Global Analysis and Transformations in Preprocessed Languages

Diomidis Spinellis, *Member*, *IEEE*

Abstract—Tool support for refactoring code written in mainstream languages such as C and C++ is currently lacking due to the complexity introduced by the mandatory preprocessing phase that forms part of the C/C++ compilation cycle. The defintion and use of macros complicates the notions of scope and of identifier boundaries. The concept of token equivalence classes can be used to bridge the gap between the language proper semantic analysis and the nonpreprocessed source code. The *CScout* toolchest uses the developed theory to analyze large interdependent program families. A Web-based interactive front end allows the precise realization of rename and remove refactorings on the original C source code. In addition, *CScout* can convert programs into a portable obfuscated format or store a complete and accurate representation of the code and its identifiers in a relational database.

Index Terms—Refactoring, preprocessor, program families, renaming, C, C++, reverse engineering.

1 Introduction

Refactoring program transformations are widely regarded as a significant method for performing design level changes. The complexity of design changes performed on an established source code base is often harnessed by having incremental refactoring operations performed by humans assisted by specialized tools. However, tool support for the mainstream languages C and C++ is currently lacking, although the theory behind the concept is clearly understood. The reason behind this state of affairs is the complexity introduced by the mandatory preprocessing phase that forms part of the C/C++ compilation cycle. The problem, in short, is that macros complicate the notion of scope and the notion of an identifier. For one, preprocessor macros and file inclusion create their own scopes. This is, for example, the case when a single textual macro using a field name that is incidentally identical between two structures that are not otherwise related is applied on variables of those structures. In addition, new identifiers can be formed at compile time via the preprocessor's concatenation operator.

The source code analysis problems introduced by the C preprocessor can be overcome by considering the scope of preprocessor identifiers during a program's language proper semantic analysis phase. Having performed this analysis, refactoring transformations can be performed by tagging all identifiers with their original source code position and taking into account the identifier equivalence classes formed by the combined preprocessor and language proper scopes.

In the following sections, we describe the problems introduced by preprocessing and introduce algorithms for precisely mapping a C/C++ program's semantic information to its nonpreprocessed source code. Furthermore, we demonstrate the application of our methods in the *CScout* toolchest[1] that programmers can use to perform rename and remove refactorings.

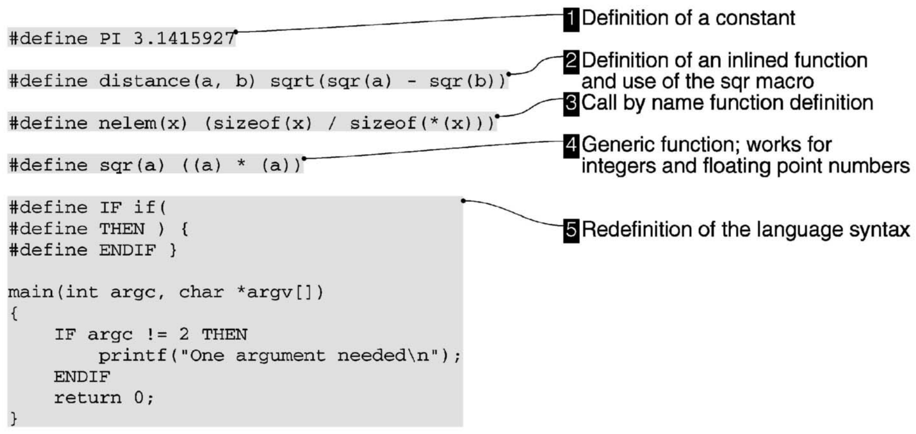
2 Work Context

Source-to-source transformations [1] in a body of code can serve a variety of goals. The resulting new code may be easier to maintain and reuse, be more readable, operate faster, or require less memory than the old code; many of the transformations can be described under the general term of *refactoring* [2], [3], [4], [5]. Common examples of refactorings include the encapsulation of fields, the hiding of methods, the replacement of conditionals with polymorphism, various rename and removal operations, and the movement of fields and methods up and down a class hierarchy. The automation of some of these transformations is in principle straightforward; it can be implemented by rearranging a syntactic representation of the code and generating the new code from that representation. As an example, the parse tree of a Java or Ada program can be manipulated in a way that preserves its meaning and then flattened again to create a new, equivalent source code body that will represent the program after the transformation. When the result of these transformations is supposed to be code that will be read and maintained by humans, an important goal is the preservation of the original format, identifier names, and comments. In our previous example, this can be accommodated by incorporating into each parse tree node the whitespace (including comments) surrounding it and associating the original names with identifier nodes. Parse trees can also be used to analyze program code identifying interdependencies between units such as functions, classes, modules, and compilation units, locating entity definitions, and as a basis for determining program

• *The author is with the Department of Management Science and Technology, Athens University of Economics and Business, Patision 76, GR-104 34 Athina, Greece. E-mail: dds@aueb.gr.*

Manuscript received 8 Oct. 2002; revised 24 Mar. 2003; accepted 6 Aug. 2003.
Recommended for acceptance by T. Reps.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 117534.

1. http://www.spinellis.gr/cscout.



```
#define PI 3.1415927

#define distance(a, b) sqrt(sqr(a) - sqr(b))

#define nelem(x) (sizeof(x) / sizeof(*(x)))

#define sqr(a) ((a) * (a))

#define IF if(
#define THEN ) {
#define ENDIF }

main(int argc, char *argv[])
{
    IF argc != 2 THEN
        printf("One argument needed\n");
    ENDIF
    return 0;
}
```

1 Definition of a constant
2 Definition of an inlined function and use of the sqr macro
3 Call by name function definition
4 Generic function; works for integers and floating point numbers
5 Redefinition of the language syntax

Fig. 1. Preprocessor macro definitions.

slices [6]. Tools that aid program code analysis and transformation operations are often termed *browsers* [7, pp. 297-307] and *refactoring browsers* [8], respectively. A different line of research [9], [10] focuses on the detection and implementation of refactorings without human intervention; this approach is outside the scope of our work.

2.1 Preprocessing

A complication arises in languages that include a preprocessing phase as part of the compilation, such as C [11], [12], C++ [13], Cyclone [14], PL/I, and many assembly-code dialects. The preprocessing step typically performs *macro-substitutions* replacing at a purely lexical level, token sequences with other token sequences, *conditional compilation*, *comment removal*, and *file inclusion*. Commands of the C/C++ preprocessors are line-based and always start with a # character. The `#include` command inserts at the point it is encountered the contents of the file specified as its argument. The `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` commands perform conditional compilation, skipping code blocks depending on the value of a—compile-time evaluated—expression; in practice, most cases involve simple tests for macro definitions. Finally, the `#define` command is used to define identifier or function-like macro replacements. Constant, integer-valued expressions are evaluated by the preprocessor only for the purpose of conditional compilation; however, two preprocessor-specific operators allow, when performing macro-substitution, the conversion of tokens into strings (unary operator `#`) and the concatenation of adjacent tokens (binary operator `##`).

Macro substitutions are often trivial, used to define constants (Fig. 1, item 1), inline-compiled functions (Fig. 1, item 2), or implement call-by-name semantics (Fig. 1, item 3). However, macro substitutions are also often used to create *generic* functions overcoming limitations of the language's type system (Fig. 1, item 4), affect the language's reserved words in, often misguided, attempts to enhance readability (Fig. 1, item 5), create shortcuts for initializing structures (Fig. 2a), and dynamically generate new code sequences, such as function definitions, by pasting together tokens, or substituting operators and whole code blocks (Fig. 2b) [15]. File inclusion is typically used to bring into the scope of a given compilation unit declarations of elements defined in other units (and declared in separate header files), thereby providing a way to communicate declarations and type information across separately-compiled compilation units.

As a result of the changes introduced by the preprocessing phase, the parsing of the language proper and, therefore, any meaning-preserving transformation, cannot be performed unless the code is preprocessed. However, after the preprocessing step has been performed, the parse tree-based transformation approach we outlined cannot be used because the tree contains the code in the form it has after its preprocessing. This code differs significantly from the original code and, in that representation it is neither portable (since standard included files typically differ between compilers, architectures, and operating systems), nor readable or maintainable by people. In addition, file inclusion introduces complex dependencies between files, propagating bindings between declared, defined, and used identifiers upward and downward in the file inclusion tree.

As an illustrative example of the complications introduced by preprocessing, the code body of a simple loop written in C to copy characters from the program's standard input to its output expands from 48 to 214 characters, while the inclusion of the Microsoft Windows SDK `windows.h` header file results in a preprocessed source code body of 158,161 lines, with identifiers spanning as many as 39 different files. Consequently, and to the best of our knowledge, there are no known general purpose approaches for automatically performing nontrivial source-to-source transformations on C and C++ code in a way that preserves the original code structure. This is a significant problem because these languages are popular, huge collections of code are written in them, and many important maintenance-related activities such as refactoring could be automated and performed in a safe and cost-effective manner. Ernst et al. [16] provide a complete empirical analysis of the C preprocessor use, a categorization of macro bodies, and a description of common erroneous macros found in existing programs. Furthermore, recent work on object-oriented design refactoring [5] asserts that it is generally not possible to handle all problems introduced by preprocessing in large software applications.

```
extern int v_left, v_right, v_top, v_bottom;


#define sv(x) {#x, &v_ ## x}


struct symtab {
    char *name;
    int *val;
} symbols[] = {
    sv(left),
    sv(right),
    sv(top),
    sv(bottom),
};
```

Code after preprocessing:

```
extern int v_left, v_right, v_top, v_bottom;


struct symtab {
    char *name;
    int *val;
} symbols[] = {
    {"left", &v_left},
    {"right", &v_right},
    {"top", &v_top},
    {"bottom", &v_bottom},
};
```

Fig. 2a. Structure initialization using token stringization and concatenation.

2.2 Analysis and Transformation Approaches

Analysis and transformation of C and C++ code can be performed at different levels of code integration based on a variety of techniques. The concepts of noise (extraneous matches) and silence (missed matches) can be used to judge the efficacy of a given method. As an example, consider the task of locating occurrences of a given entity (e.g., a function, or a structure member) in program code using a text editor's search function for the entity's name.

Noise will occur through matches inside comments and strings, in different name spaces (e.g., label names), or outside the scope being searched.

Silence will result from matches that should occur, but are missed: For example, a search for the definition and uses of the `blit_xor` function in the code in Fig. 2b would fail to locate its definition through the `defblit` macro.

We can classify the techniques typically used based on the level of code integration they apply to.

Programmers typically deal with individual *source files*. Most current source transformation tools for C/C++ operate at a lexical level on single files. *Code beautifiers* such as the Unix-based *cb* and *indent* tools can improve the readability of programs by rearranging whitespace. As the definition of whitespace is the same for both the C preprocessor and the C/C++ language proper, the conservative transformations these tools employ can in most cases be safely applied. The tools reformat the program based on a naive parsing of the code that takes hints from delimiters such as braces and brackets; their operation can be fooled by the use of certain macro substitutions, but, since the tools only rearrange whitespace, the program will only be suboptimally formatted yet still perform its intended operation. *Source code editors* work following imperative, task-oriented, text-processing commands. Some editors can automatically indent code blocks, or display reserved words and comments using special colors; the approaches employed depend on heuristics similar to those used by the code beautifiers.

A source file together with the files it includes forms a *compilation unit*. A number of code analysis and code generation tasks are performed on this entity. Compilers typically perform (after the preprocessing phase) syntactic and semantic analysis and code generation. In addition, tools such as *lint* [17] and its descendants preprocess the file (sometimes using custom-developed include files) and employ heuristics to detect common programmer errors. The *PCp*^{3} preprocessor-aware C source code analysis tool [18] tightly integrates a customized preprocessor with a C language parser to allow code written in Perl to accurately analyze the use of preprocessor features. The object file resulting from the compilation can also be analyzed to accurately determine, at least, all globally visible symbols defined and referenced in a given compilation unit (subject to name-mangling restrictions, see below).

Multiple compilation units are linked together into a *linkage unit*, which, in most cases, comprises an executable program. The linkers typically lack information about the syntax and semantics of the language; as a result, techniques such as *name mangling* [19, pp. 121-127] are often used to perform some kind of primitive type checking at link time [20]. A number of integrated development environments and tools such as *cscope* and *ID utils* allow the user to specify the files comprising a linkage unit and thereby perform textual searches or replacements across all files. In addition, profilers [21] and debuggers [22] can be used to analyze the program's operations; however, they fail to deal with source elements that were replaced during the preprocessing step.

```
#define defblit(name, op) \

blit_ ## name(char *s, char *d, int c) \

{ \

    for (int i = 0; i < c; i++) \

        d[i] = s[i] op d[i]; \

}



defblit(xor, ^)

defblit(and, &)

defblit(or, |)
```

Code after preprocessing:

```
blit_xor(char *s, char *d, int c) { for (int i = 0; i < c; i++)\

d[i] = s[i] ^ d[i]; }

blit_and(char *s, char *d, int c) { for (int i = 0; i < c; i++)\

d[i] = s[i] & d[i]; }

blit_or(char *s, char *d, int c) { for (int i = 0; i < c; i++)\

d[i] = s[i] | d[i]; }
```

Fig. 2b. Creating new functions based on a template and an operator.

Finally, a *workspace* or program family [5] consists of a collection of source code files that form multiple linkage units. The collection, typically organized through common shared header files and separately compiled library files, contains many interdependencies between files belonging to different compilation units through the sharing of source code via source file inclusion (during preprocessing) and linking. Typical examples of workspaces are operating system distributions where both the kernel and hundreds of user programs depend on the same include files and libraries as well as software product lines [23], [24]. Again, only textual processing can be performed at this level, either using an IDE, or the text and file processing tools available in environments such as Unix [25]. As an example, locating the string `printf` in the source code of an entire operating system source code base can be performed using the command `find -name '*.[ch]' -print | xargs grep printf`.

In all the cases, we outlined above, an automated textual operation, such as a global search and replace, will indiscriminately change identifiers across macros, functions, and strings. Code analysis tasks, such as the location of all files where a particular identifier is used, are equally haphazard operations. This state of affairs makes programmers extremely reluctant to perform large scale changes across extensive bodies of C/C++ code. This observation is anecdotally supported by the persistence of identifier names (such as `creat` and `splx` in the Unix kernel) decades after the reasons for their original names have become irrelevant. The readability of existing code slowly decays as layers of deprecated historical practice accumulate [26, pp. 4-6, 184] and even more macro definitions are used to provide compatibility bridges with modern code. Two theoretical approaches proposed for dealing with the problems of the C preprocessor involve the use of mathematical concept analysis for dealing with cases where the preprocessor is used for configuration management [27], and the definition of an abstract language for capturing the abstractions for the C preprocessor in a way that allows formal analysis [28].

In the following sections, we describe a method for precisely identifying and analyzing the use of identifiers in the original, nonpreprocessed body of source code, taking into account the syntax and semantics of the C or C++ programming language. Tools using the precise classification of identifiers can then be used to perform renaming and remove refactorings, map dependencies between source files, identify the impact of architectural changes, optimize the build process, and generate high-quality metrics. Being able to globally rename an identifier name in an efficient manner can help programmers refactor and maintain their code by having identifier names correctly reflect their current use and the project's contemporary naming conventions. The practice is also important when code is reused by incorporating one body of code within a different one; in situations where the reused code will not be maintained in its original context, the naming of all identifiers of the grafted code should change to conform to the style used in the workspace into which it was added. Finally, the replacement of identifiers with mechanically generated names can be used as a method to obfuscate programs so that they can be distributed without exposing too many details of their operation [29].

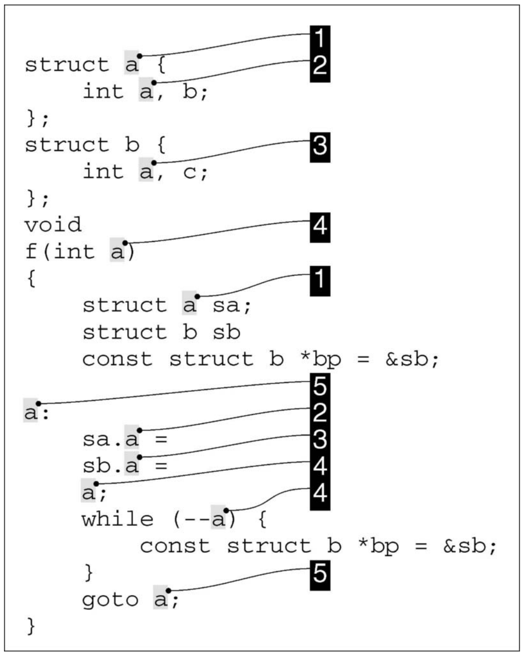
Modern language designs have tried to avoid the complications of the C/C++ preprocessor. Many features of the C++ language such as *inline* functions, and *const* declarations supplant some common preprocessor uses. Java does not specify a preprocessing step, while C# [30] specifies a distinct scope for the defined preprocessor identifiers that is visible only within the context of other preprocessor directives. However, the preprocessing technology is still relevant in a number of contexts. Preprocessing is often used as a source-to-source transformation mechanism to extend existing languages [31]. In addition, the preprocessor is also used, in all the C++ implementations we are aware of, to implement the language's standard library, and, in particular, to support—via the file inclusion mechanism—the generic programming facilities of templates and the standard template library (STL) [32].

3 Approach Description

The problem we will solve can be described as follows:

> A set of source code files $\mathcal{S}$ for a statically-scoped first order language is processed before compilation by a general purpose macro processor supporting file inclusion, macro substitution, and token concatenation. A single instance of a (identifier) token t in one of the files $\mathcal{S}_i$ is modified resulting in a new token t'. Propagate this change creating a new, syntactically and semantically equivalent set of files $\mathcal{R}$ differing from $\mathcal{S}$ only in the relevant names of identifier tokens.

With the term "first order language," we restrict our language (or our approach's domain) to systems that do not have (or utilize) metaexecution or reflection capabilities. Systems outside our domain's scope include metainterpreters implemented in nonpure Prolog, Scheme meta-evaluators [33, pp. 286-315], runtime calls to the interpreter in Perl and Tcl/Tk, and the reflection [34] capabilities of Java [35, pp. 219-223] and C# [30]. In addition, the problem definition assumes that t and t' do not directly or indirectly (after preprocessing) have a special meaning in the language the programs are written in that will result in $\mathcal{R}$ never being semantically equivalent to $\mathcal{S}$. As an example, it would in most cases not be legal to modify an existing `printf` identifier in a C program (`printf` is part of the standard library for hosted implementations), or to rename an identifier into `class` in a C++ program (`class` is a C++ reserved word). This clarification is needed because tokens treated as identifiers in the preprocessing stage can have a special meaning (reserved word, or name of a library facility) in the target language. Finally, we assume that t' does not clash in any legal context with existing identifiers in a way that would change the semantic meaning of $\mathcal{R}$. The analysis tasks we identified in the previous sections can be trivially performed by locating the identifiers that would require modification without actually changing them.



```
struct a {
    int a, b;
};
struct b {
    int a, c;
};
void
f(int a)
{
    struct a sa;
    struct b sb
    const struct b *bp = &sb;
a:
    sa.a =
    sb.a =
    a;
    while (--a) {
        const struct b *bp = &sb;
    }
    goto a;
}
```

Fig. 3. Examples of semantic identifier equivalence.

Informally, we want to automate the process followed by a programmer renaming or locating all instances of an entity's name. We will describe the approach in four steps of gradually increasing refinement. Central to our approach is the notion of *token equivalence*. Given the token to be modified t, we want to locate the set of tokens $\mathcal{E}_t$ occurring in $\mathcal{S}$ so that, by changing each $t_i : t_i \in \mathcal{E}_t$ into t', we will obtain $\mathcal{R}$. We define the set of tokens $\mathcal{E}_t$ as the *equivalence class* of t. For a complete workspace, we maintain a global set $\mathcal{V}$ containing all equivalence classes $\mathcal{V} : \forall i \mathcal{E}_i \in \mathcal{V}$.

3.1 Semantic Equivalence

We first consider *semantic equivalence* in the absence of preprocessing. We define two identifiers to be semantically equivalent if these have the same name and statically refer to the same entity following the language's scoping and namespace rules. The definition of static scope-based resolution excludes the semantic equivalence introduced by aliasing.

As an example, consider the (contrived) example of the C++ program in Fig. 3. The equivalence classes for the identifier a are marked in the right margin. The identifier a is used in five distinct namespaces defined in C++: It is the name of a structure tag (item 1), a member of two different structures (items 2 and 3) (the members of each structure reside in their own namespace), a formal argument to the function f (item 4), and a label in the function body (item 5). Significantly, the two instances of the pointer variable bp *are not* semantically equivalent under our definition because, although they both point to the same underlying object (sb), they are different according to the scoping rules of the C++ language.

Determining the semantic equivalence classes of identifiers is relatively straightforward and performed routinely by all compilers: One follows the scope rules of the language specification [36, pp. 474-479]—taking into account type inference rules for matching the declarations and corresponding uses of structure and union members.

3.2 Lexical Equivalence

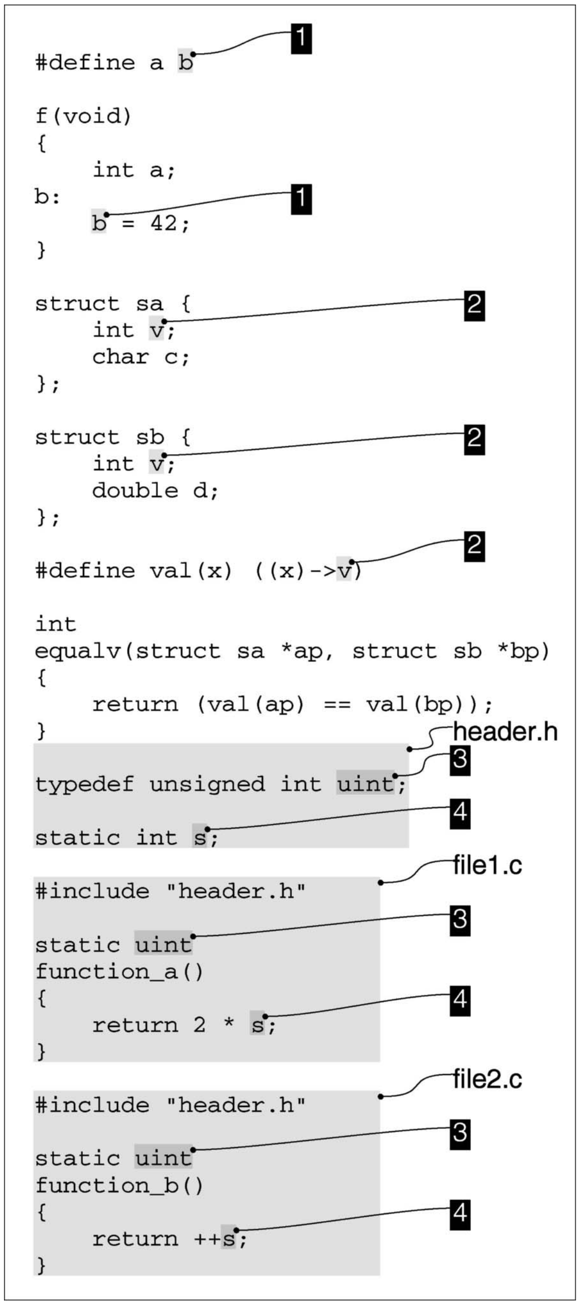
The *lexical equivalence* of tokens forms the basis of our method by taking into account the scope and semantics of preprocessing commands. A lexical equivalence class may contain identifiers appearing on the right-hand side of macro definitions and identifiers in the code body. Consider the C code in Fig. 4. The two instances of b (item 1) form a lexical equivalence class: Both must be changed to the same new value in a transformation that preserves the operation of the original program. Notice how the name of the label b is not part of the equivalence class; modifying it independently will not affect the rest of the program.

The operation of the preprocessor can also create equivalence classes between identifiers that would not be semantically equivalent under a language's rules. The macro val in Fig. 4 defines an accessor function; a common occurrence in C programs. Note that the macro is used in a polymorphic fashion: It is applied to two variables containing pointers to a different structure type. This macro definition and application brings the semantically distinct v fields of the structures a and b under the same lexical equivalence class. All three instances of v must be changed to the same new name in a meaning-preserving source transformation. Note that the two structure declarations, the definition of the the val macro, and its application could occur in four different files (that may even belong to different linkage units) combined using the preprocessor's file inclusion commands.

Lexical equivalence can also be introduced by the macro processor without macro definition, through the inclusion of the same file in multiple compilation units (a common method for importing library declarations). If the included file declares elements with static (compilation unit) scope, then identifiers in two different compilation units may be lexically equivalent, even though they are defined in isolated scopes, as a change in one of the files, must be propagated through the common included file to the other. A representative example also appears in Fig. 4. The identifiers unit and s are shared between the two C files only because they both include the same header file.

Tokens can be grouped into equivalence classes using the following algorithm.

1. Split the original nonpreprocessed source code into tokens; each token t_i is set to belong to a unique equivalence class $\mathcal{E}_i$.

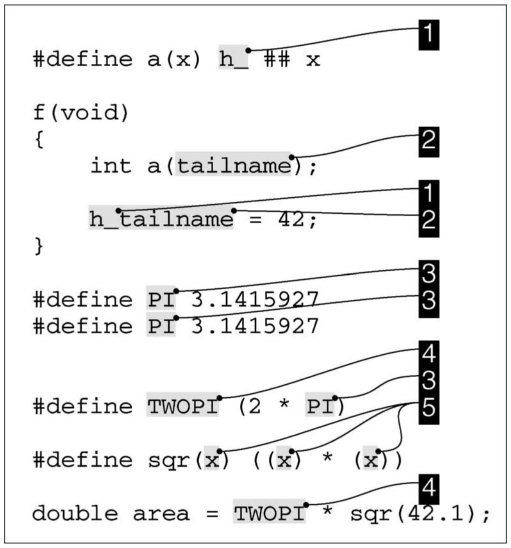


```
#define a b

f(void)
{
    int a;
b:
    b = 42;
}

struct sa {
    int v;
    char c;
};

struct sb {
    int v;
    double d;
};

#define val(x) ((x)->v)

int
equalv(struct sa *ap, struct sb *bp)
{
    return (val(ap) == val(bp));
}

typedef unsigned int uint;

static int s;

#include "header.h"

static uint
function_a()
{
    return 2 * s;
}

#include "header.h"

static uint
function_b()
{
    return ++s;
}
```

Fig. 4. Cases of lexical equivalence.

2. Perform macro substitutions; the resulting code shall consist of references to the original tokens.
3. Parse the program.
4. Perform semantic analysis according to the language's scoping and namespace rules. When two identifiers $t_a \in \mathcal{E}_a$ and $t_b \in \mathcal{E}_b$ are found to be semantically equivalent and $\mathcal{E}_a \neq \mathcal{E}_b$ merge the corresponding equivalence classes $\mathcal{E}_a$ and $\mathcal{E}_b$ into a new one $\mathcal{E}_n = \mathcal{E}_a \cup \mathcal{E}_b$ adding $\mathcal{E}_n$ into $\mathcal{V}$ and removing $\mathcal{E}_a$ and $\mathcal{E}_b$ from $\mathcal{V}$.

Having grouped tokens into equivalence classes, source code modifications can be performed by locating the equivalence class $\mathcal{E}_r$ to which a particular token t_r belongs and correspondingly modifying all tokens in that class in the source code position they were found.



```
#define a(x) h_ ## x

f(void)
{
    int a(tailname);

    h_tailname = 42;
}

#define PI 3.1415927
#define PI 3.1415927


#define TWOPI (2 * PI)

#define sqr(x) ((x) * (x))

double area = TWOPI * sqr(42.1);
```

Fig. 5. Equivalence classes in partial definitions, macro names, and arguments.

3.3 Partial Lexical Equivalence

We introduce the notion of *partial lexical equivalence* to deal with the issue of token concatenation. The preprocessor used in the ANSI C and C++ languages can concatenate tokens to form new ones using the ## operator. In addition, historical practice in pre-ANSI versions of the preprocessor also allowed token concatenation by separating them by a comment in a macro replacement pattern. The result of this capability is that there is no one-to-one equivalence between tokens in the original source code and the tokens on which the semantic analysis is performed. Parts of an identifier can belong to different equivalence classes, as shown in Fig. 5 by the identifiers marked 1 and 2.

The procedure we described in the previous section can be amended to deal with this case by handling subtokens. Each token t_i now consists of a concatenation (denoted by the $\oplus$ symbol) of n_i subtokens s_{i_j}:

$$t_i = s_{i_1} \oplus s_{i_2} \oplus \ldots \oplus s_{i_{n_i}}. \tag{1}$$

Each subtoken covers the maximal character sequence that appears continuous on all tokens associated with that part of a given equivalence class. As an example, the identifier `h_tailname` in Fig. 5 will be processed as `h_` $\oplus$ `tailname`. Semantic equivalence is still determined in terms of tokens, but equivalence classes can now consist of subtokens. When two tokens t_1 and t_2 are found to be semantically equivalent, these are processed with the following algorithm.

1. Make t_1 and t_2 consist of n' subtokens
$$t_i = s'_{i_1} \oplus s'_{i_2} \oplus \ldots \oplus s'_{i_{n'}} \tag{2}$$
of correspondingly equal length:
$$\forall j \in [1 \ldots n'] : \mathrm{len}(s'_{1_j}) = \mathrm{len}(s'_{2_j}). \tag{3}$$
This operation is performed by splitting appropriate subtokens (correctly adjusting the corresponding equivalence classes) so that, in the end, the set of split positions of both tokens is the union of the split positions of each token, i.e., both tokens are split at exactly the same positions:
$$\begin{aligned}&\forall i \in \{1,2\} \sum_{j=1}^{n'} \mathrm{len}(s'_{i_j}) \\ &\in \bigcup_{i=1}^{2} \left\{ j = 1 \ldots n_i : \sum_{k=1}^{j} \mathrm{len}(s_{i_j}) \right\}.\end{aligned} \tag{4}$$
As an example, if the token `h_` $\oplus$ `tailname` was found to be semantically equivalent with the token `h_tail` $\oplus$ `name`, both tokens would be split into `h_` $\oplus$ `tail` $\oplus$ `name`.
Every time a subtoken $u \in \mathcal{E}_\alpha$ is split into new subtokens $u_1, u_2 : u_1 \oplus u_2 = u$, its equivalence class $\mathcal{E}_\alpha$ is removed from $\mathcal{V}$ and split into $\mathcal{E}_{\alpha_1}$ and $\mathcal{E}_{\alpha_2}$ in a way that satisfies the following:
$$\begin{aligned}&\forall v \in \mathcal{E}_\alpha : v = v_1 \oplus v_2 \wedge \mathrm{len}(v_1) = \mathrm{len}(u_1) \\ &\qquad \wedge \mathrm{len}(v_2) = \mathrm{len}(u_2) \Leftrightarrow \\ &v_1 \in \mathcal{E}_{\alpha_1} \wedge v_2 \in \mathcal{E}_{\alpha_2}.\end{aligned} \tag{5}$$
2. For each equal length subtoken pair $s'_{1_i}, s'_{2_i} : s'_{1_i} \in \mathcal{E}_\alpha \wedge s'_{2_i} \in \mathcal{E}_\beta$, merge $\mathcal{E}_\alpha$ with $\mathcal{E}_\beta$ by creating a new equivalence class $\mathcal{E}_\gamma$: $\mathcal{E}_\gamma = \mathcal{E}_\alpha \cup \mathcal{E}_\beta$, adding $\mathcal{E}_\gamma$ into $\mathcal{V}$ and removing from $\mathcal{V}$ $\mathcal{E}_\alpha$ and $\mathcal{E}_\beta$.

Thus, the semantic unification of `h_` $\oplus$ `tailname` (where `h_` $\in A$ and `tailname` $\in B$) with `h_tailname` (where `h_tailname` $\in C$) would be performed by first splitting C into C_1 (containing tokens of length 2) and C_2 (containing tokens of length 8) and having

$$\mathcal{V}' = \mathcal{V} - \{A, B, C\} + \{C_1 \cup A, C_2 \cup B\}.$$

3.4 Preprocessor Equivalence

The equivalence rules we have introduced up to this point only cover the tokens emitted after the preprocessing phase. To complete our solution, we need to consider tokens that are internal to the preprocessing phase, i.e., the tokens that appear on the left-hand side of the macro definitions. We distinguish two types of tokens: macro names and macro formal parameters. Both already belong to equivalence classes as part of the initial program tokenization. These classes need to be processed following the semantics of the preprocessor. Because preprocessing is performed as part of the process we described, the rather convoluted semantics of the preprocessor have to be taken into account while actually performing the requisite evaluation operations. In particular, the scoping of macro definitions, as delineated by `#define` and `#undef` pairs, has to be calculated as part of the preprocessor operation.

Under the preprocessor regime, macro names are semantically equivalent with the tokens they replace. When a macro name token $t_a : t_a \in \mathcal{E}_a$ replaces a token $t_b : t_b \in \mathcal{E}_b$ in the program body, the equivalence classes $\mathcal{E}_a$ and $\mathcal{E}_b$ are to be merged following the process described in step 4 in Section 3.2. In addition, multiple definitions of the same macro with the same arguments and replacement should unify both the macro name and its arguments and the corresponding replacement lists since they form legal multiple definitions of the same macro [12, § 3.8.3].

The formal arguments of preprocessor macros are only meaningful within the context of the actual macro definition. Therefore, each occurrence of a formal macro argument $t_a : t_a \in \mathcal{E}_a$ found as a token $t_b : t_b \in \mathcal{E}_b$ in the macro body, shall result in the equivalence classes $\mathcal{E}_a$ and $\mathcal{E}_b$ being merged following again the process described in step 4 in Section 3.2.

An example of the issues described above is depicted in Fig. 5. Note that the merged equivalence class (item 3) of the macro `PI` will not be determined until the expansion of the macro `TWOPI` as part of the `area` initialization as under the ANSI C standard after a macro substitution the replacement token sequence is repeatedly rescanned for more defined identifiers [12, § 3.8.3.4].

4 COMPLICATIONS

The application of the method we described in the previous section on real-world programs faces a number of complications.

4.1 Conditional Compilation

Conditional compilation results in code parts that are not always processed. Some of them may be mutually exclusive defining, e.g., different operating system dependent versions of the same function. The problem can be handled with multiple passes over the code, or by ignoring conditional compilation commands. This process may need to be guided by hand, because conditionally compiled code sections are often specific to a particular compilation environment.

4.2 Imported Libraries

A useful feature of our proposed method would be the ability to protect the user from inadvertently modifying identifiers reserved by the language, or the specific programming environment. This can be accomplished by marking the set of included header files that declare such identifiers as read-only. Orderly administered compilation environments are typically organized in this way using the operating system file protection mechanisms. An equivalence class E_r that contains at least one token t_r from a file marked as read-only is considered immutable: No tokens in that class can be modified. If any part of a token belongs to an immutable class, then the whole token is considered immutable.

4.3 Code Shared among Different Programs

Workspaces or program families are often composed of a number of linkage units that loosely share many different source files. As an example, consider tightly integrated variants of Unix such as the Free/Net/OpenBSD systems and the GNU/Linux distributions. The kernel, and many of the tools share a number of project-wide, yet not part of the ANSI or a POSIX standard libraries such a *readline* and *ndbm*. It would be useful to be able to change public identifier names in libraries shared among different programs and have the changes propagated to all source code using them. The basis of our approach can be extended to cover such cases by keeping a global table of equivalence classes for all programs (linkage units) belonging to a workspace. Note that no semantic or syntactic information is associated with the equivalence classes that span linkage units. Each equivalence class is just a collection of tokens falling under the equivalence rules observed while processing each compilation and linkage unit.

4.4 Dead and Partially Used Code

The method we described will not update code in macro definitions that are never used. In order to establish semantic equivalence between identifiers appearing in the body of a macro and other code, the macro has to be applied. Identifiers in macros that are never applied will therefore never be merged with other identifiers and will consequently never be updated. The best that can be done in this case is to flag such macro definitions, warning the user that these will not be updated. A user could then create dummy code to exercise those macros. A similar problem can occur for macro definitions that are not exercised over the full range of their applicable arguments. As an example, the polymorphic accessor macro `val` we examined in Fig. 4 could not be used to identify that a field `v` of a third structure `sc` belonged to the same equivalence class unless the macro was applied to at least one variable containing or pointing to such a structure. In both cases, the transformed programs would be entirely correct and semantically equivalent to the original ones, but would be stylistically lacking; a human operator could identify by context the meaning of an unused macro or the full applicability range of a partially used one and perform additional modifications. In addition, future maintenance changes could apply the macro to other structures resulting in a program that would not compile.

4.5 Undefined Macros

In the C language the use of an undeclared function acts as an implicit declaration for an `int`-returning function without parameter information [12, § 3.3.2.2]. Undefined macros are not however handled in the same way in the C preprocessor. Thus, the common idiom used for protecting header files against multiple inclusion

```
#ifndef HEADER_INCLUDED
/* Header code */
#define HEADER_INCLUDED
#endif /* HEADER_INCLUDED */
```

will not unify the (originally undefined) macro `HEADER_INCLUDED` with its subsequent definition. Similarly, multiple tests against the same undefined macro name, often used to conditionally compile nonportable code, will fail to unify with each other. The first problem can be solved by including all headers at least twice; hardly an ideal proposition. Both problems can also be solved by employing heuristic techniques or manually defining the macros before the corresponding test. None of the solutions is completely error-proof: A heuristic unifying undefined macros with a subsequent definition may fail in cases where the two are not related; the manual definition of a macro may cause the program to be processed in ways that are not appropriate for a given compilation environment.

5 The CScout Toolchest

We tested the viability of our approach by designing and implementing the *CScout* processing engine and associated back-end tools.

5.1 Functional Description

CScout can analyze collections of C programs and *yacc* [37] parser specifications to determine the equivalence classes of their identifiers. The processing involves preprocessing, parsing, and semantic analysis according to the approach we outlined in the previous sections. The resulting data structure can then be queried to determine the equivalence class of an identifier starting at a particular offset in a given file. In addition, methods associated with equivalence class objects return the number of members of an identifier class, as well as semantic properties of the equivalence class members. A domain specific language [38] is used to specify in detail how the source files are to be processed in terms of directory contents, preprocessor definitions, include file paths, read-only files, and the composition of linkage units. Based on the facilities provided by the *CScout* processing engine, we implemented three different back end tools.

The *refactoring browser* provides a Web-based interface to a system's source code and its identifiers for examining the source and aiding the realization of *rename* (e.g., "rename-variable" [39]) and *remove* (e.g., "Remove Parameter" [4]) refactorings. Using the *swill* embedded Web server library [40], the analyzed source code collection can be browsed by connecting a Web client to the tool's HTTP-server interface. Each identifier is linked to a Web page containing details of the equivalence class it belongs to. A set of hyperlinks allows users to

- Browse file and identifier names belonging to various semantic categories (e.g., read-only files, file-spanning identifiers, or unused identifiers).
- Examine the source code of individual files with hyperlinks allowing the navigation from each identifier to its equivalence class page.
- Specify identifier queries based on the identifier's namespace, scope, and name, and whether the identifier is writable, crosses a file boundary, is unused, occurs in files of a given name, is used as a typedef, or is a (possibly undefined) macro, or macro argument. The file and identifier names can also be specified in the query as regular expressions.
- Specify file queries based on the calculated file metrics, such as the number of defined functions, C statements, or preprocessor directives.
- View the semantic information associated with the identifiers of each equivalence class. Users can find out whether the equivalence class is read-only (i.e., at least one of its identifiers resides in a read-only file), and whether its identifiers are used as macros, macro arguments, structure, union, or enumeration tags, structure or union members, labels, typedefs, or as ordinary identifiers. In addition, users can see if the identifier's scope is limited to a single file, if it is unused (i.e., appears exactly once in the file set), the files it appears in, and the projects (linkage units) that use it. Unused identifiers point to functions, macros, variables, type names, structure, union, or enumeration members or tags, labels, or formal function parameters that can be safely removed from the program.
- Substitute a given equivalence class's identifier with a new user-specified name.
- Write back the changed identifiers into the respective source code files. A single pass through each processed source file will identify file offsets that mark the starting point of an equivalence class associated with a changed identifier name and substitute the number of characters that comprised the old identifier name with the new name. Thus, the only changes visible in the program will be the modified identifier names.

The above functionality can be used to semiautomatically perform rename and identify candidates for remove refactorings. Name clashes occurring in a rename refactoring are not detected since this feature would require reprocessing the entire source code base—a time consuming process. Remove refactorings can be trivially performed by hand, after identifiers that occur exactly once have been automatically and accurately identified.

The *obfuscate* back-end systematically renames the identifiers belonging to each equivalence class, thus creating a program representation that hinders reverse-engineering attempts. This source (with appropriate selection of the specified read-only files) will remain portable across architectures, and can be used as an architecture-neutral distribution format such as the one proposed by Hansen and Toft [41]. Since the source is distributed in its nonpreprocessed form, portability problems stemming from different contents of included files across systems are obviated. Furthermore, adding to the set of read-only files the system's source files where configuration information is specified (e.g., `config.h`) will result in a source code base where the configuration commands and macros are still readable and can be tailored to a given platform.

Finally, the SQL back end provides additional analysis and processing flexibility through the use of SQL commands. The SQL back end creates an ANSI SQL-92 script that contains all equivalence class properties and represents all links between equivalence classes, tokens, files, and projects. The scripts have been tested with *Postgres* and the embeddable, Java-based *hsqldb* databases. All program text that is not part of identifiers is stored in a separate database table allowing the complete source file base to be reconstituted from a (potentially modified) database.

5.2 Implementation Outline

CScout integrates in a single process the functionality of a multiproject build engine, an ANSI C preprocessor, and the parts of a C compiler up to and including the semantic analysis based on types.

The build engine functionality is required so that *CScout* will process multiple compilation and link units as a single batch. Only in this way can *CScout* detect dependencies across different files and projects. Each compilation unit can reside in a different directory and can require processing using different macro definitions or a different include file path. In a normal build process, these options are typically specified in a *Makefile* [42]. The *CScout* operation is similarly guided by a declarative workspace definition file.

The C preprocessor part converts the source code input into C preprocessor tokens. It tags each token with the file and offset where it occurs, and creates a separate equivalence class for each token. Operating system-specific system calls are used to ensure that files included by various compilation units using different names (e.g., through distinct directory paths or by following hard and symbolic links) are internally identified as the same source file. The preprocessor part performs the normal C preprocessor functions we described in Section 2.1. When defining and replacing macros and when evaluating preprocessor expressions *CScout* will merge equivalence classes as described in Section 3.4.

The semantic analysis part reclassifies the preprocessed tokens according to the (slightly different) token definition of the C language proper and follows the language's scoping and namespace rules to determine when two identifiers are semantically equivalent in the sense we described in Section 3.1. During this phase, each object identifier token is tagged with enough type information to allow the correct classification of structure and union member names appearing on the right of the respective access operators. A vector of symbol table stacks is used to implement the language's scopes and namespaces. Each time a symbol table lookup determines that the token currently being processed is semantically equivalent with a token already stored in the symbol table the corresponding equivalence classes are merged together.

The equivalence class merge operations we described in the previous two paragraphs are always performed taking into account the partial lexical equivalence rules we described in Section 3.3. If one of the two unified tokens consists of subtokens, the corresponding equivalence classes are split, if required, and then appropriately merged.

At the end of the semantic analysis phase, every token processed can be uniquely identified by its file and offset tuple $t(f, o)$ and is associated with a single equivalence class: $t \in \mathcal{E}_t$. This data structure can be saved into a relational database for further processing. Equivalence classes containing only a single C identifier denote unused identifiers that the *CScout* user should probably delete. The first step for renaming an identifier involves associating a new name with the identifier's equivalence class. After all rename operations have been specified and the changes to the source code are to be committed, each source file is reprocessed. When the read offset o' of a file f' being read matches an existing token $t'(f', o')$, its equivalence class $\mathcal{E}_{t'}$ is examined. If the equivalence class has a new identifier n associated with it, *CScout* will read and discard $\mathrm{len}(t')$ characters from the file and write n in their place. Creating a version of a file with identifier hyperlinks involves similar processing with the difference that *all* identifier tokens are tagged with a hyperlink to their equivalence class. Finally, obfuscating the source code involves associating a new random unique identifier replacement with each equivalence class.

5.3 Performance Indicators

The operation of *CScout* spans three different processing levels that are typically abstracted and individually performed in most C/C++ environments: preprocessing, compilation, and linking. Thus, at a given point of its operation, *CScout* may contain details about identifiers in hundreds of programs, thousands of source code files, and many more include files. The resulting toll on the use of memory and CPU resources is considerable and used to be prohibitive [43], but, thanks to the rapid increases of memory capacities and CPU speeds we are constantly witnessing, is now tolerable. In fact, we argue that software engineering tools have in many cases failed to utilize the hardware resources currently available on a typical workstation; *CScout* is an exception to this phenomenon.

Quantifying the performance of *CScout*, we note that the time required to process an application is about 1.5 times that required to compile it using the GNU C compiler with the default optimization level. The memory resources required are considerable and average about 560-620 bytes per source line processed. We have successfully used *CScout* to process medium sized projects, such as the *apache* Web server and the FreeBSD kernel, and have calculated that the analysis of multimillion source code collections (such as an entire operating system distribution) is not beyond the processing capability of a modern high-end server. Specifically, we estimate that our current implementation could process the complete FreeBSD system distribution (6 million lines of C code) on a 1GHz processor in three hours using 3.5GB of memory.

6 Conclusion

This paper presents a class of algorithms and tools for automatically analyzing and renaming identifier instances in languages that employ a macro preprocessing phase as part of the compilation cycle. Although preprocessing is not part of newly developed languages, large bodies of code written in languages such as C and C++ are still being developed and also need to evolve and be maintained. It is believed that tools based on the algorithms we outlined significantly enhance the arsenal of software developers writing and maintaining programs in C and C++.

6.1 Contribution

The contributions of our research can be categorized into the areas of software engineering tools and their applications, and programming language semantics. In the previous sections, we described an approach for precisely analyzing and processing identifiers in program families written in languages that incorporate a preprocessing step such as C and C++. Earlier research [5] indicated that it was not possible to handle the problem of preprocessing for refactoring C++ programs; our approach, although limited in the scope of the supported refactorings, can be extended to cover additional cases. Our method can also be incorporated into analysis systems such as the Wisconsin

Program-Slicing Project [44] that only handle a small subset of the C preprocessor.

The *CScout* toolchest we implemented demonstrates how a family of C programs can be analyzed to support: simple refactorings, a one-to-one semantic mapping of the complete source to a relational database, a robust architecture-neutral obfuscated code distribution format, and the extraction of semantically-rich metrics with exact references to the original source code. The use of a general-purpose database schema for storing the source's semantic information, effectively isolates the analysis engine from the numerous transformations and metric extraction operations that can be performed on the source, providing flexibility, domain specificity, and efficiency.

6.2 Critique

As was outlined in Section 4, there are pathological cases of code that are not covered by our approach. Although the exceptions are minor and would probably be never encountered in routine maintenance applications, they can decrease the trust programmers place on the tools using the proposed approach.

The absence of a name clash error checking facility in the *CScout* refactoring browser can be a problem, but it is an attribute of the implementation and not an inherent shortcoming of our method. Early on, we decided to dispose semantic information from memory once a scope had been processed so as to be able to handle software systems of a realistic size.

The *CScout* tools can process C, *yacc* and many popular C language extensions, but do not (yet) cover C++. Although the method clearly applies to C++, the effort to parse and semantically analyze C++ programs is an order of magnitude larger than the scope of the project we had planned. Furthermore, enhancing the refactoring browser or the database representation to cover some common object-oriented refactorings would be an even more complex task.

6.3 Extensions

The tools developed here can be extended in a number of ways.

First of all, the *CScout* tools should be able to parse and analyze other common languages such as C++, the *lex* program generator [37], and symbolic (assembly) code. Given that many projects automatically generate C code from specifications expressed in a domain specific language [31], we are currently experimenting with lightweight front ends that transform DSL programs into C code annotated with precise backreferences to the original DSL file. *CScout* will then be able to parse the synthetic code as C, but will display and modify the original DSL source code. Such capabilities will extend *CScout's* reach to a wider set of very large program families. In addition, the database schema used can be extended to separate code lines, and separately store and identify comments, strings, and, possibly, other constants. These last modifications will allow more sophisticated operations to be performed on the source code body employing a unified approach. Identifiers could also contain additional semantic information on their precise scope, type, as well as the structures and unions members belong to.

A related aspect of improvement concerns the integration with configuration and revision management tools. Large changes to source code bases, such as those that can be made by *CScout*, should be atomic operations. Although revision-control operations can currently be introduced into the modification sequence using a scripting language, a facility offering standardized hooks for specifying how the modifications are to be performed might be preferable.

Finally, the most ambitious extension concerns not the tool, but the theory behind its operation. Given that C/C++ code can now be directly modified in a limited way in its nonpreprocessed form, it is surely worthwhile to examine what other larger refactoring operations can be safely performed using similar techniques. These would necessitate the marking of syntactic structure in the token sequences to allow rearrangements at the parse tree level. The effect the preprocessor facilities have on the representation of these structures, and methods that can mitigate these effects are currently open questions.

Acknowledgments

Alexander Leidinger, Marcel Moolenaar, and Peter Wemm supported the porting and the execution of *CScout* on 64-bit architectures. The author also wishes to thank Alexios Zavras and the anonymous reviewers for their perceptive and constructive comments on earlier versions of this paper.

References

[1] R.M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *J. ACM*, vol. 24, pp. 44-67, 1977.
[2] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD thesis, Univ. of Illinois at Urbana-Champaign, Urbana-Champaign, 1992.
[3] W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray, *AntiPatterns Refactoring Software, Architectures, and Projects in Crisis.* Wiley, 1998.
[4] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.
[5] L. Tokuda and D. Batory, "Evolving Object-Oriented Designs with Refactorings," *Automated Software Eng.*, vol. 8, pp. 89-120, 2001.
[6] F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages*, vol. 3, no. 3, pp. 121-189, Sept. 1995.
[7] A. Goldberg and D. Robson, *Smalltalk-80: The Language.* Addison-Wesley, 1989.
[8] D. Roberts, J. Brant, and R.E. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, pp. 39-42, 1997.
[9] I. Moore, "Automatic Inheritance Hierarchy and Method Refactoring," *ACM SIGPLAN Notices*, vol. 31, no. 10, pp. 235-250, Oct. 1996.
[10] E. Casais and A. Taivalsaari, "Object-Oriented Software Evolution and Re-Engineering (special issue)," *Theory and Practice of Object Systems*, vol. 3, no. 4, 1997.
[11] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, second ed. Prentice-Hall, 1988.
[12] "American National Standard for Information Systems—Programming Language—C: ANSI X3. 159–1989," New York: Am. Nat'l Standards Inst., Dec. 1989.
[13] B. Stroustrup, *The C++ Programming Language*, third ed. Addison-Wesley, 1997.
[14] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," *USENIX Technical Conf. Proc.*, June 2002.
[15] D. Spinellis, *Code Reading: The Open Source Perspective.* Effective Software Development Series, Addison-Wesley, 2003.

[16] M.D. Ernst, G.J. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," *IEEE Trans. Software Eng.,* vol. 28, no. 12, pp. 1146-1170, Dec. 2002.

[17] S.C. Johnson, "Lint, a C Program Checker," Computer Science Technical Report 65, Bell Laboratories, Murray Hill, New Jersey, Dec. 1977.

[18] G.J. Badros and D. Notkin, "A Framework for Preprocessor-Aware C Source Code Analyses," *Software: Practice & Experience,* vol. 30, no. 8, pp. 907-924, July 2000.

[19] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[20] D. Spinellis, "Checking C Declarations at Link Time," *The J. C Language Translation,* vol. 4, no. 3, pp. 238-249, Mar. 1993.

[21] S. Graham, P. Kessler, and M.K. McKusick, "An Execution Profiler for Modular Programs," *Software: Practice & Experience,* vol. 13, pp. 671-685, 1983.

[22] B. Tuthill and K.J. Dunlap, "Debugging with dbx," *UNIX Programmer's Supplementary Documents,* vol. 1, Computer Systems Research Group, Dept. of Electrical Eng. and Computer Science, Univ. of California, Berkeley, Apr. 1986.

[23] D.L. Parnas, "On the Design and Development of Program Families," *IEEE Trans. Software Eng.,* vol. 2, no. 1, pp. 1-9, Mar. 1976.

[24] J.D. McGregor, L.M. Northrop, S. Jarrad, and K. Pohl, "Initiating Software Product Lines," *IEEE Software,* vol. 19, no. 4, pp. 24-27, July/Aug. 2002.

[25] B.W. Kernighan and R. Pike, *The UNIX Programming Environment.* Prentice-Hall, 1984.

[26] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley, 2000.

[27] G. Snelting, "Reengineering of Configurations Based on Mathematical Concept Analysis," *ACM Trans. Software Eng. and Methodology (TOSEM),* vol. 5, no. 2, pp. 146-189, 1996.

[28] J.-M. Favre, "Preprocessors from an Abstract Point of View," *Proc. Int'l Conf. Software Maintenance (ICSM '96),* 1996.

[29] A. Zavras, "A New Solution to the Problem of Source Code Presentation," PhD thesis, Dept. of Electrical Eng. and Computer Science, Nat'l Technical Univ. of Athens, Greece, May 1999, (in Greek).

[30] Microsoft Corporation, *Microsoft C# Language Specifications.* Redmond, Wa.: Microsoft Press, 2001.

[31] D. Spinellis, "Notable Design Patterns for Domain Specific Languages," *J. Systems and Software,* vol. 56, no. 1, pp. 91-99, Feb. 2001.

[32] M.H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1998.

[33] H. Abelson, G.J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[34] B.C. Smith, "Procedural Reflection in Programming Languages," PhD thesis, Mass. Inst. of Technology, Jan. 1982.

[35] D. Flanagan, *Java in a Nutshell.* Sebastopol, Calif.: O'Reilly and Associates, 1997.

[36] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[37] S.C. Johnson and M.E. Lesk, "Language Development Tools," *Bell System Technical J.,* vol. 56, no. 6, pp. 2155-2176, July-Aug. 1987.

[38] D. Spinellis and V. Guruprasad, "Lightweight Languages as Software Engineering Tools," *Proc. USENIX Conf. Domain-Specific Languages,* pp. 67-76, Oct. 1997.

[39] W.G. Griswold and D. Notkin, "Automated Assistance for Program Restructuring," *ACM Trans. Software Eng. and Methodology (TOSEM),* vol. 2, no. 3, pp. 228-269, 1993.

[40] S. Lampoudi and D.M. Beazley, "SWILL: A Simple Embedded Web Server Library," *USENIX Technical Conf. Proc.,* June 2002.

[41] B.S. Hansen and J.U. Toft, "The Formal Specification of ANDF, an Application of Action Semantics," *Proc. First Int'l Workshop Action Semantics,* Apr. 1994.

[42] S.I. Feldman, "Make—A Program for Maintaining Computer Programs," *Software: Practice & Experience,* vol. 9, no. 4, pp. 255-265, 1979.

[43] D.C. Atkinson and W.G. Griswold, "The Design of Whole-Program Analysis Tools," *Proc. 18th Int'l Conf. Software Eng. (ICSE '96),* pp. 16-27, 1996.

[44] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Trans. Programming Languages and Systems (TOPLAS),* vol. 12, no. 1, pp. 26-60, 1990.

Diomidis Spinellis received the MEng degree in software engineering and a PhD degree in computer science, both from Imperial College (University of London, UK). Currently, he is an assistant professor in the Department of Management Science and Technology at the Athens University of Economics and Business, Greece. He has written more than 70 technical papers in the areas of software engineering, information security, and ubiquitous computing. His book, *Code Reading: The Open Source Perspective*, inaugurated Addison Wesley's *Effective Programming Series*. He has contributed software to the BSD Unix distribution, the X Window System, and is the author of a number of open-source software packages, libraries, and tools. Dr. Spinellis is a member of the ACM, the IEEE, the Greek Computer Society, and the Technical Chamber of Greece.