRefactory needs to know the source files, include directories, read-only directories (those that contain non-modifiable files, like standard library headers), command line macros and false conditions. The set of "false conditions" is used by CRefactory to exclude some Cpp conditional branches that it should not or cannot process. For example, the set of false conditions includes `defined(__cplusplus)` and `defined(__GNUC__)` because CRefactory cannot parse C++ code and does not currently support some GCC extensions [3] used in the code under `defined(__GNUC__)`.

5.1. Results on rm

The source code for rm is contained in a single source file: "rm.c". When this source file was loaded in CRefactory, it included 94 header files, although only 20 of them belong to the same package and are therefore modifiable (the others are GCC library headers). Note that this number of headers are included when considering all possible configurations, only excluding false conditions. The results that appear in Table 3 were obtained on the 20 files in the rm package, i.e., they filter out GCC library headers.

Table 3. Metrics on rm

Number of Cpp conditionals	262
Number of Cpp conditionals introduced by macro expansion	30
Number of incomplete Cpp conditionals	30
Perc. of code growth after completing conditionals	18%
Maximum level of nesting of Cpp conditionals	3
Percentage of conditional definitions	24%

There is one header file in the package: `system.h`, that alone contains 166 out of the 262 Cpp conditionals. It contains many macro definitions (31% of all macro definitions in the package). We can infer that this file is highly configurable and so, difficult to maintain.

From the total of 262 Cpp conditionals, 30 were introduced by P-Cpp due to macro expansion (i.e., because of calls to macros with more than one definition). All of these Cpp conditionals and only those were incomplete. Therefore, all Cpp conditionals present in the source code of the rm package are complete, which speaks very well of the readability of the source code. Moreover, 27 out of the 30 Cpp conditionals introduced by P-Cpp appear in a single file: `rm.c`. Since all of these conditionals had to be completed, the tokenized representation of `rm.c` grew 57% after conditional completion.

The depth of nesting of Cpp conditionals is low, so the source code is not complex in that sense. However, the percentage of symbol definitions that depend on configuration variables (i.e., conditional definitions) is rather high. Considering all possible configurations simultaneously is therefore very important to be able to modify this code.

We found one symbol: `getopt`, defined as a function under one configuration and as a macro under another configuration in a different header file. Refactoring this symbol or the code that uses it would be difficult without a tool

that can spot these double definitions and check for possible problems.

5.2. Results on Flex

The source code for Flex is contained in 13 source files. However, 2 of them are automatically generated from grammar specifications. Although loading Flex involved loading 54 files, Table 4 shows the results obtained on the 11 source files that are not auto-generated plus the 4 headers in the Flex package (i.e., all the modifiable files).

Table 4. Metrics on Flex

Number of Cpp conditionals	36
Number of Cpp conditionals introduced by macro expansion	9
Number of incomplete Cpp conditionals	9
Perc. of code growth after conditional completion	1%
Maximum level of nesting of Cpp conditionals	1
Percentage of conditional definitions	2%

The table above shows that the Flex package is not too complex in terms of Cpp conditionals. As with the previous case, all incomplete Cpp conditionals were introduced by P-Cpp due to macro expansion.

The files that have the most conditional directives are `flexdef.h`, `main.c` and `misc.c`. File `flexdef.h` has 15 Cpp conditionals, all present in the source code and all complete. In the case of `misc.c`, 5 out of 7 Cpp conditionals come from macro expansion. The tokenized representation of `misc.c` grew 4% after conditional completion. The file that grew the most was `yylex.c`, with a growth of 7%.

There is no nesting of the Cpp conditionals in the Flex package, although the maximum level of nesting is 23 when counting library files.

The total number of macros defined in the package is 134. There is one unreferenced variable in the package: `copyright`, defined under condition $\neg defined(lint)$.

6. Related work

Somé and Lethbridge argue that program understanding tools should provide information about each configuration in which an entity can be considered [17]. They propose some heuristics to detect the configurations that can be parsed in the same pass, therefore minimizing the number of passes needed.

The framework PCp^3 [8] allows the analysis of C source code with Cpp directives by providing "hooks" in the preprocessor or in the parser. That is, the code is preprocessed but the user can define callbacks in Perl scripting language, making use of those hooks in the preprocessor. PCp^3 can provide useful information about Cpp directives. However, the program representations that PCp^3 can produce would still be based on a single configuration and so inappropriate for refactoring.

Tokuda and Batory propose a refactoring tool on class diagrams of C++ programs [19]. Their tool also works on a single configuration. Xrefactory is a refactoring tool for C and C++. It provides some support for Cpp directives, like allowing macro renaming [7]. Xrefactory preprocess the source code but saves the original position of each element in the un-preprocessed code [20].

The best approximation to handle multiple configurations is the one provided by DMS [9]. DMS is able to parse conditional directives by allowing them at certain, predefined places in the grammar. With this approach, DMS can parse 85% of un-preprocessed C files [9]. DMS performs code restructuring like removing dead branches of Cpp conditionals by running a set of predefined rewrite rules on appropriate program representations. Our goal is instead to support interactive code manipulation through smaller refactorings that are selected by the user with a drop-down menu while visualizing the code.

7. Conclusions

Integrating conditional directives during refactoring is hard, for both the analysis of the code required before refactoring and the transformation functions themselves. The same applies to other Cpp directives, like macros and file inclusion. This is a major factor that hinders the development of refactoring tools for C or C++ code. Our work successfully integrates Cpp directives in the C grammar and the program representation.

Although CRefactory focuses on C and Cpp, our work applies to other languages that use Cpp, like C++, and probably to other preprocessors with similar directives, whose syntax is independent of the syntax of the underlying language. Moreover, analysis tools should be able to apply the same ideas for parsing and representation of Cpp conditionals.

Future work includes loading all the source code of the Linux kernel, for all possible configurations, and measure memory requirements. For this, we will need to add all GCC extensions to the C language. Also, other refactorings will be added to CRefactory. Our ultimate goal is to reach a stable version of CRefactory that we can release to the public.

Acknowledgements

We would like to thank Prof. Samuel Kamin for his valuable comments on an earlier version of this paper.

References

[1] Coreutils - GNU Project - Free Software Foundation (FSF). http://www.gnu.org/software/coreutils/coreutils.html.

[2] Eclipse.org main page. http://www.eclipse.org.

[3] Extensions to the C language family. http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html.

[4] Flex - GNU project - Free Software Foundation (FSF). http://www.gnu.org/software/flex/flex.html.

[5] IntelliJ IDEA: the most intelligent Java IDE around. http://www.intellij.com/idea/.

[6] Visual C# IntelliSense. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxoriintellisensefeatures.asp.

[7] Xrefactory - A C/C++ development tool with refactoring browser. http://xref-tech.com/xrefactory.

[8] G. J. Badros and D. Notkin. A framework for preprocessor-aware C souce code analyses. *Software Practice and Experience*, 30(8), 2000.

[9] I. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Workshop on Analysis, Slicing, and Transformation at the Eighth Working Conference on Reverse Engineering (WCRE'01)*, 2001.

[10] P. Devanbu. Genoa - a customizable, front-end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 9(2), 1999.

[11] R. Ferenc, I. Siket, and T. Gyimóthy. Extracting facts from open source software. In *20th IEEE Int. Conf. on Software Maintenance (ICSM)*, pages 60–69, Chicago, Illinois, 2004.

[12] A. Garrido and R. Johnson. Challenges of refactoring C programs. In M. Aoyama, K. Inoue, and V. Rajlich, editors, *Proc. of the Fifth International Workshop on Principles of Software Evolution (IWPSE)*, pages 6–14, Orlando, 2002. ACM.

[13] A. Garrido and R. Johnson. Refactoring C with conditional compilation. In *Proceedings of the IEEE Automated Software Engineering Conference (ASE)*, pages 323–326, Montreal, Canada, 2003.

[14] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[15] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[16] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.

[17] S. Somé and T. Lethbridge. Parsing minimization when extracting information from code in the presence of conditional compilation. In *Sixth International Workshop on Program Comprehension*, Ischia, Italy, 1998. IEEE.

[18] R. Stallman and Z. Weinberg. The C preprocessor. GNU Online documentation. http://gcc.gnu.org/onlinedocs/, 2001.

[19] L. Tokuda and D. Batory. Evolving object oriented designs with refactoring. In *Proc. IEEE Conference on Automated Software Engineering (ASE)*, 1999.

[20] M. Vittek. Refactoring browser with preprocessor. In *7th European Conference on Software Maintenance and Reengineering*, Benevento, Italy, March 2003.

Appendix

This Appendix shows the pseudo-code of the conditional completion algorithm in P-Cpp and the pretty-printing that reverses this completion. Figure 5 has the pseudo-code for the first pass of P-Cpp through the code. Figures 6 and 7 show the second pass of P-Cpp. Finally, Figure 8 describes pretty-printing.

```
case (current token = #if, #ifdef or #ifndef) {
    desc := new CppConditionalDescriptor.
    desc.startPosition := current position.
    if( top(stateStack) = End-of-construct)
        desc.badStart := false
    else {
        desc.badStart := true.
        desc.startPosShouldBe :=
                (top(stateStack)).startPosition.
    } }
case (current token = #elif or #else) {
    branch := new CppConditionalBranchDescriptor.
    branch.startPosition := current position.
    Add branch to branches of curr. Cpp conditional.
}
case (current token = #endif) {
    desc := current Cpp conditional.
    desc.endPosition := current position.
    if( top(stateStack) = End-of-construct)
        desc.badEnding := false.
    else {
        desc.badEnding := true.
        Add desc to
            (top(stateStack)).condsWBadEnding.
    } }
case (curr. token marks beginning of construct) {
    Push corresponding state into stateStack with
        newState.startPosition := current position.
}
case (curr. token marks end of curr. construct) {
    for each desc ∈
            ((top(stateStack)).condsWBadEnding)
        desc.endPosShouldBe := curr. position.
    Push appr. construct ending state into stateStack
}
```

Figure 5. Pseudo-code for first pass of P-Cpp

```
completeConditional(desc) {
  if (desc.badStart)
     desc.tokensCompletingStart :=
          tokens from desc.startPosShouldBe to
          desc.startPosition.
  nextDesc := nextConditional(desc).
  if (breakSameConstruct(desc, nextDesc))
     for each branch ∈ desc.branches
        Complete nextDesc inside branch.
  else if (desc.badEnding) {
     if (¬ intersect(desc, nextDesc))
        desc.tokensCompletingEnd :=
             tokens from desc.endPosition to
             desc.endPosShouldBe.
     else
        Complete nextDesc inside
                desc.tokensCompletingEnd.
     for each branch ∈ desc.branches
        completeConditionalBranch(desc, branch).
  } }
```

Figure 6. Second pass of P-Cpp: completing Cpp conditionals

```
completeConditionalBranch(desc, branchDesc) {
  if (desc.badStart)
     if (isFirstBranch(desc, branchDesc))
        Move desc.tokensCompletingStart to
             beginning of branch while labelling these
             tokens as 'moved forward'.
     else
        Copy desc.tokensCompletingStart to
             beginning of branch while labelling these
             tokens as 'copied'.

  for each childDesc ∈ children(desc, brachDesc)
        completeConditional(childDesc).

  if (desc.badEnding)
     if (isLastBranch(desc, branchDesc))
        Move desc.tokensCompletingEnd to
             end of branch while labelling these
             tokens as 'moved backwards'.
     else
        Copy desc.tokensCompletingEnd to
             end of branch while labelling these
             tokens as 'copied'. }
```

Figure 7. Second pass of P-Cpp: completing Cpp conditional branches

```
pretty-print(node) {
  case(node represents #if, #ifdef or #ifndef)
        if ((assocCppConditional(node)).badStart)
          queue(ifsQueue, node).
        else Print node.
  case (label(node) 'not moved' or 'moved forward')
        Check if top(ifsQueue) should be printed.
        Print node.
  case (label(node) = 'copied')
        /* do nothing */
  case (node represents #elif or #else)
        Print node.
  case (label(node) = 'moved backwards')
        queue(movedBackQueue, node).
  case (node represents #endif) {
        Print node.
        Print nodes in movedBackQueue } }
```

Figure 8. Pseudo-code for pretty-printing