CScout: A Refactoring Browser for C

Diomidis Spinellis
Athens University of Economics and Business
Department of Management Science and Technology
Patision 76, GR-104 34 Athens, Greece

Abstract

Despite the maturity and popularity of the C programming language, tool support for performing even simple refactoring, browsing, or analysis operations is currently lacking due to the identifier scope complications introduced by the C preprocessor. The *CScout* refactoring browser analyses complete program families, by tagging the original identifiers with their precise location and classifying them into equivalence classes orthogonal to the C language's namespace and scope extents. A web-based user interface provides programmers with an intuitive source code analysis and navigation front-end, while an SQL-based back-end enables more complex source code analysis and manipulation. *CScout* has been successfully applied to a number of medium and large proprietary and open source projects identifying thousands of modest refactoring opportunities.

Key words: C, browser, refactoring, preprocessor

1. Introduction

C is still the language of choice for developing systems applications, such as operating systems and databases, embedded software, and the majority of open-source projects [1, p. 16]. Despite the language's popularity, tool support for performing even simple refactoring, browsing, or analysis operations is currently lacking. Programmers typically resort to using either simplistic text-based operations that fail to capture the language's semantics, or work on the results of the compilation and linking phase that—due to the effects of preprocessing—do not correctly reflect the original code. Interestingly, many of the tools in a C programmer's arsenal were designed in the 1970s, and fail to take advantage of the CPU speed and memory capacity of a modern

Email address: dds@aueb.gr (Diomidis Spinellis).
URL: http://www.dmst.aueb.gr/dds (Diomidis Spinellis).

workstation. In this paper we describe how the *CScout* refactoring browser, running on a powerful workstation, can be used to accurately analyze, browse, and refactor large program families written in C.

CScout can process program families consisting of multiple related projects (we define a project as a collection of C source files that are linked together) mapping the complexity introduced by the C preprocessor back into the original C source code files. *CScout* takes advantage of modern hardware advances (fast processors, large address spaces, and big memory capacities) to analyze C source code beyond the level of detail and accuracy provided by current IDEs, compilers, and linkers. Specifically, the analysis *CScout* performs takes into account both the identifier scopes introduced by the C preprocessor and the C language proper scopes and namespaces.

The objective of this paper is to provide a tour of *CScout* by describing the domain's challenges, the operation of *CScout* and its interfaces, the system's design and implementation, and details of *CScout*'s application to a number of large software projects. The main contributions of this paper are the illustration of the types of problems occurring in the analysis of real-life C source code and the types of refactorings that can be achieved, the demonstration through the application of *CScout* to a number of systems that accurate large-scale analysis of C code is in fact possible, and a discussion of lessons associated with the construction of browsers and refactoring tools for languages, like C and C++, that involve a preprocessing step.

2. Problem Statement

Many features of the C language hinder the precise analysis of programs written in it and complicate the design of corresponding reasoning algorithms [2]. The most important culprit features are unrestricted pointers, aliasing, arbitrary type casts, non-local jumps, an underspecified build environment, and the C preprocessor. All features but the last two ones limit our ability to reason about the runtime behavior of programs (see e.g. the article [3] and the references therein). Significantly, the C preprocessor and a compilation environment based on external tools also restrict programmers from performing even supposedly trivial operations such as determining the scope of a variable, the type of an identifier, or the extent of a module.

2.1. *Preprocessor Complications*

In summary, preprocessor macros complicate the notion of scope and the notion of an identifier [4–6]. For one, 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 the following example, the name of the identifier `len` might need changing in all three definitions, although in C the members of each data structure belong to a different namespace.

```
struct disk_block { int len; /* ... */ };
struct mem_block { int len; /* ... */ };
#define get_block_len (b) ((b)->len)
```

In addition, new identifiers can be formed at compile time via the preprocessor's concatenation operator. As an example, the following code snippet defines a variable named `sysctl_var_sdelay`, even though this name does not appear in the source file.

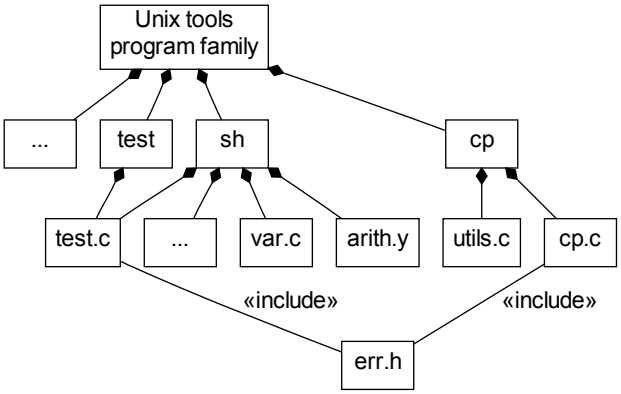


Fig. 1. Program family relationships in Unix tools.

```
#define SYSCTL(x) static int sysctl_var_ ## x
SYSCTL(sdelay);
```

An additional complication comes from the use of conditionally compiled code (see also Section 2.2). Such code may or may not be compilable under a given compilation environment, and, often, blocks of such code may be mutually incompatible.

2.2. *Build Environment Complications*

Parnas [7] defines a program family as a set of programs that should be studied by first considering the common properties of the set and then determining individual properties of family members (see also the work by Weiss & Lai [8]). For the purposes of analyzing C source code for browsing and refactoring purposes we are interested in program families consisting of programs that through their build process reuse common elements of source code. This is a property of what has been termed the build-time software architecture view [9]. We have identified three interesting instances of source code sharing in such families:

Different program configurations Often the same source code base is used to derive a number of program configurations. As an example, the FreeBSD kernel source code is used as a basis for creating kernels for five processor architectures. Major parts of the source code are the same among the different architectures, while the compilation is influenced by architecture-dependent macros specifying properties such as the architecture's native element size (32 or 64 bits) and the "endianess" of the memory layout (the order in which an integer's bytes are stored in memory).

Different programs In many cases elements of a source code base are reused to create various executable programs. Consider the case illustrated in Figure 1. Although code reuse is typically realized in the Unix environment by creating a common library (such as the libraries *math*, *dbm*, *termcap*, and *telnet*), which is linked with each program requiring the given functionality, there are cases where a simpler and less structured approach is adopted. The example in Figure 1 illustrates some dependencies between three (supposedly separate) Unix programs where

CScout was applied: *test*, *sh*, and *cp*. Among them the condition evaluation utility *test* and the shell *sh* share the source file `test.c`, while two source files both include the header `err.h`.

Program versions When there is a supported maintenance branch among different releases of the same program, then the same source code (with typically small differences between release-dependent versions) is reused among the different releases.

In all the cases we described above the sharing and the differentiation of the source code does not typically happen through mechanisms of the C language, but through extra-linguistic facilities. The most important of these are compiler invocation options that set macros and include file paths, symbolic links across files and directories, environment variables affecting the build process, macros hard-coded in the compiler, and the automated copying of files as part of the build process. Despite these complications, a viable tool should propagate browsing and refactoring operations across all files in a given program family.

2.3. *Problem Impact*

Due to the above problems, programmers are currently working with methods and tools that are neither sound nor complete. The typical textual search for an identifier in a source code base may fail to locate identifier instances that are dynamically constructed, or will also locate identifiers that reside in a different scope or namespace. A more sophisticated search using a compiler or IDE-constructed symbol table database will fail to match all macro instances, while its results will be difficult to match against the original source code. Consequently, program maintenance and evolution attributes are negatively affected as programmers, unsupported by the tools they use, are reluctant to perform even a simple rename-function refactoring. Anecdotal evidence supports our observation: consider identifier names such as the Unix `creat` system call that still persist, decades after the reasons for their original names have become irrelevant [10, p. 60]. The readability of existing code slowly decays as layers of deprecated historical practice accumulate [11, pp. 4–6, 184] and even more macro definitions are used to provide compatibility bridges between legacy and modern code.

3. Related Work

Tools that aid program code analysis and transformation operations are often termed *browsers* [12, pp. 297–307] and *refactoring browsers* [13] respectively. Related work on object-oriented design refactoring [14] asserts that it is generally not possible to handle all problems introduced by preprocessing in large software applications. However, as we shall see in the following sections, advances in hardware capabilities are now making it possible to implement useful refactoring tools that address the complications of the C programming language. The main advantage of our approach is the correct handling of preprocessor constructs, so, although we have only tested the approach on different variants of C programs, (K&R C, ANSI C, and C99 [15–17]) it is, in principle, also applicable to programs written in C++ [18], Cyclone [19], PL/I and many assembly-code dialects.

Reference [20] provides 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. Two theoretical approaches proposed for dealing with the problems of the C preprocessor involve the use of mathematical concept analysis for handling cases where the preprocessor is used for configuration management [21], and the definition of an abstract language for capturing

File Metrics
- Number of: statements, copies of the file, defined project-scope functions, defined file-scope (static) functions, defined project-scope variables, defined file-scope (static) variables, complete aggregate (struct/union) declarations, declared aggregate (struct/union) members, complete enumeration declarations, declared enumeration elements, directly included files

File and Function Metrics
- Number of: characters, comment characters, space characters, line comments, block comments, lines, character strings, unprocessed lines, preprocessed tokens, compiled tokens, C preprocessor directives, processed C preprocessor conditionals (ifdef, if, elif), defined C preprocessor function-like macros, defined C preprocessor object-like macros
- Maximum number of characters in a line

Function Metrics
- Number of: statements or declarations, operators, unique operators, numeric constants, character literals, if / switch / break / for / while / do / continue / goto / return statements, goto / case / default labels, else clauses, project-scope / file-scope (static) / macro / object and object-like / unique project-scope / unique file-scope (static) / unique macro / unique object and object-like / label identifiers, global namespace occupants at function's top, parameters
- Maximum level of statement nesting
- Fan-in and fan-out
- Cyclomatic, extended cyclomatic, and maximum (including switch statements) cyclomatic complexity

Table 1
File and function metrics that *CScout* collects.

the abstractions for the C preprocessor in a way that allows formal analysis [4]. The two-way mapping between preprocessor tokens and C-proper identifiers used by *CScout* was first suggested by Livadas and Small [22]. A tool adopting an approach similar to ours is *Xrefactory* [23]. However, *Xrefactory* is unable to handle identifiers generated during the preprocessing stage; its author claims that the problem is in general unsolvable. Other related work has proposed the integration of multiple approaches views, and perspectives into a single environment [24], the full integration of preprocessor directives in the internal representation [25,26], the use of an abstract syntax graph for communicating semantic information [27], and the use of a GXL [28] schema for representing either a static or a dynamic view of preprocessor directives [29].

The handling of multiple configurations implemented through preprocessor directives that *CScout* implements, has also been studied in other contexts, such as the removal of preprocessor conditionals through partial evaluation [30], the type checking of conditionally compiled code [31], and the use of symbolic execution to determine the conditions associated with particular lines of code [32].

4. The CScout Refactoring Browser

To be able to accurately and efficiently map and rename identifiers across program families *CScout* integrates in a single processing engine functions of a compiler driver (such as *make* or *ant*), a C preprocessor, a C compiler front-end, a parser of *yacc* files, a linker, a relational database export facility, and a web-based GUI.

CScout as a source code analysis tool can:
- annotate source code with hyperlinks to a detail page for each identifier
- list files that would be affected by changing a specific identifier
- determine whether a given identifier belongs to the application or to an external library, based on the accessibility and location of the header files that declare or define it
- locate unused identifiers taking into account inter-project dependencies

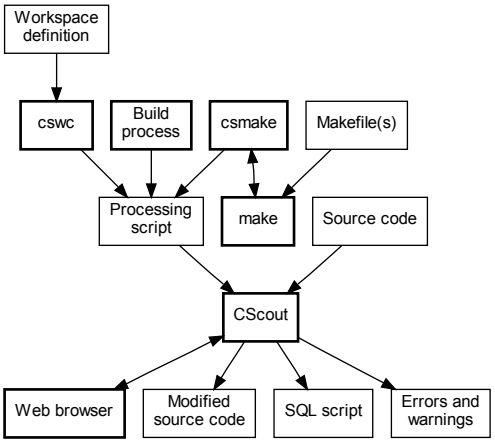


Fig. 2. *CScout* system operation.

– perform queries for identifiers based on their namespace, scope, reachability, and regular expressions of their name and the filename(s) they are found in,
– perform queries for files, based on their metrics or properties of the identifiers they contain
– perform queries for functions, based on their metrics, their callers, or the functions they call
– monitor and report superfluously included header files
– provide accurate metrics on identifiers, functions, and files (see Table 1)

More importantly, *CScout* helps programmers in refactoring code by identifying dead objects to remove, and can automatically perform accurate global *rename identifier* refactorings [33]. One might question whether support for a single refactoring type merits calling *CScout* a refactoring tool. However we should take into account that the *rename identifier* operation is by far the most common refactoring operation performed in practice [34], and that performing this operation reliably on production C source code is very tricky. Specifically, *CScout* will automatically rename identifiers

– taking into account the namespace of each identifier: a renaming of a structure tag, member, or a statement label will not affect, for example, variables with the same name
– respecting the scope of the renamed identifier: a rename can affect multiple files, or variables within a single block, exactly matching the semantics the C compiler would enforce
– across multiple projects (linkage units) when the same identifier is defined in common shared include files or even code
– across conditionally compiled units, if a corresponding workspace (a set of interrelated linkage units) has been defined and processed,
– occurring in macro bodies and even *parts* of other identifiers, when these are created through the C preprocessor's token concatenation feature

Figure 2 illustrates the model of *CScout*'s operation. The operation is directed by a processing script, which contains a sequence of imperative processing commands. These commands setup an environment for processing each source code file. The environment is defined by the current directory, the include file directory path, externally defined macros, and the linkage unit to

be associated with global identifiers. The script is a C file comprised mostly of *#define* directives and *CScout*-specific *#pragma* directives, like *project*, *block_enter*, *block_exit*, *includepath*, *clear_defines*, *clear_include*, and *process*. In cases where the source code can contain multiple configurations under conditional compilation the script will contain directives to process the source code multiple times, once for each configuration.

Creating the processing script is not trivial; for a large project, like the Linux kernel, the script can be more than half a million lines long. The script can be created in three ways.

(i) A declarative specification of the source components, compiler options, and file locations required to build the members of a program family is processed by the *CScout* workspace compiler *cswc*.

(ii) A separate program, *csmake*, can monitor compiler, archiver, and linker invocations in a *make*-driven build process, and thereby gather data to automatically create the processing script.

(iii) The build process can be instrumented to record the commands executed. This transcript can then be semi-automatically converted into the *CScout* processing script. For instance, a 74-line Perl script was used to convert the 1149 line output of Microsoft's *nmake* program compiling the Windows Research Kernel into a 51,288 line *Cscout* processing script.

To handle conditionally compiled code, a single script can contain instructions for multiple passes over the source code, with different options enabled in each pass.

As a by-product of the processing *CScout* generates a list of error and warning messages in a standard format that typical editors (like *vi* and *Emacs*) and IDEs can process. These warnings go beyond what a typical compiler will detect and report

– unnecessarily included header files,
– identifiers for functions, variables, macros, labels, tags, and members that are never used across the complete workspace, and
– elements that should have been declared with file-local (*static*) visibility.

Many interesting maintenance activities can be performed by processing this standardized error report. In one case we automatically processed those warnings to remove 765 superfluous *#include* directives (out of a total of 5429) from a 190KLOC CAD program [35].

After processing all source files, *CScout* operates as a web server, allowing members of a team to browse and modify the files comprising the program family. All changes performed through the web interface (currently rename operations on identifiers) are mirrored in an in-memory copy of the source code. These changes can then be committed back to the source code files, optionally under the control of a version control system. A separate backend enables *CScout* to export its data structures to a relational database management system for further processing.

4.1. *Web-Based Interface*

The most productive use of *CScout* stems from its interactive web-based interface (see Figure 3).

Using the SWILL embedded web server library [36] *CScout* presents the analyzed source code collection in a way that enables browsing by connecting a web client to the tool's HTTP-server interface. A set of hyperlinks enables users to perform the following tasks.

– Browse file and identifier names belonging to specific semantic categories (e.g. read-only files, file-spanning identifiers, or unused identifiers).
– Examine the source code of individual files, with hyperlinks providing navigation from each

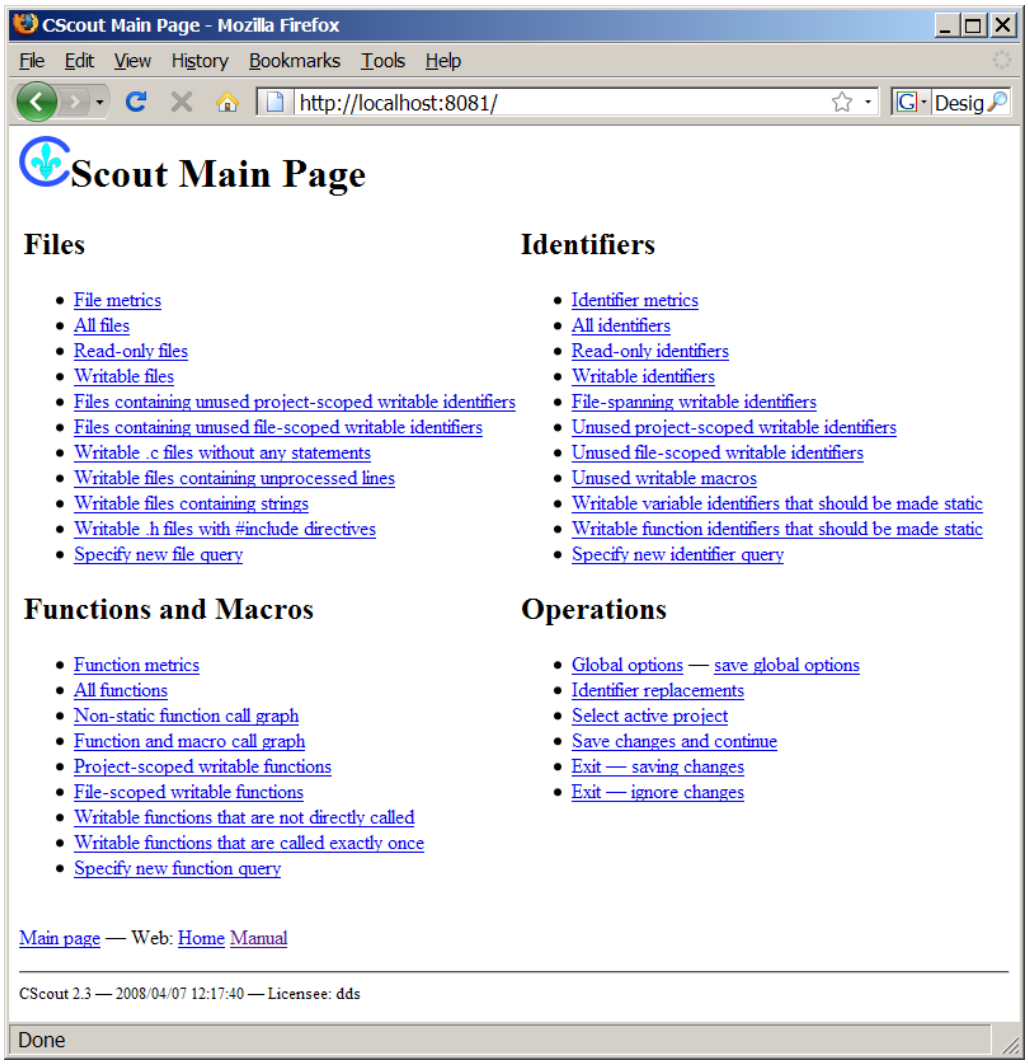
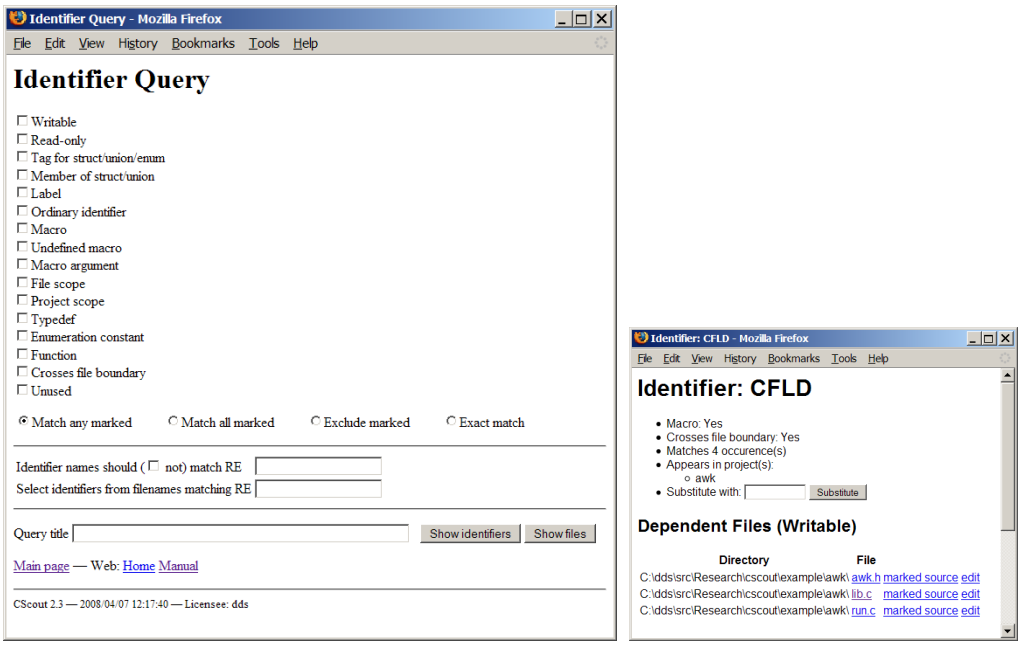


Fig. 3. A screen dump of the *CScout* web interface.

identifier to a separate page providing details of its use.

– 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 type definition, or is a (possibly undefined) macro, or macro argument. The file and identifier names to include or exclude can also be specified in the query as extended regular expressions.
– Specify simple form-based file and function queries based on the calculated metrics listed in Table 1.
– View the semantic information associated with a class of identifiers. Users can find out whether the identifier is read-only (i.e. at least one of its instances resides in a read-only file), and whether its instances are used as macros, macro arguments, structure, union, or enumeration



(a) The identifier query form.

(b) An identifier page.

Fig. 4. *CScout* in operation.

tags, structure or union members, labels, type definitions, 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 allow the programmer to find 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.

– View information associated with a function or a function-like macro: the identifiers comprising its name, its declaration and definition, the callers and the called functions, and their transitive closure. Uniquely, *CScout* can calculate metrics and call graphs that take into account both functions and function-like macros (see Figure 5—derived while browsing the source code of *awk* and drawn using *dot* [37]). This matches the reality of C programming, where the two are used interchangeably.
– Substitute all matching instances of a given identifier with a new user-specified name. This process can be repeated multiple times, allowing the incremental improvement of the code, without the expensive reprocessing step.
– Write back the changed identifiers into the respective source code files.

The above functionality can be used to semi automatically perform two important refactoring operations: *rename*, e.g. Griswold and Notkin's [33] "rename-variable", and *remove*, e.g. Fowler's [38] "Remove Parameter". Name clashes occurring in a rename refactoring are not detected, because this feature would require reprocessing the entire program family source code base—a time consuming process. This restriction however is rarely a problem in practice, as programmers can easily spot a name clash in a given scope, and the compiler will typically flag any remaining clashes as errors. Remove refactorings can be trivially performed by hand, after identifiers that

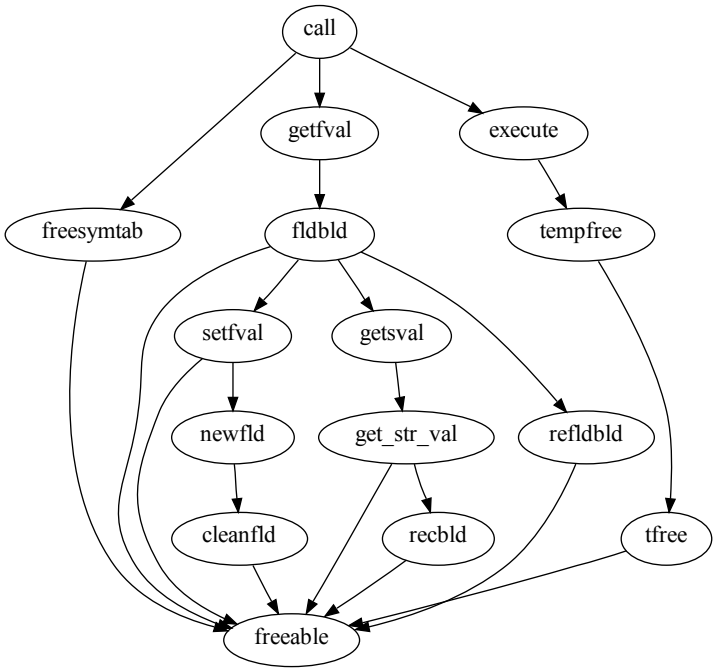


Fig. 5. A call graph spanning functions and macros (`tempfree` is a function-like macro).

occur exactly once have been automatically and accurately identified. Again, fully automating this process is hard (there are many rare special cases that have to be handled), but performing it by hand is in most cases very easy.

The web server follows the representational state transfer (REST) architecture [39], and therefore its URLs can be used for interoperating with other tools. Furthermore, as all web pages that *CScout* generates are identified by a unique URL, programmers can easily mark important pages (such as a particular identifier that must be refactored, or the result of a specialized form-based query) using their web browser's bookmarking mechanism, or even email an interesting page's URL to a coworker. In fact, most links appearing on *CScout*'s main web page are simply canned hyperlinks to the general queries we outlined above.

4.2. *SQL Backend*

CScout can also save the data structures representing its source code analysis into a relational database. We chose to use a relational database over a specialized and more expressive logic query language, such as SOUL [40] or JQuery [41,42], in order to exploit the performance and maturity of existing RDBMS systems for the offline storage of very large data sets—one particular study we performed [43] involved storing and processing more than 160 million records.

Figure 6 shows the most important parts of the corresponding schema. (Four tables associated with reasoning about include file dependencies are not shown.) Through the database one can

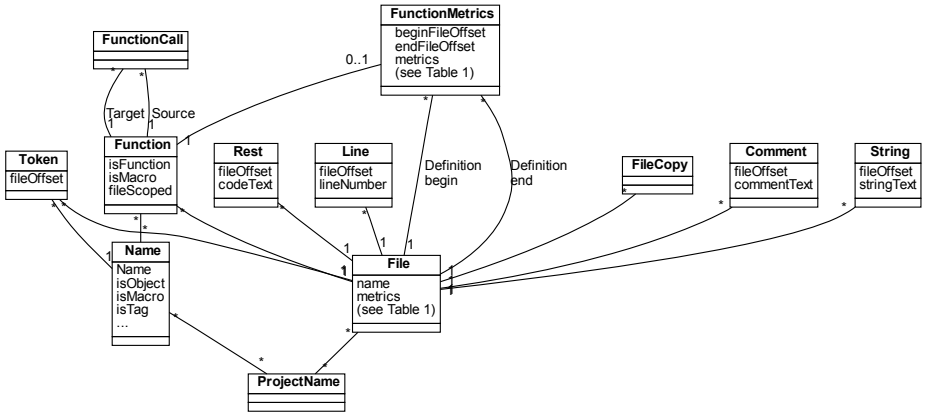


Fig. 6. The logical schema of the exported database.

issue all the queries available through the GUI, and many more.

For instance, the following simple SQL query will find all type definitions that don't end in "_t" (a common naming convention).

```
select distinct name from
ids left join tokens on ids.eid = tokens.eid
where ids.typedef and not name like '%_t'
order by name
```

The following, more complex, query

```
select name, count(*) as nfile from (
 select fid, tokens.eid, count(*) as c from
 tokens
 group by
 eid, fid) as cl inner join ids on
cl.eid = ids.eid
group by ids.eid, ids.name
order by nfile desc;
```

will show all identifiers, ordered by the number of different files in which they occur (an indication of coupling):

```
+---------+-------+
| name    | nfile |
+---------+-------+
| NULL    |  3292 |
| u       |  2560 |
| printk  |  1922 |
| ...     |   ... |
```

The program's SQL representation contains all elements of the corresponding source code. Therefore, one can also perform large-scale refactorings through SQL commands. Then, the

source code of each file (e.g. file number 42 in the following SQL query) can be fully reconstituted from its (refactored) parts.

```
select s from
( select name as s, foffset
 from ids inner join tokens on
 ids.eid = tokens.eid
  where fid = 42
 union select code as s, foffset from rest
  where fid = 42
 union select comment as s, foffset from comments
  where fid = 42
 union select string as s, foffset from strings
  where fid = 42
)
order by foffset
```

5. Design and Implementation

Bringing *CScout* into life required careful analysis of the principles of its operation and substantial implementation work.

5.1. *Operation*

The theory behind *CScout*'s operation is described in detail elsewhere [6]; this paper focuses on the tool's design, implementation, and application. The basic principle of operation is to tag each identifier appearing in the original source code with its precise location (file and offset) and to follow that identifier (or its part when new identifiers are composed by concatenating original ones) across preprocessing, parsing, (partial) semantic analysis, and (notional) linking. Each identifier also belongs to exactly one *equivalence class*: a set of identifiers that would have to be renamed in concert for the program family to remain semantically and syntactically correct. As each identifier token is read, a new equivalence class for that token is created. The notion of an equivalence class is orthogonal to the language's existing namespace and scope extents, taking into account the changes to those extents introduced by the C preprocessor. Every time a symbol table lookup operation for an identifier matches an existing identifier (e.g the use of a declared variable or the use of an argument of a function-like macro) the two corresponding equivalence classes are *unified* into a single one.

In total, 20 different equivalence class unifications are performed by *CScout*. These can be broadly classified into the following categories: macro formal parameters and their use inside the macro body, macros used within the source code, macros being redefined or becoming undefined, tests for macros being defined, identifiers used in expressions, structure or union member access (direct, through a pointer indirection, or through an initializer designator), declarations using `typedef` types, application of the `typeof` operator (a *gcc* extension) to an identifier, use of structure, union, and enumeration tags, old-style [15] function parameter declarations with the respective formal parameter name, multiple declarations and definitions of objects with compilation or linkage unit scope, and `goto` labels as labels and targets.

By classifying all identifiers into equivalence classes, and then creating and merging the classes following the language's rules, we end up with a data structure that can identify many interesting relationships between identifiers.

- A rename operation simply involves changing the name of all identifiers belonging to the same equivalence class.
- Unused identifiers are those belonging to an equivalence class with exactly one member.
- If at least one identifier in an equivalence class is located in a read-only file—for instance a system library header file—then all the identifiers of that class are considered immutable.

Although complete semantic analysis and type checking is not performed, a symbol table containing basic type information for identifiers is maintained. Furthermore, support for the C99 initializer designators also requires the evaluation of compile-time constants. (The array position of an initializer can be specified by a compile-time constant. When elements of nested aggregates—structures, unions, and arrays—are specified in comma-separated form without enclosing them in braces, the array position constant must be evaluated in order to determine the type of the next element.) The type checking subsystem is mainly used to identify the underlying structure or union for member access and initialization, and to handle type definitions. In addition, its implementation provided us with a measure of confidence regarding the equivalence class unification operations dictated by the language's semantics.

The symbol table design follows the language's block scoping rules, with special cases handling prototype declarations, and compilation and linkage unit visibility. Thankfully, between the processing of two different projects (linkage units) the complete symbol table is cleared and only equivalence classes remain in memory, thus reducing *CScout*'s memory footprint. This optimization can be performed, because if we ignore extra-linguistic facilities (such as shared libraries, debug symbols, and reflection) linked programs operate as standalone processes and do not depend on any program identifiers for their operation.

5.2. *Implementation Details*

CScout has been actively developed for five years, and currently consists of 25 KLOC. Most of the code is written in C++ with Perl being used to implement the *CScout* processing script generators. Two more Perl scripts automatically extract from the source code the documentation for the SQL database schema and the reported error messages. The lexical analyzer and the C preprocessor are hand-crafted, while the parsing of preprocessor expressions and the C code are handled by two separate *btyacc* grammars. Handling the various language extension dialects hasn't proven to be difficult; probably because *CScout* is quite permissive in what is accepts. Therefore, currently *CScout*'s input is the union of all possible language extensions. If in the future some extensions are found to be mutually exclusive, this can be handled by adding *pragma* directives that will change the handling of the corresponding keywords.

Although no fancy algorithms and data structures were used to achieve the *CScout*'s scalability extreme care was taken to adopt everywhere data structures and corresponding algorithms that would gracefully scale. This was made possible by the C++ STL library. For each data structure we simply chose a container that would handle all operations on its elements efficiently in terms of space and time. Thus, all data lookup operations are either $O(1)$ for accessing data through a pointer indirection or at a vector's known index, or $O(\log N)$ for operations on sets and maps. Up to now algorithmic tuning was required only once, to fix a pathological case in the implementation of the C preprocessor macro expansion [44].

The aggressive use of STL complicated *CScout*'s debugging. Navigating STL data structures with *gdb* is almost impossible, because *gdb* provides a view of the data structures' implementation details, but not their high-level operations. This problem was solved by implementing a custom logging framework [45]: a lightweight and efficient construct that allowed us to instrument the code with (currently 175) log statements. As the following example shows, writing such a *debugpoint* statement is trivial:

```
if (DP()) cout << "Gather args returns: " << v << endl;
```

Each debugpoint can be easily enabled by specifying in a text file at runtime its corresponding file name and line number. The overhead of debugpoints can be completely disabled at compile time, but even when they get compiled, if none of them is enabled, their cost is only that of a compare and a jump instruction.

The testing of *CScout* consists of stress and regression testing. Stress testing involves applying *CScout* to various large open-source systems. Problems in the preprocessor, the parser, or the semantic analysis quickly exhibit themselves as various errors or crashes. In addition, by having *CScout* replace all identifiers of a system with mechanically-derived names and then recompiling and testing the corresponding code builds confidence in *CScout*'s equivalence algorithms and the rename-identifier refactoring. Regression testing is currently used to check corner cases and accidental errors. The *CScout*'s preprocessor is tested through 68 test cases whose output is then compared with the hand-verified output. The parser and analyzer are further tested through 42 small and large test cases whose complete analysis is stored in an RDBMS and compared with previous (correct) results.

CScout benefits from the use of existing mature open source components and tools: the *btyacc* backtracking variant of the *yacc* parser generator, the SWILL embedded web server library [36], the *dot* graph drawing program [37]), and the *mySQL* and *PostgreSQL* relational database systems. The main advantages of these components were their stability, efficiency, and hassle-free availability. In addition, the source code availability of *btyacc* and SWILL allowed us to port them to various platforms and to add some minor but essential features: a function to retrieve an HTTP's request URL in SWILL, and the ability for multiple grammars to co-exist in a program in *btyacc*.

6. Applying CScout

CScout has been applied on many commercial and open source program families running on a variety of hardware and software platforms [43,46,35]. The workspace size ranges from 6 KLOC (*awk*) to 4.1 MLOC (the Linux kernel). In all cases *CScout* was applied on the unmodified source code of each project. (*CScout* supports the original K&R C [15], ANSI C [16], and many C99 [17], *gcc*, and Microsoft C extensions.) Details of some representative projects can be seen in Table 2. The projects listed are the following.

awk The *one true awk* scripting language. [1]

Apache httpd The Apache project web server, version 1.3.27.

FreeBSD The source code of the FreeBSD kernel HEAD branch, as of 2006-09-18, in three architecture configurations: i386, AMD64, and SPARC64.

Linux The Linux kernel, version 2.6.18.8-0.5, in its AMD64 configuration.

[1] `http://cm.bell-labs.com/who/bwk/index.html`

	awk	Apache httpd	FreeBSD kernel	Linux kernel	Solaris kernel	WRK
Overview						
Configurations	1	1	3	1	3	2
Modules (compilation units)	1	3	1,224	1,563	561	3
Files	14	96	4,479	8,372	3,851	653
Lines (thousands)	6.6	59.9	2,599	4,150	3,000	829
Identifiers (thousands)	10.5	52.6	1,110	1,411	571	127
Defined functions	170	937	38,371	86,245	39,966	4,820
Defined macros	185	1,129	727,410	703,940	136,953	31,908
Preprocessor directives	376	6,641	415,710	262,004	173,570	35,246
C statements (thousands)	4.3	17.7	948	1,772	1,042	192
Refactoring opportunities						
Unused file-scoped identifiers	20	15	8,853	18,175	4,349	3,893
Unused project-scoped identifiers	8	8	1,403	1,767	4,459	2,628
Unused macros	4	412	649,825	602,723	75,433	25,948
Variables that could be made static	47	4	1,185	470	3,460	1,188
Functions that could be made static	10	4	1,971	1,996	5,152	3,294
Performance						
CPU time	0.81”	35”	03:43:40”	07:26:35”	01:18:54”	00:58:53”
Required memory (MB)	21	71	3,707	4,807	1,827	582

Table 2. Details of representative processed applications.

Solaris Sun's OpenSolaris kernel, as of 2007-07-28, in three architecture configurations: Sun4v, Sun4u, and SPARC.

WRK The Microsoft Windows Research Kernel, version 1.2, into two architecture configurations: i386 and AMD64.

In the cases of awk, Apache, and WRK, the program family included one main project and a number of small peripheral ones (such as add-on modules, or post-processing tools) sharing a few source or header files. The three Unix-like kernels (FreeBSD, Linux, and OpenSolaris) were different: all consist of a main kernel and hundreds more modules providing functionality for various devices, filesystems, networking protocols, and additional features. With *CScout* these were processed as a single workspace, allowing browsing and refactoring to span elements residing in different linkage units.

Another interesting task was the processing under different configurations for FreeBSD, OpenSolaris, and WRK [43]. A kernel configuration specifies the CPU architecture, the device drivers, filesystems, and other elements that will be included in a kernel build. Through conditional compilation directives, the processed source code of one configuration can differ markedly from another. By processing multiple configurations as a single workspace *CScout* can present the source code as the union of the corresponding source code elements, and therefore ensure that the refactorings won't break any of the configurations processed.

The processing time required appears to be acceptable for integrating *CScout* in an IDE for small (e.g. up to 10 KLOC) projects. Memory requirements also appear to be tolerable for up to medium sized workspaces (e.g. up to 100 KLOC) for a typical developer workstation. Large workspaces will require a high-end modern workstation or a server equipped with multi-gigabyte memory and a 64-bit CPU.

Up to now the most useful application of *CScout* has been the cleanup of source code, performed by removing unused objects, macros, and included header files and by reducing the visibility scope of defined objects. This is an easy target, since all it entails is letting an editor automatically jump to each affected file location by going through *CScout*'s standardized warning report.

To test *CScout*'s identifier analysis we instrumented the refactoring engine to rename all the writable identifiers in the *apache* source code into new, mechanically derived, random strings. The resulting obfuscated version of the source code compiled and appeared to work without any problems. Such an approach could be applied on proprietary code to derive an architecture-neutral distribution format.

7. Lessons Learned

The operation of program analysis and transformation tools can be characterized as *sound* when the analysis will not generate any false positive results, and as *complete* when there are not missing elements in the results of the analysis. The analysis performed by *CScout* over identifier equivalence classes is in the general case sound, because it follows precisely the language's semantic rules. The incompleteness of the produced results stems from three different complications; addressing those with heuristics would result in an analysis that would no longer be sound.

Predictably, the main complications in our scheme arise from preprocessing features: 1) unifying undefined macros, 2) dealing with macros that do not cover the complete semantic spectrum of their possible application, and 3) handling conditional compilation. In practice, the last is-

sue has caused the greatest number of problems. 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. When processing the FreeBSD kernel we used both approaches: a special predefined kernel configuration target named LINT to maximize the amount of conditionally compiled code that the configuration and processing would cover, and a separate pass for each of five different processor architectures. Yet, even this approach did not adequately cover the complete source code, as evidenced by an aborted attempt to remove header files that appeared to be unused.

Another problem we encountered when applying *CScout* in realistic situations concerned language extensions. The first version of *CScout* supported the 1989 version of ANSI C [16] and a number of C99 [17] extensions. In practice we found that *CScout* could not be applied on real-world source code without supporting many compiler-specific language extensions. Even if specific programs were written in a portable manner, the platform-specific header files they included used many of those extensions, and could not be processed by a tool that did not support them. This was a significant problem for a number of reasons:

- Compiler-specific language extensions are typically far less carefully documented than the standardized language. In a number of cases we had to understand an extension's syntax and semantics by looking for examples of its use, or reading the corresponding compiler's source code.
- Significant effort that could have been spent on improving the usefulness of *CScout* on all platforms was often diverted toward the support of a single proprietary compiler-specific extension.
- Some language extensions were mutually incompatible.
- Unintended extensions arising from a compiler's sometimes haphazard checking of a program's syntactic correctness restrict the portability of supposedly portable programs that use the extension out of oversight.

Finally, we have yet to find a practical way to handle meta programming approaches where a project-internal domain specific language (DSL) is used to produce C code. In such cases, changes to the C source code may need to be propagated to the DSL code, or even to the DSL compiler. Integrating the support into *CScout*, as we have done for *yacc*, solves the problem for one specific case, but this approach cannot scale in a realistic manner. Currently identifiers residing in an automatically generated C file can be easily tagged as "read-only", but this will restrict the number of identifiers that can be renamed. In the future, accurate file and offset tagging of the automatically created source code, in a way similar to the *#line* directives currently emitted by generators such as *lex* and *yacc* may offer a viable solution.

8. Conclusions

The application of CPU and memory resources toward the analysis of large program families written in C is an effective approach that yields readily exploitable refactoring opportunities in legacy code. *CScout* has already been successfully applied on a wide range of projects for performing modest, though not insignificant, refactoring operations. Our approach can be readily extended to cover other preprocessed languages like C++. Open issues from a research perspec-

tive are the automatic identification and implementation of more complex refactoring operations, increasing the accuracy of flow graphs by reasoning about function pointers [47], the production of source code views for given macro values, and the efficient maximization of code coverage.

Acknowledgements and Tool Availability

This work was partially funded by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software (SQO-OSS)".

The tool and its documentation are available at `http://www.dmst.aueb.gr/dds/cscout/`. CScout currently runs under the FreeBSD, Linux, and Microsoft Windows operating systems under several 32 and 64-bit architectures.

References

[1] D. Spinellis, Code Reading: The Open Source Perspective, Addison-Wesley, Boston, MA, 2003.

[2] A. Garrido, R. Johnson, Challenges of refactoring C programs, in: IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution, ACM, New York, NY, USA, 2002, pp. 6–14.

[3] R. Ghiya, D. Lavery, D. Sehr, On the importance of points-to analysis and other memory disambiguation methods for C programs, ACM SIGPLAN Notices 36 (5) (2001) 47–158, proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI).

[4] J.-M. Favre, Preprocessors from an abstract point of view, in: Proceedings of the International Conference on Software Maintenance ICSM '96, IEEE Computer Society, 1996.

[5] G. J. Badros, D. Notkin, A framework for preprocessor-aware C source code analyses, Software: Practice & Experience 30 (8) (2000) 907–924.

[6] D. Spinellis, Global analysis and transformations in preprocessed languages, IEEE Transactions on Software Engineering 29 (11) (2003) 1019–1030.

[7] D. L. Parnas, On the design and development of program families, IEEE Transactions on Software Engineering SE-2 (1) (1976) 1–9.

[8] D. M. Weiss, C. T. R. Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, 1999.

[9] Q. Tu, M. Godfrey, The build-time software architecture view, in: ICSM'01: Proceedings of the IEEE International Conference on Software Maintenance, 2001, pp. 398–407.

[10] D. Cooke, J. Urban, S. Hamilton, Unix and beyond: An interview with Ken Thompson, IEEE Computer 32 (5) (1999) 58–64.

[11] A. Hunt, D. Thomas, The Pragmatic Programmer: From Journeyman to Master, Addison-Wesley, Boston, MA, 2000.

[12] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison-Wesley, Reading, MA, 1989.

[13] D. Roberts, J. Brant, R. E. Johnson, A refactoring tool for Smalltalk, Theory and Practice of Object Systems 3 (4) (1997) 39–42.

[14] L. Tokuda, D. Batory, Evolving object-oriented designs with refactorings, Automated Software Engineering 8 (2001) 89–120.

[15] B. W. Kernighan, D. M. Ritchie, The C Programming Language, 1st Edition, Prentice Hall, Englewood Cliffs, NJ, 1978.

[16] American National Standard for Information Systems — programming language — C: ANSI X3.159–1989, (Also ISO/IEC 9899:1990) (Dec. 1989).

[17] International Organization for Standardization, Programming Languages — C, ISO, Geneva, Switzerland, 1999, iSO/IEC 9899:1999.

[18] B. Stroustrup, The C++ Programming Language, 3rd Edition, Addison-Wesley, Reading, MA, 1997.

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang, Cyclone: A safe dialect of C, in: USENIX Technical Conference Proceedings, USENIX Association, Berkeley, CA, 2002.

[20] M. D. Ernst, G. J. Badros, D. Notkin, An empirical analysis of C preprocessor use, IEEE Transactions on Software Engineering 28 (12) (2002) 1146–1170.
[21] G. Snelting, Reengineering of configurations based on mathematical concept analysis, ACM Transactions on Software Engineering and Methodology (TOSEM) 5 (2) (1996) 146–189.
[22] P. E. Livadas, D. T. Small, Understanding code containing preprocessor constructs, in: IEEE Third Workshop on Program Comprehension, 1994, pp. 89–97.
[23] M. Vittek, Refactoring browser with preprocessor, in: CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2003, p. 101.
[24] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, E. Merlo, Program understanding and maintenance with the CANTO environment, in: ICSM '97: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 1997, p. 72.
[25] A. Garrido, R. Johnson, Analyzing multiple configurations of a C program, in: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2005, pp. 379–388.
[26] A. Garrido, Program refactoring in the presence of preprocessor directives, Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, adviser: Ralph Johnson (2005).
[27] S. Lapierre, B. Laguë, C. Leduc, Datrix source code model and its interchange format: lessons learned and considerations for future work, SIGSOFT Softw. Eng. Notes 26 (1) (2001) 53–56.
[28] R. C. Holt, A. Schürr, S. E. Sim, A. Winter, GXL: a graph-based standard exchange format for reengineering, Science of Computer Programming 60 (2) (2006) 149–170.
[29] L. Vidács, A. Beszédes, R. Ferenc, Columbus schema for C/C++ preprocessing, in: CSMR '04: Proceedings of the Eighth European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2004, pp. 75–84.
[30] I. D. Baxter, M. Mehlich, Preprocessor conditional removal by simple partial evaluation, in: WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, 2001, pp. 281–292.
[31] L. Aversano, M. D. Penta, I. D. Baxter, Handling preprocessor-conditioned declarations, in: SCAM'02: Second IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society, Los Alamitos, CA, USA, 2002, pp. 83–93.
[32] Y. Hu, E. Merlo, M. Dagenais, B. Lagüe, C/C++ conditional compilation analysis using symbolic execution, in: ICSM '00: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2000, pp. 196–207.
[33] W. G. Griswold, D. Notkin, Automated assistance for program restructuring, ACM Transactions on Software Engineering and Methodology 2 (3) (1993) 228–269.
[34] G. C. Murphy, M. Kersten, L. Findlater, How are Java software developers using the Eclipse IDE?, IEEE Software 23 (4) (2006) 76–83.
[35] D. Spinellis, Optimizing header file include directives, Journal of Software Maintenance and Evolution: Research and Practice.To appear.
[36] S. Lampoudi, D. M. Beazley, SWILL: A simple embedded web server library, in: USENIX Technical Conference Proceedings, USENIX Association, Berkeley, CA, 2002, fREENIX Track Technical Program.
[37] E. R. Gasner, E. Koutsofios, S. C. North, K.-P. Vo, A technique for drawing directed graphs, IEEE Transactions on Software Engineering 19 (3) (1993) 124–230.
[38] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, 2000.
[39] R. T. Fielding, R. N. Taylor, Principled design of the modern Web architecture, ACM Transactions on Internet Technology 2 (2) (2002) 115–150.
[40] R. Wuyts, Declarative reasoning about the structure of object-oriented systems, in: TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems, IEEE Computer Society, Washington, DC, USA, 1998, pp. 112–124.
[41] D. Janzen, K. D. Volder, Navigating and querying code without getting lost, in: AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2003, pp. 178–187.
[42] K. De Volder, JQuery: A generic code browser with a declarative configuration language, in: Practical Aspects of Declarative Languages, Springer Verlag, 2006, pp. 88–102, lecture Notes in Computer Science 3819.
[43] D. Spinellis, A tale of four kernels, in: W. Schäfer, M. B. Dwyer, V. Gruhn (Eds.), ICSE '08: Proceedings of the 30th International Conference on Software Engineering, Association for Computing Machinery, New York, 2008, pp. 381–390.
[44] D. Spinellis, Code finessing, Dr. Dobb's 31 (11) (2006) 58–63.

[45] D. Spinellis, Debuggers and logging frameworks, IEEE Software 23 (3) (2006) 98–99.
[46] D. Spinellis, The way we program, IEEE Software 25 (4) (2008) 89–91.
[47] A. Milanova, A. Rountev, B. G. Ryder, Precise call graphs for c programs with function pointers, Automated Software Engineering 11 (1) (2004) 7–26.

Id: cscout.tex 1.26 2008/06/15 22:42:10 dds Exp