Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2013; **25**:1285–1304
Published online 26 June 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1603

Embracing the C preprocessor during refactoring

Alejandra Garrido[1,*] and Ralph Johnson[2]

[1]*LIFIA, Facultad de Informática, Universidad Nacional de La Plata and CONICET, La Plata, Bs. As., Argentina*
[2]*University of Illinois at Urbana-Champaign, Urbana, IL, USA*

ABSTRACT

C preprocessor directives are heavily used in C programs because they provide useful and even necessary additions to the C language. However, they are usually executed and discarded before any analysis is applied on C programs. In refactoring, preprocessor directives must be preserved through the whole process of parsing, analysis and transformation to retain editable yet correct source code. We propose a new preprocessing approach and special program representations that allow a program to be analyzed and transformed without losing its preprocessor directives, but treating them as first-class program entities. These representations are essential for a correct refactoring tool. We also describe the challenges that preprocessor directives bring to refactoring and how the program representations that we propose solve those challenges. Finally, we give details of two refactorings and present some case studies with our successfully applied solution. Copyright © 2013 John Wiley & Sons, Ltd.

Received 23 February 2012; Revised 24 February 2013; Accepted 15 May 2013

KEY WORDS: refactoring; preprocessor directives; code analysis; software maintenance

1. INTRODUCTION

Many large software projects are written in C. Sourceforge.com lists 527 projects coded in C and the only other language with more projects is C++ with 762 projects. C is also one of the most popular programming languages in IT companies [1]. These projects require state-of-the-art maintenance processes and tools that can deal with their complexity and size. Unfortunately, there are no refactoring tools that can completely and safely transform C code. The main reason for this is the C preprocessor (Cpp) [2].

Cpp enriches the C language with many useful features. It lets programmers divide a program into manageable parts, customize code for different platforms or C dialects, and define constants and constructs that can be used on any data type. Cpp is controlled by special commands called preprocessor directives. Cpp directives start with '#' and their syntax is completely independent of the syntax of the C language [3]. Cpp is heavily used in C programs because it provides for significant flexibility, but its ability to perform arbitrary source code manipulations complicate the understanding of C programs by programmers and tools [4]. Liebig *et al*. analyzed 40 open-source software systems with more than 30 million lines of code and found that Cpp's mechanisms are preferred by programmers to enclose variable or extension code, which on average accounts for 23% of the code base in a project [5].

Preprocessing occurs before parsing, transforming a program by a series of textual replacements. These replacements include removing comments, converting the input file into a sequence of tokens

*Correspondence to: Alejandra Garrido, LIFIA, Facultad de Informática, Universidad Nacional de La Plata. 50 y 120. CP 1900. La Plata, Bs. As., Argentina.
†E-mail: garrido@lifia.info.unlp.edu.ar

(tokenization), executing directives and expanding macros [6]. The most important and used directives in Cpp are: #include, for the inclusion of header files; #define, to create macros; and the family of conditional directives to control the inclusion of code based on configuration setting.

A refactoring tool for C must be able to allow transformations directly in the given source code, not its preprocessed version, for three main reasons:

1. Programmers rarely work with preprocessed source code. Although macro expansion might help to understand a code snippet, preprocessing file inclusion directives merges all files in a single unit, losing modularization.
2. Programmers would rather not use a tool to make changes to their working programs if they cannot even recognize their code, and thus they cannot trust that programs' behavior will be preserved.
3. If changes are applied on the preprocessed version of a program, it may be impossible to go back to the original source code. On the one hand, some changes to macro expansions may prohibit recovering macro definitions. Figure 1 shows to the left a piece of the definition of macro 'MODINFO_ATTR', which applies concatenation on its parameter to create the function name. The macro references at the end (named 'macro calls' through this article) expand to the function definitions on the right of Figure 1. If, for example, the first function is renamed (e.g., to 'exist_mod_ver'), it becomes impossible to translate the code back to the unpreprocessed source code that called the macro 'MODINFO_ATTR'. On the other hand, if a refactoring tool preprocesses conditional directives so a single configuration is selected at a time, a refactoring applied on a single configuration may cause the code under the other configurations to become invalid. That would be the case in Figure 2 if the variable 'pointer' is renamed to 'pointer_t' only under configuration '__STDC__' (which would rename all but the 2nd occurrence in Figure 2); when the other configuration is used, there will be no definition for the uses of 'pointer_t' and the code will not compile. Thus, refactorings could not guarantee to preserve behavior [2].

For these reasons, any reengineering tool for C must process a program in a way that does not remove Cpp directives but includes them in the internal representations of the program. We propose for this reason a specialized preprocessor, called *pseudo-preprocessor* (*P-Cpp*), which evaluates Cpp directives but does not remove them. In a previous article we described how P-Cpp handles conditional directives [7]. In this article, we extend the description showing how CRefactory, our refactoring engine, handles macro directives and file inclusion. With a slight modification of the parser and extensions to the program representations that the parser creates, CRefactory can handle the analysis and transformation of C programs even in the presence of a heavy use of Cpp directives.

```
#define MODINFO_ATTR(field) \
static int modinfo_##field##_exists(struct module *mod) { \
        return mod->field != NULL;                        \
}                                                         \
…
MODINFO_ATTR(version);
MODINFO_ATTR(srcversion);
```

```
static int modinfo_version_exists(struct module *mod) {
        return mod->version != NULL;
}
static int modinfo_srcversion_exists(struct module
*mod) {
        return mod->srcversion != NULL;
}
```

Figure 1. Piece of source code on the left from file 'module.c' in the Linux kernel and its preprocessed version on the right.

```
#if __STDC__
   typedef void *pointer;
#else
   typedef char *pointer;
#endif
…
pointer alloca (unsigned size) {
...
   free ((pointer) hp);
```

Figure 2. Piece of source code from 'alloca.c' in Make-3.82.

DOI: 10.1002/smr

Moreover, the presence of Cpp directives complicates refactoring in many ways [2]. Because directives are not legal C code, they can change the scope and definition of program elements and can allow for special manipulations that may violate otherwise correct refactorings. We describe each of these problems and how CRefactory accounts for them to ensure that refactorings preserve behavior.

2. PSEUDO PREPROCESSING AND PARSING IN A REFACTORING TOOL FOR C

Refactoring tools are interactive tools that a programmer uses to automate simple changes to source code. The purpose of these changes is usually to improve maintainability and readability, not to change functionality [8]. Thus, refactoring tools should be powerful enough to allow improving design quality of software while robust enough to preserve behavior. To combine these properties, a refactoring tool requires at least two internal representations of the program: the symbol table and the abstract syntax tree (AST) [8]. These program representations are constructed during parsing and semantic analysis.

Thus, the first problem we face is how to preserve Cpp directives, include them in the internal representations, analyze them during refactoring and transform them together with C code. One solution for this would be to avoid preprocessing at all and have the parser and semantic analyzer deal directly with Cpp directives. However, it is not possible to apply this solution without introducing ambiguities in the grammar or being too restrictive. Conditional compilation directives and macro calls often 'break' statements. A conditional compilation directive or macro call breaks a statement when it produces a fragment of C code that is not a complete syntactic unit. For example, Figure 3 shows a preprocessor conditional breaking the condition of the 'if' statement. Macro references cause a similar problem because they may represent any arbitrary fragment of C code.

2.1. *Pseudo-preprocessing*

Because we aimed for a general and complete solution, our tool could not be too restrictive in the kind of macros it accepts or the placement of directives. Therefore, CRefactory preprocesses the input in a way that does not remove directives but makes the input parseable. We call this 'pseudo-preprocessing'. We have developed a pseudo-preprocessor, P-Cpp, which tokenizes the input and executes directives as Cpp does, but does not remove directives nor comments from the tokenized output [9]. Each token in this output holds its original position, so the exact same formatting can also be preserved.

To solve the problem of Cpp conditionals breaking statements (Figure 3), one approach is to parse each Cpp conditional branch independently [10]. The drawback is that there may be a huge number of possible configurations created by Cpp conditionals, which would take a long processing time, and make recombining the result very complex. A more efficient approach is to parse each branch in parallel, by 'cloning' the parser for each branch as suggested in [11]. The drawback of this approach is that it requires writing a table-driven parser, and it is not possible to use a standard parser.

Our approach to parse multiple configurations in a single pass is to have P-Cpp apply a behavior-preserving transformation to the tokenized source code. We call this transformation "conditional completion algorithm" (CCA) because it turns Cpp conditionals into *complete* syntactical units, that is, they enclose complete statements or other directives. The CCA is thoroughly explained elsewhere [7]. This algorithm expands code, often moving and copying code from outside a Cpp conditional into each of its branches to complete them. P-Cpp labels the tokens that were moved or copied so

```
            if (instring == ''
#if defined (__MSDOS__) || defined (__EMX__) || defined (WINDOWS32)
                                || !unixy_shell
#endif
                                )
                                    ++p;
```

Figure 3. Code excerpt from file 'job.c' in the source code of Make-3.82. Conditional directives break the condition.

DOI: 10.1002/smr

the CCA can be later reversed. Figure 4 shows the code of Figure 3 after P-Cpp has applied the CCA. Because Cpp conditionals can be nested, the algorithm covers several cases of conditionals' combination. In the worst case, this expansion can be exponential, but we have not found any exponential blowup in practice (Section 7 shows results on typical C code).

Because macros calls often break statements, they need to be expanded for parsing to work. Such as Cpp, P-Cpp expands macro calls but labels the tokens in the expansion so the call can be traced back and reconstructed. Because we process multiple configurations, there may be multiple definitions of a macro, and so a single macro call may bind to multiple macro definitions, therefore having more than one possible expansion. In that case P-Cpp expands the macro call to a Cpp conditional, with one branch for each possible macro expansion. Figure 5 shows an example. The reference to BO_EXBITS at the end of Figure 5(a) becomes expanded as shown in Figure 5(b), after which the CCA completes the conditional because it breaks the assignment, and the results appear in Figure 5(c).

Summarizing the work of P-Cpp, we can say that: (a) on pure C code, the output of P-Cpp is the same than Cpp; (b) on code with a preprocessor directive, the output of P-Cpp is larger than the output of Cpp, because it includes the preprocessor directive as a token; and (c) on code with a macro call, P-Cpp either expands the call equally to Cpp or it inserts a conditional with multiple alternative expansions; nonetheless, one of those expansions must be the same used by Cpp on the chosen configuration [9]. That is, in all cases, the output of P-Cpp includes the output of Cpp. We can state it precisely as follows: P-Cpp(SC) $\supseteq$ Cpp(SC) for a given source code SC. Consequently, we can state the following important equality:

$$Cpp_{Conf}(SC) = Cpp_{Conf}(P{-}Cpp(SC)) \tag{1}$$

where Cpp_{Conf} means applying Cpp under a particular configuration $Conf$. Formally proving this equality requires the formal specification of the semantics of Cpp and P-Cpp, which is out of the scope of this article (the interested reader can find it in [9]). Succinctly, proving (1) by induction would proceed as follows: suppose $SC = L\ LS$ (a line L and a list of lines, LS), then

$$Cpp_{Conf}(L\ LS) = Cpp_{Conf}(L) + Cpp_{Conf}(LS)$$

that is, because Cpp is line oriented, it works line by line concatenating the output.

Base case: $SC = L$:

- if L is a preprocessor directive, $P\text{-}Cpp(L) = L$, so $Cpp_{Conf}(P\text{-}Cpp(L)) = Cpp_{Conf}(L)$.
- if L is not a preprocessor directive, Cpp and P-Cpp tokenize the output equally, except in the presence of macro calls that (in P-Cpp's case) may bind to multiple definitions and expand to a conditional directive with one branch for each possible expansion. One of these branches will

```
#if defined (__MSDOS__) || defined (__EMX__) || defined (WINDOWS32)
        if (instring == "" || !unixy_shell)
                ++p;
#else
        if (instring == "")
                ++p;
#endif
```

Figure 4. Code of Figure 3 after Cpp conditionals have been completed.

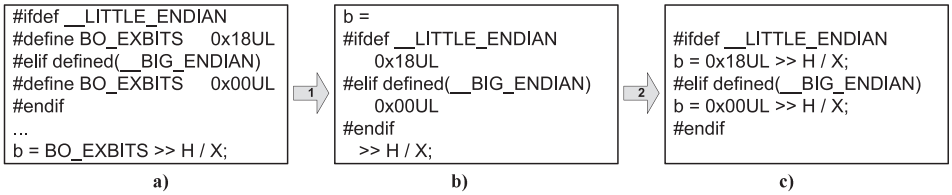

Figure 5. (a–c) Macro expansion (arrow 1) and conditional completion (arrow 2).

DOI: 10.1002/smr

have the same expansion that Cpp creates under *Conf*. When Cpp_{Conf} is applied on P-Cpp's output, the conditional directives will be removed except for the one corresponding with *Conf*, which results in the same output than $Cpp_{Conf}(L)$.

- The inductive case can be proven similarly. Moreover, the transformation that the CCA applies on conditionals is behavior preserving for all configurations [7].

2.2. Creating the program database

After pseudo-preprocessing, CRefactory still needs to construct the AST. For that reason, CRefactory's parser (CRParser) accepts an extended C grammar that integrates Cpp directives as terminal tokens and builds nodes in the AST to represent them. The integrated grammar is the standard C grammar [6] where five non-terminals were extended with an alternative that accepts a Cpp directive terminal: at the same level of statements and declarations, enclosing some structure fields, or a subset of values in an array initializer or an enumerator. Figure 6 shows two of these grammar productions, where controlLine is a terminal that represents a Cpp directive (see Appendix A for complete details of the integrated grammar). This grammar has been successfully used to parse the code we have used for testing (the Linux Kernel, the GNU library, GNU Make, etc.). It's important to note that this grammar is only used by CRefactory to construct the program database that allows refactoring of C together with Cpp code, but works independently of the compiler; that is, developers should still use their preferred compiler, and call CRefactory only as a refactoring engine (i.e., we are not changing the way C programs are compiled).

CRefactory's ASTs have standard C program nodes together with nodes that represent Cpp directives. Source comments are also preserved inside nodes with their exact position. Token labels are transparent to the parser but AST nodes inherit the labels of their tokens for analysis. The semantic analyzer obtains all the information about Cpp directives from the AST to construct the symbol table. On the other end of the process, CRefactory's pretty-printer reverses the manipulation performed by P-Cpp, so, except for refactored code, the end result is the original, unpreprocessed source code, with the exact same layout and comments. This means precisely that, given a source code *SC*:

$$CRPrettyPrinter(CRParser(P{-}Cpp(SC))) = SC \tag{2}$$

The next three sections describe, for the three main types of directives, more details of their role, the challenges that they cause during refactoring, and how CRefactory solves these challenges with the data structures it constructs for them.

3. HANDLING CONDITIONAL DIRECTIVES DURING REFACTORING

A preprocessor conditional defines separate code blocks that are included in the final program only under certain conditions. These conditions are evaluated by Cpp in the context of configuration settings (macro definitions) to choose the block of code that makes it into the final compilation unit [6]. A conditional directive is one of the following: #if, #ifdef, #ifndef, #elif, #else or #endif. Both #if and #elif take a constant expression as condition, and #ifdef and #ifndef take an identifier and check whether it has been defined as a macro. The #if, #ifdef and #ifndef directives start a Cpp

```
externalDeclaration
        : functionDefinition
        | declaration
        | controlLine
        ;

structDeclaration
        : specifierQualifierList structDeclaratorList <SEMI_COLON>
        | controlLine
        ;
```

Figure 6. Extended C grammar for external declarations and structure fields.

DOI: 10.1002/smr

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 source text inside a branch may include Cpp directives, so Cpp conditionals can also be nested.

As previously described, P-Cpp does not select a single configuration but all possible ones. Yet the user can specify 'false conditions' as input to CRefactory, to exclude some Cpp conditional branches that P-Cpp should not or cannot process (e.g., 'defined(*__cplusplus*)' or conditions used for commenting). Additionally, the user should identify 'incompatible conditions' to prevent semantic inconsistencies between declarations in different Cpp conditionals that cannot be true at the same time.

3.1. Challenges of conditional directives during refactoring

There are two issues that may arise during refactoring of C code with Cpp conditionals:

1. *Multiple definitions for a program entity*: Because we process multiple configurations simultaneously, a program entity may have multiple definitions (such as the case for variable 'pointer' in Figure 2). Thus, a use of a symbol may bind to one or several definitions. Depending on this, some refactorings may apply to a single definition (e.g., 'replace type'), whereas others may only be correct when applied to all definitions (such as the case of 'rename variable' in the example of Figure 2).
2. *Conditionals affecting code movement*: In refactorings that extract or move code around, problems may arise unless the following is ensured:
   - If the code being moved includes conditional directives, the code must include the whole Cpp conditional construct(s) to which the individual directives belong.
   - The code being moved should be placed under a condition *compatible* with the original one (C_1 is *compatible* with C_2 if the user did not marked as incompatible and it cannot be derived that $C_2 \Rightarrow \neg C_1$).

3.2. Data structures for conditional directives

The following data structures allow CRefactory to handle the previous challenges.

Cpp conditionals tree models the nesting of Cpp conditionals in a file. Each node of this tree is a Cpp conditional descriptor, which stores the condition plus the start and end positions of each branch. The conditional completion algorithm uses this information to complete Cpp conditionals and the pretty printer uses it to reverse pseudo-preprocessing. During refactorings that involve code movement, this tree is queried to check that the selected code includes complete Cpp conditional constructs (i.e., all branches). Moreover, the conditions stored in Cpp conditional descriptors are first-class objects that can be composed, analyzed and compared, for example, to check for compatibility of conditions during code movement.

Guarding condition. While tokenizing the code, P-Cpp labels each token with the current condition in terms of the nesting of Cpp conditionals that surround it. Note that the current condition is the conjunction among the condition in the current node of the Cpp conditionals tree and its ancestors. The nodes in the AST inherit the 'guarding condition' from their tokens [7].

Multiple definitions for a symbol table entry. Because a program element may have more than one applicable definition, the symbol table in CRefactory was enhanced to support multiple definitions for a given symbol, each one labeled with the guarding condition under which the definition applies (Figure 9). The lookup operation on this enhanced symbol table takes two arguments: a name and a condition. The name selects an entry in the table, and the condition selects among the possible definitions for the symbol. Moreover, given a symbol S with multiple definitions: $D_1, D_2, \ldots, D_n$, the semantic analyzer binds a use of S under condition Q to all definitions D_i under condition C_i for which C_i is compatible with Q. With this representation, the symbol table can easily be queried to find out if a refactoring is applicable to a single or multiple definitions of a symbol.

4. HANDLING MACROS DURING REFACTORING

Macros are defined through the #define directive, which associates a name with an arbitrary fragment of C code [6]. The name of a macro may be any valid identifier. The scope of a macro definition starts right after

DOI: 10.1002/smr

its #define and ends with the compilation unit. A macro may be undefined through the #undef directive followed by the macro name, to shorten its scope. Macros with parameters are called *function-like macros*. Furthermore, the replacement text of macros may use two special operators: the *stringification operator* '#' and the *concatenation operator* '##'.

When Cpp encounters a #define directive, it creates an entry in a *macro table* that associates the given name with its replacement text. Then, each time Cpp scans an identifier that appears in its macro table, Cpp replaces the identifier (and arguments if it has) by the replacement text after the appropriate argument substitution. The replacement text may in turn call other macros. Therefore, Cpp repeatedly rescans the macro expansion for more instances of macros to expand. While Cpp rescans the macro body, it checks for the occurrence of the '#' or '##' operators. When a macro parameter is immediately preceded by '#', Cpp converts the parameter name into a string constant. When a macro body contains a '##' operator, Cpp concatenates the tokens surrounding the '##'. See [3, 6] for a complete list of the complex rules of macro substitution and common pitfalls.

4.1. Challenges of macros during refactoring

In the presence of macros, the following issues may arise during refactoring:

1. *Extended scope of refactoring*: When a refactoring is applied to a C language entity, macro definitions may change if their body refers to that entity. For example, Figure 7 shows a function 'print_spaces', and two macro definitions in a different file that refer to it. Thus, refactoring the function signature should change all macro definitions that are called in the scope of the function definition.
2. *Different contexts calling the same macro*: Another problem that appears when a macro has unbound references to a program element is that there may be different definitions for that element depending on the context from where the macro is called. For example, Figure 8 shows a macro 'yyerror' that refers to a symbol 'yyerrstatus' and two functions that call 'yyerror', each with its own declaration of variable 'yyerrstatus'. If the variable is renamed in the context of only one of the functions (including the macro body), it will break the other function, that is, renaming is incorrect unless applied to all contexts that define the variable.

```
void print_spaces (unsigned int n) {   /* function definition in file misc.c */
…
/* macro definitions in file debug.h */
#define DBS(_l,_x) do{ if(ISDB(_l)) {print_spaces (depth); \
            printf _x; fflush (stdout);} }while(0)

#define DBF(_l,_x) do{ if(ISDB(_l)) {print_spaces (depth); \
            printf (_x, file->name); \
            fflush (stdout);} }while(0)
```

Figure 7. Function print_spaces and two macros that reference it in Make-3.82.

```
#define yyerror (yyerrstatus = 0)
...
int yyparse() {
  int yyerrstatus;
  …
  if (bottom < 0)
      yyerror;
  …
}

int main() {
  int yyerrstatus;
  …
  if (input == 0)
     yyerror;
  …
}
```

Figure 8. Macro 'yyerror' with calls from two contexts that define 'yyerrstatus'.

DOI: 10.1002/smr

3. *Use of concatenation in macro bodies*: The operator '##' can cause a replacement text to refer indirectly to a C language entity by concatenating two tokens (Figure 1). Thus, when renaming is applied to an entity *E* and there is a macro *M* called in the scope of refactoring that refers to *E* indirectly through concatenation, there is no safe way of modifying *M* so that it refers to the new name of *E* without affecting other calls to *M* and other results of the concatenation.
4. *Macros affecting code movement*: Macros can be defined, undefined and redefined at any point. Therefore, a refactoring that moves a piece of code (e.g., 'Extract Function') may not preserve behavior if any of the macros called from that piece of code bind to a different definition in the new location.
5. *Symbols as C entities and macros*: Extending the idea of Section 3.1, not only a program entity may have multiple definitions, but some of those definitions may be C language entities and others may be macros. This is very common in the case of functions and function-like macros (such as the example at the top left of Figure 9). Thus, refactoring a C language entity may cause refactoring on macros and vice versa.

4.2. Data structures for macros

Macro definition entry: When P-Cpp encounters a #define directive, it acts as Cpp creating an entry for the macro definition in its macro table. Unlike Cpp, P-Cpp creates a token for the #define line in the tokenized output (which will later go into an AST node), and the token has a reference to the macro definition entry in the macro table. An entry in P-Cpp's macro table contains data about the location of the macro definition, the location of its #undef if it has one and references to all calls to the macro. The information on locations is used during code movement, to account for the fourth challenge described previously. To handle the fifth challenge in the previous discussion, P-Cpp's macro table is actually a symbol table that holds entries for both macro definitions and for C language elements, each entry tagged by its guarding condition. Details about each C language element's definition is filled out by the semantic analyzer after parsing, but knowing that a symbol has different roles allows P-Cpp to expand a use of the symbol to a Cpp conditional construct with a branch per role. For example, Figure 9 shows at the top left a code extract with three definitions of symbol unblock_sigs and its associated symbol table entry.

Macro call object: As described in Section 2, macro calls need to be expanded for parsing to work. P-Cpp expands macro calls as Cpp does, but labels the tokens in the expansion with a MacroCall object. If a token comes from the expansion of nested macro calls, the token is labeled with all MacroCall objects in the sequence that created it, starting with the innermost macro call and ending with the outermost. This also causes the introduction of *layers* in the representation of token positions, that is, a position is a collection of relative offsets inside the nested containers of a token.

Macros in the AST: The token that P-Cpp creates for each macro definition becomes translated into an AST node, as shown in Figure 10 for macro 'ER1'. Moreover, the nodes in the AST that derive from a macro call obtain the MacroCall label from their terminals. As depicted in Figure 10, a MacroCall refers back to the AST subtree that represents its expansion or the smallest subtree that contains its expansion. Having these labels in the AST allows CRefactory to analyze refactoring preconditions directly on the tree and, after refactoring, allows pretty printing of the AST to reverse the macro expansion. For instance, to handle the first challenge presented in Section 4.1, if the AST nodes that require

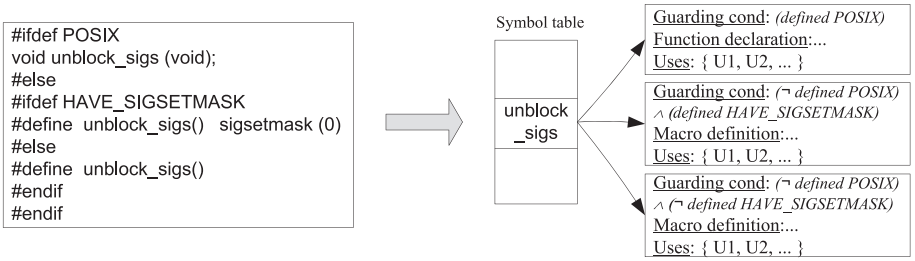

Figure 9. Code extract from Make-3.82 with three definitions for unblock_sigs() and its symbol table entry.

DOI: 10.1002/smr

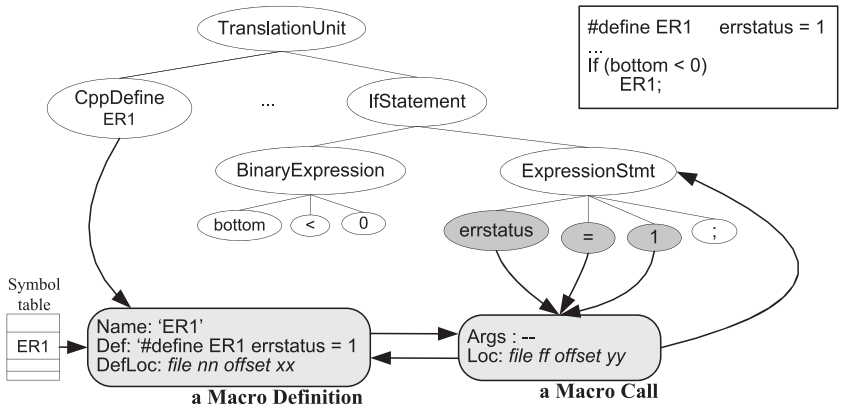

Figure 10. Abstract syntax tree for the source code in the top right, with macro definition node and macro call labels.

changes have a MacroCall label, the MacroCall object and its associated definition(s) are inspected to decide if a change is required on the macro call (if the changing entity was an argument of the macro call) or on the macro definition (if its body refers to the changing entity). However, if the macro definition uses concatenation on the name of a changing entity, the refactoring is aborted to solve the third challenge.

To solve the second challenge in Section 4.1, when a changing program entity has a MacroCall label in the AST, and the entity is not an argument of the call, the symbol table is used to inspect all macro definition entries to which the call binds, and all associated calls of those macros. If the context of any of the calls has a different definition of the changing entity, the refactoring cannot proceed.

5. HANDLING FILE INCLUSION DURING REFACTORING

The #include directive allows programmers to divide the program into smaller and more manageable parts, and ties common declarations together [6]. Included files are usually called header files. The behavior of Cpp with #include directives is to merge all files into a single compilation unit, which is not appropriate for a refactoring tool. Therefore, P-Cpp does not merge all files but creates separate representations for them. Section 5.2 describes these representations in detail. Maintaining separate representations of files brings new challenges that we present next.

5.1. Challenges of file inclusion during refactoring

The #include directive brings the following challenges:

1. *Calculating the scope of refactoring*: Changing a program entity correctly in the presence of #include directives depends on calculating the exact scope of the refactoring, that is, the visibility of the program entity. This is solved in CRefactory following the *Visibility Rule* [9]: 'A definition for a symbol *N*, located at position *P* inside a scope *S* and under condition *C* is visible:
   a) Inside *S* after *P*. If *S* is a local scope, *N* is not visible outside *S*.
   b) In inner scopes of *S* that do not contain a redefinition of *N*. A redefinition of *N* is a definition of *N* under the same condition *C*.
   c) If *S* is the global scope of a file *F* and *F* is a header file, *N* is visible in all files that include *F* directly or indirectly, after the file inclusion (i.e., including a file makes its definitions visible).
   d) If *S* is the global scope of a file *F*, *N* is visible inside the files that *F* includes after *P* (i.e., including a file makes all current definitions be visible for that file).
   e) In the four previous subrules, visibility is restricted to the pieces of code under conditions compatible with *C*.'

DOI: 10.1002/smr

2. *Different macro definitions depending on file inclusion order*: The Visibility Rule, which derives from Cpp's implementation of #include as recursive streaming of tokens instead of a module importation, shows that a definition is visible not only in the files that include the one with the definition but also in the files included after the definition. For example, Figure 11 shows a file 'B.h', which includes 'A.h'. Subsequent lines of code in 'B.h' may use any macros defined in 'A.h', such as the case of MAX_LFS_FILESIZE. Moreover, 'A.h' may also use the definitions that occur in 'B.h' prior to the #include 'A.h' line, as the case of macro BO_EXBITS, called by 'A.h' but defined in 'B.h' before the inclusion of 'A.h'. This situation is dangerous, because 'A.h' may also be included by other files, as the case of 'C.h' in the example, with a different definition of macro BO_EXBITS. There is a file 'X.c' that includes both 'B.h' and 'C.h'. Thus, the value of *var1* in the code of 'X.c' that follows the #include 'B.h' line is 0x18UL. However, the value of *var1* after the #include 'C.h' line is 0x00UL.

5.2. Data structures for file inclusion

Program file: Upon a #include directive, P-Cpp creates a token to represent the file inclusion line (which will go into its own AST node). Note that if the target of the #include is a macro, P-Cpp will expand it as usual and label the resulting token with the corresponding MacroCall object. Moreover, P-Cpp tokenizes each file separately. A file is represented through a ProgramFile object, which will contain its own token stream, its own AST and its own symbol table. Representing each file separately eases analysis, transformation and pretty printing. Because the source code that appears after the #include line depends on the declarations of the file being included, it is still necessary to process the included file before continuing with the rest of the code in the current file, just as Cpp would. For this purpose, the pseudo-preprocessor and the parser in CRefactory maintain a stack of the ProgramFiles being processed, and push a ProgramFile on the stack as they find #include directives. The tokens or nodes created at any point are added to the ProgramFile at the top of the stack.If a given file is included more than once in a compilation unit, Cpp processes the file completely each time. Instead, P-Cpp tries to reuse previous representations (which speeds up the process considerably). However, it needs to take care of the second challenge presented previously: macros may have changed from the previous inclusion of the file. Therefore, when P-Cpp finds a line '#include *X*', and *X* has been already tokenized (there is a ProgramFile for *X*), P-Cpp checks if the current definition of macros that *X* calls are the same as the previous time *X* was preprocessed. In that case, no further work is necessary, but if the macro definitions differ, P-Cpp adds the conflicting macro definitions to the macro table, where the guarding condition that labels each macro definition names the file that contains the definition. P-Cpp then tokenizes *X* again to reflect the new possible macro expansions.

Include dependencies graph: Because each file is represented separately, it is necessary to model the relationships or dependencies among files to handle the first challenge. A suitable representation of file dependencies is a graph, which we call 'include dependencies graph' (IDG). The nodes in the IDG are ProgramFiles, and there is an edge in the graph from ProgramFile *A* to ProgramFile *B* if *B* includes *A*. Edges are labeled with two elements:

- the position at which the file is included (necessary to calculate the definitions reaching a certain line of code);

```
/*** FILE A.h ***/
#define MAX_LFS_FILESIZE   0x7fffffffffffffff

int var1 = BO_EXBITS;
```

```
/*** FILE C.h ***/
#undef BO_EXBITS
#define BO_EXBITS      0x00UL

#include "A.h"
```

```
/*** FILE B.h ***/
#undef BO_EXBITS
#define BO_EXBITS      0x18UL

#include "A.h"
int var2 = MAX_LFS_FILESIZE;
```

```
/*** FILE X.c ***/

#include "B.h"
/* code that uses var1 */

#include "C.h"
/* other code that uses var1 */
```

Figure 11. Different definitions of a macro depending on the order of file inclusion.

DOI: 10.1002/smr

- the condition under which the file is included (because file inclusion frequently occurs inside a Cpp conditional, i.e., for certain configurations).

6. PUTTING IT ALL TOGETHER

This section shows how two refactorings deal with Cpp in their preconditions and mechanics, thanks to CRefactory's powerful program representations (previously described) and transformation engine: refactoring objects transform ASTs using an AST rewriter (a parse tree visitor that aims at creating the minimum disruption on code formatting) and create change objects to update the program database and log all changes.

6.1. Rename function

This refactoring replaces a C function name under all binding configurations. Moreover, if there is a function-like macro with the same name, which binds at the same calls, it must also be renamed. Input values are as follows: a string identifying the current function name, *oldName*, the user selected *position* in a *file* where this identifier appears, and the *newName*. The initial preconditions may be stated as

$$(file \in \text{IDG}) \wedge (file \text{ is writeable}) \wedge (\text{source}(file) \text{ has reference to } oldName \text{ at } position) \\ \wedge (newName \text{ is not visible in } file \text{ at } position)$$

that is, *oldName* appears at *position* in a writeable *file*, and there is no definition of *newName* visible at *position* (under the Visibility Rule). Additional preconditions are as follows:

$$(\text{user agrees to rename all } Di \in DEFS) \wedge (\forall\ S \in SCOPES : S \text{ is writeable})$$

where

DEFS = {all function and macro definitions named *oldName* visible at *position* that share uses with the function selected by the user}
SCOPES = {scope(*Di*): *Di* ∈ *DEFS*}

The mechanics appear in Figure 12.

Appendix B provides a proof of correctness of the algorithm in Figure 12. Furthermore, let's here review how the preconditions and mechanics of this refactoring account for the challenges engaged by the refactoring:

Conditional directives: Challenge 3.1.1 is covered in the preconditions, when calculating the set *DEFS* (which considers all definitions that share uses with the function selected by the user, including macro definitions) and assuring the user agrees to rename all definitions.

```
1. Set GCONDS = {guarding-condition(Di) : Di ∈ DEFS}        //the conditions of all binding configs
∀ S ∈ SCOPES : ∀ Node ∈ AST(S)
    If (Node is a match)              // (Node is function def ∨ function call ∨ macro def = oldName)
                                      // ∧ (guarding-condition(Node) ∈ GCONDS)
    Then
        If (macro-call(Node) = ⊥)                 // Node doesn´t come from macro-call
        Then
            2. Replace oldName by newName in Node.
        Else
            3. Set MC = macro-call(Node).         // The macro-call from where Node was derived
            4. Set MDEFS = {all macro definitions M : MC binds to M }
            If (∃ M ∈ MDEFS : (M body uses '##' on oldName) ∨
                    (the oldName in M binds to a D ∉ DEFS)
            Then 5. Exit & Rollback.
            If (Node is argument of MC)
            Then 6. Replace reference in argument of MC.
            Else 7. Replace oldName by newName in all M ∈ MDEFS.
```

Figure 12. Mechanics of 'Rename Function'.

DOI: 10.1002/smr

Macros:

- Challenge 4.1.1 is dealt with in Step 7 of the mechanics;
- Challenges 4.1.2 and 4.1.3 are accounted for in the condition of Step 5 of the mechanics. The presence of any of both problems invalidates the refactoring, but instead of checking them during the analysis of preconditions, they are more efficiently analyzed during the transformation itself. However, upon finding any of these problems it is necessary to exit the refactoring and roll back any changes (to preserve program's behavior, it needs to have a transactional semantics).
- Challenge 4.1.5 is taken care of when calculating *DEFS*.

File inclusion: Challenge 5.1.1 is covered in the preconditions when calculating the sets *DEFS* and *SCOPES* using the Visibility Rule.

6.2. Extract macro

With this refactoring the user selects a piece of code to extract into a new macro. The selection can be any list of tokens except for Cpp directives (but may include macro calls). Input values are as follows: the new macro name *MName*, the piece of code *E* to extract into the new macro, and the *filename* and *position* to place the new macro definition. The scope of this refactoring is calculated as $S = \{$all ProgramFiles *PF* in IDG: *MName* can be visible in *PF*$\}$. Preconditions are the following:

(Cpp directives $\notin E$) $\wedge$ ($E \in S$) $\wedge$ (S is writeable)
$\wedge$(*MName* is not visible in *S*)
$\wedge$(*position* is at statement level in *filename*)
$\wedge$(guarding condition at *position* is compatible with guarding-condition(*E*))
$\wedge$($\forall$ macro-call *MC* in *E*: bindings(*MC*) are preserved at *position*).

that is, *E* does not contain Cpp directives, and if *MName* is placed in *filename* at *position*, the whole scope where *MName* may be visible includes *E* and is writeable, there is no other definition for *MName*, *position* does not break a statement, and guarding condition and macro call bindings are preserved at *position*.

The mechanics appear in Figure 13. Note the power and flexibility of this refactoring: the macro can be placed in a widely used header file, increasing the chances of reusing it; also, the selection can be any list of tokens thanks to the AST rewriter's capability of matching incomplete subtrees by using 'wildcard' nodes. Yet it has simple mechanics: as opposed with the previous refactoring, all possible problems are analyzed by the preconditions, so it just creates the new definition and labels the corresponding nodes with a new macro call object. Afterwards, the pretty printer takes care of the rest (printing the macro call instead of the labeled nodes' tokens).

Because of the simplicity of this refactoring, we do not provide a proof of correctness, although we review how it solves the preprocessor's challenges it compromises:

Conditional directives:

- Challenge 3.1.1 is covered in the preconditions checking that *MName* is not visible in the whole scope *S* where it may be called.
- Challenge 3.1.2 is also covered in the preconditions when checking the compatibility of the guarding condition for the macro definition.

1. In symbol-table(*filename*), add entry *MDef* for macro definition of *MName*.
2. In AST(*filename*), add a MacroDefinitionNode *MDNode* pointing at *MDef* at place corresponding with *position*.
$\forall$ *N* $\in$ AST(*filename*) : *N* appears after *MDNode* in a depth-first-search traversal
    3. Set position(token(*N*)) += length(*MDef*).
4. Set *INST* = {instances of *E* with compatible guarding condition present in any AST(*PF*), $\forall$ *PF* $\in$ *S*}.
$\forall$ *T* in *INST* (which may not be a complete subtree) :
    5. Create a macro-call object *MC* pointing to *MDef*.
    $\forall$ *N* in *T* : 6. Add label *MC* to *N*.

Figure 13. Mechanics of 'Extract Macro'.

DOI: 10.1002/smr

Macros: Challenge 4.1.5 is taken care of when calculating the visibility of *MName* among C entities and macros.

File inclusion: Challenge 5.1.1 is accounted for when calculating the set *S* using the Visibility Rule.

7. EVALUATION

CRefactory is implemented in VisualWorks Smalltalk™ (Cincom Systems, Inc. Cincinnati, OH, USA). To load a program, CRefactory takes as input: the source files, including directories, read-only directories, command line macros, and false and incompatible conditions. It can handle GNU Cpp although it does not support GNU extensions. Because CRefactory is not a production system, we did not optimize it to make it fast, but we did test it on many C programs to make it correct. Because VisualWorks is cross-platform, we were able to test CRefactory in both a Linux platform and a Windows platform.

Table I shows three case studies that, although not large, make sufficient use of the preprocessor to test our solution: *Linux/init*, the init directory of the Linux kernel; *Make*, GNU utility for generating executables and GNU *Binutils* package. In the table, A) is the sum of the sizes of all .c and .h source files that CRefactory loads in the IDG for a package (the size of a file is counted once, even if it is included more than once); B) is the total number of Cpp conditional constructs of all files in the IDG; C) is the number of Cpp conditional constructs (from the total in B) that the CCA had to complete to make them parseable; D) is the percentage of source code growth after the CCA has completed all branches of incomplete conditionals; E) is, from the total of all symbols defined in a package, the percentage of definitions that are placed inside conditional directives (i.e., that depend on the Cpp configuration); and F) is the total number of symbols defined as macros in a package, for all possible configurations.

Let us first compare the case studies in terms of Cpp directives and how CRefactory's program representations and the CCA behave in each case. In terms of handling file inclusion, the value of reusing previously generated representations of programs becomes evident when we compare a program's total lines of code (LOC) number versus the LOC number actually processed by CRefactory when loading the program. For example, in Binutils, the total program LOC is around 750 KLines (which for instance includes 83 copies of <stdio.h> and 100 copies of <types.h>), whereas CRefactory actually processes 90 KLines when loading Binutils (each file is process only once).

Going from left to right in Table I, the proportion of incomplete Cpp conditionals becomes larger, and so is the code growth after completing those conditionals. Most incomplete conditionals are introduced by P-Cpp upon symbols that bind to multiple definitions and appear in the middle of expressions. Note that when many conditionally defined symbols appear in the same statement, the conditionals that P-Cpp inserts become nested and combined by the CCA upon completion [7]. The maximum nesting depth for these packages is 8. The large code growth in Binutils, which is the largest we have seen in testing (even in programs with a conditionals' nesting depth of 20) happens mostly in the MinGW library. Because it was loaded as read-only, it does not require further processing. Yet, pseudo-preprocessing the whole package and libraries takes less than 3 min in a 2.10 GHz dual-core machine. Moreover, adding less than five false conditions created 380 less incomplete Cpp conditionals and with that, the code growth was only 27%. Thus, it is important to discard discontinued configurations because it can drastically reduce the cost of conditional completion.

Let us now look at the power of CRefactory during refactoring itself. We can see from Table I that all packages have a large percentage of conditional definitions. Back in Figure 9, we showed an

Table I. Metrics while loading C code in CRefactory.

Metrics	Linux/init	Make	Binutils
A) Size (MB)	1.68	1.56	2.65
B) Number of Cpp conditionals	1988	3778	3592
C) Number of incomplete Cpp conditionals	918	1928	1999
D) Percentage of code growth after conditional completion (%)	13	32	47
E) Percentage of conditional definitions (%)	35	27	20
F) Number of macro definitions	8805	11956	3877

DOI: 10.1002/smr

example from package Make of the conditional definitions of symbol unblock_sigs. Not only there are shared uses of unblock_sigs among the three defining configurations but also among the configurations for seven possible platforms, involving a total of 21 possible configurations that use unblock_sigs. When we applied Rename function on unblock_sigs, it found the shared uses between the function and the macros and renamed all 3 definitions and 10 uses (addressing challenge 3.1.1). Any other solution leaving configurations out would permanently invalidate those configurations.

Another refactoring we applied is Extract Macro, selecting the code '($pid < 0$)' in file 'job.c' to create the macro FAILED_PID. The information in the program database indicated that placing the new macro in file 'job.h' would allow other two files that include it to use the macro. After selecting a position in 'job.h' outside any Cpp conditional (addressing challenge 3.1.2), the mechanics in the refactoring took care of replacing all eight matches of the text in three files by a call to the new macro.

Note that after the previous refactoring, the variable *pid* is unbound in the body of macro FAILED_PID. Thus, when we selected Rename on this variable, CRefactory replaced the variable name in all contexts that called the macro (handling challenge 4.1.2).

8. RELATED WORK

There are several papers that discuss the need to map the results of program analysis back to the unpreprocessed source code for program understanding tools. Livadas and Small [12] describes a preprocessor developed for the Ghinsu environment that captures several mappings between preprocessed tokens and their counterparts in the original source code. Their tool may then highlight the results of program analysis in the original source code. Cox and Clarke [13] present a much fine-grained approach where they mark up every character in the preprocessor output with the history of preprocessor replacements from which it was derived, by way of XML tags. The paper does not address the issues of representing the generated tagged output in program representations such as the AST. Kullbach and Riediger [14] present a similar approach where the preprocessor generates, besides the usual Cpp output, a representation that maintains original source coordinates, conditionals and macro replacements. Their program understanding tool called GUPRO can display the source code with unexpanded macros, or expand them one level at a time. The framework PCp^3 [15] allows the analysis of C source code with preprocessor directives by providing 'hooks' in the preprocessor or in the parser. The code is preprocessed but the user can define callbacks in the PERL scripting language, making use of those hooks in the preprocessor. PCp^3 therefore allows for more power in the analysis of Cpp directives than the previous approaches, although it is still not possible to have Cpp directives in the AST of a program. Latendresse proposes symbolic evaluation of Cpp conditionals, attaching to every line of code, the Boolean expression under which it is compiled [16]. We do the same but for every token in the code. However, his purpose is simplification of Boolean expressions.

Regarding transformation approaches, Xrefactory is a refactoring tool for C, C++ and JAVA that comes as a plug-in and allows transformations on source code including macros [17]. Their approach also consists of adding information to each piece of code about its position in the original unpreprocessed code [10]. In contrast with CRefactory, Xrefactory applies transformations directly on the source code, not in the AST, matching symbols by position. Because updating positions after a refactoring is too expensive in Xrefactory, it allows for a 'relaxed' mode of operation where positions are only approximate and refactorings may not be behavior preserving. To handle Cpp conditionals, Xrefactory proposes multiple passes over the source code, each one with different configuration settings [17]. Our analysis shows that in a large project, the number of possible configurations may be too large to make multiple passes feasible. The same problem is present in CScout [18], which performs an adequate analysis of identifiers in the presence of macros and include directives, but does not support multiple configurations (suggesting multiple passes over the code or ignoring conditional compilation commands). DMS's CloneDr can detect and extract duplicated code in large projects under multiple C dialects, but again, it does not support Cpp conditionals [19]. This situation persist in other tools such as Microsoft's Visual Studio and Eclipse's CDT, which support refactoring on a particular configuration (the code in other configurations is dimmed as a comment). In [20], authors recognize the problem of style disruption as a key for developers to reject refactoring tools; so similarly to our approach, they retain the positions of

DOI: 10.1002/smr

every literal and label the tokens that come from macro expansion. Unlike our approach, they do not treat Cpp directives as first-class entities, but as comments, so they do not propose refactoring for Cpp directives. They can create branches in the AST associated with different conditional branches, but report working with a single build at a time [20]. This work, as well as in [21], does not report the extra analysis required to deal with Cpp directives, nor the extra data structures needed for the analysis and transformation, and to the best of our knowledge, there is no other work that reports them.

9. CONCLUSIONS

We have presented a unique solution to handle the C preprocessor during program refactoring. The solution is also complete in that it handles all presented challenges of Cpp directives. We claim that any other approach that does not handle all listed challenges would fail to perform correct refactorings. Although we have not built a complete integrated development environment for our refactoring engine, this article shows how to build one, that is, what are the data structures that can support the required analysis and sound transformation of programs while preserving the unpreprocessed source code. The program representations, mainly symbol table and AST, integrate Cpp directives together with C language elements, thus treating Cpp directives as first-class entities that may also be transformed.

Future works include studying how to best visualize the information on the program representations, so that it may help a developer to understand the code better and suggest refactorings, evaluating the results on large programs.

ACKNOWLEDGEMENTS

The first author acknowledges the support of Universidad Abierta Interamericana, Argentina through grant agreement No. 2163 with CONICET.

REFERENCES

1. 10 Popular programming languages in IT companies – BCA HUB. http://bcahub.shareittips.com/4-bca-help/bca-stuffs/10-popular-programming-languages-in-it-companies/ [15 February 2012]
2. Garrido A, Johnson R. Challenges of refactoring C programs. Fifth International Workshop on Principles of Software Evolution (IWPSE 2002). IPSJ SIGSE and ACM SIGSOFT, Florida. 2002; 6–14.
3. Harbison SP, Steele GL, Jr. C, A Reference Manual (3rd edn) Prentice-Hall: New Jersey, 1991.
4. Ernst MD, Badros GJ, Notkin D. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering* 2002; **28**(12): 1146–1170.
5. Liebig J, Apel S, Lengauer C, Kästner C, Schulze M. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. ACM/IEEE International Conference on Software Engineering* – Vol. 1 (ICSE'10) Cape Town, South Africa 2010; 105–114.
6. ISO/IEC 9899:2011 – Information technology – Programming languages – C, 2011.
7. Garrido A, Johnson R. Analyzing multiple configurations of a C program. In *Proc. IEEE Int. Conf. on Software Maintenance (ICSM'05)*. Budapest, Hungary, 2005; 379–388.
8. Fowler M. Refactoring. Improving the Design of Existing Code. Addison-Wesley: Massachusetts, 1999.
9. Garrido A. Program refactoring in the presence of preprocessor directives. PhD Thesis. Univ. of Illinois at Urbana-Champaign, 2005.
10. Vittek M. Refactoring browser with preprocessor. In 7th European Conference on Software Maintenance and Reengineering (CSMR'2003). Benevento, Italy, 2003.
11. Overbey J, Johnson R. Generating rewritable abstract syntax trees. SLE 2008. LNCS **5452**, 2009; 114–133.
12. Livadas P, Small D. Understanding code containing preprocessor constructs. In IEEE 3rd Workshop on Program Comprehension. Washington D.C, USA, 1994.
13. Cox A, Clarke C. Relocating XML elements from preprocessed to unprocessed code. In *Proc. of IWPC 2002: International Workshop on Program Comprehension*. Paris, France, 2002.
14. Kullbach B. Riediger V. Folding: an approach to enable program understanding of preprocessed languages. Eighth Working Conference on Reverse Engineering. Stuttgart, Germany, 2001.
15. Badros G, Notkin D. A framework for preprocessor-aware C source code analyses. *Software Practice and Experience* 2000; **30**(8): 907–924.
16. Latendresse M. Rewrite systems for symbolic evaluation of C-like preprocessing. CSMR 2004; 165–173.
17. Vittek M, Borovansky P, Moreau PE. A collection of C, C++ and Java code understanding and refactoring plugins. *Proc. 21st IEEE Int. Conference on Software Maintenance - Industrial and Tool volume*. ICSM 2005; 61–64.

DOI: 10.1002/smr

18. Spinellis D. CScout: a refactoring browser for C. *Science of Computer Programming* 2010; **75**(4): 216–231.
19. C CloneDr. Semantics design. http://www.semanticdesigns.com/Products/CloneDR/CCloneDR.html [August 2012]
20. Waddington D, Yao B. High-fidelity C/C++ code transformation. *Electronic Notes in Theoretical Computer Science* 2005; **141**(4): 35–56.
21. Padioleau Y. Parsing C/C++ code without pre-processing. In *Proc. 18th Int. Conf. on Compiler Construction (CC'09)*, Berlin 2009.

APPENDIX A: Extension to the C Grammar

This appendix shows how we extended the standard C grammar in five nonterminals to support Cpp directives in these grammar productions: externalDeclaration, structDeclaration, enumeratorList, initializerList and statement. The only Cpp directives that in practice appear breaking other productions are conditional directives, which are repositioned by the CCA before parsing occurs. In the following productions, terminal tokens are denoted between angle brackets.

1. External declaration: A C program is a translationUnit composed of a list of externalDeclarations (functionDefinition or declaration). We add to these two options a third possibility, called controlLine, which is a Cpp directive:

```
externalDeclaration
        : functionDefinition
        | declaration
        | controlLine
        ;
```

2. Struct declaration: the standard C grammar defines a structOrUnionSpecifier containing a structDeclarationList. Each item in this list is a structDeclaration, to which we added the possibility of having a controlLine:

```
structDeclaration
        : specifierQualifierList structDeclaratorList <SEMI_COLON>
        | controlLine
        ;
```

3. Enumerator list: Similarly to the previous, an enumeratorSpecifier contains an enumeratorList, to which we added the possibility of having a controlLine as an element:

```
enumeratorList
        :
        | enumerator
        | enumerator <COMMA> enumeratorList
        | controlLine enumeratorList
        | enumerator controlLine enumeratorList
        ;
```

4. Initializer list: An arrayInitializer in the standard C grammar contains an initializerList between curly braces. We added to this list the possibility of having a conditionalDirective, which is the only controlLine we have found in an arrayInitializer:

```
initializerList
        :
        | initializer
        | initializer <COMMA> initializerList
        | conditionalDirective initializerList
        | initializer conditionalDirective initializerList
        ;
```


DOI: 10.1002/smr

5. Statement: Finally, a functionDefinition in the standard C grammar may also contain a list of statements. We added to this list the possibility of having a controlLine:

```
statement
        : labeledStatement
        | compoundStatement
        | expressionStatement
        | selectionStatement
        | iterationStatement
        | jumpStatement
        | controlLine
        ;
```

Following is the grammar for controlLine. In this case, we also show with the grammar, the expressions that create the nodes in the AST for each combination. Those expressions appear between curly braces and are written in Smalltalk. The expression 'lnodel' declares a variable, which is then assigned a new node constructed with the single token that represents the controlLine (denoted with the expression '1'). This node also receives whatever comments were scanned with the line. Note that all Cpp directives are represented with a single token created by P-Cpp, so that the parser is not confused with the line-oriented aspect of Cpp. The pseudo-preprocessor consumes the whole directive, including line splicing, and creates a single token for it that contains all the data and labels that define it.

```
controlLine: <INCLUDE>
                {|node| node := CRControlIncludeNode token: '1'.
                 node comments: scanner getComments}
        | macroDirective
                {|node| node := '1'.
                 node comments: scanner getComments}
        | conditionalDirective
                {|node| node := '1'.
                 node comments: scanner getComments}
        | <OTHER_DIRECTIVE>
                {|node| node := CRControlOtherNode token: '1'.
                 node comments: scanner getComments}
        ;

macroDirective
        : <DEFINE>
                {CRControlDefineNode token: '1'}
        | <UNDEF>
                {CRControlUndefineNode token: '1'}
        ;

conditionalDirective
        : <CONDITIONAL_START_IF>
                {CRControlConditionalStartIfNode token: '1'}
        | <CONDITIONAL_START_IFDEF>
                {CRControlConditionalStartIfdefNode token: '1'}
        | <CONDITIONAL_ELIF>
                {CRControlConditionalElifNode token: '1'}
        | <CONDITIONAL_ELSE>
                {CRControlConditionalElseNode token: '1'}
        | <CONDITIONAL_END>
                {CRControlConditionalEndNode token: '1'}
        ;
```

APPENDIX B: Proof of correctness of 'Rename Function'

We next demonstrate the correctness of rename function by proving the equivalence of the refactored AST with the AST that would result from applying an ideal 'rename function' on the preprocessed

DOI: 10.1002/smr

version of the same piece of code. In precise terms, if we assume that there is a correct version of 'rename function' over pure C code (called *CorrectRF*), such that

$$SC \xrightarrow{Cpp_{Conf}+Parser} AST_{Conf} \xrightarrow{CorrectRF} ReAST_{Conf} \xrightarrow{Pretty-printer} ReSC$$

where

- *SC* is a piece of source code;
- Cpp_{Conf} means applying Cpp under a configuration *Conf*;
- AST_{Conf} is the AST that results from applying Cpp_{Conf} and the standard C parser;
- $ReAST_{Conf}$ is the resulting AST after applying *CorrectRF*; and
- *ReSC* is the refactored source code after pretty printing,

and CRefactory's version of 'rename function' (*CR-RF*, which appears in Figure 11) is such that

$$SC \xrightarrow{PCpp+CRParser} AST_{CR} \xrightarrow{CR-RF} ReAST_{CR} \xrightarrow{Pretty-printer} ReSC_{CR}$$

where AST_{CR} is the tree that results from applying P-Cpp on the same source code *SC* and the CRParser, and the rest is similar to the former. Assuming a correct *pretty printer*, we will prove the correctness of *CR-RF* by proving the equivalence of the refactored AST with the result of *CorrectRF*, that is,

$$\boxed{ReAST_{CR} \overset{equivalent-under-Cpp_{Conf}}{\Longleftrightarrow} ReAST_{Conf} \quad (\forall\, Conf)}$$

where '*equivalent-under-*Cpp_{Conf}' means that applying Cpp_{Conf} on the pretty-printed version of each tree yields the same result, that is, $ReSC_{CR} \xrightarrow{Cpp_{Conf}} ReSC \qquad (\forall\, Conf)$.

The demonstration proceeds by induction on the number of nodes in AST_{Conf}.

Proof

Firstly note that the starting ASTs (the trees before refactoring) must be *equivalent-under-*Cpp_{Conf}, that is,

$$AST_{CR} \overset{equivalent-under-Cpp_{Conf}}{\Longleftrightarrow} AST_{Conf} \tag{B.1}$$

This is because of equality (1) stated in Section 2 and considering that the CRParser does not disrupt the output of P-Cpp when creating the AST.

Another important fact is that *CR-RF* is more conservative than *CorrectRF*, because it has extra conditions to handle the challenges posed by Cpp directives and to allow CRefactory to preserve them. Therefore, it is possible that *CR-RF* exits without refactoring the AST, even when *CorrectRF* may apply the refactoring. Therefore, we will actually prove the previous equivalence for the case that *CR-RF* applies the refactoring. Thus, in the proof, we assume that the condition for Step 5 in *CR-RF* is never true.

Base case

AST_{Conf} has a single node. Let us call it N_{Conf}. There are two possibilities for AST_{CR}: either it has also a single node or it has more than one.

Base case 1. AST_{CR} has a single node, let us call it N_{CR}: This means that there are no Cpp directives in the source code, and thus, neither in the tree. Then, by (B.1), $N_{CR} = N_{Conf}$.
Now, if N_{Conf} is a match for *CorrectRF* and so it becomes renamed, it must also be a match for *CR-RF*, because there are no conditional directives involved and no

DOI: 10.1002/smr

guarding condition that may discard N_{CR} from being a match. Moreover, N_{CR} does not have a macro call label (because there are no macro definitions) so it will be renamed in Step 2 of *CR-RF*. Similarly, if N_{Conf} is not a match, N_{CR} will not be a match either, so none of them will be renamed.

Base case 2. AST_{CR} has more than one node; then, by (B.1), the extra nodes can only be Cpp directives, or nodes for C code enclosed in different branches of a Cpp conditional construct (such that Cpp_{Conf} discarded them in AST_{Conf}). Let us analyze how the extra nodes may influence the behavior of *CR-RF* for each Cpp directive:

a. If AST_{CR} has nodes for a Cpp conditional construct, then by (B.1), there must one branch of that construct with a single node N_{CR} such that $N_{CR}=N_{Conf}$, and the guarding condition of N_{CR} (call it *GC*) belongs to the configuration *Conf*. For this reason, if N_{Conf} is a match for *CorrectRF*, then it must be true that $GC \in GCONDS$ (set in Step 1 of *CR-RF*), and thus N_{CR} is also a match for *CR-RF*. That is, both N_{Conf} and N_{CR} will be renamed. Similarly, if N_{Conf} is not a match, N_{CR} will not be a match either, so none of them will be renamed. If there are no Cpp conditionals in AST_{CR}, the extra nodes can only be other Cpp directives, and again by (B.1), there must be a single node N_{CR} such that $N_{CR}=N_{Conf}$.
b. If AST_{CR} has nodes for macro definitions that do not affect N_{CR} (i.e., N_{CR} does not have a macro call label), and N_{Conf} is a match and becomes renamed, N_{CR} will also be a match by 2.a and will be renamed in Step 2. Similarly, if N_{Conf} is not a match, N_{CR} will not be a match either, so none of them will be renamed.
c. If *NCR* has a macro call label *MC*, and suppose N_{Conf} is a match and becomes renamed, N_{CR} will also be a match by 2.a. Here, two cases are possible: (i) if *oldName* was an argument of *MC*, then the reference to *oldName* in *MC* is changed to *newName* in Step 6. When $ReAST_{CR}$ is pretty printed, the macro will have *newName* as argument, and Cpp_{Conf} will expand it to the same that in the pretty-printed version of $ReAST_{Conf}$; (ii) if *oldName* was not an argument of *MC*, then *oldName* must be in the body of all macro definitions to which *MC* binds, and in that case, the bodies of all those macro definitions will be updated to refer to *newName* in Step 7 of *CR-RF*. After $ReAST_{CR}$ is pretty printed and preprocessed by Cpp_{Conf}, *MC*'s expansion will have *newName*, equally to the pretty-printed version of $ReAST_{Conf}$.
d. If AST_{CR} has nodes for file inclusion directives under a guarding condition that belongs to *Conf*, then the included files must be empty (otherwise, AST_{Conf} would have more than one node). Thus, the file inclusion directive does not have any influence on the algorithm.

Because we have not set any restrictions on *Conf*, the former discussion is true for any *Conf*.

Inductive hypothesis: $ReAST_{CR} \overset{equivalent-under-Cpp_{Conf}}{\Longleftrightarrow} ReAST_{Conf} \quad (\forall\, Conf)$ when the starting tree AST_{Conf} has *N* nodes.

Inductive step: AST_{Conf} has *N* nodes plus one node that we will call N_{Conf}. Similarly to the base case, there must be a node $N_{CR} \in AST_{CR} : N_{CR}=N_{Conf}$. The proof for the case where there are no Cpp directives involved proceeds similarly to Base case 1. When AST_{CR} has nodes for Cpp directives, the proof proceeds similarly to Base case 2, except for 2.d, because now there can be file inclusion directives involving nonempty files. In that case, suppose we split AST_{Conf} in different trees ($AST_{Conf\text{-}i}$) that represent each file separately, in the same way that CRefactory splits the ASTs of each file. For each $AST_{Conf\text{-}i} \in AST_{Conf}$, there is by (B.1) an AST_{CR} such that $AST_{CR} \overset{equivalent-under-Cpp_{Conf}}{\Longleftrightarrow} AST_{Conf-i}$. Because each $AST_{Conf\text{-}i}$ has less than *N* nodes, the equivalency of the refactored trees holds by the inductive hypothesis. Moreover, joining the trees does not produce changes in them, so the equivalency holds also for the complete trees.

DOI: 10.1002/smr

AUTHORS' BIOGRAPHIES

Alejandra Garrido is an Assistant Professor at Facultad de Informática, Universidad Nacional de La Plata, Argentina, where she's a member of LIFIA (Research and Development in Advanced IT Lab). She is also a researcher at CONICET (Argentina's National Scientific and Technical Research Council). Her research interests include refactoring and Web engineering, focusing on design patterns, frameworks, refactoring for the C language, and refactoring for usability. Garrido has a PhD in Computer Science from the University of Illinois at Urbana-Champaign, 2005. She's a member of the Hillside Group. Contact her at garrido@lifia.info.unlp.edu.ar.

Ralph E. Johnson is a Research Associate Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He is a co-author of the influential computer science textbook Design Patterns: Elements of Reusable Object-Oriented Software. He was one of the originators of the software patterns movement. Johnson was an early pioneer in the Smalltalk community. He has held several executive roles at the ACM Object-Oriented Programming, Systems, Languages and Applications conference (OOPSLA). His current research interests are program transformation and parallel programming. Johnson has a PhD in Computer Science from Cornell University, 1987. Contact him at rjohnson@illinois.edu.

DOI: 10.1002/smr