CScout: A Refactoring Browser for C

Diomidis Spinellis[a]

[a]*Athens University of Economics and Business*
Department of Management Science and Technology
Patision 76, GR-104 34 Athens, Greece

Abstract

Despite its maturity and popularity, the C programming language still lacks tool support for reliably performing even simple refactoring, browsing, or analysis operations. This is primarily due to 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 backend allows more complex source code analysis and manipulation. *CScout* has been successfully applied to many medium and large proprietary and open source projects identifying thousands of modest refactoring opportunities.

Key words: C, browser, refactoring, preprocessor

1. Introduction

C remains the language of choice for developing systems applications, such as operating systems and databases, embedded software, and the majority of open-source projects [44, 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 source 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 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. The theory behind *CScout*'s operation is described in detail elsewhere [45]; this paper focuses on the tool's design, implementation, and application.

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) correctly handling most of the complexity introduced by the C preprocessor. *CScout* takes advantage of modern hardware (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, *CScout*'s analysis 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

Science of Computer Programming, 75(4):216–231, April 2010.

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

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 [15]. The most important culprits 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 [18] and the references therein). Significantly, the C preprocessor and a compilation environment based on loosely-coupled tools, like *make* and a language-agnostic linker, 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 [11, 4, 45]. 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, a renaming operation of the identifier `len` will require changing in all three definitions, although in C the members of each data structure belong to a different namespace.

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

int s = get_block_len(db) + get_block_len(mb);
```

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.

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

An additional complication comes from the use of conditionally compiled code (see also Sections 4.1 and 7). 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. Code Reuse Complications

Parnas [38] 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 and Lai [61]). When 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 [57]. We have identified three interesting instances of source code sharing in such families.

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).

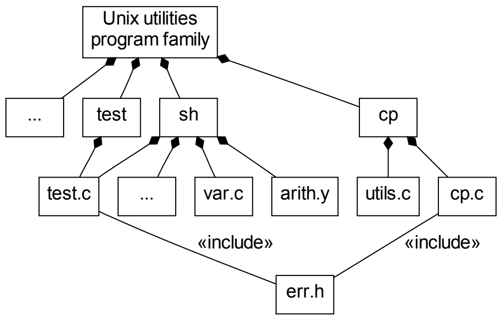


Figure 1: Program family relationships in the FreeBSD implementation of the Unix utilities.

Ad-hoc code reuse. In many cases elements of a source code base are reused to create various executable programs. Although code reuse is typically realized by creating a common library (such as the Unix 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`.

Version branches. 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 three cases we described, 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 allow browsing and propagate refactoring operations across all files in a given program family.

2.3. Problem Impact

Due to the previously described 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. When working with a compiler or IDE-constructed symbol table there is another problem. Many C implementations treat preprocessing as a separate phase and fail to pass information about C macros down through the other compilation phases. Therefore, a more sophisticated search using such a 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 suffer, because programmers, unsupported by the tools they use, are reluctant to perform even a simple rename-function refactoring. Anecdotal evidence supports our observation: consider mutilated identifier names such as that of the Unix `creat` system call that still persist, decades after the reasons for their original names have become irrelevant [7, p. 60]. The readability of existing code slowly decays as layers of deprecated historical practice accumulate [23, 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* [19, pp. 297–307] and *refactoring browsers* [40] respectively. Related work on object-oriented design refactoring [56] asserts that it is generally not possible to handle all problems introduced by preprocessing in large software applications. However,

Table 1: Comparison of C and C++ Refactoring and Transformation Tools

	CScout	**Xrefactory**	**Proteus**	**CDT**	**Refactor!**
Number of supported refactorings	4	11	∞	5	150
Handle C namespaces	√	√	√	√	√
Rename preprocessor identifiers	√	√	×	√	√
Handle scopes introduced by the C preprocessor	√	√	×	×	×
Handle identifiers created by the C preprocessor	√	×	×	×	×
C++ support	×	√	√	√	√
Yacc support	√	×	×	×	×
User environment	Web	Emacs	—	Eclipse	Visual Studio
Reference		[59]	[60]	[42]	[9]

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 [28, 1, 25]) it is, in principle, also applicable to programs written in C++ [53], Cyclone [27], PL/I and many assembly-code dialects.

Reference [10] 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 [43], and the definition of an abstract language for capturing the abstractions for the C preprocessor in a way that allows formal analysis [11]. The two-way mapping between preprocessor tokens and C-proper identifiers used by *CScout* was first suggested by Livadas and Small [34].

A number of tools support the refactoring and transformation of C and C++ code. A summary of their capabilities appears in Table 1; below we provide a brief description of each tool in comparison to *CScout*.

A tool adopting an approach similar to ours is Vittek's *Xrefactory* [59]. Its functionality is integrated with the Emacs editor [51]. Compared to *CScout*, *Xrefactory* supports C++, and thus also offers a number of additional refactorings: field and method moving, pushing down and pulling up fields and methods, and the encapsulation of fields. However, *Xrefactory* is unable to handle identifiers generated during the preprocessing stage; its author writes that deciding how to handle the renaming of an identifier that is constructed from parts of other identifiers is, in general, an unsolvable problem. The case refers to the renaming of the identifier `sysctl_var_sdelay` we showed in Section 2.1 into, say, `foo`. Vittek, correctly writes that there is no way to perform this renaming in a natural way. We sidestep this restriction by only allowing the renaming of an identifier's constituent parts. Thus, in this case, a *CScout*'s user can rename individually the identifier's `sysctl_var` part and the `sdelay` part, with each renaming affecting the other corresponding parts in the program.

Another related tool, *Proteus* [60], analyzes C and C++ code, faithfully preserving preprocessor directives, comments, and formatting information by integrating these elements into an abstract syntax tree (AST). This has the advantage of allowing more sophisticated transformations than those that *CScout* can perform. The changes are specified using a domain-specific language, YATL—Yet Another Transformation Language. *Proteus* handles all preprocessor directives as layout elements. Consequently, because *Proteus* does not consider and handle macro definitions as first-class entities these cannot be changed. Furthermore, the code reconstructed from the AST can differ from the original one, even if no transforms were applied; the authors conducted three large studies and found that 2.0%–4.5% of the lines differed.

In the recent years two IDEs have evolved to support the refactoring C and C++ code through add-on modules. Compared to *CScout* these support C++ and offer many more refactoring operations, but with less fidelity. The Eclipse C/C++ Development Tooling (CDT) project features the following refactorings: extract constant, extract function, generate getters and setters, hide method, and implement method [42]. The refactoring support of Visual Studio 2008 does not support C, but a third-party add-on *Refactor!* supports C/C++, offering 150 refactorings [9]. However, the most recent versions of these two systems (Eclipse CDT 5.0.2—2009-02-13 and Refactor! for Visual Studio 3.2—

Table 2: File and Function Metrics that *CScout* Collects

File Metrics

- Number of: statements, copies of the file, defined project-scoped functions, defined file-scoped (`static`) functions, defined project-scoped variables, defined file-scoped (`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 (e.g. `max(a, b)`), defined C preprocessor object-like macros (e.g. `EOF`)
- Maximum number of characters in a line

Function Metrics

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

2009-02-27) cannot handle the preprocessor complications listed in Section 2.1. Specifically, when attempting to rename identifiers appearing in Section's 2.1 source examples Eclipse CDT reports "The selected name could not be analyzed", whereas Refactor! renames the identifier specified, but fails to rename other associated instances.

Other related work has proposed the integration of multiple approaches, views, and perspectives into a single environment [2], the full integration of preprocessor directives in the internal representation [16, 14], the use of an abstract syntax graph for communicating semantic information [30], and the use of a GXL [21] schema for representing either a static or a dynamic view of preprocessor directives [58].

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 [5], the type checking of conditionally compiled code [3], and the use of symbolic execution to determine the conditions associated with particular lines of code [22].

4. The CScout Refactoring Browser

To be able to map and rename identifiers across program families accurately and efficiently *CScout* integrates in a single processing engine functions of a build tool (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,

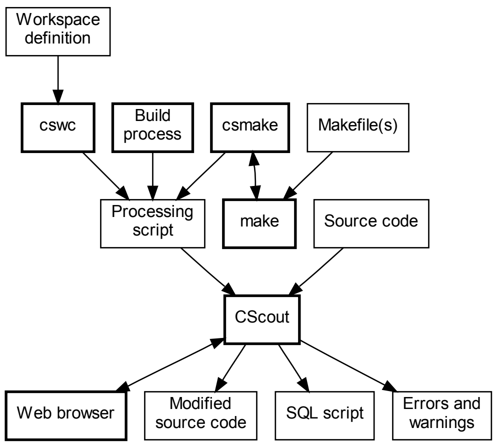


Figure 2: *CScout* system operation.

- 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,
- perform sophisticated queries for identifiers, files, and functions,
- monitor and report superfluously included header files, and
- provide accurate metrics over functions and files (see Table 2).

More importantly, *CScout* helps programmers in refactoring code by identifying dead objects to remove, and can automatically perform accurate global *rename identifier*, *add parameter*, *remove parameter*, and *change parameter order* refactorings [20]. One might question whether support for a few simple refactoring types merits calling *CScout* a refactoring tool. To answer this, consider that the *rename identifier* operation is by far the most common refactoring operation performed in practice [36], and that performing refactoring operations reliably on production C source code is very tricky. Specifically, *CScout* will automatically rename identifiers and refactor function arguments

- 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 identifiers: a refactoring operation 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 header files or even code,
- across conditionally compiled units, if an appropriate workspace (a set of interrelated linkage units) has been defined and processed.

Uniquely, *CScout* will rename identifiers occurring in macro bodies and even *parts* of other identifiers, when these are created through the C preprocessor's token concatenation feature.

4.1. Source Code Processing

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 header file directory search path, externally

Table 3: The `#pragma` Directives of the *CScout* Processing Script

Pragma	Action
`echo` *string*	Display the *string* on *CScout*'s standard output when the directive is processed.
`ro_prefix` *string*	Add *string* to the list of filename prefixes that mark read-only files. This is a global setting used for bifurcating the source code into the system's (read-only) files and the application's (writable) files.
`project` *string*	Set the name of the current project (linkage unit) to *string*. All identifiers and files processed from then on will be set to belong to the given project.
`block_enter`	Enter a nested scope block. Two blocks are supported, the first `block_enter` will enter the project scope (linkage unit); the second encountered nested `block_enter` will enter the file scope (compilation unit).
`block_exit`	Exit a nested scope block. The number of `block_enter` pragmasshould match the number of block_exit pragmas and there should never be more than two `block_enter` pragmas in effect.
`process` *string*	Analyze (*CScout*'s equivalent to compiling) the C source file named *string*.
`pushd` *string*	Set the current directory to *string*, saving the previous current directory in a stack. From that point onward, all relative file accesses will search the specified file from the given directory.
`popd`	Restore the current directory to the one in effect before a previously pushed directory. The number of `pushd` pragmas should match the number of `popd` pragmas.
`includepath` *string*	Add *string* to the list of directories used for searching included files (the include path).
`clear_include`	Clear the include path, allowing the specification of a new one from scrarch.
`clear_defines`	Clear all defined macros allowing the specification of new ones from scrarch. Should normally be executed before processing a new file. Note that macros can be defined in the processing script using the normal `#define` C preprocessor directive.

defined macros, and the linkage unit name to be associated with global identifiers. The script is a C file comprised mostly of `#define` directives and *CScout*-specific `#pragma` directives (see Table 3). In cases where the source code involves multiple configurations implemented through conditional compilation the script will contain directives to process the source code multiple times, once for each configuration with different options (defined macros or include file paths) set in each pass.

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

1. A *CScout* companion 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. This method has been used for processing all code listed in Table 4 (apart from the Solaris and Windows kernels), as well as tens of other Unix-based systems.
2. 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*. This method offers precise control of *CScout*'s processing. It is also useful in cases when *csmake* is not compatible with the platform's compilation process; *csmake* currently handles the programs *make*, *gcc*, *cc*, *ld*, *ar*, and *mv* running in a POSIX shell environment. A 27-line *csmake* specification has been used for processing the Unix utilities illustrated in Figure 1 and a 125-line specification for processing a 350 KLOC proprietary CAD system.
3. 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 1,149-line output of Microsoft's *nmake* program compiling the Windows Research Kernel into a 51,288-line *Cscout* processing script. Similarly, a 137-line Perl script was used to convert the 26,704-line output of Sun's *dmake* program [54] compiling the OpenSolaris kernel into a 140,552-line *Cscout* processing script.

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.

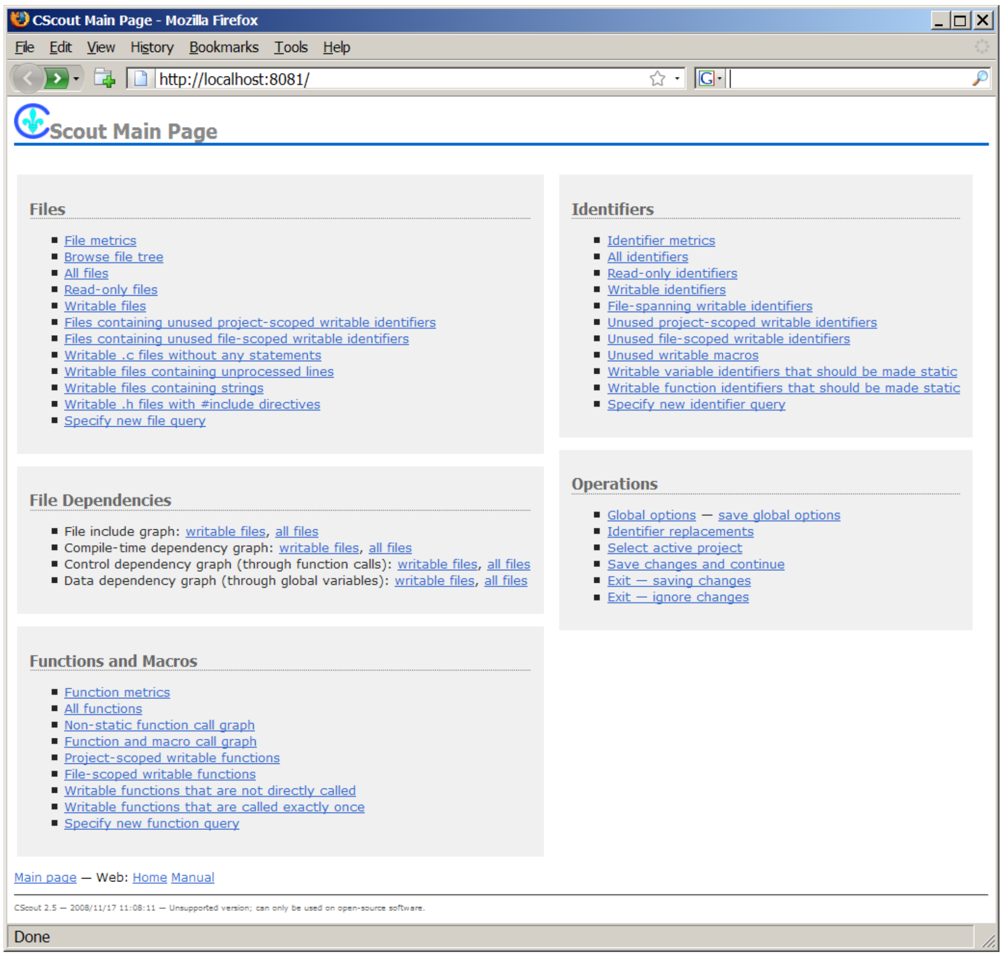
Many worthwhile maintenance activities can be performed by processing this standardized error report. In one case we automatically processed those warnings to remove 765 superfluous `#include` directives (out of a total of 5429) from a 190KLOC CAD program [50], thereby increasing its maintainability by reducing namespace pollution.

After processing all source files, *CScout* can operate 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 and various function argument refactorings) are mirrored in an in-memory copy of the source code. These changes can then be committed back to the source code files, optionally issuing commands for a version control system. When *CScout* writes back the refactored source code the only changes made are the renamed identifiers and the changed function arguments. Therefore, *CScout*'s effect on the source code's formatting is negligible. A separate backend enables *CScout* to export its data structures to a relational database management system for further processing.

4.2. Web-Based Interface

The easiest way to use *CScout* is through its interactive web-based interface (see Figure 3). Using the SWILL embedded web server library [29], *CScout* allows the connection of web clients to the tool's HTTP-server interface. Through a set of hyperlinks users can 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 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—see Figure 4(a).
- Specify simple form-based file and function queries based on the calculated metrics listed in Table 2.
- Perform queries for functions based on their callers, the functions they call, identifiers they contain, and the filenames where they reside.
- 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 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 a workspace), 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—see Figure 4(b). Uniquely, *CScout* can calculate metrics and call graphs that take into account both functions and function-like macros—see Figure 5(b) derived while browsing the source code of *awk* and drawn using *dot* [17]. This matches the reality of C programming, where the two are used interchangeably.


CScout Main Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8081/

CScout Main Page

Files

- File metrics
- Browse file tree
- All files
- Read-only files
- Writable files
- Files containing unused project-scoped writable identifiers
- Files containing unused file-scoped writable identifiers
- Writable .c files without any statements
- Writable files containing unprocessed lines
- Writable files containing strings
- Writable .h files with #include directives
- Specify new file query

File Dependencies

- File include graph: writable files, all files
- Compile-time dependency graph: writable files, all files
- Control dependency graph (through function calls): writable files, all files
- Data dependency graph (through global variables): writable files, all files

Functions and Macros

- Function metrics
- All functions
- Non-static function call graph
- Function and macro call graph
- Project-scoped writable functions
- File-scoped writable functions
- Writable functions that are not directly called
- Writable functions that are called exactly once
- Specify new function query

Identifiers

- Identifier metrics
- All identifiers
- Read-only identifiers
- Writable identifiers
- File-spanning writable identifiers
- Unused project-scoped writable identifiers
- Unused file-scoped writable identifiers
- Unused writable macros
- Writable variable identifiers that should be made static
- Writable function identifiers that should be made static
- Specify new identifier query

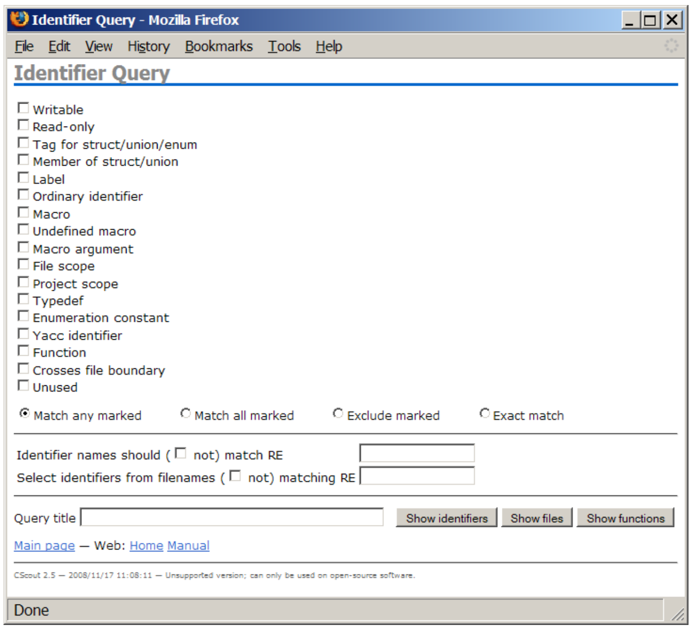
Operations

- Global options — save global options
- Identifier replacements
- Select active project
- Save changes and continue
- Exit — saving changes
- Exit — ignore changes

Main page — Web: Home Manual

CScout 2.5 — 2008/11/17 11:08:11 — Unsupported version; can only be used on open-source software.

Figure 3: A screen dump of the *CScout* web interface.


Identifier Query - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Show identifiers
Show files
Show functions
Done

Identifier Query

☐ Writable
☐ Read-only
☐ Tag for struct/union/enum
☐ Member of struct/union
☐ Label
☐ Ordinary identifier
☐ Macro
☐ Undefined macro
☐ Macro argument
☐ File scope
☐ Project scope
☐ Typedef
☐ Enumeration constant
☐ Yacc identifier
☐ Function
☐ Crosses file boundary
☐ Unused

[✓ Match any marked] Match all marked Exclude marked Exact match

Identifier names should (☐ not) match RE

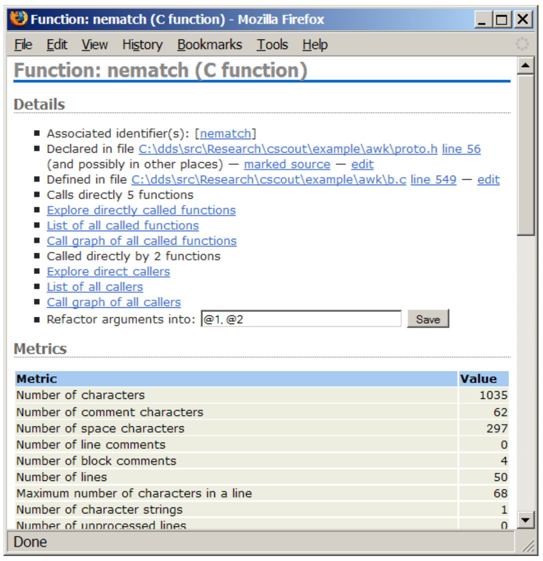
Select identifiers from filenames (☐ not) matching RE

Query title

Main page — Web: Home Manual

CScout 2.5 — 2008/11/17 11:08:11 — Unsupported version; can only be used on open-source software.

(a) The identifier query form.


Function: nematch (C function) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Save
Done

Function: nematch (C function)

Details

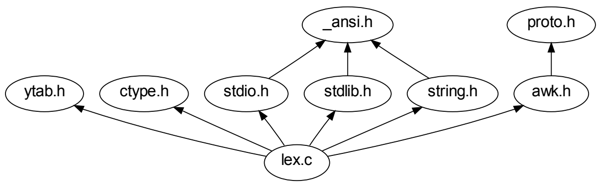
- Associated identifier(s): [nematch]
- Declared in file C:\dds\src\Research\cscout\example\awk\proto.h line 56 (and possibly in other places) — marked source — edit
- Defined in file C:\dds\src\Research\cscout\example\awk\b.c line 549 — edit
- Calls directly 5 functions
- Explore directly called functions
- List of all called functions
- Call graph of all called functions
- Called directly by 2 functions
- Explore direct callers
- List of all callers
- Call graph of all callers
- Refactor arguments into: @1, @2

Metrics

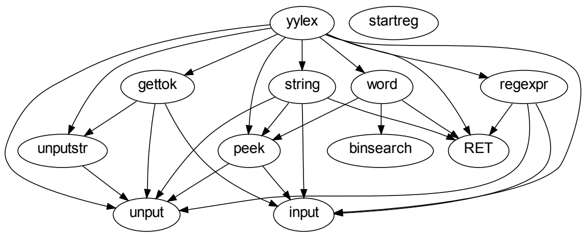
Metric	Value
Number of characters	1035
Number of comment characters	62
Number of space characters	297
Number of line comments	0
Number of block comments	4
Number of lines	50
Maximum number of characters in a line	68
Number of character strings	1
Number of unprocessed lines	0

(b) A function or macro page.

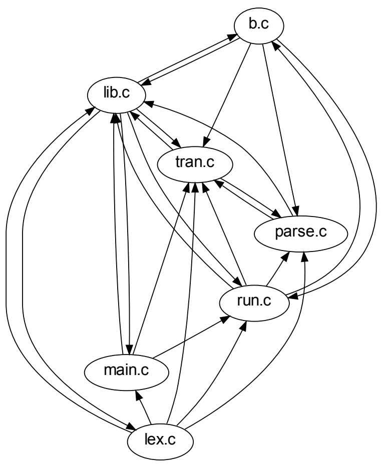
Figure 4: The *CScout* web front-end in operation.



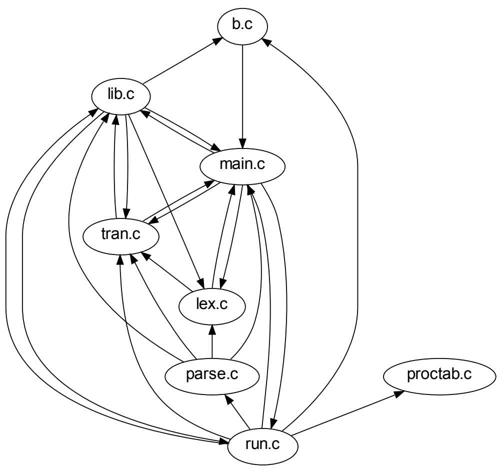
(a) Included files.



(b) Call graph spanning functions and macros.



(c) Control dependencies between files.



(d) Data dependencies between files.

Figure 5: *CScout*-generated graphs for the *awk* source code file `lex.c`.

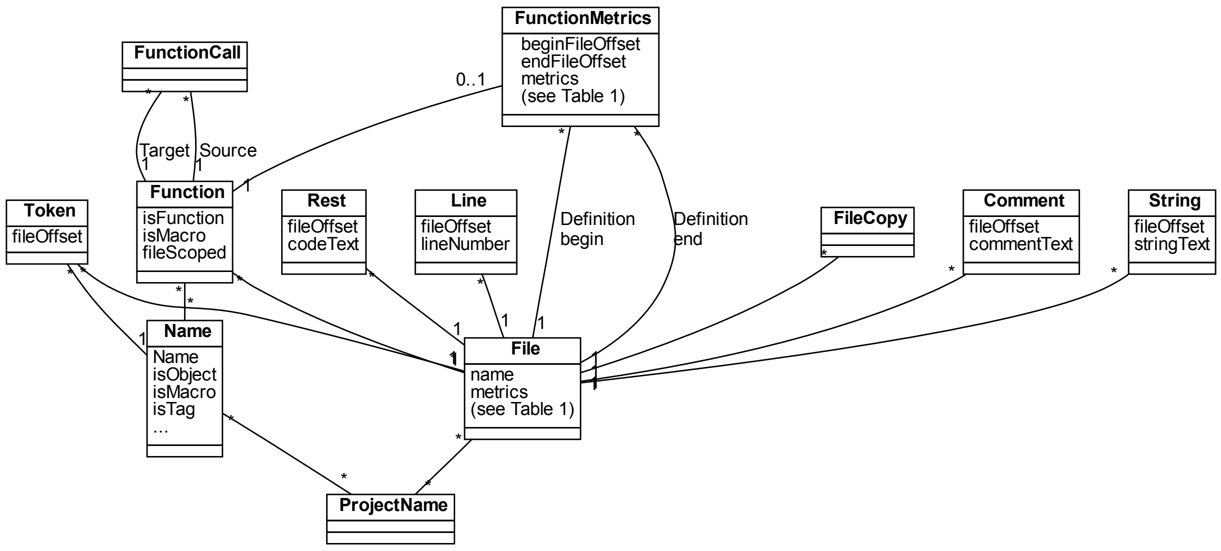


Figure 6: The logical schema of the exported database.

- Generate graphs of compile-time and run-time control and data–dependencies—see Figures 5(c) and 5(d). .
- Perform *rename identifier* and various *function argument* refactorings. Specifically, users can substitute all matching instances of a given identifier with a new user-specified name. In addition, in a function's web page users can specify a substitution template for the function's parameters. This can include the original parameters (denoted by placeholders for the original arguments, named `@1`, `@2`, *etc.*), as well as other arbitrary text, like constants and expressions. Refactorings can be specified multiple times, allowing the incremental improvement of the code, without the expensive reprocessing step. A separate operation will permanently save the modified code.

The rename functionality can be used to semi automatically perform two important refactoring operations: *rename*, e.g. Griswold and Notkin's [20] "rename-variable", and *remove*, e.g. Fowler's [13] "Remove Parameter". Remove refactorings can be trivially performed by hand, after identifiers that occur exactly once have been automatically and accurately identified. 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 substitution template for function parameters can be used for adding a function argument (with a user-specified default value), for removing a function argument (by omitting its placeholder from the substitution pattern), and for changing the order of a function's arguments.

The web server follows the representational state transfer (REST) architecture [12], and therefore its URLS can be used for interoperating with other tools. For instance, a build tool could use the URL

```
http://localhost:8081/fgraph.txt?gtype=C
```

to obtain the compile-time dependencies between a project's files. 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, many links appearing on *CScout*'s main web page are simply canned hyperlinks to the general queries we previously outlined.

4.3. SQL Backend

CScout can also dump the data structures representing the source code analysis it performed in an SQL script suitable for loading the data into a relational database. There is considerable history behind storing source code in a relational schema both for procedural languages in general [33] and, in particular, for C [6]. We chose to use a relational model over a specialized and more expressive logic query language, along the lines of SOUL [62] or JQuery [26, 8], 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 [48] 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 omitted.) Through the database one can issue all the queries available on the GUI front-end 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 (a measure 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, a design that matched the software and computing resources at hand, and substantial implementation work. The major challenges can be divided into: preprocessing and parsing, the enforcement of C namespaces, the handling of C preprocessor complications, the handling of code reuse complications, testing, achieving adequate performance, and keeping the project in a manageable scale.

5.1. Preprocessing and Parsing

Preprocessing C is anything but trivial. *CScout*'s lexical analyzer and the C preprocessor are hand-crafted; converging toward a correct preprocessor proved to be tricky. For many years *CScout* would be patched to fix misbehaviors occurring in obscure cases of macro invocations. The situation was becoming increasingly difficult, because often fixing one case would break another. In the end we realized that the only way to achieve correct behavior was to locate (through a personal communication with its author) and implement the so-called Prosser's macro expansion algorithm [39]. Almost miraculously all test cases worked correctly, and after two years of use and many millions of processed code, no other problems were reported in the area of macro expansion.

In contrast to C++, C is not difficult to parse, but the grammar supplied as part of the C standards is not suitable for generating *yacc*-based parsers, because such parsers then contain numerous rule conflicts. *CScout*'s C grammar is based on Roskind's work [41] extended to support the parsing of *yacc* files, and many C99 [25], *gcc*, and Microsoft C extensions. It comprises 144 productions and, after 149 revisions, it is 2,670 lines long. The parsing of preprocessor expressions and the C code are handled by two separate *btyacc* grammars. *Btyacc* was selected over *yacc* for its portability, better support for C++, superior handling of syntax errors through backtracking, and the ability to customize it in order to support the side-by-side linking of two separate 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.

5.2. Enforcement of C Namespaces

The separation of identifiers into C namespaces is achieved through a symbol table containing basic type information for identifiers in the current scope. Furthermore, support for the C99 initializer designators also requires the evaluation of compile-time constants. This non-trivial functionality is needed, because 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 a tag's 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. 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.

5.3. Handling C Preprocessor Complications

The basic principle of *CScout*'s 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 [45]. To handle the scoping rule mix-ups generated by the C preprocessor (see Section 2.1), every identifier is set to belong 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. 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. When each identifier token is read, a new equivalence class for that token is created. 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 a parameter of a function-like macro) the two corresponding equivalence classes are *unified* into a single one.

In total, 20 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 [28] function parameter declarations with the respective formal parameter name, multiple declarations and definitions of objects with compilation or linkage unit scope, and `goto` labels and targets, respectively.

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.
- Verifying that a renamed identifier does not clash with other identifiers means checking that no new equivalence class unifications occur when reprocessing the code. This method handles correctly all the language's scoping rules, a problem for many other refactoring tools [52].
- 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.

5.4. Handling Code Reuse Complications

CScout handles the code reuse complications outlined in Section 2.2 by providing an integrated build system that can process multiple linkage units as a whole.

An early design choice based this build system on extending the C language that *CScout* can process with a few `#pragma` directives (see Table 3). Making the build language an extension of C means that existing C facilities can be used for a number of tasks. Thus, external macro definitions that other build systems pass to the compiler as flags are simply defined through `#define` directives. Furthermore, internally-defined macro definitions, such as those handling *gcc*'s built-in intrinsic functions, can be easily introduced simply by processing the file that defines them with a `#include` directive.

Making the build language textual, rather than GUI-based as is typically the case in many IDES, means that other more sophisticated tools can create build scripts. This is the case with the *cswc*, the *CScout* workspace compiler and *csmake*, the *make*-driven build process monitor and build script generator.

5.5. Testing

The complexity of *CScout*'s analysis requires a framework to ensure that it remains functional and correct as the code evolves. 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 parsing 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 verify corner cases and check for accidental errors. The *CScout*'s preprocessor is tested through 70 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 previously verified results.

5.6. Performance

With *CScout* processing multi-million line projects as a single entity, time and space performance have to be kept within acceptable bounds, with increases at most linearly dependent on the size of the input. 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. These choices also allow the elegant and efficient expression of complex relationships, using STL functions like `set_union`, `set_-intersection`, and `equal_range`. Up to now algorithmic tuning was required only once, to fix a pathological case in the implementation of the C preprocessor macro expansion [46].

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 [47]: a lightweight and efficient construct that allowed us to instrument the code with (currently 200) 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 at runtime by specifying in a text file 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.

5.7. Project Scale

Implementing a tool of *CScout*'s complexity proved to require considerable effort. *CScout* has been actively developed for five years, and currently consists of 27 KLOC. Most of the code is written in C++ with Perl being used to implement the *CScout* processing script generators *csmake* and *cswc*. Two more Perl scripts automatically extract from the source code the documentation for the SQL database schema and the reported error messages.

Eight class hierarchies allow for some inheritance-based code reuse. Ordered by decreasing number of classes in each inheritance tree, these cover C's types, graph rendering, the handling of user options, SQL output, query processing, C's tokens, metrics, and functions.

More importantly, *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 [29], the *dot* graph drawing program [17], 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 tens of open source and commercial software systems running on a variety of hardware and software platforms [48, 49, 50]. The workspace sizes range 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 [28], ANSI C [1], and many C99 [25], *gcc*, and Microsoft C extensions.) Details of some representative projects can be seen in Table 4, while data of the hosting hardware appears in Table 5. 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.

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.

PostreSQL The PostgreSQL relational database, version 8.2.5.

GDB The GNU debugger, version 6.7.

In the cases of awk, Apache httpd, GDB, WRK, and GDB 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 run time–loadable modules providing functionality for various devices, filesystems, networking protocols, and additional features. Similarly, PostgreSQL included in its build numerous commands, tests, and dynamically-loadable localization libraries. With *CScout* all linkage units were processed as a single workspace, allowing browsing and refactoring to span elements residing in different linkage units.

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

Table 4: Details of representative processed applications.

	awk	Apache httpd	FreeBSD kernel	Linux kernel	Solaris kernel	WRK	PostgreSQL	GDB
Overview								
Configurations	1	1	3	1	3	2	1	1
Modules (linkage units)	1	3	1,224	1,563	561	3	92	4
Files	14	96	4,479	8,372	3,851	653	426	564
Lines (thousands)	6.6	59.9	2,599	4,150	3,000	829	578	363
Identifiers (thousands)	10.5	52.6	1,110	1,411	571	127	32	60
Defined functions	170	937	38,371	86,245	39,966	4,820	1,929	7,084
Defined macros	185	1,129	727,410	703,940	136,953	31,908	4,272	6,060
Preprocessor directives	376	6,641	415,710	262,004	173,570	35,246	13,236	20,101
C statements (thousands)	4.3	17.7	948	1,772	1,042	192	70	129
Refactoring opportunities								
Unused file-scoped identifiers	20	15	8,853	18,175	4,349	3,893	2,149	2,275
Unused project-scoped identifiers	8	8	1,403	1,767	4,459	2,628	2,537	939
Unused macros	4	412	649,825	602,723	75,433	25,948	1,763	2,542
Variables that could be made static	47	4	1,185	470	3,460	1,188	29	148
Functions that could be made static	10	4	1,971	1,996	5,152	3,294	133	69
Performance								
CPU time	0.81”	35”	3h 43’40”	7h 26’35”	1h 18’54”	58’53”	3’55”	11’13”
Lines / s	8,148	1,711	194	155	634	235	2,460	539
Required memory (MB)	21	71	3,707	4,807	1,827	582	463	376
Bytes / line	3,336	1,243	1,496	1,215	639	736	840	1,086

Table 5: Performance measurements' hardware configuration.

Item	Description
Computer	Custom-made 4U rack-mounted server
CPU	4 × Dual-Core Opteron
CPU clock speed	2.4GHZ
L2 cache	1024KB (per CPU)
RAM	16GB 400 MHZ DDR2 SDRAM
System Disks	2 × 36GB, SATA II, 8 MB cache, 10k RPM, software RAID-1 (mirroring)
Storage Disks	8 × 500GB, SATA II, 16 MB cache, 7.2k RPM, hardware RAID-10 (4-stripped mirrors)
Database Disks	4 × 300GB, SATA II, 16 MB cache, 10k RPM, hardware RAID-10 (2-stripped mirrors)
RAID Controller	3ware 9550SX, 12 SATA II ports, 226MB cache
Operating system	Debian 5.0 stable running the 2.6.26-1-amd64 Linux kernel

Another interesting element of the analysis was the handling of different configurations for FreeBSD, OpenSolaris, and WRK [48]. 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. The time required to write-back the refactored files is negligible. For instance, saving the 96 files of Apache httpd (60 KLOC) with all identifiers replaced with a unique random name required in our configuration 331 ms—about 1% of the total processing time. However, if the user opts to check rename refactorings for clashes with other identifiers, then a complete reprocessing of the source code is required; this takes about the same time as the original processing.

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 added an option in the refactoring engine to rename all the writable identifiers into new, mechanically derived, random identifier names. We applied this transformation to the *apache* HTTP server source code; the resulting version compiled and appeared to work without a problem. This source code transformation can be applied on proprietary code to derive an architecture-neutral software distribution format: a (minimally) obfuscated version of the source code, which, like compiled code, unauthorized third parties cannot easily comprehend and modify, but, unlike compiled code, can be configured and compiled on each end-user platform to match its processor architecture and operating system.

7. Lessons Learned

The main lessons learned from *CScout*'s development are the value of end-to-end whole-workspace analysis of C source code, and the many practical difficulties of dealing with real-world C software. Researchers can apply these lessons by adopting a similar depth of analysis, such as the analysis already done in the LLVM compiler infrastructure project [31]. Alternatively, researchers at the forefront of tool technology, can save a lot of effort and pain by steering their energy toward more tractable languages, like Java. Furthermore, commercial tool builders should plan and budget for the difficulties we outline.

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, these complications in our scheme arise from preprocessing features.

Unifying undefined macros. In the absence of a shared `#undef` directive two undefined macros with the same name can only be unified into a single identifier through a heuristic rule that considers them to be referring to the same entity. This is typically a correct assumption, because testing through undefined macros is used for configuring software through a carefully managed namespace, with identifiers such as `HAS_FGETPOS` and `HAS_GETPPID`.

Coverage of macro applications. Dealing with function-like macros whose application does not cover all possible cases needed for semantically correct refactoring can be problematic. Consider the first case in Section 2.1. If the code does not apply `get_block_len` on at least one element of type `disk_block` and one of type `mem_block` *CScout* has no way to know that all three instances of `len` are semantically equivalent and should be renamed in concert.

Handling conditional compilation. In practice, this issue 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 the three supported 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 [1] and a number of C99 [25] extensions. In practice we found that *CScout* could not be applied on real-world source code without supporting many compiler-specific language extensions. Even programs that were written in a portable manner included platform-specific header files, which used many compiler extensions, and could therefore 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 by 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 and seldom-used 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 mistakenly rely on the extension.

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.

8. Conclusions

We have plans to extend CScout in a number of directions. One challenging and worthwhile avenue is support for the C++ language and object-oriented refactorings.

The web front-end is beginning to show its age. It should probably be redesigned to use of AJAX technologies, communicating with the *CScout* engine through XML requests. This interface would also allow the implementation of a more sophisticated testing framework. Queries can be made considerably more flexible by allowing the user to specify them in an embedded scripting language, like Lua [24]. Such a change would probably also require the provision of an asynchronous mechanism for aborting expensive queries. An alternative approach would be to provide a built-in SQL interface, perhaps through virtual tables of an embedded database, like SQLite [37].

Currently, many URLs of the web front end are fragile, breaking across *CScout* invocations or when the web-front end source code changes. These URLs can be made more robust by expressing them at a higher level of abstraction. Logging of *CScout*'s HTTP requests can provide research data on its actual use.

Source code browsing can also be improved. The source code views can be enhanced through the use of configurable syntax coloring and easier navigation to various elements. An interface can be provided for showing identifiers shared between two files. Refactoring opportunities can be pointed out by identifying bad smells in the code. These can be located through the judicious provision of some key metric-based queries, and through the automatic detection of duplicated code [32].

CScout could support file names as first class citizens. This should allow the renaming of a file name, correcting all references to it in include directives. Furthermore, the web front-end should hyperlink file names appearing in include directives. On the same subject, *CScout* could provide a header refactoring option to support the style guideline that requires each included file to be self-sufficient (compile on its own) by including all the requisite header files [55, p. 42].

CScout's support for DSLs can be improved along a number of lines. For one, *csmake* should also support *yacc* invocations. More generally, it would probably be worthwhile to provide *CScout* with an option to perform best-effort identifier substitutions in files it can't parse. These substitutions would be performed simply by matching whole words; developers will enable this option when they are reasonable confident that there are no spurious matches of the identifiers they rename in DSL files. In the future, ubiquitous 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 more robust solution.

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 perspective are the automatic identification and implementation of more complex refactoring operations, increasing the accuracy of dependency graphs by reasoning about function pointers [35], the production of source code views for given macro values, and the efficient maximization of code coverage.

Acknowledgements and Tool Availability

We would like to thank the anonymous reviewers for their many excellent suggestions to improve this paper and *CScout*. The following people have helped over the years the development of *CScout* with advice, comments, and feature requests: Walter Briscoe, Wilko Bulte, Munish Chopra, Georgios Gousios, Poul-Henning Kamp, Kris Kennaway, Alexander Leidinger, Sandor Markon, Marcel Moolenaar, Richard A. O'Keefe, Igmar Palsenberg, Wes Peters, Dave Prosser, Jeroen Ruigrok van der Werven, Remco van Engelen, and Peter Wemm. The tool, its documentation, and representative examples are available at `http://www.dmst.aueb.gr/dds/cscout/`. *CScout* currently runs under the FreeBSD, Linux, Mac OS X, Microsoft Windows, and Solaris operating systems under several 32 and 64-bit architectures. The freely-downloadable version of *CScout* can be used on open-source code; the supported commercial version is licensed for use on proprietary code, and includes the obfuscation and SQL back-ends.

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] Y.-F. Chen, M. Y. Nishimoto, C. V. Ramamoorthy, The C information abstraction system, IEEE Transactions on Software Engineering 16 (3) (1990) 325–334.

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

[8] 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.

[9] Developer Express Inc., Refactoring your code with Refactor!, Online `http://www.devexpress.com/Products/Visual_Studio_Add-in/Refactoring/whitepaper.xml`. Accessed 2009-03-17. Archived by WebCite at `http://www.webcitation.org/5fMxaOP4n`, white paper (2009).
URL `http://www.webcitation.org/5fMxaOP4n`

[10] 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.

[11] 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.

[12] R. T. Fielding, R. N. Taylor, Principled design of the modern Web architecture, ACM Transactions on Internet Technology 2 (2) (2002) 115–150.

[13] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, 2000.

[14] 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).
URL `http://www.lifia.info.unlp.edu.ar/papers/2005/Garrido2005.pdf`

[15] 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.

[16] 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.

[17] 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.

[18] 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, pLDI '01: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.

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

[20] W. G. Griswold, D. Notkin, Automated assistance for program restructuring, ACM Transactions on Software Engineering and Methodology 2 (3) (1993) 228–269.

[21] 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.

[22] 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.

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

[24] R. Ierusalimschy, Programming in Lua, 2nd ed., Lua.org, Rio de Janeiro, 2006.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: CGO '04: Proceedings of the 2004 International Symposium on Code Generation and Optimization, 2004.

[32] Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-miner: Finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering 32 (3) (2006) 176–192.

[33] M. A. Linton, Implementing relational views of programs, in: SDE 1: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, New York, NY, USA, 1984.

[34] P. E. Livadas, D. T. Small, Understanding code containing preprocessor constructs, in: IEEE Third Workshop on Program Comprehension, 1994.

[35] A. Milanova, A. Rountev, B. G. Ryder, Precise call graphs for C programs with function pointers, Automated Software Engineering 11 (1) (2004) 7–26.

[36] G. C. Murphy, M. Kersten, L. Findlater, How are Java software developers using the Eclipse IDE?, IEEE Software 23 (4) (2006) 76–83.

[37] M. Owens, The Definitive Guide to SQLite, Apress, Berkeley, CA, 2006.

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

[39] D. F. Prosser, Complete macro expansion algorithm, Standarization committee memo X3J11/86-196, ANSI, New York, online `http://www.`

spinellis.gr/blog/20060626/x3J11-86-196.pdf. Accessed 2009-03-21. Archived by WebCite at X3J11/86-196 (Dec. 1986).
URL http://www.webcitation.org/5fRS2iru0

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

[41] J. A. Roskind, Grammar file for the dpANSI C language, Available online at http://www.ccs.neu.edu/research/demeter/tools/master/doc/headers/C++Grammar/c4.y. Accessed: 2009-03-13. Archived by WebCite at http://www.webcitation.org/5fF8fX28Q (Mar. 1990).
URL http://www.webcitation.org/5fF8fX28Q

[42] D. Schaefer, Code analysis and refactoring with CDT, Available online http://cdtdoug.blogspot.com/2008/11/code-analysis-and-refactoring-with-cdt.html. Accessed 2009-03-15. Archived by WebCite at http://www.webcitation.org/5fMyo3trp, Eclipse Summit Europe presentation (Nov. 2008).
URL http://www.webcitation.org/5fMyo3trp

[43] G. Snelting, Reengineering of configurations based on mathematical concept analysis, ACM Transactions on Software Engineering and Methodology 5 (2) (1996) 146–189.

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

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

[46] D. Spinellis, Code finessing, Dr. Dobb's 31 (11) (2006) 58–63.

[47] D. Spinellis, Debuggers and logging frameworks, IEEE Software 23 (3) (2006) 98–99.

[48] 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.

[49] D. Spinellis, The way we program, IEEE Software 25 (4) (2008) 89–91.

[50] D. Spinellis, Optimizing header file include directives, Journal of Software Maintenance and Evolution: Research and Practice 21 (4) (2009) 233–251.

[51] R. M. Stallman, EMACS: The extensible, customizable, self-documenting display editor, in: D. R. Barstow, H. E. Shrobe, E. Sandwell (eds.), Interactive Programming Environments, McGraw-Hill, 1984, pp. 300–325.

[52] F. Steimann, A. Thies, From public to private to absent: Refactoring Java programs under constrained accessibility, in: S. Drossopoulou (ed.), ECOOP '09: Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 2009, Lecture Notes in Computer Science.

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

[54] Sun Microsystems, Inc., Santa Clara, CA, Sun Studio 12: Distributed Make (dmake), part No: 819-5273. Available online http://docs.sun.com/app/docs/doc/819-5273. Accessed 2009-03-13 (2007).
URL http://docs.sun.com/app/docs/doc/819-5273

[55] H. Sutter, A. Alexandrescu, C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Addison Wesley, 2004.

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

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

[58] 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.

[59] M. Vittek, Refactoring browser with preprocessor, in: CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2003.

[60] D. G. Waddington, B. Yao, High-fidelity C/C++ code transformation, Electronic Notes in Theoretical Computer Science 141 (4) (2005) 35–56.

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

[62] 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, 1998.