Challenges of Refactoring C Programs

Alejandra Garrido
University of Illinois at Urbana-Champaign
1304 W. Springfield Av.
Urbana, IL 61801
1-217-333-5219
garrido@cs.uiuc.edu

Ralph Johnson
University of Illinois at Urbana-Champaign
1304 W. Springfield Av.
Urbana, IL 61801
1-217-244-0093
johnson@cs.uiuc.edu

ABSTRACT

Refactoring has become a well-known technique for transforming code in a way that preserves behavior. Refactorings may be applied manually, although manual code manipulation is error prone and cumbersome, so maintainers need tools to make automatic refactorings. There is currently extensive literature on refactoring object-oriented programs and some very good tools for refactoring Smalltalk and Java code. Although there is more code written in C or C++ than in any other language, refactoring tools for C with full support for preprocessor directives have not yet appeared.

The C programming language, especially the preprocessor directives that coexist with it, complicates refactorings in different ways as directives are not legal C code and may violate otherwise correct refactorings.

Refactoring C poses two major research challenges. On the one hand, as preprocessor directives may violate correctness, new precondition and execution rules must be defined for existing refactorings to preserve behavior. On the other hand, the automated execution of refactorings requires specialized program analysis tools to represent and manipulate preprocessor directives. After studying the area we have found some results to overcome these challenges and make a correct tool for the C language attainable.

This paper first discusses the difficulties in refactoring C code with preprocessor directives. It then defines preconditions and execution rules to maintain correctness of refactoring in the presence of macros and conditional directives. Moreover, new refactorings are proposed for macro definitions and conditionals. Lastly, the paper suggests enhancements to program analysis and program representation tools to correctly manipulate preprocessor directives.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *program editors*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *restructuring*; D.3.2 [**Programming Languages**]: Language Classifications – *C*; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors – *parsing, preprocessor.*

General Terms

Design, Languages.

Keywords

Refactoring. C programming. Preprocessor directives.

1. INTRODUCTION

Software systems must adapt to changing environments. Requirements are transformed or new ones are elicited. Designers are challenged to create and maintain highly reusable components that accommodate to evolving requirements. Code is often rewritten to reflect new functionality or new designs.

Unless changes are carefully incorporated, code that is constantly modified is in danger of becoming unmaintainable and buggy. The reasons that degrade the code are having different people working on the same code, adding changes fast but without elegance and without any documentation. Many systems still reach the point where they better be thrown away and rewritten from scratch upon a single change in requirements. In conclusion, companies are wasting money, time and resources and cannot keep up with good timing and quality. It is imperative that before incorporating changes in functionality, the code is reshaped and reorganized in ways that would make it more open to changes, easier to read and easier to maintain, in short, become more reusable.

Refactoring is about code evolution [14]. It allows improving the design of the code, making it more reusable and flexible to subsequent semantic changes. Moreover, refactoring is a disciplined process so that changes do not affect program behavior and consequently, tests are not violated. We consider that a refactoring does not affect or preserves program behavior when the versions of the program before and after refactoring are semantically equivalent. That is, the mapping of input to output values remains the same [14]. A typical example of a refactoring is the renaming of a variable. A complex example of refactoring is the application of a design pattern [13].

Our research group at the University of Illinois at Urbana-Champaign has been researching on refactoring for a long time ([14], [4], [16], [15], [7]). William Opdyke was the first to coin the term "refactoring" in his PhD thesis [14]. Turning research into practice, Don Roberts and John Brant built the first successful refactoring tool for the Smalltalk language [16].

Research on refactoring has spread, and well-known refactoring techniques have been catalogued as step-by-step recipes to help maintainers with a manual process [5]. However, most of the literature about refactoring is concentrated in object-oriented languages like Smalltalk or Java and transformations in the inheritance hierarchy ([5], [13], [18]).

Our goal is to build a refactoring tool for a non object-oriented language like C. A language like C calls for a different set of refactorings meaningful to its imperative, pointer & struct based nature and that considers its preprocessor directives. In an earlier work [7], we have proposed a catalog of refactorings for the C language that were implemented in a prototype tool. However, that catalog does not include refactorings for preprocessor directives. Those refactorings are the aim of our current research as presented in this paper.

Although catalogs of refactorings are helpful, manual code manipulation is error prone and cumbersome. A program of more than ten thousand lines of code may even turn manual manipulation impossible. Therefore, maintainers need tools to make automatic refactorings. Many people now think that refactoring tools are going to make a big impact in the software engineering process [6]. The model for refactoring tools is the Refactoring Browser [16]. It is integrated in the Smalltalk development environment and allows for powerful and fast refactorings that guarantee to preserve behavior. As Martin Fowler says in [6], not many people use Smalltalk, but "the Refactoring Browser makes one thing very clear. Tool support for refactoring is both possible and valuable." Tools for refactoring Java code are also improving. Good examples are jFactor 6 and IDEA [9].

We found two transformation tools that handle C and provide some level of preprocessor awareness. The tool Xrefactory from Xref-Tech [19] provides a limited support for preprocessor conditionals by allowing users to specify multiple pre-processing passings through source code [Marian Vittek, personal communication]. However, some refactorings are still based on a single preprocessor pass, so they cannot guarantee correctness. The reengineering toolkit DMS from Semantic Designs [2] supports the representation of macro definitions and macro calls, but not of conditional directives [Ira Baxter, personal communication].

Consequently, there are still no refactoring tools for C that handle preprocessor directives completely and correctly. We believe that full support for refactoring in the presence of preprocessor directives is extremely important. As analyzed in [3], the preprocessor is heavily used in C programs, as it provides many advantages that the article categorizes. Stroustrup also recognizes "Occasionally, even the most extreme uses of Cpp are useful, but its facilities are so unstructured and intrusive that they are a constant problem to programmers, maintainers, people porting code, and tool builders" [17].

There are many difficulties in refactoring C. One is pointer manipulation, which is not the focus of our research. The other is the presence of preprocessor directives, as they are not legal C code, they change scope and definition of program elements, and allow for special manipulations that may violate otherwise correct refactorings. We call correct refactorings those that preserve program behavior.

Refactoring C and preprocessor directives poses two major research challenges. On the one hand, since directives may violate correctness in refactoring, new precondition and execution rules must be defined for existing refactorings to preserve behavior. Moreover, new refactorings can be proposed for preprocessor directives. On the other hand, the automated execution of refactorings requires specialized program analysis tools so that preprocessor directives can be incorporated in program representations and be transformed.

This paper describes what we believe is the first serious attempt to refactoring C with preprocessor directives. The next section explains the problems posed by preprocessor directives in refactoring. Then, the special cases for refactoring macros and conditional compilation directives are presented, characterizing the problems and our solutions. Then, modifications to program analysis tools and program representations are proposed for the implementation of C refactoring tools.

2. WHY REFACTORING C IS DIFFICULT. PREPROCESSOR DIRECTIVES.

A typical C file uses not only terminals defined in the C grammar but also constructs called preprocessor directives. Preprocessor directives allow for file inclusion, macro definition and conditional compilation [11]. File inclusion allows having separate compilation units. Macros are defined with an arbitrary replacement text (usually not a complete or legal C expression), may have parameters and may use special operators, and gets substituted at every occurrence of the macro name. Conditional compilation provides a way to include code selectively depending on the value of conditions evaluated during compilation.

These directives use special commands and a rudimentary syntax that is understood and evaluated by a preprocessor. The C preprocessor takes as input a C file with directives and outputs pure C code, where directives have been stripped out and substituted accordingly. Therefore, parsing C does not involve dealing with preprocessor directives.

A refactoring tool for C can be constructed to perform transformations on preprocessed C code (that is, code without preprocessor directives). If this is the case, rules for behavior preservation of transformations are similar to those defined for other languages (e.g., the rules for renaming a variable in any language include modifying the variable name exclusively in its scope and namespace). Moreover, parsing and abstract syntax tree construction, the basic program analysis functions that a refactoring tool must perform, are standard and transformations deal only with the rules of the C language. This is the approach traditionally taken for C and C++ visualization and analysis tools (Star Diagram [8], Visualization of class diagrams [18]).

Tools that work on preprocessed C code have many disadvantages. The refactorings allowed are very restricted; otherwise, directives can be irrecoverable or may no longer apply. If directives are irrecoverable, the code may become unmaintainable. The tool may be very inconvenient for code visualization, as the code would look different than the source code. Therefore, the code will lack all the benefits of using preprocessor directives (as separate program units, reuse of macro definitions, use of #define constants instead of hard-coded values, etc.). Furthermore, such a tool cannot be integrated in a development environment where users may edit, debug, or perform refactorings interchangeably. Roberts and Brant claim that when they integrated the Refactoring Browser directly into the Smalltalk browser (the editor), the tool became useful [16].

A more powerful and useful refactoring tool would allow C programmers to transform C files from their original code, i.e., the code they wrote with preprocessor directives. This imposes new requirements on the refactoring tool:

- users should be able to transform C code that has interleaving preprocessor directives; e.g., renaming a variable in some code with macro calls;
- users should be able to transform preprocessor directives; e.g., adding a parameter to a macro definition;
- the presence of directives should not affect the correctness of refactorings.

Incorporating the ability to handle preprocessor directives in a refactoring tool is significantly difficult. An exception is file inclusion. As include directives change the scope of program elements declared global to a file, refactoring with #include directives means broadening the scope of refactoring accordingly. The preprocessor directives that present a real challenge are macro definitions and conditional compilation.

Macros can be defined and undefined, their bodies may reference global program elements and they may refer indirectly to global elements by using a special concatenation operator. With this, macros may easily violate the correctness of refactoring. For example in Figure 1, if the variable "errstatus" of "function1" is renamed but the body of macro "ERR" is not modified, the call to "ERR" in "function1" will cause an error of undefined variable. On the contrary, if the macro body is modified renaming "errstatus", the call to "ERR" in the function "main" will cause the error of undefined variable.

Conditional directives have alternative branches that are mutually exclusive. If during a refactoring a single branch is considered, other branches may become obsolete and erroneous. If all branches are considered, each program element may have multiple definitions, like a variable with multiple types.

These issues and some more that we discuss below call for a careful definition of the precondition and execution rules of each refactoring. Following the same approach as in [14], we discuss the preconditions under which a refactoring preserves behavior and we argue how the proposed execution of the refactoring does not alter that behavior. Although a formal proof of the validity of the proposed rules is out of the scope of this paper, the rationale behind each rule is carefully described and exemplified.

```
#define ERR  errstatus = 1

int function1() {
  int errstatus;
   ...
  if (bottom < 0)
      ERR;
   ...
}

int main() {
   int errstatus;
   ...
   if (function1() == 1)
      ERR;
   ...
}
```

Figure 1. Error in a rename refactoring

A major challenge in the construction of a C refactoring tool that deals with preprocessor directives lies in processing the code before refactoring: the standard C scanner, parser and abstract syntax tree builder no longer apply. The reason is that if code is not first preprocessed, it does not respond to the C grammar. In the Section "Enhanced program analysis process and program representations" we propose modifications to program analysis and its output data.

The next two sections discuss the circumstances in which we found macros and conditional directives to violate correctness in refactoring. They also present new preconditions and execution rules to reinstate and guarantee correctness.

3. REFACTORING OF MACROS

3.1 Macro Definitions

A macro definition has the form that appears in Figure 2. It calls for a simple macro substitution: subsequent occurrences of the token name will be replaced with the replacement_text [11]. The replacement text is arbitrary. The name may also include arguments and so each occurrence of the formal parameter in the replacement text is replaced by the corresponding actual parameter.

```
#define name replacement_text
```

Figure 2. A macro definition

Macro definitions do not tend to change much since they usually are general utilities, short pieces of code and highly reusable from their conception. Nevertheless, some refactorings that may apply to macro definitions include macro renaming, macro parameter renaming, macro parameter addition or removal (see Table 1).

The preconditions and execution rules for refactorings on macro definitions are similar to those that apply for function definitions. For example, in parameter renaming, the preconditions still include checking that the new name does not clash with the name of another program element in the scope. The execution involves renaming the parameter in its scope, but leave unmodified any other program element with the same name but a different usage (in Figure 3 below, if the parameter to rename is "x", the reference in "st.x" should not be renamed, since that "x" is a structure field).

```
#define M1(x, st)      x = st.x
```

Figure 3. A macro with different uses of the name "x"

There is a difference, however, from refactoring functions, with the definition of the scope. As macros can be undefined through #undef directives, the scope is restricted to the code from the #define up to the #undef. This has the consequence that subsequent definitions of the same macro are not transformed. A different approach would be to transform all definitions of the macro with the same name. In either case, the tool must make its transformation procedure very clear to the user.

Other refactorings related to macro definitions are: the creation of a macro definition replacing an expression in the code and the removal of a macro definition. For the creation of a macro, the user should be able to select the expression from the code that is to be turned into a macro, and the parts of the expression that should be parameterized. For example, suppose that in the code of Figure 4 a user chooses to create a macro from the statement that appears selected. The user is prompted for a name, say COND, and for the expressions to parameterize, which she chooses to be "c > 'a'" and "count". The result of the refactoring appears in Figure 5.

```
...
while (1) {
        getc(c);
        if c > 'a' goto count;
        else
...
```

Figure 4. Example of macro creation (before refactoring)

```
#define COND(E,P)    if (E) goto (P)
...
while (1) {
        getc(c);
        COND(c > 'a', count);
        else
...
```

Figure 5. Example of macro creation (after refactoring)

It is important to pay attention to the parentheses surrounding the parameters in the body of the macro (i.e., "(E)" and "(P)" in Figure 5). The absence of those parentheses may cause an error in subsequent substitutions of the macro, and therefore behavior would not be preserved. As a simple example from [11], suppose the expression from which the macro is created is "x * x", with parameter "x", and there is another matching expression "(z+1) * (z+1)" that gets replaced by a macro call with the result of the refactoring looking as in Figure 6. The substitution of "SQUARE(z+1)" will obviously not yield the same result as the original code.

```
#define SQUARE(A)    A * A

... SQUARE(x)

... SQUARE(z+1)
```

Figure 6. Error in the absence of parentheses

In the case of macro removal, uses of the macro are replaced with the macro's body. If the macro has parameters, this refactoring works similarly to "Inline function refactoring", in the sense that it replaces formal parameters by actual parameter expressions in every call.

Table 1 shows a list of refactorings that we propose for macro definitions.

Table 1. Refactorings for macro definitions

Rename macro
Rename macro parameter
Add parameter to macro definition
Remove parameter from macro definition
Add macro definition replacing value in code
Remove macro definition

The previous discussion proposes preconditions and execution rules for refactorings on macro definitions that can be summarized as follows:

- If the refactoring involves a macro name or its parameters, the refactoring must modify the macro definition and all calls to it, in the code or in other macro definitions (except for macro parameter rename where calls are unaffected).
- For the creation of macro definitions, the scope is circumscribed to the file where the expression is selected plus the files that include it. All the code in the scope, including the body of other macro definitions, needs to be searched and matched for the parameterized expression. In the body of the macro, every reference to a formal parameter must be parenthesized.
- For macro removal, the scope is defined as the file where the macro is selected plus all files that include it, until an #undef for this macro, if it exists. That is, subsequent definitions for the same macro after its #undef are not removed.

3.2 Code that Contains Macro Calls

When refactorings are applied on C code as opposed to preprocessor directives, macro definitions are likely to change if their body refers to global program elements. Therefore, when refactoring C code, the normal scope must be extended with the body of all macros that are called in that scope. Furthermore, our study shows that correctness can be jeopardized in three cases:

- if a macro is defined but never called in the scope of refactoring;
- if a macro refers to a variable with different declarations, and the macro is called from the different contexts of the variable;
- if a macro definition uses the concatenation operator ##.

In the case of a macro defined but never called, the problem can be elucidated by the code in Figure 7. There is a macro definition with name QUEUE that extends several lines. The body of the macro refers to a global variable "nelems".

```
#define QUEUE(q, ch)              \
  if (nelems < 10)                \
    { (q)->nelems = ch;           \
      nelems++;                   \
    }
```

Figure 7. Macro definition with global references

Suppose that there is a function that defines "nelems" as a local variable, and the user chooses to rename that variable. The macro "QUEUE" refers to "nelems" but as there is no call to "QUEUE" in the function, the tool has no way of parsing the body of the macro in the context of the call. Doing a string search on the body of the macro would find a reference to "nelems", but it cannot discern the use of "nelems". Actually, in the third line the macro refers to a different "nelems" that is a field of a structure pointed by "q". A string search cannot distinguish the reference to "nelems" in the second line from the reference in the third line. Therefore, there is no way of refactoring the macro. If the macro remains unchanged and is incorporated in the future, it will break the code. The problem can be compared to what happens when some piece of code is commented out and uncommented after refactoring: the code may break in the same way. To preserve correctness, a precondition must be added to "renaming" refactorings to check for possible references to the renaming element in the body of macros that are never called, if there is any. If there is a possible reference, conservatively the refactoring should not execute.

The second cause of errors in refactoring listed above also originates from global references in macros' body. Figure 8 repeats the example in Figure 1 that presented the case of a variable "errstatus" declared locally in two different functions and a macro "ERR" referring to that variable. In each macro call, the macro will make reference to a different definition of "errstatus". Consequently, the variable cannot be renamed. A precondition rule must be added to "renaming" refactorings to check if there is a macro that makes global reference to the element to be renamed. If there is, the scope of every call to the macro must be searched for a different definition of the element. If there is a different definition, the refactoring cannot proceed.

```
#define ERR   errstatus = 1

int function1() {
  int errstatus;
   ...
  if (bottom < 0)
       ERR;
   ...
}

int main() {
   int errstatus;
   ...
   if (function1() == 1)
       ERR;
   ...
}
```

Figure 8. Error in a rename refactoring

The replacement text of macros may use two special operators: # and ##. If a parameter is preceded by #, it causes the argument to be quoted. We did not find # to be the cause of any violation to correctness in refactoring. The second operator, ##, concatenates the adjacent tokens. With it, a replacement text may refer indirectly to a global program element. For example let us look at the code in Figure 9.

```
#define CAT(x, y)    x ## y

int main() {
  int tablesize;

   ...
  if (CAT(table, size) < 10) {
   ...
```

Figure 9. Use of the concatenation operator

Suppose the variable "tablesize" is selected to be renamed inside "main". A refactoring tool would not be able to find a direct use or reference to the variable in its scope or any macro definitions that are called from the scope. However, since there is a call to the macro CAT(x, y) in the scope of the variable, and the macro contains a concatenation operator, it is possible that the resulting value of that call matches the variable to be renamed, as it is the case in the example. For this reason, another precondition must be added to renaming refactorings, as well as to refactorings for adding or removing program elements, to first check if there is a call to a macro using concatenation in the scope of the element. If there is not, the refactoring can proceed safely. If there is, the refactoring may only continue if the user decides to expand macro definitions and an exhaustive search can take place.

The next section presents the issues arising from the other important type of preprocessor directives: conditional compilation.

4. REFACTORING CONDITIONAL DIRECTIVES

Conditional directives provide a way to include code selectively, depending on the value of conditions evaluated during compilation [11]. Conditional directives lines are those starting with #if, #ifdef, #ifndef and #elif, plus #else lines and #endif lines. The text in between each conditional directive line can be any material, including other preprocessor directives, or it may be empty. The standard C preprocessor evaluates the conditions and eliminates the text for which conditions are false, along with conditional directive lines.

There are no refactoring tools that can deal correctly with conditional directives. The usual approach is to preprocess conditional directives and therefore some code is discarded, so refactoring the end code causes transformations on partial pieces. When combined again, the code that was discarded for refactoring might not be valid. That makes refactorings incorrect because they cannot guarantee to preserve behavior. For example, if the code in Figure 10 is preprocessed and left with the first alternative, renaming variable "nelems" will make the other alternative invalid when recombined.

```
#ifndef _BUFFER
    #define _BUFFER
    int nelems;
    cqueue * q;
#else
    unsigned short nelems;
#endif
```

Figure 10. An example of conditional directive

The above paragraph leads to a new rule of correctness of refactoring in the presence of conditional directives: refactorings should be applied to all alternatives of a conditional.

In the same way that we propose additional refactorings for macro definitions, we propose some new refactorings for conditional directives, as listed in Table 2.

Table 2. Refactorings for conditional directives

Eliminate an alternative
Complete an statement inside a conditional branch with the code that follows the conditional
Move common code outside the conditional

A good use of "Eliminate an alternative" originates from refactoring legacy systems written for several platforms, some of which are no longer used. For example, if the VAX platform is discontinued, an alternative like "#ifdef _VAX" could be removed from all conditionals. To guarantee correctness, the scope of this refactoring should be all application files. Figure 11 shows the rules for transforming a conditional directive when an alternative is eliminated. The figure depicts these rules showing how the code in the left is transformed to the code in the right when eliminating the alternative for condition "X" with arbitrary text "aaaaa".

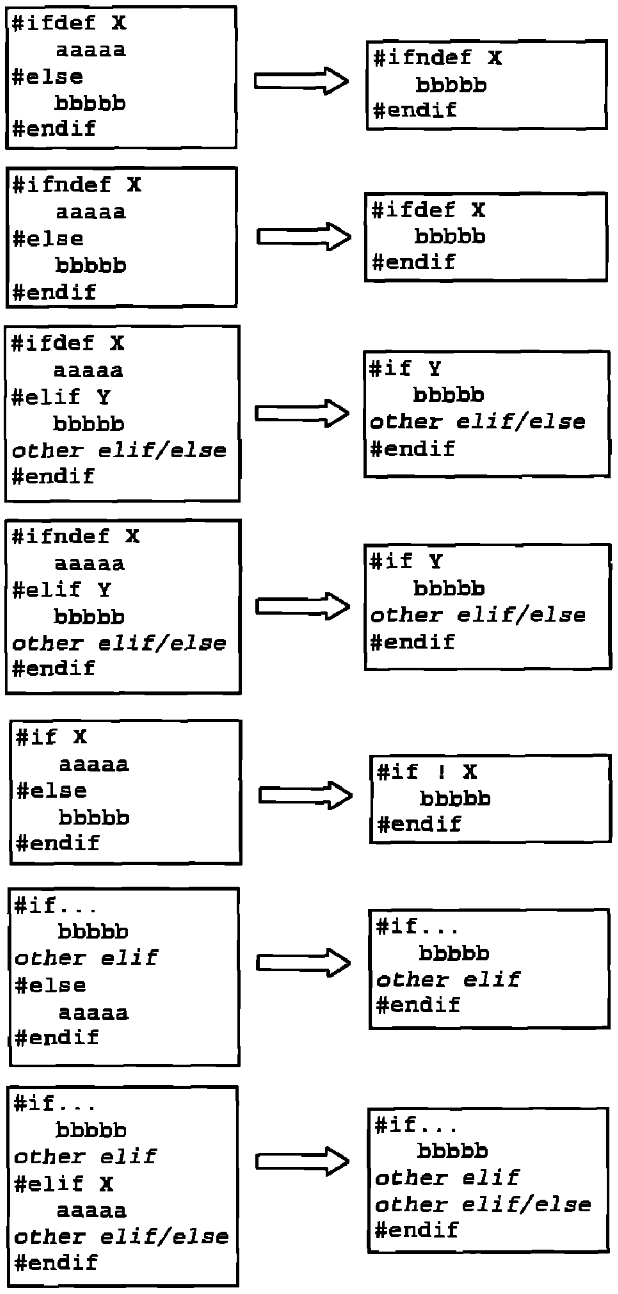

```
#ifdef X
   aaaaa
#else
   bbbbb
#endif
```
```
#ifndef X
   bbbbb
#endif
```
```
#ifndef X
   aaaaa
#else
   bbbbb
#endif
```
```
#ifdef X
   bbbbb
#endif
```
```
#ifdef X
   aaaaa
#elif Y
   bbbbb
other elif/else
#endif
```
```
#if Y
   bbbbb
other elif/else
#endif
```
```
#ifndef X
   aaaaa
#elif Y
   bbbbb
other elif/else
#endif
```
```
#if Y
   bbbbb
other elif/else
#endif
```
```
#if X
   aaaaa
#else
   bbbbb
#endif
```
```
#if ! X
   bbbbb
#endif
```
```
#if...
   bbbbb
other elif
#else
   aaaaa
#endif
```
```
#if...
   bbbbb
other elif
#endif
```
```
#if...
   bbbbb
other elif
#elif X
   aaaaa
other elif/else
#endif
```
```
#if...
   bbbbb
other elif
other elif/else
#endif
```

Figure 11. Rules for eliminating an alternative of a conditional

The second refactoring listed in the table originates from the fact that the text between conditional lines is not necessarily a complete C declaration or statement or sequence of them. By a "complete conditional directive" we mean one for which alternatives may be parsed in sequence without producing a parse error. That is, if the conditional directive keywords were not present, it would parse correctly. For example, Figure 12 shows an incomplete version of a conditional to the left, and its complete version to the right.

The "complete" version of a conditional directive looks more readable and is more flexible to changes. This refactoring allows completing the alternatives of a conditional so that each alternative is a syntactically complete piece of C code.

Figure 12 shows another feature of conditional directives: they allow alternative definitions of the same program element. In the example, the variable "var1" is defined of a different type in each alternative. Two new issues arise from this feature. Firstly, a refactoring like "replace type of program element" makes sense in a single alternative of the conditional, the one in which the program element is selected. Secondly, program representation tools must allow storing different definitions of a program element, but this issue is discussed in the next section.

```
#ifndef _C1
    #define _C1
    int
#else
    long int
#endif
var1;
```

Incomplete conditional

```
#ifndef _C1
    #define _C1
    int var1;
#else
    long int var1;
#endif
```

Complete conditional

Figure 12. Incomplete and complete versions of a conditional directive

Other elements that can have alternative definitions are macros, and this poses special problems. We consider alternative definitions of the same macro those that share the same macro name and the same number of parameters, although parameter names may not match. If a macro has alternative definitions, the refactorings "macro rename", "macro remove", "macro parameter addition" and "macro parameter removal" must be applied to all definitions of the macro to guarantee correctness. In the case of "parameter rename" or "parameter removal", the parameter at issue may be matched by position.

If alternative definitions of a macro exist with different number of parameters, refactoring cannot guarantee to preserve behavior, as the code may not work in the first place.

To motivate the third refactoring listed in Table 2, "Move common code outside the conditional", suppose we first refactor completing the conditional showed to the left of Figure 12, and then we change the type of variable "var1" in the #else branch of the conditional to be "int". Then, both branches of the conditional will contain the line "int var1;" at the end. In the same spirit as the refactoring "Consolidate Duplicate Conditional Fragments" [5], the common fragment of code can be moved outside of the conditional directive.

5. ENHANCED PROGRAM ANALYSIS PROCESS AND PROGRAM REPRESENTATIONS

The previous sections presented the problems that may appear with refactoring in the presence of macros and conditional directives and the considerations that must take place to guarantee correctness in refactoring. These considerations, like new definitions of the scope of refactoring, new preconditions and execution rules, the possibility of having more than one type for the same program element and even new refactorings, impose special requirements in refactoring tools for C. Tools need specific program analysis functions and program representations.

There are some tools that implement transformations in the code by string search & replace functions (e.g., the Moose Refactoring Engine for Java [12]). Although string search & replace can be effective sometimes, it cannot guarantee to preserve behavior. As a simple example, renaming cannot discern between different uses of the same name. Therefore, while these tools make code transformations, they do not perform refactorings. To guarantee correctness, parsing and abstract syntax tree construction are the minimum requirements.

The abstract syntax tree (AST) of a program contains enough information for many of the analysis functions required by the transformation engine of a refactoring tool. In some cases, more complex program representations are used, as program dependency graphs (PDGs). In [1], the authors employ PDGs to find uses of a variable. However, our research group believes that PDGs or similar program representations are complex and require considerable time to build [15]. We believe that for refactoring tools to be useful, speed is very important. ASTs contain sufficient information to implement powerful and fast refactorings, as demonstrated by the Refactoring Browser [16].

As mentioned under the Section "Why refactoring C is difficult", the standard C scanner, parser and AST builder no longer apply when directives are not preprocessed, as the code does not respond to the C grammar. Extending the C grammar with the syntax of directives is not possible. Alternatives of a conditional directive may not be complete statements so they cannot be represented by a non-terminal unit of the grammar. Moreover, macro calls may appear anywhere in the code and may represent any part of a statement or sequence of statements. From one side, listing all the cases where a macro call may appear would lead to an exceptional number of grammar productions. On the other side, it is not possible to parse a statement with a macro call without knowing what the macro represents. As a simple example, suppose the code in Figure 13. MACRO1 may contain the closing curly brace, in which case the 'else' in the code is paired with the 'if' on the first line, or the macro may contain another 'if' statement, in which case the 'else' would be paired with the 'if' in the macro. The parser cannot determine how to match the 'else'.

```
...
if (x > y) {
MACRO1
else
MACRO2
...
```

Figure 13. Unparseable piece of code with macro calls

Consequently, a specialized program analysis process is required. We propose an analysis process with changes and/or enhancementes to the parsing components: preprocessor, scanner, parser and AST builder. The job of the preprocessor will be integrated with the scanner so as to leave the parser as close as the standard C parser as possible. The grammar in the parser will only be enhanced with directive lines, that is, those starting with '#' The constructed ASTs will contain all the information of directives so refactorings do not need other complex program representations.

We envision a preprocessor-like component that will analyze the #include directives and conditional directives (without performing substitution). Analyzing #include directives provides an order of parsing the files so that a file is parsed after the one it includes. Moreover, storing the dependencies generated from #includes will allow latter on to compute the scope of refactorings global to a file.

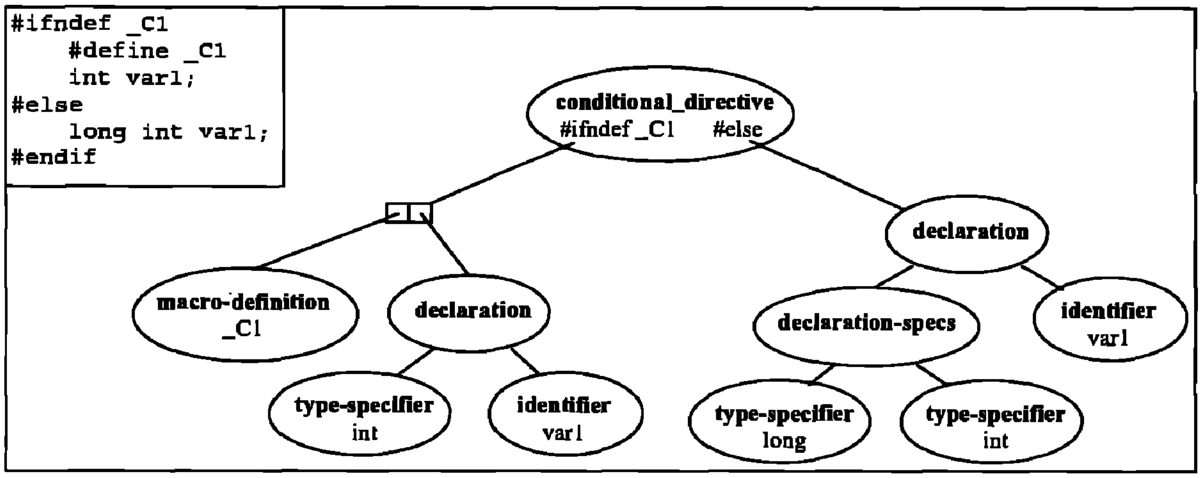
To deal with conditional directives, a possible approach would be to perform multiple passes on the code, one for each possible alternative. Then, a single file will have different ASTs. This poses two problems: first, the combinations of nested conditional directives can be exponential, and take a considerable amount of time to parse. Second, the trees need to be recombined for unparsing, and the work of combining a considerable number of trees can be very expensive. Since we are interested in fast refactoring tools, this is not an option.

Instead, we have found that if conditional directives are completed as discussed in the Section "Refactoring conditional directives", they can be incorporated in the grammar by expressing a conditional as a list of alternatives. In this way, a single pass will suffice to parse all the code. We plan on having the preprocessor-like component be able to recognize incomplete conditional directives and complete them accordingly. To recognize an incomplete conditional, the preprocessor needs to be able to separate C statements and recognize when a conditional directive starts or ends at a mid-statement. For the purpose of completing the conditional, it needs to keep track of various pointers to the text that should be moved and where.

Contrarily to conditional directives, macros cannot be completed since each call to a macro would provide a different ending. In order to parse macros, we propose having the scanner work as a filtering process so that the parser receives tokens corresponding to the C grammar. That is, when the scanner finds a macro identifier, instead of sending the identifier to the parser it will send the replacement text. Then, we need to reconstruct the original code (with macro calls) in the other end of the parser. For that purpose, the scanner will label each token that comes from a replacement text with the name of the macro that defines it. The parser does not need to look at the label but it will pass the information to the AST builder.

The AST builder will receive these labeled or decorated terminals and non-terminals and construct a colored AST. In this tree, nodes get colored depending on the type of preprocessor directive they come from. Some examples of node types are: "macro-definition", "macro-derived" (text replaced from a macro), and "conditional". Conditional nodes represent conditional directives and have a branch for each alternative. In this way, we can combine all alternatives of a conditional in the same AST. Figure 14 shows the AST resulting from parsing the code at the right of Figure 12 (the code is repeated in Figure 14 for easier reference).

Besides the enhanced AST, additional components of program representations like symbol tables, must allow for alternative definitions of the same program element (like we saw in the example of Figure 12, that a variable had two different types).



```
#ifndef _C1
    #define _C1
    int var1;
#else
    long int var1;
#endif
```

Figure 14. AST with nodes representing conditional and macro directives

6. CONCLUSIONS

Refactoring has been growing in importance and it is now recognized as a very important part of the software engineering process. Refactoring tools are getting better but there are no tools for refactoring C code with preprocessor directives, the reasons being the complexity to guarantee correctness in the transformations.

Some people use macros extensively in C and C++ programs, while others prefer a construction like templates in C++. In any case, there is a vast amount of code that uses macros and we have to provide tools that can refactor it.

This paper scopes the problems that we found where macros and conditional compilation directives may cause errors in refactoring. Although the list may not be exhaustive, our study shows that it covers the most important cases. The paper also provides new refactorings for preprocessor directives, new definitions of scope, additional preconditions and execution rules for existing refactorings. Moreover, special parsing components are proposed that we believe will solve the problems with program analysis and program representations that include preprocessor directives.

Although we have not looked into other languages, our approach may be applicable to other than C, where there is a preprocessor that works on a separate pass from the parser, and the syntax of the preprocessor is not integrated with that of the language. It should be noted, however, that our approach does not apply to C++ templates. The difference is that C++ templates are part of the C++ syntax.

7. REFERENCES

[1] Bowdidge, R. and Griswold, W. Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization. ACM Transactions of Software Enginnering and Methodology 7 (2), April 1998, 109-157.

[2] DMS Software Reengineering Toolkit. http://www.semdesigns.com/Products/DMS/DMSToolkit.html

[3] Ernst, M., Badros, G. and Notkin, D. An Empirical Analysis of C Preprocessor Use. Revision of Technical Report UW-CSE-97-04-06. Dept. of Computer Science and Engineering. Univ. of Washington, Seattle, 1999.

[4] Foote, B. and Opdyke, W. Lifecycle and Refactoring Patterns that Support Evolution and Reuse. Pattern Languages of Program Design I, Coplien and Schmidt *eds.*, Addison-Wesley 1995, 239-257.

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

[6] Fowler, M. Crossing Refactoring's Rubicon. http://www.martinfowler.com/articles/refactoringRubicon.html

[7] Garrido, A. Software Refactoring Applied to C Programming Language. MS Thesis. University of Illinois at Urbana-Champaign, 2000.

[8] Griswold, W., Chen, M., Bowdidge, R. and Morgenthaler, J. Tool Support for Planning the Restructuring of Data Abstractions in Large Systems. Proceedings of the ACM SIGSOFT'96 Symposium on the Foundations of Software Engineering (FSE-4). San Francisco, Oct. 1996.

[9] IntelliJ IDEA: Java IDE with refactoring support. http://www.intellij.com/idea/

[10] jFactor home page. http://www.instantiations.com/jfactor/

[11] Kernighan, B. and Ritchie, D. The C Programming Language. Prentice Hall. 1988.

[12] Moose Refactoring Engine. http://scgwiki.iam.unibe.ch:8080/SCG/19

[13] Ó Cinnéide, M. Automated Application of Design Patterns: a Refactoring Approach. PhD thesis, University of Dublin, Trinity College, 2001.

[14] Opdyke, W. Refactoring Object-Oriented Frameworks. PhD Thesis, University of Illinois at Urbana-Champaign, 1992.

[15] Roberts, D. Eliminating Analysis in Refactoring. PhD Thesis, University of Illinois at Urbana-Champaign, 1999.

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

[17] Stroustrup, B. The Design and Evolution of C++. Addison-Wesley, Reading, Massachusetts, 1994, p. 424

[18] Tokuda, L. and Batory, D. Evolving Object-Oriented Designs with Refactorings. In: Proceedings of Conference on Automated Software Engineering, Florida, 1999.

[19] Xref- Technologies - refactoring development tools. http://xref-tech.com/xrefactory/