18th IEEE International Conference on Automated Software Engineering (ASE 2003). Montreal, Canada. 2003.

Refactoring C with Conditional Compilation

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

Abstract

Refactoring, an important technique for increasing flexibility of the source code, can be applied with much ease and efficiency by using automated tools. There is currently a lack of refactoring tools for C with full support for preprocessor directives because directives complicate refactorings in many ways.

This paper describes refactoring of C programs in the presence of conditional compilation directives and how we propose to support them in a refactoring tool.

1. Introduction

Refactoring has become a well-known technique for transforming code while preserving behavior [6], [4]. However, research on refactoring has not been much related to procedural languages like C, even when there are many legacy systems written in C for which a refactoring tool would be extremely beneficial. One of the main problems that C poses for refactoring is the presence of preprocessor directives.

C programs depend on a preprocessor that provides file inclusion, macro definition and conditional compilation [5]. The C preprocessor (also known as "cpp") takes as input a C file with directives and outputs pure C code, where directives have been stripped out and substituted accordingly. Therefore, cpp directives are not part of the C grammar.

Analysis tools for C generally ignore the preprocessor. They apply their analysis to the output of the preprocessor, which, in the case of refactoring tools, is inappropriate. Programmers expect the results of a refactoring tool to still contain preprocessor directives. Therefore, refactoring tools cannot ignore the preprocessor. However, preprocessor directives are hard to handle for two main reasons: it is difficult to carry information of directives from the source code to abstract program representations and it is difficult to guarantee correctness in the transformations.

This paper concentrates on conditional compilation directives. Conditional directives allow defining separate code branches, which are included or excluded from the final compilation unit depending on the value of conditions evaluated by the preprocessor. If the code is preprocessed under a particular set of conditions, applying refactoring on the resulting code will probably leave the rest of the source code in an inconsistent state. This makes conditional directives complicated to analyze and poses an array of problems, which are presented in this paper, together with our solution.

Part of our solution is defining program representations that allow for incompatible conditional branches to be analyzed at the same time. These program representations conform our program model. Another part of the solution is to define new preconditions and execution rules for existing refactorings. We are implementing this solution in a refactoring tool called CRefactory and our goal is to test its feasibility and usability by refactoring the Linux kernel.

2. Enhanced program model with conditional compilation directives

A refactoring tool for C that supports conditional compilation directives must have a program model able to represent multiple configurations and analyze them together for refactoring. This is the only way that refactoring can guarantee behavior preservation. An example where refactoring a single configuration does not preserve behavior appears in the code taken from the Linux kernel shown in Figure 1. If this code is preprocessed assuming `BUILDING_PTY_C` is defined, renaming variable `unix_98_max_ptys` will cause a bug in the other branch, because the variable is defined in both branches.

On the other hand, there may be some conditional directive branches that are not meant to be parsed or may be not applicable, i.e., their conditions should be considered always false. Examples are "#if 0" or "#ifdef __cplusplus". We let the user be responsible for providing a list of conditions that should be considered always false. Then, conditional directives are analyzed considering that, except for those listed by the user, all other conditions are true.

```
#ifdef BUILDING_PTY_C
  void (*devpts_upcall_new)(int,kdev_t) = NULL;
  void (*devpts_upcall_kill)(int)       = NULL;
  unsigned int unix98_max_ptys          =
                    NR_PTYS * UNIX98_NR_MAJORS;
#else
  extern void (*devpts_upcall_kill)(int);
  extern unsigned int unix98_max_ptys;
#endif
```

Figure 1. An example of conditional directive

In addition to preserving behavior, a usable refactoring tool must be fast. Behavior preservation requires at least parsing and abstract syntax tree construction. The abstract syntax tree (AST) of a program contains sufficient information to implement powerful and fast refactorings, as demonstrated by the Refactoring Browser [7] and it does not require much time to build.

In the same spirit as the Refactoring Browser, our tool uses the AST of a program not only to perform analysis of the code but also to transform it. That is, refactorings are executed by first checking their preconditions and then manipulating the nodes in the abstract syntax tree to match the refactoring's result tree. Since our refactorings work on the ASTs, the trees must include information about conditional directives so we can refactor them and pretty print them. This section describes a novel approach to include conditional directives in the AST and other program representations of our model.

2.1. Parsing conditional compilation directives

The easiest way to have conditional directives represented in the AST is to extend the C grammar with them. However, conditional directives may appear anywhere, so extending the C grammar directly would require an enormous number of grammar productions. DMS restricts the places where preprocessor directives can occur so as to bind the number of productions [2], but we want to support as much cases as possible. On the other hand, we cannot afford to use an approximate grammar like LCLint [3] because refactoring needs exact information to assure correctness.

We could alternatively parse the files in multiple passes for each possible configuration, generating multiple ASTs for the same file. As far as we know, this is the approach used by Xrefactory [8]. However, having multiple ASTs for the same file requires recombining the analysis and pretty printing performed in each tree to perform refactorings. When the number of ASTs is large, recombining information from them would be very complex and expensive, two things that interactive refactoring tools must avoid.

Instead, we have found that conditional directives can be manipulated or transformed so that they appear at the same level as external declarations or statements in the C grammar. This transformation consists of completing the branches of a conditional directive with the text that precedes and/or follows the conditional, until each branch is a complete syntactical unit, as exemplified in Figure 2.

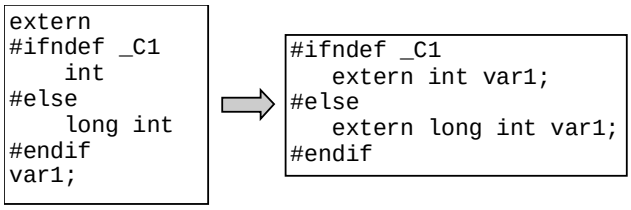

Figure 2. Incomplete and complete versions of a conditional directive

Therefore, if we complete conditional directives, we can add conditional directive to the C grammar at the same level of external declarations or statements. By doing this, we can parse incompatible conditional branches (those that could not otherwise be parsed together, as the ones in the left of Figure 2) in a single pass. Another advantage is that this approach generates a single AST for a file, avoiding recombining ASTs.

The transformation to complete a conditional must occur in the internal representation of the code in the program model, since our ultimate goal is to show the user exactly the same source code she wrote. We have built a pseudo-preprocessor component (that we call P-Cpp) that is responsible for recognizing incomplete conditional directives and completing them.

P-Cpp outputs a stream of characters that are labeled with their original position in the source code so the pretty printer may later reverse the transformation on conditionals, printing them in their original location. Note that the output of our pseudo-preprocessor will still contain conditional directives, as they will be parsed and included in the AST. P-Cpp also labels characters with the condition that guards them. When parsing takes place on the P-Cpp output, AST nodes will inherit the labels of the tokens they represent, which in turn inherit the labels of their characters.

2.2. Representing program elements with multiple definitions

Some conditions are semantically incompatible. In other words, when considered true at the same time, they generate conflicting definitions for the same program

element. Figure 3 shows an example simplified from the Linux kernel where the function `ntohl` is defined differently depending on the condition. Standard program representations are not prepared to allow more than one definition of the same program element.

```
#if defined(__KERNEL__) || defined (__GLIBC__)
extern __u32             ntohl(__u32);
#else
extern unsigned long int ntohl(unsigned
                                long int);
#endif
```

Figure 3. Multiple definitions of a function

When the definition of a program element depends on the configuration, our enhanced symbol table represents it by allowing multiple labeled branches to each of the element's definitions. The branches are labeled with the condition that guards the definition. In this way we solve the problem of semantically incompatible conditions.

Aversano, Di Penta and Baxter propose a similar symbol table with multiple entries depending on the configuration parameter values [1].

2.3. Handling multiple definitions of a macro

Semantically incompatible conditions also give rise to multiple definitions of the same macro name, which makes parsing very complicated. Figure 4 shows an example taken from [5].

```
#if SYSTEM == SYSV
       #define HDR "sysv.h"
#elif SYSTEM == BSD
       #define HDR "bsd.h"
#else
       #define HDR "default.h"
#endif

#include HDR
```

Figure 4. Multiple definitions of the same macro

If all conditions in Figure 4 are considered true, which definition of the macro is used for macro expansion in the #include of the last line? In all possible configurations, all three files should be included.

To solve this problem, P-Cpp manipulates the code once again and transforms it by inserting a conditional with the statement containing the macro call, with one branch for each definition of the macro. The example in Figure 4 is translated as it appears in Figure 5. P-Cpp labels the generated characters as "fabricated" so the pretty printer does not include them in the final code.

```
#if SYSTEM == SYSV
       #define HDR "sysv.h"
#elif SYSTEM == BSD
       #define HDR "bsd.h"
#else
       #define HDR "default.h"
#endif
#if SYSTEM == SYSV
       #include "sysv.h"
#elif SYSTEM == BSD
       #include "bsd.h"
#else
       #include "default.h"
#endif
```

Figure 5. Transformation to support multiple definitions of the same macro

3. Applying Refactorings

The problem of semantically incompatible conditions call for a careful definition of the preconditions and mechanics of each refactoring so that behavior is still preserved when conditional directives are present. In this section we discuss the new preconditions of two well-known refactorings: rename and extract function.

3.1. Rename refactoring

Renaming a program element is probably the best known and used refactoring. In C, the program element to rename can be a variable, a structure field, a function or a user-defined type. Macros can also be renamed but the preconditions and mechanics of macro renaming are different and out of the scope of this paper.

The standard preconditions say that the new name N shall not clash with any other symbol in the scope.

In the presence of conditional directives, the precondition must be enhanced because, as we discussed earlier, semantically inconsistent conditions bring multiple definitions of the same program element. Therefore, if a program element E has multiple definitions, the refactoring must check if the changes apply to the selected definition of E only or to more than one. Let us call this the *cardinality* of the refactoring.

The above leads to the following rule for renaming in the presence of conditional directives. Assume a user selects to rename a definition D_s of a program element E, where D_s is guarded by a condition C_s (this condition can be atomic or a conjunction of conditions of nested conditional directives). Assume there are M definitions of E, each one guarded by a different condition in the set $\Gamma = \{C_1,\ldots,C_M\}$ that includes C_s. The refactoring is safe

and may execute successfully in the presence of conditional directives if:

1. there is a single definition of E, i.e., $M = 1$, OR
2. if there are multiple definitions of E and each use of E is guarded by only one of the conditions in Γ, only the occurrences of E guarded by C_s are renamed, OR
3. if there are multiple definitions of E and each use of E is guarded by the disjunction of more than one condition in Γ, all occurrences of E are renamed.

3.2. Extract function refactoring

This is a complex refactoring which is considered very important for a refactoring tool to have. With it, a statement list L is extracted into a new function named N placed at a point P in the code.

The standard preconditions establish that the statement list must be convertible to a legal function, that the point P may hold a function definition and that the name N should not clash with any other symbol in the scope.

The meaning of "convertible to a legal function" must be enhanced to account for conditional directives. Moreover, an additional precondition is needed to check the condition that guards the point P where the extracted function is selected to go.

In the presence of preprocessor conditionals, a statement list that is convertible to a legal function is one that either has no conditional directives or contains a whole preprocessor conditional. For example, if the statement list L to be extracted is the piece of code in gray in Figure 6, since the selection includes only part of a preprocessor conditional, extracting the selection would turn both the remaining code and the extracted code incorrect in terms of preprocessor syntax.

```
int f1() {
        nelems++;
        #ifdef _C1
                q+= j;
                nelems -= q;
        #else
                nelems *= j;
        #endif
}
```

Figure 6. Incorrect selection for function extraction

The additional precondition must check that the condition C_1 guarding the point P where the new function goes, is the same as the condition C_0 guarding the selected code before refactoring, or that C_1 is a conjunction that includes C_0. If this precondition is not met, the code selected for extraction may, for example, use a variable defined in C_0 which is not defined in C_1.

4. Conclusions

Conditional directives are extensively used in C programs, so being able to handle them is an important step towards realizing a C refactoring tool. This paper presents our approach to handle conditional directives in the refactoring tool we are developing, CRefactory.

The techniques discussed in this paper are specific to the C preprocessor but not to the C language, so we believe they are applicable to other languages that use the cpp, as C++. Moreover, the program model that we propose may be useful to other than refactoring tools, for instance, analysis tools that may want to provide functions and metrics on conditional directives.

In the near future we expect to finish the implementation of CRefactory and use it extensively on large projects such as the Linux kernel to validate its usability and correctness.

5. References

[1] Aversano, L., Di Penta, M. and Baxter, I. Handling Preprocessor-Conditioned Declarations. 2nd IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM'02). Montreal, 2002.

[2] Baxter, I. and Mehlich, M. Preprocessor Conditional Removal by Simple Partial Evaluation. Workshop on Analysis, Slicing, and Transformation (AST) at the Eighth Working Conference on Reverse Engineering (WCRE'01)

[3] Evans, D. LCLint User's Guide. MIT Laboratory for Computer Science. Cambridge, MA, v2.2 edition, August 1996.

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

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

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

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

[8] Vittek, M. Refactoring Browser with Preprocessor. 7th European Conference on Software Maintenance and Reengineering (CSMR'2003). Benevento, Italy. March 2003.