A Foundation for Refactoring C with Macros

Jeffrey L. Overbey
joverbey@auburn.edu

Farnaz Behrang
fzb0012@auburn.edu

Munawar Hafiz
munawar@auburn.edu

Department of Computer Science and Software Engineering
Auburn University, AL, USA

ABSTRACT

This paper establishes the concept of *preprocessor dependences* as a foundation for building automated refactoring tools that transform source code containing lexical macros and conditional compilation directives, such as those provided by the C preprocessor (CPP). We define a *preprocessor dependence graph* (PPDG) that models the relationships among macro definitions, macro invocations, and conditional compilation directives in a file—the relationships that must be maintained for the semantics of the C preprocessor to be preserved. For many refactorings, a tool can construct a PPDG from the code before and after it is transformed, then perform a linear-time comparison of the two graphs to determine whether the refactoring will operate correctly in the presence of macros and conditional compilation directives. The proposed technique was implemented in OpenRefactory/C and tested by applying refactorings to GNU Coreutils version 8.21. Empirical results indicate that the technique is effective; it successfully handled refactoring scenarios in which Eclipse CDT, Visual Assist X, and XRefactory all refactored code incorrectly.

Categories and Subject Descriptors

D.1.2 [**Automatic Programming**]: Program Transformation; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

General Terms

Languages, Theory

Keywords

Refactoring, C, Preprocessor

1. INTRODUCTION

Automated refactoring has received widespread attention over the last decade. Refactoring support is included in most major integrated development environments, including Eclipse, Visual Studio, and Xcode. Tools for refactoring languages like Java, C#, and Smalltalk are reasonably robust. Unfortunately, the same is not true for C. In a previous work, we applied Eclipse's refactorings for Java and C to a number of open source codes; 1.4% of the Java test cases failed, while 7.5% of the C tests failed [10].

There are many factors that make C difficult to refactor correctly, including unrestricted pointers, vendor-specific language extensions, pragmas, and inline assembly code. But one seemingly simple aspect of the C language makes it deviously hard to refactor: the presence of preprocessor directives. The problem is that C has a *lexical* preprocessor, not a syntactic preprocessor: macros do not necessarily expand to complete syntactic constructs, nor do `#ifdef` blocks necessarily surround complete statements.

Building correct, complex refactorings is hard enough in languages like Java that do not have preprocessors. The literature has an abundance of work on improving the correctness of refactorings in such languages [21, 24–26]. Preprocessing adds yet another layer of complexity. Garrido [6, 8], in her seminal work on refactoring C code containing preprocessor directives, discusses many issues at length, including how to incorporate preprocessing information into an abstract syntax tree, how to refactor in the presence of `#include` directives, and the situations under which a refactoring can modify `#define` directives. She observes that code movement refactorings are sensitive to conditional directives and changes in the macro environment. However, she does not provide a uniform framework for handling these issues, instead assuming that refactorings will handle them on a case-by-case basis.

This paper introduces a program representation called a *preprocessor dependence graph*, or PPDG, that models the relationships between macro definitions and invocations, as well as the relationships between conditional compilation directives and the code affected by them. It also discusses how this graph can be used to establish the correctness of certain kinds of refactorings—notably, those that reorder, remove, or copy code within a file. Of the five most commonly used refactorings [20], three of them—EXTRACT METHOD, EXTRACT LOCAL, and MOVE—perform code movement and, therefore, must respect preprocessor dependences.

PPDGs integrate nicely into the "transform-then-check" paradigm for establishing refactoring correctness, which has been suggested by a number of authors in recent years [21, 24–26]. Rather than checking *a priori* whether the preprocessor dependence structure will be preserved by a transformation, a refactoring tool can simply build a PPDG prior to refactoring, then build another PPDG after refactoring, and compare the two. The differences between the two can be used to determine whether the transformation will cause any changes in preprocessor behavior.

Using PPDG comparison to analyze the effects of a transformation on preprocessing behavior has several advantages:

- *Modularity.* Central to our approach is determining whether a transformation preserves preprocessor dependences. This check is performed in the same way, regardless of what transformation is performed, allowing it to be written once and reused in numerous refactorings.
- *Separation of Concerns.* With our approach, developers constructing automated refactorings can, in many cases, focus on establishing *baseline correctness*—i.e., using name bindings, control flow, aliasing, etc., to ensure that the refactoring will preserve behavior in the absence of preprocessor directives. *Macro correctness*—guaranteeing that the code modifications will not inadvertently change the effects of the preprocessor under various macro configurations—can often be established independently, both in specification and implementation.
- *Testing.* As a side effect of the above two items, when refactoring engine developers test new refactorings, they can limit the number of preprocessor-related test cases that must be written. Since macro and conditional compilation constructs are handled uniformly, there is less need to exhaustively test each refactoring under arbitrarily complex combinations of macro definitions and conditional compilation constructs.

We implemented a preprocessor dependence preservation analysis and integrated it with three refactorings implemented in OpenRefactory/C [12]. We selected three refactorings for testing: Move External Declaration, Move Statement, and Extract Function. We automatically applied these refactorings on a large number of random but appropriate targets in GNU Coreutils version 8.21. We identified if the preprocessor dependence preservation analysis successfully prevented a refactoring from making a change that breaks the program. On average, 10% of the refactoring test cases were blocked by the preprocessor dependences preservation analysis; i.e., errors would be introduced if the preprocessor dependences were not considered. In our implementation, the overhead of the preprocessor dependence preservation analysis was minimal (3.2%), but it had a high impact on the correctness of refactorings.

During the process of testing Coreutils, we identified several common scenarios where the preprocessor dependence preservation analysis was utilized. We developed minimized programs representative of these scenarios and tested the refactoring capabilities of well-known C/C++ IDEs with these inputs. Surprisingly, all of the IDEs failed to block the refactorings, producing code that either did not compile or showed different behavior.

This paper makes the following contributions:

- It introduces the concepts of preprocessor dependences and preprocessor dependence graphs, or PPDGs (§§4–5).
- It describes sufficient conditions for refactorings to remove, copy, and reorder code in the presence of macros and conditional compilation, based on PPDGs (§6).
- It defines baseline correctness and macro correctness as separate concerns, and it discusses how comparing PPDGs before and after refactoring can be used to establish macro correctness in an automated refactoring tool (§7).
- It provides an empirical evaluation of the effectiveness and applicability of the technique (§8).

2. ASTS FOR C PREPROCESSED CODE

The C preprocessor (CPP) is a *lexical* preprocessor: preprocessing is performed on a token stream, not on syntactic constructs in the C language. Thus, at least conceptually, the preprocessor operates separately from the C parser. The original file is read, preprocessed, and then the resulting token stream is parsed. The grammar implemented in the C parser—the grammar including function and variable declarations, `while` loops, etc.—does not include macros, `#include` directives, conditional compilation (`#if`) directives, or any other preprocessor constructs. These must be eliminated by the preprocessor. In fact, it is practically impossible to incorporate preprocessor constructs into the C grammar. Preprocessor directives, such as `#include` and `#define`, can occur between any two adjacent tokens in the input. Macros can expand to completely arbitrary sequences of tokens. Moreover, `#if` sections can enclose arbitrary token sequences; they do not have to enclose entire expressions or statements.

Refactoring tools, static analysis tools, and others tools that analyze source code almost always use an abstract syntax tree (AST) as their program representation of choice, since (if the AST is designed appropriately) there is a direct mapping between AST nodes and source code. There are essentially two approaches to construct an AST from source code containing preprocessor directives.

The first approach is to preprocess the code exactly as a compiler would (i.e., configure the preprocessor with a particular set of macros and preprocess the code under that configuration), then parse the result to construct an AST. AST nodes are attributed to distinguish those that can be mapped directly to text in the source file from nodes that resulted from file inclusion or macro expansion. The Eclipse C Development Tools (CDT) use this approach, for example. Unfortunately, this approach "loses" information about the original code. For example, if the code contains `#if` directives, all but one branch will be ignored by the preprocessor.

The second (better) approach is to *pseudo-preprocess* the code [6–8]. A pseudo-preprocessor is aware of *all* feasible configurations. If a macro may have multiple expansions, it expands the macro in all possible ways; if the code contains `#if` directives, it handles all feasible branches. Moreover, it internally "expands" the regions enclosed by conditional compilation directives and macro expansions so that they enclose complete syntactic units in the C grammar (functions, statements, etc.). This allows these constructs to be incorporated into the phrase structure grammar and therefore included in the AST. When a macro may expand to several possible expressions, or conditional compilation may vary the statements in a function, all of these possibilities are included in the AST, and the corresponding nodes are attributed with information about the preprocessor configuration(s) governing their inclusion.

In either case, the AST represents the code *after* it has been preprocessed, and it contains just enough information to determine which nodes resulted from preprocessing and which nodes can be mapped directly to text in the original file. When a pseudo-preprocessor is used, the AST contains all feasible preprocessings of the code, but its structure reflects the code *after* preprocessing has been applied. Directives such as **#define** are typically omitted from the tree since they do not necessarily fit into the syntactic structure.

Therefore, ensuring the correctness of a refactoring in the presence of preprocessor constructs requires additional machinery that is not necessarily evident in the tree structure. Fortunately, for many refactorings—notably, those that move, remove, or copy code—there is a straightforward, uniform way to detect and prevent such changes in preprocessing behavior: using *preprocessor dependences.*

3. RUNNING EXAMPLE

Consider the following code segment extracted from GNU Coreutils version 8.21, copy.c, lines 227–238.

```
227 #ifdef __linux__
228 # undef BTRFS_IOCTL_MAGIC
229 # define BTRFS_IOCTL_MAGIC 0x94
230 # undef BTRFS_IOC_CLONE
231 # define BTRFS_IOC_CLONE _IOW (BTRFS_IOCTL_MAGIC, 9, int)
232   return ioctl (dest_fd, BTRFS_IOC_CLONE, src_fd);
233 #else
   ...
238 #endif
```

The **return** statement in line 232 uses a macro BTRFS_IOC_CLONE that is redefined in line 231, after being undefined in line 230. If a refactoring attempts to move the **return** statement anywhere above line 232, the macro will have a different definition (detected by *macro flow dependence*); consequently, the behavior of the program may change, or the program will not compile if the statement is moved between lines 230 and 231. Also, the **return** statement is conditionally compiled, guarded by the directive in line 227. Moving the statement below line 232 to be included in the **#else** part or outside the **#ifdef** part will change the conditions under which it is included in the preprocessed code (detected by *preprocessor control dependence*). Both macro flow dependence and preprocessor control dependence must be considered before allowing a code movement refactoring to proceed. We will use this program as a running example throughout Section 4.

4. PREPROCESSOR DEPENDENCES

4.1 Overview

The C preprocessor is oblivious to the syntactic structure of the program being preprocessed. In fact, the C preprocessor can even be applied to programs not written in the C language. For example, it is common for Fortran code to be preprocessed using CPP. From the preprocessor's perspective, a file consists of *parts.* A *part* is either a conditional compilation construct (**#if** or **#ifdef**), a control line (**#define** or **#error**), or a text line (a line not beginning with **#**).

Figure 1 is a grammar for a subset of the language accepted by the C preprocessor, based on the grammar in Section 6.10 of the ISO C99 standard [13]. The remainder of the technical content in this section will be given in terms of this grammar. In Section 5, we will discuss how our proposed technique can be extended in a straightforward manner to handle the full feature set of CPP, including function-like macros, token pasting, **#elif** clauses, **#error** directives, etc. (Such features do not change our technique in any fundamental way; they were intentionally omitted from the current presentation in order to convey the essence of our technique precisely while respecting space limitations.)

We assume there is a set I of identifiers and a set O of other tokens (including keywords, punctuation, etc.). The grammar of valid C preprocessor inputs is

$$\begin{array}{rcl}
\textit{file} & \rightarrow & \textit{parts} \\
\textit{parts} & \rightarrow & \textit{parts part} \mid \textit{part} \\
\textit{part} & \rightarrow & \textit{if-section} \mid \textit{control-line} \mid \textit{text-line} \\
\textit{if-section} & \rightarrow & \textbf{\#if}\ \textit{tokens} \hookleftarrow \textit{parts} \\
 & & \textbf{\#else} \hookleftarrow \textit{parts} \\
 & & \textbf{\#endif} \hookleftarrow \\
\textit{control-line} & \rightarrow & \textbf{\#define}\ \text{IDENTIFIER}\ \textit{tokens} \hookleftarrow \\
 & \mid & \textbf{\#undef}\ \text{IDENTIFIER} \hookleftarrow \\
\textit{tokens} & \rightarrow & \textit{tokens token} \mid \epsilon \\
\textit{text-line} & \rightarrow & \textit{tokens} \hookleftarrow \\
\textit{token} & \rightarrow & \text{IDENTIFIER} \mid \text{OTHER-TOKEN} \\
\text{IDENTIFIER} & \in & I \\
\text{OTHER-TOKEN} & \in & O
\end{array}$$

where tokens are taken from the set $T = \{$ **#if**, **#else**, **#endif**, **#define**, **#undef**, $\hookleftarrow \} \cup I \cup O$, where $O \cap I = \varnothing$.

Figure 1: Grammar for a subset of the language accepted by CPP. Newlines are syntactically significant, indicated by hooked arrows.

The essence of our proposed technique is as follows.

(1) We will define two dependence relations between *part* nodes in a parse tree constructed from the preprocessor grammar.

   (i) The *control dependence* relation relates a *part* corresponding to an **#if** directive to each *part* nested under the directive, i.e., the parts that are conditionally included or excluded based on the guarding condition.

   (ii) The *macro flow dependence* relation relates a *part* corresponding to a **#define** directive to every *part* node in which that macro may be used, either directly or indirectly via the expansion of another macro.

(2) We will show how these dependence relations can be used to ensure the correctness of refactorings that modify source code containing preprocessor directives.

To define the control dependence relation, we will define a *preprocessor control flow* relation on *part* nodes in a parse tree (Section 4.3). This relates each *part* to the next *part* that may be handled by the preprocessor. It is mainly intended to model the fact that only one of the *parts* in an **#if**/**#else**/**#endif** construct will be analyzed by the preprocessor, depending on the configuration in which it is run. Then, we will use a straightforward data flow analysis (*reaching macro definitions*) to construct the macro flow dependence relation (Section 4.5).

4.2 Parts: Indices and Forms

Starting from a parse tree constructed from the above grammar, we will define preprocessor dependences such that they relate one *part* node in the tree to another. So, for convenience, we will number the *part* nodes from 1 through n in the order they appear in a preorder traversal of the parse tree. We will call this integer the **index** of the part, and for a *part* p, we will denote its index by p.INDEX. Since every *part* node has a unique index, we will often treat the integers 1 through n and the *part* nodes of a parse tree interchangeably.

The parse tree for the running example program is shown in Figure 2, with part indices indicated.

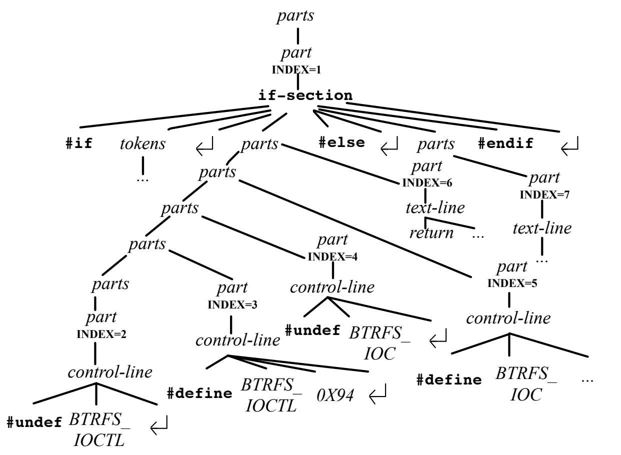

Figure 2: CPP parse tree for the running example.

From the grammar in Figure 1, a *part* can be one of three things: an **#if** construct, a control line (**#define** or **#undef**), or a text line. Hence, in a parse tree, the subtree rooted at a *part* node will have one of the four forms in Figure 3. We will refer to these as Form I, Form T, Form D, and Form U.

4.3 Preprocessor Control Flow

To define a control flow relation, we need two auxiliary definitions. The productions *parts* → *parts part* | *part* define sequences of *part* nodes. We will define functions *first* and *last* that identify the first *part* node and the last *part* nodes in such a sequence. As these are functions computed on the nodes of a parse tree, we use a straightforward notation to describe tree patterns; subscripts ($parts_0$ vs. $parts_1$) are assigned arbitrarily to distinguish nodes of the same type.

DEFINITION 1. *The function first maps each* parts *node to a* part *node as follows:*

$$first\left(\begin{array}{c} parts_0 \\ \diagup\diagdown \\ parts_1 \quad \mathrm{part}_2 \end{array}\right) \overset{def}{=} first(parts_1), \ \ and$$

$$first\left(\begin{array}{c} parts_0 \\ | \\ \mathrm{part}_1 \end{array}\right) \overset{def}{=} \mathrm{part}_1.$$

DEFINITION 2. *The function last maps each* parts *node to a set of* part *nodes as follows:*

$$last\left(\begin{array}{c} parts_0 \\ \diagup\diagdown \\ parts_1 \quad \mathrm{part}_2 \end{array}\right) \overset{def}{=} last'(\mathrm{part}_2), \ \ and$$

$$last\left(\begin{array}{c} parts_0 \\ | \\ \mathrm{part}_1 \end{array}\right) \overset{def}{=} last'(\mathrm{part}_1),$$

where if a part *subtree has Form I, then*

$$last'(\mathrm{part}_0) \overset{def}{=} last(parts_3) \cup last(parts_4),$$

and if a part *subtree has Form T, Form D, or Form U,*

$$last'(\mathrm{part}_0) \overset{def}{=} \{\mathrm{part}_0\}.$$

DEFINITION 3. *The control flow relation* **has-succ** *relates* part *nodes in a parse tree (via their indices, $1 \ldots n$) and is the smallest relation constructed by the following rules.*

1. *For each subtree* $\begin{array}{c} parts_0 \\ \diagup\diagdown \\ parts_1 \quad \mathrm{part}\ _2 \end{array}$,

$$last(parts_1).\text{INDEX } \overset{def}{\textbf{has-succ}} \ \mathrm{part}_2.\text{INDEX}.$$

2. *For each* part *of Form I,*

$$\mathrm{part}_0.\text{INDEX } \overset{def}{\textbf{has-succ}} \ first(parts_3).\text{INDEX}, \ \ and$$

$$\mathrm{part}_0.\text{INDEX } \overset{def}{\textbf{has-succ}} \ first(parts_4).\text{INDEX}.$$

THEOREM 1. *The* **has-succ*** *relation is a partial order on the integers $1, 2, \ldots, n$. Equivalently, the graph of the* **has-succ** *relation forms a DAG from the* part *nodes in a parse tree. Moreover, 1* **has-succ*** *p for every p, and the total order $1 < 2 < \cdots < n$ is a linear extension of* **has-succ****. That is, every* part *is reachable, and a preorder traversal of the parse tree will visit* part *nodes in an order that is compatible with their control flow ordering.* □

4.4 Preprocessor Control Dependences

In the compilers literature, a statement j is said to be *control dependent* on statement i (written $i\ \delta^c\ j$) iff j postdominates some successor of i, and j does not postdominate all successors of i [28, p. 71]. With the appropriate accommodations for entry and exit nodes, this definition can be applied to flow graphs constructed from the **has-succ** relation in the previous section.

However, the only control flow-like construct in the C preprocessor is **#if**, so we can use a more straightforward definition of control dependence instead—a definition that is more useful in a refactoring tool. We will define unconditionally compiled parts as having no control dependences, and conditionally compiled parts as being control dependent on the nearest enclosing **#if**.

DEFINITION 4. *A* part *with index j is said to be* ***preprocessor control dependent*** *on a* part *(of Form I) with index i (written $i\ \delta^c\ j$) iff there is a directed path*

$$part_0 \ \ if\text{-}section \ \ldots \ part_k$$

in the parse tree that does not contain any part *nodes other than* part_0 *and* part_k*, with indices such that* $\mathrm{part}_0.\text{INDEX} = i$ *and* $\mathrm{part}_k.\text{INDEX} = j$*. Where it is necessary to distinguish between the* if *and* else *branches, we will write $i\ \delta^c_T\ j$ if the path contains the node labeled as* parts_3 *in Figure 3 and $i\ \delta^c_F\ j$ if the path contains the node labeled as* parts_4*.*

In Figure 4, control dependences for the running example program are represented by edges labeled $\mathbf{c}_T$ and $\mathbf{c}_F$. (The graph and the edges labeled **f**, will be described next.)

4.5 Reaching Macro Definitions

Now, working from the definition of preprocessor control flow given previously, we will compute reaching definitions for macros. We define two auxiliary relations, **is-defined-at** and **is-undefined-at**, that relate an identifier to the (indices of) *part* nodes for **#define** and **#undef** directives that may define or undefine a macro with that name.

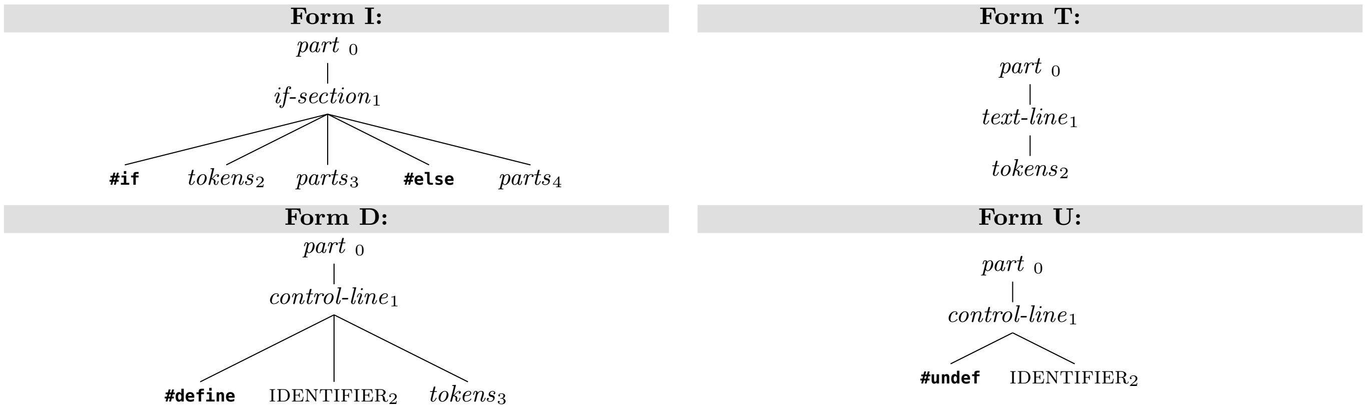

Figure 3: Four forms that a *part* subtree can assume (#endif and newline tokens omitted).

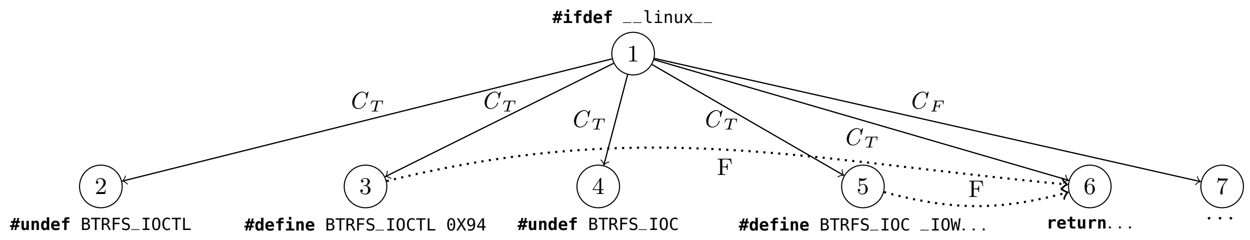

Figure 4: Preprocessor Dependence Graph (PPDG) for the running example.

Definition 5. *We define the three relations* **is-defined-at**, **is-undefined-at**, *and* **is-defundef-at** *as follows: for each subtree of Form D,*

$$\text{IDENTIFIER}_2 \ \overset{def}{\textbf{is-defined-at}} \ \text{part}_0.\text{INDEX};$$

for each subtree of Form U,

$$\text{IDENTIFIER}_2 \ \overset{def}{\textbf{is-undefined-at}} \ \text{part}_0.\text{INDEX}; \ \textit{and}$$

$$\textbf{is-defundef-at} \overset{def}{=} \textbf{is-defined-at} \cup \textbf{is-undefined-at}.$$

Definition 6. *Given a* part *with index j, the set of* ***reaching macro definitions*** *is given by RD(j), which is defined by the following data flow equations.*

$$IN(1) = \varnothing$$

$$IN(j) = \bigcup_{i \ \textbf{has-succ} \ j} OUT(i)$$

$$OUT(j) = GEN(j) \cup (IN(j) - KILL(j))$$

$$GEN(j) = \begin{cases} \{j\} & \textit{if the part with index } j \textit{ has Form D} \\ \varnothing & \textit{otherwise} \end{cases}$$

$$KILL(j) = \{i \mid j \ \textbf{is-defundef-at}^{-1} \ \textbf{is-defined-at} \ i\}$$

$$RD(j) = IN(j)$$

(The *KILL* equation is the least straightforward. Intuitively, it says: If a macro is defined or undefined by part j, then part j kills all definitions of that macro.)

Because the preprocessor control flow graph is a DAG and a preorder traversal of the tree is a linear extension of the control flow ordering (Theorem, Section 4.3), the sets of reaching definitions can be computed by a single, preorder traversal of the tree.

4.6 Macro Flow Dependences

Finally, we can relate occurrences of identifiers to **#define** directives that may define them as macros.

We will define a function *ids* that collects all of the identifiers in a *part* that may be expanded as macros. These include identifiers in *text-line* parts, as well as identifers in the condition of an **#if** directive and identifiers in the replacement token portion of a **#define** directive.

Definition 7. *Given a* part *with index i, let* $ids(i) \subseteq I$ *denote a set of identifiers as follows.*

- *If the* part *has Form I or T, then* $ids(i) \overset{def}{=} ids'(tokens_2)$,
- *If the* part *has Form D, then* $ids(i) = ids'(tokens_3)$, *and*
- *If the* part *has Form U, then* $ids(i) = \varnothing$,

where $ids'(tokens_0)$ *is the set consisting of every* IDENTIFIER *appearing in the subtree rooted at* $tokens_0$. *(Note that* $ids'(tokens_0)$ *includes both identifiers from the program text—e.g., variable names—as well as identifiers used as macros.)*

Now, we can use data flow information to compute a set of macro definitions that may be referenced during macro expansion. Per the grammar from Figure 1, we will presently be concerned only with object-like macros. (Function-like macros will be discussed in Section 5.)

Definition 8. *Given a* part *of Form I or Form T with index i, define refs(i) to be the least set of identifiers computed inductively as follows.*

- $ids(i) \subseteq refs(i)$.
- *If* $x \in refs(i)$, x **is-defined-at** j, $j \in RD(i)$, *and* $y \in ids(j)$, *then* $y \in refs(i)$.

For a part *of Form D or Form U with index i,* $refs(i) \overset{def}{=} \varnothing$.

Effectively, this computes the transitive closure of the identifiers that could be expanded as macros during the macro expansion process. The second bullet captures the notion that, if there is a **#define** directive that defines x as a macro, that macro definition could reach the current part, and some identifier y appears in that **#define** directive, then y could potentially be expanded as a macro as well. Also, note that when a **#define** directive contains other macros in its expansion, those macros are expanded at every *use*, not in the definition itself; thus, for Form D, the *refs* set is $\varnothing$.

Now, we can define macro dependences as follows.

Definition 9. *Suppose a* part *node has index j. For each $i \in RD(j)$, if x* **is-defundef-at** *i and $x \in refs(j)$, then* part *j is said to be* ***preprocessor flow dependent*** *or* ***macro flow dependent*** *on* part *i (written $i\ \delta^f\ j$).*

Finally, we can define a preprocessor analog of the program dependence graph [5,16], which we call the *preprocessor dependence graph*, or PPDG.

Definition 10. *Given a parse tree T, the* ***preprocessor dependence graph*** *$PPDG(T)$ is a labeled directed acyclic graph whose nodes are the* part *nodes of T and whose edges are defined as follows.*

- *If $i\ \delta^c_T\ j$, then there is an edge labeled $\boldsymbol{c}_T$ from the* part *with index i to the* part *with index j.*
- *If $i\ \delta^c_F\ j$, then there is an edge labeled $\boldsymbol{c}_F$ from the* part *with index i to the* part *with index j.*
- *If $i\ \delta^f\ j$, then there is an edge labeled $\boldsymbol{f}$ from the* part *with index i to the* part *with index j.*

The PPDG for the running example is shown in Figure 4. Flow dependences are also called read-after-write dependences. There are other types of dependences—for example, antidependences (δ^a) or write-after-read dependences, and output (δ^o) or write-after-write dependences. Defining preprocessor analogs of these dependences is a straightforward extension; they could be included in a PPDG as well. However, in our experience, they are unnecessary for the purposes of refactoring.

5. EXTENSIONS

As noted previously, the grammar in Figure 1 omits some C preprocessor constructs, and some simplifications were made in order to keep the presentation manageable. Our implementation (evaluated in Section 8) handles the complete set of C preprocessor constructs, as described in the ISO C standard [13]. In this section, we will discuss some of the simplifications that were made in the grammar of Figure 1 and how the definitions in Section 4 extend, in a fairly straightforward way, to the full C preprocessor grammar.

C refactorings do not necessarily reorder/remove/copy entire part *subtrees.* Note, however, that *text-line* parts are delimited by linefeeds, which are whitespace according to the C language grammar. So, the tokens on a single, physical *text-line* can be internally treated as multiple *text-lines* with finer granularity.

CPP allows nodes for parts *to have no children.* In the full CPP grammar, *parts* nodes can be empty, i.e., have no children. However, we required them to have at least one child *part*. This kept the definition of control flow concise. Extending the definitions of Section 4 to handle empty *parts* is not unlike handling empty if/else branches in an ordinary programming language. Alternatively, empty *parts* nodes can be internally populated with an empty *text-line* to more closely align with the definitions of Section 4.

Conditional compilation constructs are more complex. The full CPP grammar includes **#ifdef** and **#ifndef**; these are equivalent to `#if defined(...)` and `#if !defined(...)`. It does not require an **#else** branch and also permits **#elif** branches. These require straightforward changes to the definitions in Section 4.

There are additional control-line constructs: **#pragma**, **#error**, *etc.* These may have preprocessor control dependences, for example, and there may be additional constraints on the ways in which they can be modified. For example, **#line** directives may be nonsensical if they are moved arbitrarily within a file. Pragmas can be constrained too; `#pragma omp parallel` (which indicates an OpenMP parallel loop) can precede a **for** loop but not an **if** statement. Handling these correctly may require significant extensions.

The `__LINE__` *macro changes values if it is moved within a file.* In practice, this is a non-issue. Strictly speaking, any refactoring that may move an occurrence of `__LINE__` to a different line does not preserve preprocessor behavior. However, `__LINE__` is almost always used to display a source code line number in an error message; if a refactoring moves it, then the error message should display the new line number. Only in pathological cases (`if (__LINE__ == 3)`) would the semantics of a program depend in any critical way on the value of `__LINE__`.

CPP contains predefined macros. There are ISO-standard predefined macros such as `__FILE__` and `__DATE__`; many compilers add others (e.g., `__GNUC__`). CPP also allows macros to be defined on the command line (e.g., `cpp -DM=5`). In the reaching macro definitions data flow analysis, an implementation must modify $IN(1)$ to accommodate macros so defined. It must also accommodate macro flow dependences on predefined macros.

CPP has function-like macros, stringification, and token pasting. These only affect the *refs* set. Otherwise, they can be handled similarly to object-like macros. Due to space limitations, we will not attempt to precisely define a *refs* set for function-like macros, although the following example illustrates the key ideas.

```
#define f(a,b,c) a##b #c
#define gh i
#define i hello
f(g,h,world)
```

This code preprocesses to `hello "world"`. Its preprocessor parse tree contains 4 *part* nodes, where the macro invocation is the *part* with index 4. Here, $refs(4) = \{$`f`, `gh`, `i`, `hello`$\}$. This is computed as follows.

- From the macro invocation, the macro name (`f`) is in *refs*(4). The arguments (`g`, `h`, and `world`) are not included immediately; they will be added at a later point if they result from the expansion of a macro. (Macro arguments are not expanded at the invocation site—they are expanded later during the expansion of the function-like macro—so there is no reason to add them to the *refs* set immediately.)
- `f(g,h,world)` expands to `gh "world"`, so `gh` is in *refs*(4). The formal parameters `a` and `b` are not in *refs*(4), since they are local to the definition of the macro `f`.

- `gh` expands to `i`, so `i` is in *refs*(4).
- `i` expands to `hello`, so it is in *refs*(4).

Thus, the *part* for `f(g,h,world)` is macro flow dependent on all three of the preceding macro definitions.

6. REFACTORING

Preprocessor control and data dependences identify how *part* nodes affect conditional compilation and macro expansion in the preprocessor. Notably, the way in which a *part* is preprocessed is affected only by the transitive closure of its dependences. Given two *part* nodes with indices i and j, if it is neither the case that $i\ \delta^{c+}\ j$ nor that $i\ \delta^{f+}\ j$, then part i can be removed without affecting how part j is preprocessed.

In a refactoring tool, this observation has three essential applications: code removal, copying, and reordering.

6.1 Removal

There are a few refactorings that remove code from a file. One notable example is Safe Delete, which removes a function definition if the function has no known callers (it can be applied to other declarations as well). In C, where the function may contain preprocessor directives, we must impose an additional requirement: no parts outside the function to be deleted may be preprocessor dependent upon any parts inside that function. This would, for example, prevent the removal of a function containing a macro definition that is used after the function before being redefined. In general,

> A set of parts can be removed without affecting the behavior of the preprocessor if every preprocessor control or flow dependence on a part being removed emanates from a part that also being removed.

6.2 Copying

A number of refactorings can be implemented by copying parts of the user's code to another location. Consider a refactoring that takes a function definition and adds a forward declaration at the top of the file. If the function definition begins with

```
INT_TYPE func(INT_TYPE n) {
```

where `INT_TYPE` is a macro, then the user will presumably want the forward declaration declared identically. (Typically, macros are used in this way so that types can be varied at compile time, depending on the target architecture or compiler.) A similar situation occurs in the Extract Function refactoring, which moves a sequence of statements into a new function, replacing the statements with a call to that function: Extract Function must pass local variables as arguments to the new function, so if any local variables' types are declared using macros, then the new function's parameters should be declared using the same macros. Code copying also occurs in refactorings like Unroll Loop (a refactoring analog of the compiler optimization).

In all of these cases, a sufficient condition for correctness is the following:

> If a *part* q is copied to q', then it must be the case that q is control or flow dependent upon p if and only if q' is control or flow dependent upon p, respectively, and no statements are dependent upon q' in the transformed program.

In other words, q and q' have the "same" dependences, and the insertion of q' must not introduce any directives (e.g., **`#define`**) that might cause macro expansions to change in the transformed program. This condition is a bit more strict than necessary—for example, there are situations where duplicating a **`#define`** directive would be acceptable—but it is reasonable enough for the refactorings listed above.

6.3 Reordering

Perhaps the most important application of preprocessor dependences is the application to refactorings that move code within a file. In optimizing compilers, dependence analysis is primarily used in reordering transformations, so the application of preprocessor dependences to refactorings that reorder code is a natural analog. A refactoring known in the aggregate as Move is empirically one of the most-used refactorings in the Eclipse Java Development Tools [20]. In C, it has a number of variations as well: sorting **`#include`** directives alphabetically, moving function declarations closer to the functions that call them, reordering sequences of loop nests to enable fusion, etc.

Correctly handling preprocessor dependences is essential in transformations like these—transformation that relocate statements or declarations within a file. In the context of refactoring, there is an obvious analog to what Allen and Kennedy call the "Fundamental Theorem of Dependence" [15, p. 43]:

> If a transformation simply reorders the parts of a file (without adding or removing content), and the transformation preserves the relative order of the head and tail of every preprocessor dependence, then every *part* in the transformed file will be preprocessed in exactly the same way in which it was preprocessed in the original file.

This follows from the observation above: by preserving both preprocessor control and macro flow dependences, every statement will be conditionally compiled under the same conditions and every macro will be expanded in exactly the same way as it was in the original program.

If a refactoring simply reorders code, this provides an easy way to check for correctness: build the preprocessor dependence graph (PPDG) for the original program, transform it, build the PPDG for the transformed program, and then verify that the two graphs are isomorphic. Since the graphs are DAGs and, with careful implementation, it can be known which nodes in the original program correspond to which nodes in the transformed program, this verification can be performed by a single traversal of the two graphs.

7. IMPLEMENTATION

Here, we describe how a preprocessor dependence analysis can be integrated into a refactoring tool to ensure that source code transformations are correct even when the source code contains macros and conditionally-compiled code.

Refactoring tools use abstract syntax trees as their primary program representation. Name bindings, control flow, and definition-use relationships are all computed from (or mapped to) ASTs. Refactorings use these analyses to determine whether a transformation should be allowed to proceed. If a refactoring determines that a transformation will preserve behavior based on these static analyses—ignoring preprocessor directives—we will say that it has established *baseline correctness*. In an AST constructed from pseudo-preprocessed code, baseline correctness essentially guarantees that, if the transformation were applied to any preprocessed version of the code, it would be correct.

For the transformation to be correct when applied to the original code—code containing preprocessor directives—the refactoring must also establish that the transformation will not adversely affect the way the code is handled by the C preprocessor. Garrido [6, 8] discusses refactoring preprocessed code at length, including handling `#include` directives and modifying macro definitions. The scope of the present paper is narrower: we are concerned specifically with *macro correctness*, i.e., ensuring that the transformation will have no unintended effects on the macro expansion and conditional compilation behavior of the preprocessor.

For refactorings that reorder, remove, or copy parts of a preprocessed file, establishing macro correctness amounts to checking for the preservation of preprocessor dependences. We call this check a preprocessor dependence preservation analysis. It can be implemented in a component and reused by several refactorings.

We implemented such a component in OpenRefactory/C [12], an infrastructure for developing C refactorings, as follows:

- Before making any source code modifications, the preprocessor dependence preservation analysis constructs a PPDG for the file that will be modified by the refactoring.
- The refactoring determines what modifications will be made, notifying the preprocessor dependence preservation analysis of the changes so that it can establish a mapping between parts of the original code and parts of the modified code.
- If a refactoring adds new preprocessor dependences or removes existing preprocessor dependences, the preprocessor dependence preservation analysis informs the preprocessor dependence preservation analysis of the expected changes. For example, in the SAFE DELETE example above, it would be acceptable to delete a preprocessor dependence whose head and tail are both inside the function being deleted.
- The preprocessor dependence preservation analysis constructs a PPDG for the modified file.
- The preprocessor dependence preservation analysis compares the two PPDGs and determines whether there are any unexpected differences between the two graphs.

8. EVALUATION

To evaluate our technique, we implemented a preprocessor dependence preservation analysis as described earlier, integrated it into three refactorings, and applied the refactorings on open source software. We implemented the preservation analysis on the OpenRefactory/C [12] infrastructure as an extension to the C refactorings already available. Then we evaluated three refactorings: (1) MOVE EXTERNAL DECLARATION, which moves an external variable declaration, function declaration, or function definition within a translation unit; (2) MOVE STATEMENT, which moves statements to a new location in the same function; and (3) EXTRACT FUNCTION, which moves a sequence of statements into a new function, replacing the original statements with a call to that function. All of these perform code movement and thus are sensitive to preprocessor dependences.

Table 1: Execution results on GNU Coreutils 8.21

Refactoring	Tests	Step 1	Step 2	Step 3
Move Ext. Decl.	400	272	36	92
Move Stmt.	951	437	91	423
Extract Func.	1,046	46	113	887
$\sum$	2,397	755	240	1,402
Percentage	-	31.50%	10.01%	58.49%

* Step 1: Stopped by Baseline Correctness Preservation Analysis * Step 2: Stopped by Preprocessor Correctness Preservation Analysis * Step 3: Completed successfully

We tested the refactorings by running them on GNU Coreutils version 8.21[1] and collected data on the cases where the preprocessor dependence preservation analysis successfully prevented a refactoring from making an illegal change to the program. We identified common scenarios in which the preprocessor dependence analysis was empirically effective. We also tested other well-known refactoring IDEs to gauge their ability to refactor code under these scenarios. We asked three questions:

(1) **RQ1: Effectiveness.** Is a preprocessor dependence preservation analysis effective in ensuring that refactorings perform correct transformations on real-world software?

(2) **RQ2: Coverage.** In what kinds of preprocessor scenarios is a preprocessor dependence preservation analysis most commonly applicable, empirically?

(3) **RQ3: Comparison.** Can other refactoring IDEs successfully refactor codes illustrative of the scenarios from RQ2?

8.1 RQ1: Effectiveness

For our evaluation, we applied refactorings to GNU Coreutils version 8.21, which contains 195,267 lines of code. Coreutils contains the basic file, shell and text manipulation utilities of the GNU operating system; it consists of 108 files.

Following the approach used in our previous work on testing refactorings [10], our testing approach consisted of three main steps: (1) finding all the program elements where a given refactoring can be applied, running the refactoring on each of the elements, and recording the failures with associated message; (2) splitting the failures into clusters based on the recorded messages; and (3) manually analyzing the clusters to identify bugs. The first step is entirely automated; step 2 is semi-automated and step 3 is manual.

The existing refactorings in OpenRefactory/C used name binding and scalar dependence analyses to ensure correctness on unpreprocessed code (baseline correctness in the presence of macros and conditional compilation). We added a CPP preservation analysis to the refactorings to ensure macro correctness in the event that the baseline correctness checks pass. Refactorings were only permitted to proceed if both correctness checks passed.

Table 1 shows the execution results of applying the MOVE EXTERNAL DECLARATION, MOVE STATEMENT, and EXTRACT FUNCTION refactorings to a selection of random but appropriate targets from GNU Coreutils. For each refactoring, we recorded the total number of targets on which the refac-

[1]`http://www.gnu.org/software/coreutils/`

toring was applied, the number of times the refactoring was stopped by baseline correctness checks, the number of times that baseline checks passed but the refactoring was stopped by macro correctness checks, and the number of times that the refactoring was allowed to complete successfully.

For Extract Function, we extracted 1,046 statements out of which 113 (10.8%) were stopped by macro correctness checks. To make sure that the preprocessor dependence checks were not overly conservative, we extracted those 113 statements again with CPP preservation analysis disabled; all of the transformed codes either had compile errors or they changed the behavior of the original program. In total, 240 tests in all three refactorings that were successfully blocked by the CPP preservation analysis—on average, 10.01% for each refactoring. If these refactorings were allowed to make the change, most (237) would result in code containing compile errors; only three would have introduced behavioral changes. We detected behavioral changes by running `make test` both before and after refactoring.

To determine the overhead of running CPP preservation analysis, we recorded the time taken to run the 240 tests blocked by the CPP preservation analysis (Table 1) with and without this analysis. Enabling the analysis had only 3.2% overhead. We ran the experiments on an Intel Core i5-520M @2.40GHz with 4 GB of RAM.

8.2 RQ2: Coverage

We identified several common scenarios in which code movement refactorings may impact macro dependences and preprocessor control dependences, resulting in either a compiler error or a behavioral change in the transformed code.

8.2.1 Handling Macro Definitions

We identified five scenarios in which code movement is impacted by macro definitions. Four out of five may introduce bugs if preprocessor dependences are not considered.

1. **Move macro to undefined region.** A macro is moved with code to a region that does not define the macro. Consider this example from Coreutils file who.c:

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

Applying Extract Function on line 352 will move the use of the `DEV_DIR_WITH_TRAILING_SLASH` macro above line 341. In this case, the result is a compiler error.

2. **Move to same macro identifier with different replacement.** A macro is moved to an area in code in which the macro is defined with a different replacement. This does not cause a compile error, but the behavior of the code will change. Consider this example from GNU Coreutils file copy.c:

```
230 #undef BTRFS_IOC_CLONE
231 #define BTRFS_IOC_CLONE _IOW (BTRFS_IOCTL_MAGIC, 9,
    int)
232 return ioctl (dest_fd, BTRFS_IOC_CLONE, src_fd);
```

Applying the Move Statement refactoring on line 232 to move the `return` statement before line 230 will redefine the macro `BTRFS_IOC_CLONE`. The behavior of the program will change if the Move Statement refactoring is allowed.

3. **Change non-macro definition to macro identifier.** A non-macro definition is moved to a region in the code where it is captured by a macro definition with the same name. This results in compiler error after the refactoring. Consider this example from GNU Coreutils file make-prime-list.c when applying Move External Declaration:

```
 32 #undef malloc
     ...
160 static void *
161 xalloc (size_t s)
162 {
       ...
165    void *p = malloc (s);
       ...
169 }
```

`malloc(s)` on line 165 is defined as a function in stdlib.h. If `xalloc` function is moved above line 32, the program after refactoring will not compile since `malloc` will be redefined as a macro in that region of the code.

4. **Change macro identifier to non-macro definition.** A macro reference is moved to a region in code where a non-macro definition is defined with the same name. The refactored program will not compile; in some cases, the program will compile but demonstrate a different behavior. We did not find any example of this scenario while running test refactorings on GNU Coreutils. We show a minimized example:

```
1 int M = 0;
2 int main(){
3   #define M 3
4   printf("MACRO_defined_with_value_[%d]\n", M);
5   return 0;
6 }
```

If we apply Extract Function to move line 4 to a new function, the program will print 0 as the value of the macro definition. However, before refactoring this value is 3.

5. **Move without any change of macro definition.** The CPP preservation analysis allows a refactoring to proceed since it moves code that does not change the macro environment. The refactoring will preserve the original program's behavior and will not cause a compile error.

8.2.2 Handling Conditional Compilation Directives

Refactoring in the presence of conditional compilation directives may lead to two scenarios:

1. Move code that is guarded by a conditional compilation directive to a region outside the conditional.

2. Move code to a region guarded by a new directive.

Consider this example from GNU Coreutils file install.c:

```
802 #ifdef SIGCHLD
803 /*System V fork+wait does not work if SIGCHLD is
     ignored.*/
804   signal (SIGCHLD, SIG_DFL);
805 #endif
806 break
```

Table 2: Testing other refactoring engines

Scenarios (From Section 8.2.1)	Refactoring Engines			
	Eclipse	VAX	XRef	OpenRefactory
Undef. region	X	X	X	✓*
Same Macro Id	X*	X*	X*	✓*
Non-macro to macro	X	X	X	✓*
Macro to non-macro	X*	X*	X*	✓*
No change in def	✓	✓	✓	✓

X: Incorrect transformation with compiler error
X*: Incorrect transformation with behavioral changes
✓: Correct behavior by allowing refactoring to proceed
✓*: Correctly stops refactoring from continuing

Applying the MOVE STATEMENT refactoring on line 804 to move it before line 802 will result in a compile error since `SIGCHLD` is undefined in that region (Scenario 1). Moving the **`break`** statement on line 806 inside the conditional directive may change the behavior of the program (Scenario 2).

8.3 RQ3: Comparison

The refactorings implemented on OpenRefactory/C that are supported by the CPP preservation analysis correctly handle the scenarios identified in Section 8.2 that may lead to errors. We wanted to test if other refactoring engines handle these scenarios correctly. We tested three well-known C refactoring engines: Eclipse CDT, Visual Assist X (VAX), and XRefactory. CDT adds C and C++ support to the Eclipse IDE; VAX is a plugin for Microsoft Visual Studio; and XRefactory is a plugin for Emacs, xEmacs, and jEdit. We used VAX running on Visual Studio 2008 and XRefactory version 2.0.14 running on Emacs version 24.1.

We compared only EXTRACT FUNCTION, since the other two refactorings are not supported by the refactoring engines under test. For each of the three scenarios in Section 8.2.1, we wrote a minimized example and manually tried to execute the EXTRACT FUNCTION refactoring.

Table 2 shows the results. All of the refactoring engines except OpenRefactory/C failed to prevent EXTRACT FUNCTION from breaking the program after the refactoring. In the first and the third scenario, the refactored program failed to compile. The program behavior changed in the second and the fourth scenario. The fifth scenario was successfully handled by all refactoring engines.

9. RELATED WORK

Dependences have a long history, being universally used in optimizing compilers (e.g., [5, 15, 16, 19, 28]), but they also have history specifically in the refactoring/restructuring literature. Griswold's dissertation [11], which introduced one of the first code refactoring tools, used PDGs as its program representation. Schäfer [24–26] has argued that a broader concept of dependence, which includes relationships like name bindings and relationships among `synchronized` entities in Java code, provides a more concise and reliable means to specify and implement refactorings.

One of the earliest papers on handling the C preprocessor in a software maintenance tools is due to Platoff et al. [23]. Badros and Notkin [2] describe an infrastructure for handling the C preprocessor in static analysis tools, which invokes user-defined callbacks when "interesting" preprocessing events occur. Their infrastructure is based on GCC's preprocessor, so it only handles a single configuration, although they provide access to unpreprocessed **`#ifdef`** regions as unstructured text, and they provide access to the parser stack in order to aid analysis of this text. The first comprehensive work on handling the C preprocessor in a refactoring tool is due to Garrido [6–8]; this was discussed previously. Notably, Garrido's work identifies rules for refactoring code containing macros and file inclusions, it recognizes the need to "guard" symbol table entries with the preprocessor configurations under which they are valid, and it addresses the difficulties of refactoring codes where conditional compilation directives do not enclose complete syntactic constructs.

The importance of preprocessor concerns in program analyses (and transformations) is confirmed in an empirical study by Ernst et al. [3]. Several authors have attempted to understand how commonly preprocessors are used in C code. These studies rely on heuristics to approximate the actual usage [1, 3, 7, 17, 22]. Of course, the problems of using **`#ifdef`** directives have also been noted [27].

In recent years, several authors have proposed and evaluated techniques for parsing and analyzing (e.g., type checking) code containing conditional compilation directives and macros, respecting the full space of feasible preprocessor configurations [9, 14, 18]. These techniques improve over previous proposals (including [6, 22]) since they do not rely on heuristics and do not impose any restrictions on the location of preprocessor constructs.

Erwig and Walkingshaw [4] propose the Choice Calculus to reason about variational structures (like **`#ifdef`**). For example, the Choice Calculus formalizes the notion that common tokens in all branches of an **`#ifdef`** can be "factored out." Choice Calculus-based reasoning could be used to build refactorings to move or modify **`#ifdef`** directives, while PPDGs are appropriate for ensuring the correctness of more typical C refactorings that do not modify such directives.

10. FUTURE WORK AND CONCLUSION

In this paper, we introduced the concept of preprocessor dependences and showed how they provide a uniform framework for ensuring that refactorings—particularly those that move, delete, or copy code—operate correctly on code containing lexical macros and conditional compilation.

Our formal definitions of preprocessor dependences were constructed in a way intended to make their correctness relatively apparent. Extending the concepts to handle the full suite of CPP directives was also intended to be straightforward (the authors' implementation of this technique in OpenRefactory/C handles the full C preprocessor). Nevertheless, ongoing work includes a complete, formal specification and proof of correctness (relative to a formal specification of the C preprocessor). For more information, see `https://sites.google.com/site/cpprefactoring/`.

We expect this technique to be applied to other preprocessed languages. Fortran code often includes C preprocessor constructs, for example. In fact, many Fortran programs utilize other macro processors (e.g., M4), as well as custom preprocessors. Some use several preprocessors in sequence. How to refactor such programs—and the role that preprocessor dependences may play—remains an open question.

11. ACKNOWLEDGMENTS

This work was supported by NSF grant CCF-1217271.

12. REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. Hassan. Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 243–254, New York, NY, USA, 2009. ACM.

[2] G. Badros and D. Notkin. A framework for preprocessor-aware C source code analyses. *Softw. Pract. Exper.*, 30(8):907–924, July 2000.

[3] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, Dec. 2002.

[4] M. Erwig and E. Walkingshaw. The Choice Calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, Dec. 2011.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[6] A. Garrido. *Program refactoring in the presence of preprocessor directives.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005.

[7] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *Proceedings of 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 379–388, 2005.

[8] A. Garrido and R. Johnson. Embracing the C preprocessor during refactoring. *J. Softw. Evol. and Proc.*, June. 2013.

[9] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 323–334, New York, NY, 2012. ACM.

[10] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, volume 7920 of *ECOOP '13*, pages 629–653. Springer Berlin Heidelberg, 2013.

[11] W. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, University of Washington, Seattle, WA, 1991.

[12] M. Hafiz, J. Overbey, F. Behrang, and J. Hall. OpenRefactory/C: An infrastructure for building correct and complex C transformations. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, WRT '13, 2013.

[13] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C.* Dec 1999.

[14] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 773–792, New York, NY, 2012. ACM.

[15] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann, San Francisco, 2002.

[16] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[17] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 105–114, New York, NY, USA, 2010. ACM.

[18] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 81–91, New York, NY, USA, 2013. ACM.

[19] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, 1997.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, 2009.

[21] J. Overbey and R. Johnson. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11.

[22] Y. Padioleau. Parsing C/C++ code without pre-processing. In *ETAPS '09*, CC '09, pages 109–125, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] M. Platoff, M. Wagner, and J. Camaratta. An integrated program representation and toolkit for the maintenance of C programs. In *Proceedings of IEEE International Conference on Software Maintenance*, ICSM '91. IEEE, 1991.

[24] M. Schaefer and O. de Moor. Specifying and implementing refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 286–301, New York, NY, 2010. ACM.

[25] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip. Correct refactoring of concurrent Java code. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 225–249, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping stones over the refactoring rubicon – lightweight language extensions to easily realise refactorings. In *Proceedings of the 23rd European Conference on Object-oriented Programming*, ECOOP'09.

[27] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer 1992 Technical Conference*, pages 185–198, Berkeley, CA, USA, June 1992. Usenix Association.

[28] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, USA, 1990.