Static Validation of C Preprocessor Macros

Andreas Sæbjørnsen† Lingxiao Jiang† Daniel Quinlan‡ Zhendong Su†

†*University of California, Davis*
‡*Lawrence Livermore National Laboratory*
{andsebjo, jiangl, su}@cs.ucdavis.edu, dquinlan@llnl.gov

***Abstract*—The widely used C preprocessor (CPP) is generally considered a source of difficulty for understanding and maintaining C/C++ programs. The main reason for this difficulty is CPP's purely lexical semantics, *i.e.*, its treatment of both input and output as token streams. This can easily lead to errors that are difficult to diagnose, and it has been estimated that up to 20% of all macros are erroneous. To reduce such errors, more restrictive, replacement languages for CPP have been proposed to limit expanded macros to be valid C syntactic units. However, there is no practical tool that can effectively validate CPP macros in legacy applications.**

In this paper, we introduce a novel, general characterization of inconsistent macro usage as a strong indicator of macro errors. Our key insight is that all applications of the same macro should behave similarly. In particular, we map each macro call c in a source file f to c's *normalized* syntactic constructs within the abstract syntax tree (AST) for f's preprocessed source, and use syntactic similarity as the basis for comparing macro calls of the same macro definition. Utilizing this characterization, we have developed an efficient algorithm to statically validate macro usage in C/C++ programs. We have implemented the algorithm; evaluation results show that our tool is effective in detecting common macro-related errors and reports few false positives, making it a practical tool for validating macro usage.

***Keywords*-preprossing, macro errors, inconsistencies**

I. Introduction

C and C++ are two of the most commonly used programming languages for system-level and general purpose programming. Both languages support a meta-level programming mechanism by providing a set of preprocessor directives, such as source file inclusion (**#include**), macro definitions (**#define**), and conditional compilation (**#if**). A C/C++ source file is normally a mix of preprocessor code and actual C/C++ code [16], [17], and the C preprocessor (CPP) [2] is used to preprocess C/C++ source code to pure C/C++ syntax.

This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, US Air Force under grant FA9550-07-1-0532, and an LLNL LDRD subcontract. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and funded by the Laboratory Directed Research and Development Program at LLNL under project tracking code 07-ERD-057. (LLNL-PROC-406820)

CPP is widely used to make C/C++ code more manageable and portable through its preprocessor directives. However, CPP operates at a purely lexical level, transforming input character sequences directly to output character sequences when preprocessing C/C++ source code (*a.k.a. macro expansion*). It does not consider syntactic or semantic information related to the source code. A sequence of characters, even if they appear exactly the same, may be interpreted in different ways later by a C/C++ parser or compiler, depending on their surrounding contexts. Because the C/C++ parser/compiler operates on the preprocessed code and generally has no knowledge about the CPP directives in the original source, it cannot guarantee correct macro usage.

Because of CPP's lexical nature, many programming errors exist due to inappropriate macro definition and usage. The most common source of errors in CPP usage stems from the macro expansion process, and it has been estimated that up to 20% of all macros contain errors [13]. Although some errors lead to syntactically invalid C/C++ code and will result in compilation errors, many do not reveal themselves until the compiled code exhibits strange behavior at runtime. Thus, much research has attempted to characterize what constitutes an error in preprocessor usage so that tools may be developed to automatically detect such errors accurately. However, there is no general characterization that captures macro errors accurately without compromising the programming flexibility offered by the current preprocessor language.

There are two main general approaches to this problem: (1) developing replacement languages for CPP to avoid such errors altogether, and (2) designing analysis tools to detect certain incorrect CPP usage. The first approach advocates the close integration of preprocessor languages with the host languages so that there is no difference between macro errors and other programming errors. This way, no specific tools are needed for detecting macro errors. This approach can be beneficial for developing new software, but it is less useful for detecting and eliminating macro errors in the large amount of legacy C/C++ code using CPP.

The second approach has a few alternatives: (a) ignoring the preprocessor and performing simple lexical analysis on C/C++ code, (b) assuming that the C/C++ parser/compiler can catch all preprocessor errors and only analyzing the

preprocessed code and occasionally reporting macro-related errors by accident, or (c) attempting to emulate the preprocessor and checking how it affects the way the code is being parsed. Alternative (a) is too simplistic since many macro usage patterns cannot be captured purely lexically, and syntactic or semantic information from the preprocessed code is often needed for precise analysis. When utilizing good macro coding conventions to check for violations, a tool effectively imposes restrictions on the preprocessor language and is likely to report high numbers of false positives (because violating a coding style may not necessarily be an error) and false negatives (because the enforced coding styles may not capture certain classes of errors).

Alternative (b) can be used to detect those macro-related errors that result in compilation errors, but some more difficult-to-detect errors do not result in compilation errors. The following is an example of such an error:

```
#define MULTIPLY(arg1,arg2) arg1*arg2

MULTIPLY(1+1,2);
```

where the macro call will expand into "1+1∗2", which is parsed as "1+(1∗2)". By just inspecting the expanded expression, such tools cannot decide whether the expression is an error or not.

As for alternative (c), it cannot reliably catch errors in CPP usage since CPP is very permissive in changing how a piece of code gets parsed. For the same example as the above, such tools may notice the expanded expression has a different parse tree structure than the structure for the macro definition itself, but they cannot know whether the difference is intended or not. Reporting such differences as errors may result in high false positive rates and affect adoption of such tools.

In this paper, we take the second general approach by developing a static validation tool for CPP macros. Our goal is to design a tool that can detect general macro usage errors, has few false positives, and does not impose any restrictions on the preprocessor language, thus overcoming disadvantages of previous approaches. To this end, we formalize a novel, general notion of *inconsistent macro usage*, capturing our intuition that macro calls of the same macro definition should be similar. We utilize information from CPP for the original source code and syntax trees of the preprocessed source code. In particular, we normalize each macro call to capture how it influences the abstract syntax tree of the preprocessed code and use syntactic similarity to compare macro calls with the same macro name. Any inconsistency is reported as a potential macro usage error. Our characterization has a number of benefits. It is general and does not rely on any macro usage patterns or coding conventions; is flexible and does not impose restrictions on how developers define macros, thus preserving the flexibility of the preprocessor language. It only relies on our observation that the same macro should be used in a similar manner.

We have implemented our approach in a prototype tool, CPPCHECKER, to detect inconsistent macro usages. CPPCHECKER is designed to detect inconsistencies across all uses of a macro in every source file of a program. Ernst *et al.* [13] performed a study on macro usage and errors. We have performed empirical evaluations of our tool on the top four test subjects used in Ernst *et al.*'s study [13]. The results show that the tool can detect common, difficult-to-detect macro errors categorized by Ernst *et al.* [13] through error injection. It is precise and reports very few false positives. It has also detected previously unknown inconsistencies in the latest releases of these test subjects. It is efficient, adding negligible overhead over parsing.

The rest of the paper is structured as follows. Section II gives the details of our approach: it gives some background on CPP and macros, defines inconsistent macro usage, and presents the detailed algorithms to detect inconsistent macros. We discuss details of our tool implementation in Section III and present the evaluation results in Section IV. Section V discusses possible ways to enhance our current technique and tool. Finally, we survey related work (Section VI) and conclude (Section VII).

II. Our Approach

In this section, we give some background information on CPP and macros, discuss our notion of inconsistent macro usage, and present details of our algorithms.

A. Background on Macros

CPP, the C macro preprocessor, expands all macro calls and handles all preprocessing directives. There are two types of macros, *object-like* and *function-like*. Object-like macros do not take any arguments, while function-like macros take arguments. The syntax of a macro definition is given below:

```
#define < identifier > <replacement token list >
#define < identifier >(<parameter list >) \
                      <replacement token list >
```

where the first form is for object-like macros and the second for function-like macros. In both cases, the "identifier" (with arguments in the second case) corresponding to a macro definition in the tokenized text is replaced by its replacement

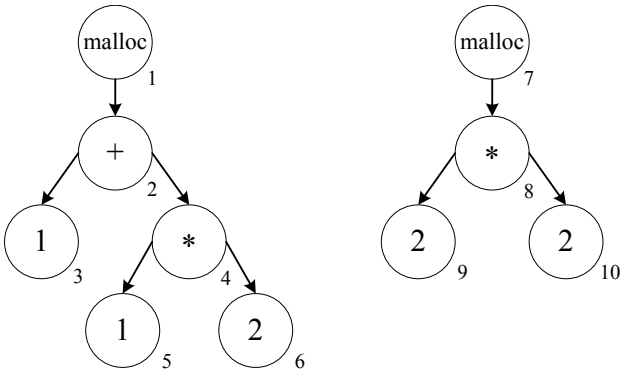


Figure 1. Left is AST of ALLOCATE_MEM(1+1); right is the AST of ALLOCATE_MEM(2).

token list. This is called a *macro call*. As other macros can be called in a replacement token list, we define an expanded macro as a series of tokens that contain no macro calls. Such a token stream is created by recursively expanding macros in the replacement token list until there are no macros left. The following is an example of a function-like macro with two calls:

```
#define ALLOCATE_MEM(arg) malloc(arg*2)
int main(){
    ALLOCATE_MEM(1+1);      // 1
    ALLOCATE_MEM(2);        // 2
};
```

which after preprocessing becomes:

```
int main(){
    malloc(1+1*2);          // 1
    malloc(2*2);            // 2
};
```

B. Defining Inconsistent Macros

Ernst *et al.* [13] classify ten categories of macro usage and nine categories of erroneous macro usage. Erroneous macros are mainly results of the lexical treatment of token streams in macro expansions. Our goal is to define a simple, but general, notion to capture erroneous macro usage.

Our key insight is that all macro calls to the same macro definition must be similar because like other types of abstraction mechanisms, a macro definition abstracts an "intended," shared usage pattern. To compare macro calls, we use structural similarity between the subtrees of the abstract syntax trees (ASTs) that two macro calls correspond to. If two macro calls to a macro definition are not structurally similar then we have detected an inconsistency.

The ALLOCATE_MEM example shown earlier (Section II-A) is an example of inconsistent macro usage. Because the operator $*$ has higher precedence over +, "malloc(1+1$*$2)" will be parsed as "malloc(1+(1$*$2))" and allocate 3 bytes of memory, while the other call "malloc(2$*$2)" will allocate 4 bytes of memory.

Stylistic conventions [30] like parenthesizing arguments can be used to avoid macro errors, but unfortunately variable names cannot be parenthesized in all macro definitions [3]. For example, in macros requiring that their argument be textually inserted in the particular position inside an expression, enclosing the correspondent variable name in parentheses may inhibit required order of evaluation. For example,

```
#define MKSTR(args) static_cast<std :: ostringstream &>\
    ( std :: ostringstream () << std :: flush << args ). str ()

std :: string msg = MKSTR("There are " << 4 <<
                          " lines in the program");
```

requires that the insertion to the stringstream occurs first. This macro cannot be re-written to parenthesize the macro parameter. Manually applying style conventions is also tedious and error prone, especially for large or complex macros, so having a tool that can verify correct application of these conventions is useful even for code that is properly styled.

In general, stylistic conventions can often be applied to macros that are expressions or analogs of complete statements. At a high level these macros can be viewed as functions. Some macros just generate composite pieces of source code and therefore do not fit into these categories. Styling conventions cannot be applied to these as adding parentheses or brackets would interfere with their functionality.

Our definition of inconsistent macro usage is based on how a macro call impacts the syntactic structure of the preprocessed code. If two calls to the same macro definition have different impact, we say there is an *inconsistent macro usage*. The question remains how to define "impact." It is clear this cannot be directly based on the syntactic structure of the macro calls since many macros take arguments or call other macros. We need to normalize the macro calls by removing nodes coming from these sources before checking for similarity. However, we do not remove all nodes that come from arguments or calls to other macros; we should keep those nodes that "influence" the parsing structure, which will be made more precise in the following.

Our definition of inconsistent macro usage utilizes the structure of an AST to decide which nodes to remove and which ones to keep. The reason we use an AST is because preprocessed code is parsed into an AST, which is the right context to discuss the syntactic impact of a macro call.

Definition 1 (Structurally Significant): Let $M(\overrightarrow{a})$ be a macro call and $\overrightarrow{t} = t_1, \ldots, t_k$ be the corresponding expanded token sequence of $M(\overrightarrow{a})$. An AST node n_i that is parsed from t_i of $\overrightarrow{t}$ is *structurally insignificant* if

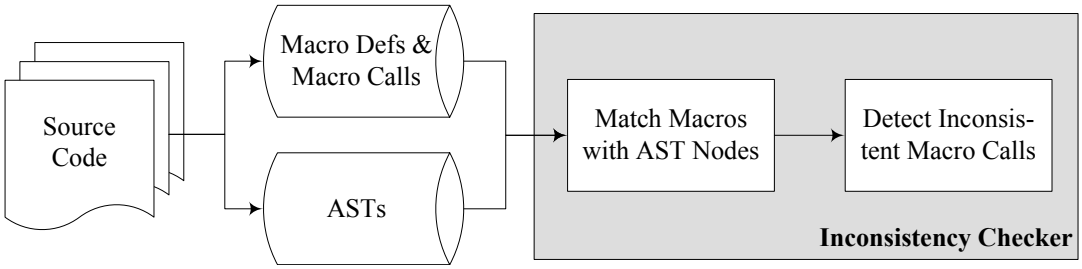


Figure 2. High-level view of detecting inconsistent macro usage.

t_i is part of the macro argument $\overrightarrow{a}$ or comes from the macros $M_1, \ldots, M_l$ called by M. Otherwise, we say n_i is *structurally significant*.

For example, the ASTs for the two expanded "ALLOCATE_MEM" macro calls in our earlier example are illustrated in Figure 1. Nodes 1, 4, 6, 7, 8, and 10 are structurally significant and nodes 2, 3, 5, and 9 are structurally insignificant.

Definition 2 (Normalized Syntax Tree): Let T denote the smallest syntactic AST unit containing a macro call $M(\overrightarrow{a})$. We create the *normalized syntax tree* of T with respect to $M(\overrightarrow{a})$ by deleting every structurally insignificant node n_i with no structurally significant node in the subtree of T rooted at n_i.

Note that in the example in Figure 1, nodes 3, 5, and 9 are removed because they are structurally insignificant and subtrees rooted at those nodes do not contain any significant nodes. On the other hand, although node 2 comes from a macro argument, it is kept because the subtree rooted at node 2 contains structurally significant nodes.

Now with the definition of normalized syntax tree of a macro call, we can define consistent and inconsistent macro usage.

Definition 3 (Consistent Macro Usage): Let $M(\overrightarrow{a})$ and $M(\overrightarrow{b})$ be two macro calls to the same macro definition M. Further, let T_1 and T_2 be the corresponding normalized syntax trees of $M(\overrightarrow{a})$ and $M(\overrightarrow{b})$ respectively. We say the calls are *consistent* if $T_1 = T_2$. Otherwise, the calls are *inconsistent*.

Consider the example in Figure 1 again. The normalized syntax trees for the two calls are different, thus we detect an inconsistent macro usage and a possible macro error.

C. Detailed Algorithms

This section gives a detailed description of our algorithm, structured according to the flow-chart in Figure 2. Since ASTs usually do not contain detailed information on macro calls, we construct annotated ASTs using a separate preprocessor to provide such information. An annotated AST is created by matching a post-CPP token sequence and an AST (Section II-C1). In order to compare macro calls we construct a normalized tree representation for each macro call. A normalized tree is constructed by traversing an AST to find the smallest embedding subtrees containing all the tokens from a macro call (Section II-C2) and deleting structurally insignificant nodes (Section II-C3). If we have two macro calls to a macro definition, we consider the two calls to be inconsistent if the normalized trees for the two macro calls are dissimilar (Section II-C4). A macro definition that has only one call is consistent by definition.

Algorithm 1 Matching AST nodes with post-CPP tokens

Input: $L \leftarrow$ in-order linearization of the AST
Input: $T \leftarrow$ post-CPP token sequence
Output: $M \leftarrow$ set of nodes paired with tokens

```
1: for n ∈ L do
2:   if n is not a structural node then
3:     for t ∈ T do
4:       T = T \ {t}
5:       if n corresponds to t then
6:         M = M ∪ {(n, t)}
7:         break
```

1) Matching ASTs with post-CPP token sequences: A C/C++ parser generates ASTs based on post-CPP token sequences and has no knowledge of the CPP syntax in pre-CPP source files. Because our notion of inconsistent macro calls are based on normalized trees, we need a detailed mapping from the macro calls in pre-CPP token sequences to their corresponding AST nodes.

We annotate an AST with detailed information on macro calls, such as which nodes in the AST correspond to the formal macro body, the actual macro arguments, and the calls to other macros in the macro body during expansion. Such information can be provided by a separate preprocessor that keeps track of token transformations during macro expansion and annotates each token in post-CPP token sequences with its corresponding pre-CPP token.

Algorithm 1 annotates each AST node with a post-CPP token. Each node in an AST is checked to see if it corresponds to a token in expanded macro calls. The correspondence

between a node and a token is determined by the in-order linearization of an AST as that always has the same ordering as its post-CPP token sequence.

In Algorithm 1, we also require that "n" is not a structural node. This is needed because an AST often contains structural nodes which are generated as auxiliary nodes to represent language semantics and do not have any corresponding token in the original source files, such as constant folded expressions and structural place-holders. The worst-case time complexity of this algorithm is $O(N)$, where N is the number of nodes in an AST.

2) Finding Smallest Embedding Subtrees: Our notion of inconsistent macro usage relies on representing only the parts of an AST that are truly relevant to each expanded macro call. Here, we aim to find a set of embedding subtrees in an AST that contains all tokens from each expanded macro call and use such subtrees for consistency checking.

Since an expanded macro can often correspond to only parts of a complete syntactic unit in C/C++ language, *e.g.*, a subexpression in an expression, we look for the *smallest* embedding subtrees that contain all tokens from each expanded macro call and the fewest irrelevant tokens. Especially, when an expanded macro call spans more than several statements but does not correspond to a complete syntactic unit, we generate a *set* of embedding subtrees that are the most essential to the expanded macro. For instance in the following example:

```
#define SPANS_STATEMENTS (i+2); i++;
int i; 2*SPANS_STATEMENTS;
```

the smallest embedding subtrees for "SPANS_STATEMENTS" expanded are the subexpression "(i+2)" and the statement "i++;", instead of all the code "**int** i; 2*(i+2); i++;" or the two statements "2*(i+2); i++;".

Algorithm 2 finds such smallest set of subtrees for each macro call. This is done in two steps. First, in a pre-order traversal, the top-most AST nodes that are matched with a token from the macro call (*c.f.* Section II-C1) are kept (Lines 2–5). These top-most nodes can be disjoint parts of the same macro call, so we traverse the parents of these nodes bottom-up, up to statement boundaries (the condition on Line 8), to locate nodes that encapsulates larger parts of the macro call (the condition on Line 9) so that the number of embedding subtrees can also be reduced. Then such parent nodes replace the encapsulated subtrees as a new embedding subtree in the result set. The worst-case time complexity of Algorithm 2 is $O(N)$, where N is the number of nodes in the AST.

3) Normalizing Macro Calls: Our notion of inconsistent macro usage operates on normalized syntax trees where all insignificant nodes are considered for removal. An insignificant node comes from a macro argument or another macro called during macro expansion, but it is only removed if there are no significant nodes in its subtree.

Algorithm 2 Finding smallest embedding subtrees of a macro call

Input: $T \leftarrow$ pre-order numbering of the AST
Input: $M \leftarrow$ set of nodes matched with the macro call
Output: $S \leftarrow$ root of subtrees containing the macro call

```
1:  I = ∅
2:  for n ∈ T do /* find top-most nodes from M */
3:    if n ∈ M and n ∉ I then /* n is a top-most node */
4:      I = I ∪ children(n)
5:      S = S ∪ {n}
6:  for n ∈ S do /* find common ancestors */
7:    p = n
8:    while p is not a statement boundary do
9:      if size(S ∩ children(p)) > 1 then
10:       S = S \ children(p)
11:       S = S ∪ {p}
12:     p = parent(p)
```

Algorithm 3 Normalizing macro calls

Input: $T \leftarrow$ post-order numbering of the AST nodes
Input: $S \leftarrow$ list of significant nodes

```
1: for n ∈ T do /* post-order traversal */
2:   if n ∉ S then /* n not significant */
3:     if children(n) ∩ S = ∅ then /* no significant node in subtree */
4:       Remove t
5:     else
6:       S = S ∪ n
```

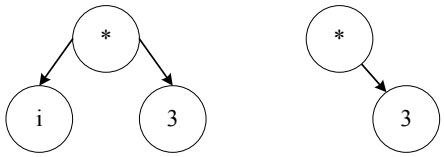
As an example, we show how to normalize two macro calls to a function-like macro:

```
#define TEST_MACRO(arg) arg*3
int i,j;
TEST_MACRO(i);    //1
TEST_MACRO(i+j);  //2
```

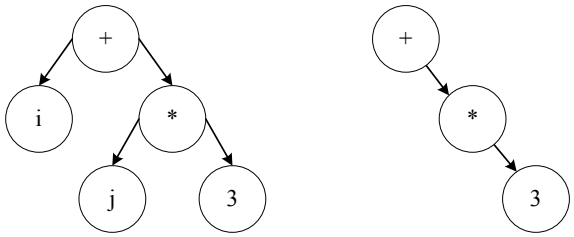
The example code expands into:

```
int i,j;
i*3;              //1
i+j*3;            //2
```

The left hand side of Figure 3(a) shows the AST for the first macro call. As the 'i' node comes from the macro argument we remove it to get the normalized tree on the right hand side. Similarly, the left hand side of Figure 3(b) shows the AST of the second macro call. Nodes 'i', '+' and 'j' come from the macro argument so we consider removing them, but since '+' has the structurally significant nodes '*' and '3' in its subtree we keep it in the normalized tree on the right hand side of Figure 3(b).



(a) The left graph is for TEST_MACRO(i) before argument removal; the right hand side is after argument removal.



(b) The left graph is for TEST_MACRO(i+j) before argument removal; the right side is after argument removal.

Figure 3. Examples of normalized syntax trees.

Algorithm 4 Detection of inconsistent macro uses

Input: $M_1 \leftarrow$ numbering of normalized tree for call 1
Input: $M_2 \leftarrow$ numbering of normalized tree for call 2
Input: $c \leftarrow$ allowed tree difference

1: **if** $edit\_distance(M_1, M_2) > c$ **then**
2: return "*inconsistent*"
3: **else**
4: return "*consistent*"

Using algorithm 3 we remove insignificant nodes to create a normalized tree representation of a macro call. As input the algorithm takes a post-order numbering of the AST nodes in the smallest subtrees containing a macro call. The worst-case time complexity of the algorithm is $O(N)$, where N is the number of nodes corresponding to a macro call.

4) Detecting Inconsistent Macro Usage: We detect incorrect usage of a macro definition by comparing the the normalized syntax trees corresponding to each macro call to the macro definition. According to our notion, two macro calls to the same macro definition with structurally dissimilar normalized trees indicate a possible macro error.

The right hand sides of Figure 3(a) and Figure 3(b) show the normalized trees for the two macro calls in the example in Section II-C3. Since the two normalized trees are clearly dissimilar, the macro usage is inconsistent and it indicates an inappropriate macro definition.

Algorithm 4 detects inconsistencies based on normalized trees for macro calls. Each macro call is represented by a numbering of its normalized tree, and we use the *edit distance* among the numbering of trees to determine inconsistency. The edit distance between M_1 and M_2 is the number of edit operations (either insertion, deletion, or modification) required to transform them into each other. An advantage of using edit distance in the algorithm is that it may tolerate certain inconsistencies among normalized trees if a user expects different usage of a macro and sets the threshold c to a value larger than 0. For our evaluation we only use edit distance 0. The running time of the algorithm is $O(min(|M_1|, |M_2|))$.

III. Implementation

We have implemented our algorithm as a macro error detection tool, CppChecker. We use the ROSE open source compiler infrastructure [6], [28] for parsing and the Boost Wave C99/C++ preprocessor [1] for annotating ASTs with macro information. Our tool supports both C and C++.

We designed our tool to use an SQLite database for communication between different phases of our analysis, allowing new software packages to be added without requiring existing packages to be re-analyzed. Each analysis phase is implemented as a separate C++ program. The first step is to create the normalized syntax tree for each macro call and store both macro definitions and calls and the trees into database. Then, for each macro definition M, we query the database for the normalized trees for all calls to M and compare the trees to detect inconsistent macro usage. The database can be easily accessed to create macro error reports or suggest possible fixes. We designed our tool based on database to make whole program and even cross-program analysis easy, and all macro calls to a macro definition are analyzed only once per definition. For example, if file f_1 contains calls to a macro M defined in file f_2, all calls to M in f_1 and f_2 are analyzed together for detecting inconsistencies.

A. Constructing AST

Our definition of inconsistent macro usage and our algorithms are formulated over abstract syntax trees (ASTs). Although it is also possible to apply our algorithms to parse trees, we utilize the higher-level structure of ASTs to get the normalized trees for macro calls. For instance, a binary operator in an AST has two subtrees, one for the left operand and one for the right operand, which simplifies tree structures and the normalization process.

We use ROSE [6], [28] to construct ASTs for source files. ROSE supports Fortran, C, and C++ among other languages. Like other compiler infrastructures, ROSE consists of front-ends, a mid-end, and back-ends. The intermediate representation used in ROSE is high-level, builds ASTs, and provides a framework for writing static analysis tools. The mid-end contains a set of analyzes that we use to

construct CPPCHECKER. In particular, ROSE has built-in traversal mechanisms, which are useful for implementing our algorithms. The traversal mechanisms support pre-order, post-order, and pre-post-order traversals.

When creating an annotated AST, we need to know how to map between an AST and its corresponding post-CPP token sequence. Algorithm 1 creates such a mapping, but it takes as input an in-order traversal of the AST nodes. Because ROSE does not provide in-order traversal, we construct our own in-order traversal using a pre-order traversal. The AST structure for certain nodes in ROSE can be ambiguous as how to create an in-order traversal. For instance, a node representing a conditional expression "$c\ ?\ e_1 : e_2$" has three subtrees: one each for the condition c, the true expression e_1, and the false expression e_2. We need to number the node twice, one for "?" (between c and e_1) and one for ":" (between e_1 and e_2) to ensure that the in-order traversal of the AST nodes is equivalent to the ordering of the token sequence.

B. Annotating AST

Like most compilers, ROSE generates ASTs based on post-CPP token sequences and has no knowledge of macro calls except for their expanded forms in the post-CPP token sequences. Because our notion of inconsistent macro usage is based on normalized trees, we need a detailed mapping from AST nodes to macro calls. We use a separate preprocessor, integrated with ROSE, to get the detailed information on macro calls.

Our implementation currently uses the Boost Wave preprocessor [1]. Wave is an implementation of the C99/C++ preprocessor standard with a simple-to-use interface for observing macro expansions and directive handling. Wave exposes a token iterator interface, a context object, and preprocessing hooks. It iterates over a pre-CPP token stream and expands macros according to the C99/C++ preprocessor standard in a way equivalent to the CPP step in the ROSE compiler. In our implementation, Wave does a second CPP pass over all source files and we utilize its preprocessing hooks to annotate each ROSE AST node with information about preprocessor directives during its process.

C. Whole Program & Incremental Analysis

The use of a database facilitates whole program and incremental analysis. We chose an SQLite database [7] as it is a small C library that implements a self-contained, zero-configuration SQL database engine. Zero-configuration is a significant advantage in terms of usability as no setup or administration is needed. The database is stored in a single disk file which is easily shared across different machines. Based on our experience this has proven to be useful in sharing information about macro usage as the database presents an easy interface for sharing detailed macro information. Another benefit is that once the database is constructed no recompilation is necessary if for instance a new threshold value c is chosen for edit distance in Algorithm 4. We can also check additional software packages without recompiling packages that are already in the database to enable incremental analysis.

Package	Version	LOC	NCNB	Files
bc	1.06	21870	15831	39
gzip	1.3.12	18391	11916	58
rcs	5.7	30570	20903	46
remind	3.01.06	32677	24197	50

Table I
ANALYZED PACKAGES AND THEIR SIZES; NCNB LINES ARE NON-COMMENT, NON-BLANK LINES.

IV. EMPIRICAL EVALUATION

This section presents our evaluation of CPPCHECKER. It discusses the setup for the evaluation, including the test subjects and system setup. It also lists the types of macro errors categorized by Ernst *et al.* [13] for use in the evaluation. CPPCHECKER's effectiveness is evaluated in terms of recall, precision, and efficiency.

A. Experimental Setup

We evaluated the effectiveness of CPPCHECKER in validating macro usage on the packages in Table I. Out of the programs evaluated by Ernst *et al.* [13], these are the ones with the highest usage of CPP macros. All analysis runs were done on a workstation with two Xeon X5355 2.66 GHz quad-core processors and 16 GiB of RAM, of which we used one core for our experiments. The workstation has a 4-disk RAID with 15,000 RPM 300 GB disks. It runs Red Hat Enterprise Linux 4 with kernel version 2.6.9-78.

B. Types of Macro Errors

We detect four types of macro errors defined by Ernst *et al.* [13]: *inconsistent arity*, *unparenthesized formal*, *unparenthesized body*, and *swallows else*. Out of all the errors defined by Ernst *et al.*, these are the most difficult-to-detect categories because they cannot be easily detected by inspecting the macro definition or macro call.

Inconsistent Arity: The macro name is defined multiple times with different definitions, like for instance:

```
#define MAX(a,b) a> b? a : b
#define MAX(a,b) a>=b? a : b
```

This may either indicate a real bug or a macro name used for different purposes in different parts of the package. CPP requires an "#undef MACRO_NAME" directive before redefining a macro in the same file, while this is not necessary when redefining the macro in a different file within the same package. We treat all macro definitions with the

```
#define PARSENUM(var, s) \
var = 0;                 \
while ( isdigit (*(s ))) {  \
    var *= 10;           \
    var += *(s) - '0';   \
    s++;                 \
}
```

```
#define UPPER(c) toupper(c)
```

```
#define PARSENUM(var, string)      \
 if (! isdigit (*( string ))) return;  \
var = 0;                           \
while ( isdigit (*( string ))) {   \
    var *= 10;                     \
    var += *( string ) - '0';      \
    string++;                      \
}
```

```
#define UPPER(c) islower(c) ? toupper(c) : (c)
```

Figure 4. Example macros of inconsistent arity found in Remind.

same formal name as the same macro, thus enforcing that all calls to this macro should be consistent.

Unparenthesized Formal: An argument is used as part of an expression without being parenthesized, so that precedence rules can cause two different macro expansions parsed into dissimilar AST subtrees. The following is a concrete example:

```
#define SQUARE(x) x*x
 ... SQUARE(1) ...
 ... SQUARE(1+1) ...
```

where the second macro invocation evaluates to 3, not 1.

Unparenthesized Body: The macro body is an expression that should be parenthesized to avoid precedence issues. For example, consider the following code:

```
#define ADD(x,y) x+y
 ... 3*ADD(1,2) ...
```

which evaluates to the value 5 instead of 9.

Swallows Else: The macro ends with an if statement without a corresponding else clause, and swallows any else following it in the token stream. For instance

```
#define AN_IF(x) if(x){/* do something */}
 ...
if ( condition )
  AN_IF(true)
else ...
```

will parse into an AST construct where the if-statement from the macro expansion has an else-statement. Other invocations of the same macro may not have an else-statement in the context immediately following the macro call so that they are inconsistent with this invocation.

Multiple Formal Uses: Some argument is used as an expression several times, so any side effect in the actual macro argument will occur several times. For instance:

```
#define SQUARE(x) (x)*(x)
 ... SQUARE(x++) ..
```

will perform the "++" operation twice. This type of macro errors is easy to detect with a simple scan of the macro body. Because this is not a result of inconsistent usage across two macro calls, we do not model this type of errors.

Ernst *et al.* presents two additional types of errors that cause a compiler parsing error: dangling semicolon after a macro call and side-effecting a formal argument that cannot be side-effected in the target language.

There are also three macro error types that we do not model because they are easy to detect. Usage of free variables in the macro to achieve dynamic scoping is the only one which can result in runtime error. The two other errors will not result in runtime error, but causes problems for a programmer trying to understand the code: macro arguments that is not used in the macro definition and macro names that are reserved words in the target language.

C. Evaluation on Mature Packages

All the packages in Table I are stable and mature packages that have been screened for the types of errors we detect. In addition, they utilize styling conventions to avoid macro errors. In particular, all the projects parenthesize macro arguments wherever applicable in order to enforce a particular operator precedence. The following is an example of parenthesized arguments in "bc−1.06/h/number.h":

```
#define MAX(a,b)      ((a)>(b)?(a):(b))
```

Although such tactics are effective when applicable, they cannot be applied to all macro arguments because a macro argument is not restricted to representing expressions or complete statements (see Section II for a concrete example).

For these reasons, we expect our tool to find few macro inconsistencies in these mature and stable projects. Table II

```
#define  fixangle(a)  (a − 360.0 ∗ \
                       ( floor (a / 360.0)))
#define  elonge       278.833540
#define  elongp       282.596403

double Day, N, M;

N = fixangle((360 / 365.2422) ∗ Day);
M = fixangle(N + elonge − elongp);
```

```
double Day, N, M;

N = ((360 / 365.2422) ∗ Day − 360.0 ∗
     ( floor ((360 / 365.2422) ∗ Day / 360.0)));

M = (N + 278.833540 − 282.596403 − 360.0 ∗
     ( floor (N + 278.833540 − 282.596403 / 360.0)));
```

Figure 5. Example bug injection in Remind. Code to the right is the preprocessed version of the code to the left.

	Swallows-Else			Unparenthesized-Body			Unparenthesized-Formal		
Package	**Inject**	**Cons**	**Incons**	**Inject**	**Cons**	**Incons**	**Inject**	**Cons**	**Incons**
bc	0	0	0	3	0	3	0	0	0
gzip	3	0	3	13	0	13	3	0	3
rcs	2	0	2	6	0	6	2	0	2
remind	0	0	0	12	0	12	8	0	8
Total	5	0	5	32	0	32	13	0	13

Table III
EVALUATION OF HOW WELL CPPCHECKER DETECT INJECTED BUGS OF MACRO ERROR TYPES SWALLOWS-ELSE, UNPARENTHESIZED-BODY AND UNPARENTHESIZED-FORMAL.

Package	**MD**	**MC**	**Consistent**	**Inconsistent**
bc	39	321	36	3
gzip	78	403	78	0
rcs	63	547	62	1
remind	71	1072	69	2
Total	251	2343	247	6

Table II
ANALYSIS OF CONSISTENCY OF FUNCTION-LIKE MACRO CALLS ACROSS PROJECTS. MD IS THE TOTAL NUMBER OF FUNCTION-LIKE MACRO DEFINITIONS, MC IS THE TOTAL NUMBER OF MACRO CALLS TO THE FUNCTION-LIKE MACRO DEFINITIONS.

shows that 6 out of the 251 macro definitions are reported inconsistent by CPPCHECKER. Investigating each inconsistency by hand, we find that two of them are actual inconsistencies. Both of the inconsistencies are of type inconsistent arity, and the macro definitions are shown in Figure 4. The macros PARSENUM and UPPER have different definitions in the same package, which can lead to potential errors.

Due to an implementation limitation in the ROSE framework, four of the reported inconsistencies are not actual errors. Because of this limitation in ROSE, we cannot properly identify the nodes originating from the macro arguments when the macro expands into an enum definition. The following is such an example:

```
#define  MACRO_CALL(x) (x+2)
enum x{
  en1 = MACRO_CALL(1),
  en2 = MACRO_CALL(2)
};
```

D. Evaluation on Injected Errors

We also injected errors into the packages in Table I to evaluate our tool's effectiveness in detecting actual inconsistency errors. Errors were only injected for the macro error types that do not require writing new macro definitions: swallows else, unparenthesized body, unparenthesized formal. The injection was done by making one copy of the packages for each macro error type. We removed parenthesization from macros that use styling conventions. In each copy of the package for a macro error type, we injected one erroneous macro call for every macro definition that is vulnerable to this type of errors.

An example error that we injected is shown in Figure 5. This injected error causes a detected inconsistency. In the original code the macro argument "a" to the function-like macro "fixangle" is parenthesized. By removing the parentheses, the two macro calls to "fixangle" become inconsistent with respect to each other.

In Table III the injected column shows how many macro definitions are vulnerable to each type of macro errors. The inconsistent column shows that CPPCHECKER detected all the injected errors.

E. Efficiency

We measured the efficiency of our tool on the four packages. Table IV lists the results, which show that 64% of the analysis time was spent on constructing the ASTs, while 36% was spent on generating the normalized trees

Package	AST	Tree Normalization	Inconsistency
bc	99s	47s	0.021s
gzip	71s	65s	0.015s
rcs	306s	123s	0.016s
remind	266s	188s	0.012s
Total	742s	423s	0.064s

Table IV
TIMING INFORMATION IN SECONDS FOR THE CONSTRUCTION OF THE AST, THE CONSTRUCTION OF THE NORMALIZED TREE FOR ALL MACROS AND THE DETECTION OF INCONSISTENCIES.

for every macro call. The timing for the creation of the normalized trees includes the time to map between pre-CPP token streams and ASTs. Detecting inconsistencies took less than a second for all of the projects as it is a simple database lookup. The results indicate that our tool is efficient and only adds negligible overhead over standard parsing.

V. DISCUSSION

We detect the most difficult to detect macro errors using our notion of inconsistent macro usage. These are errors that cannot be easily found through inspection of the macro definitions or the macro calls. For instance, detecting unparenthesized-body errors involves understanding complex interactions between the CPP language grammar and the target language grammar. The complexity of doing this increases with the size of the macro and the complexity of its expressions as the developer must carefully observe how the target language grammar interprets the expanded macro. This is particularly burdensome since the developer usually works in the pre-CPP form of the source code. Conversely, multiple formal uses can be detected by inspecting the macro definition when the macro definition is given a side-effecting argument to see if that argument is used multiple times. Although simple heuristics can be used to detect the types of macro errors that are not covered by our current notion it is attractive to have a single notion that detect as many macro error types as possible. One possibility is to extend our notion with program dependency graphs (PDGs) to detect these error types, which we leave for future investigation.

Ernst *et al.* indicate that many packages likely contain macro errors. Many open source projects apply stylistic conventions in order to avoid macro errors, but from our experience some projects do not use these conventions and others do not strictly follow them. It would be interesting to perform a large scale study of the frequency of macro errors in open source packages.

We applied CPPCHECKER as a separate step of the development process. Integrating the tool into the development tool chain is likely to make macro usage less error-prone because it can be used to provide early detection of macro errors. This is particularly useful for new projects that use macros from other packages since the tool can compare the new usage case of a macro with existing usage patterns of the same macro.

It is common that Fortran programs use CPP and we expect our notion of inconsistent macro usage is applicable to that domain as well. It would be interesting to extend our tool to support Fortran. Constructing an annotated AST for Fortran is likely to pose some interesting challenges since Fortran code uses traditional text-based C preprocessors as opposed to token-based ISO C preprocessors. As ROSE currently has a Fortran front-end, it is likely that we can reuse most of the existing code for CPPCHECKER.

Since CPPCHECKER provides an accurate mapping from the pre-CPP and post-CPP token streams to ASTs, we believe it presents an opportunity for developing better refactoring tools for C/C++ programs. It is particularly interesting to develop tools for analyzing macro usage across programs for across-program understanding and maintenance. In the context of ROSE, our algorithms can be essential for source-to-source analysis as it provides a mechanism for refactoring code without losing the abstraction provided by macros.

VI. RELATED WORK

Our work builds on the study of macro usage by Ernst *et al.* [13]. In order to analyze macro usage, they used a tool called PCP3 [10] that extracts parse trees representing source code and provides its own preprocessor to record information about macro expansions. However, PCP3 does not detect errors in macro usage and as a refactoring tool it fails on complex macros. Our work uses abstract syntax trees because of its higher-level syntax. An AST representation is also richer since type information is available for analysis.

Previous research has also proposed replacement languages for CPP. More restrictive preprocessors are created to avoid the possible macro errors defined by Ernst *et al.* [13]. An example is a replacement macro language called ASTEC [24] by McCloskey *et al.*. They also developed a translator called MacroScope that converts existing code using CPP to use their macro language. MacroScope uses an algorithm for finding the smallest set of statements containing a macro call to detect errors in macro usage. However, MacroScope enforces coding styles to detect potential errors and therefore likely has high number of false positives. For instance, macros expanding into an **if** statement that swallows an **else** statement from the macro context will always be marked by MacroScope as an error. This will incorrectly identify macro calls as errors where "swallows-else" is intended. Also, this approach is only useful if developers either adopt the more restrictive replacement preprocessor language or adopt a style convention that can be quite restrictive.

Extending the target language with a macro syntax language has been proposed to facilitate the construction of syntactically valid macros [32], [19]. Weise *et al.* [32] introduced explicit code template operators into the macro language, and used the type system to guarantee that the

macro definition is a valid syntactic fragment in the target language. Syntax macros produce ASTs that replace the code of the macro invocation and therefore restricts the macro definition to be syntactically valid units. Our approach will also catch errors other than syntax errors. Using syntax macros requires changes in the compiler to extend the language syntax and the user code to adopt the new preprocessor syntax.

Many style guides are also available. They describe rules for how to avoid macro errors [4], [5], [30]. These style guides will only work to the extent that developers correctly apply them to their code. There are tools available to automatically enforce such guidelines. Various lint [18], such as LC-lint [14] and PC-lint/Flexelint [8], can check whether C programs satisfy certain restrictions, such as unparenthesized macros and unused macro definitions. Programmers are allowed, to some extent, to define their own rules for detecting bad code, so that the tools can check for those rules as well. These tools enforce certain pre-specified coding guidelines, while as we have a simple, uniform definition of macro inconsistencies that captures a large class of (difficult-to-detect) macro errors.

A standard compiler without knowledge of macros can detect some types of CPP errors. However, as the compiler is unaware of the CPP syntax, it is limited to detecting errors where CPP generates incorrect C/C++ language syntax or semantics. The compiler can for instance detect where a macro call expands into an assignment expression where the type on the right hand side cannot be assigned to the left hand side. Some studies thus provide more comprehensive error checking capability in addition to that is provided by a compiler. Krone *et al.* [21] apply *concept analysis* to infer configuration structures caused by macro uses in source code and represent such structures in *concept lattices* to gain insight into the structures and properties of possible configurations. Sajeev *et al.* [27] examine the most common macro errors and present a number of algorithms for detecting those errors statically. In order to provide a formal, common ground for understanding preprocessor languages, Favre [15] describes a denotational semantics for CPP. Our goal is different; we aim at statically validating macro usage.

Many studies have also proposed different refactoring, maintaining techniques for programs written in preprocessed languages. Padioleau *et al.* [26] propose a special parsing technique that can parse most C/C++ programs without preprocessing. The technique also finds some coding mistakes related to macros, but it may require adapting to some different conventions and idioms for different programs. Spinellis *et al.* [29] use the concept of *token equivalence classes* to bridge the gap between the language semantics and non-preprocessed source code. Their CScout tool is able to analyze large interdependent program families and allows rename and remove refactorings on the original C source code. Mennie *et al.* [25] describes a method of fact extraction and code manipulation to create a set of transformations which will remove preprocessor directives from the original source, converting them into regular C/C++ code with as few changes as possible, while maintaining readability in the code. Vidács *et al.* [31] propose a preprocessor schema. Instances of the schema may be used to model preprocessor constructs in the original source code, the preprocessed compilation unit, and the transformations made by the preprocessor, and thus may help program understanding and maintenance.

The concept of *inconsistencies* is also helpful for identifying abnormalities and has been applied to detect different kinds of bugs. Engler *et al.* [12] observe that what correctness rules a system must obey are often undocumented or specified in an ad hoc manner, while such information may be inferred from source code itself, rather than programmers, thereby avoiding the need for a priori knowledge of correctness rules for detecting violations. CP-Miner [22] utilizes inconsistent identifier names among similar pieces of code to find Copy-paste related bugs. PR-Miner [23] detects inconsistent API usage among programming rules as potential bugs. Ammons *et al.* [9] and Kremenek *et al.* [20] considered inconsistencies in the context of program specifications, while Xie *et al.* [33] use redundancies in programs, such as idempotent operations, unused values, dead code, un-taken conditional branches, and redundant null-checks, to flag possible errors. More semantic-aware inconsistencies can also be useful for bug detection. For example, Dillig *et al.* [11] use inconsistent uses of a same pointer to find null-pointer dereference errors.

VII. Conclusions

The pure lexical semantics of CPP macros can easily result in programming errors. Thus far no effective tool exists to statically validate the correct usage of macros. In this paper, we have introduced a general characterization of macro errors and developed an algorithm to detect many difficult-to-detect macro errors while preserving the flexibility of CPP. Our technique is based on normalizing macro calls and comparing them for structural dissimilarity to detect potential macro-related errors. Evaluated on a number of open-source projects, our tool, CppCheck, is precise and reports very few false positives. It is also efficient, adding negligible overhead over traditional parsing. We believe it is practical and can be used routinely to check for macro-related errors and validate macro usage.

Acknowledgments

We thank Hartmut Kaiser for his support of and help with Boost Wave. We would also like to thank Mark Gabel, Jeremiah Willcock and the anonymous reviewers for useful feedback on earlier drafts of this paper.

References

[1] Boost Wave Preprocessor. http://www.boost.org/libs/wave/index.html.

[2] The C Preprocessor. http://gcc.gnu.org/onlinedocs/cpp/.

[3] CERT: Use parentheses within macros around parameter names. https://www.securecoding.cert.org/confluence/display/cplusplus/PRE01-CPP.+Use+parentheses+within+macros+around+parameter+names.

[4] Cpp macro pitfalls. http://gcc.gnu.org/onlinedocs/cpp/Macro-Pitfalls.html.

[5] Fun with C++: Macro pitfalls. http://yaserzt.com/blog/2008/02/29/fun-with-c-macro-pitfalls-batch-1/.

[6] ROSE open source compiler infrastructure. http://rosecompiler.org/.

[7] SQLite database. http://www.sqlite.org/.

[8] Gimpel Software, PC-lint and FlexeLint. http://www.gimpel.com/html/lintinfo.htm, 2006.

[9] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages (POPL)*, 2002.

[10] G. J. Badros. PCP3: A C front-end for preprocessor analysis and transformation. Technical report, University of Washington, 1997.

[11] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 435–445, 2007.

[12] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP)*, pages 57–72, 2001.

[13] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[14] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96, New Orleans, USA, 6–9 Dec. 1994.

[15] J.-M. Favre. CPP denotational semantics. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 22–31. IEEE Computer Society, 2003.

[16] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Sept. 1998.

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

[18] S. C. Johnson. Lint, a C program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[19] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *LFP'86: Conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM.

[20] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 161–176, 2006.

[21] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 49–57. IEEE Computer Society Press, May 1994.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–302, 2004.

[23] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 306–315, 2005.

[24] B. McCloskey and E. A. Brewer. ASTEC: a new approach to refactoring C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 21–30, 2005.

[25] C. A. Mennie and C. L. A. Clarke. Giving meaning to macros. In *12th International Workshop on Program Comprehension (IWPC)*, pages 79–88. IEEE Computer Society, 2004.

[26] Y. Padioleau. Parsing C/C++ code without pre-processing. In *18th International Conference on Compiler Construction (CC)*, pages 109–125, 2009.

[27] A. S. M. Sajeev and D. A. Spuler. Static detection of preprocessor macro errors in C. Technical Report JCU-CS-92/7, Department of Computer Science, James Cook University, July 1 1992.

[28] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations, 2003.

[29] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Trans. Software Eng*, 29(11):1019–1030, 2003.

[30] S. Sýkora. Writing C/C++ macros: Rules, tricks and hints. *Stan's Library*, 1, 2004.

[31] L. Vidács, Á. Beszédes, and R. Ferenc. Columbus schema for C/C++ preprocessing. In *CSMR*, pages 75–84. IEEE Computer Society, 2004.

[32] D. Weise and R. Crew. Programmable syntax macros. In *Programming Language Design and Implementation(PLDI) 1993*, pages 156–165, New York, NY, USA, 1993. ACM.

[33] Y. Xie and D. R. Engler. Using redundancies to find errors. In *Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 51–60, 2002.