Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code

Jörg Liebig
University of Passau
joliebig@fim.uni-passau.de

Christian Kästner
Philipps-University Marburg
kaestner@informatik.uni-marburg.de

Sven Apel
University of Passau
apel@fim.uni-passau.de

ABSTRACT

The C preprocessor *cpp* is a widely used tool for implementing variable software. It enables programmers to express variable code (which may even crosscut the entire implementation) with conditional compilation. The C preprocessor relies on simple text processing and is independent of the host language (C, C++, Java, and so on). Language-independent text processing is powerful and expressive—programmers can make all kinds of annotations in the form of `#ifdef`s—but can render unpreprocessed code difficult to process automatically by tools, such as refactoring, concern management, and variability-aware type checking. We distinguish between disciplined annotations, which align with the underlying source-code structure, and undisciplined annotations, which do not align with the structure and hence complicate tool development. This distinction raises the question of how frequently programmers use undisciplined annotations and whether it is feasible to change them to disciplined annotations to simplify tool development and to enable programmers to use a wide variety of tools in the first place. By means of an analysis of 40 medium-sized to large-sized C programs, we show empirically that programmers use *cpp* mostly in a disciplined way: about 84 % of all annotations respect the underlying source-code structure. Furthermore, we analyze the remaining undisciplined annotations, identify patterns, and discuss how to transform them into a disciplined form.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

General Terms

Languages

Keywords

preprocessor, ifdef, conditional compilation, virtual separation of concerns, crosscutting concerns

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'11, March 21–25, 2011, Pernambuco, Brazil.
Copyright 2011 ACM 978-1-4503-0605-8/11/03 ...$10.00.

1. INTRODUCTION

The preprocessor *cpp* is a text processing tool that extends the programming language C by lightweight metaprogramming facilities [24]. It was originally designed for the programming language C and is nowadays also part of C++ and used with several other languages such as Fortran. The preprocessor provides three capabilities: file inclusion, textual substitution (macro substitution), and conditional inclusion (a.k.a. conditional compilation). Here, we concentrate on conditional inclusion and problems related to this capability [8, 26].

Conditional inclusion allows programmers to selectively include source code. To this end, a programmer *annotates* source code using the preprocessor directives `#ifdef`, `#ifndef`, and so on, which wrap lines of source code to make them optional. Programmers influence the inclusion of annotated code with configuration files or compiler flags and, to this end, generate different program variants, some of which include certain code fragments and some not. The application of conditional inclusion is not limited to a single file. Programmers use it for the implementation of features (end-user visible concerns) that often crosscuts the entire code base [26].

In academia, contemporary textual preprocessors are heavily criticized as error prone and as rendering code hard to read and maintain [1, 11, 10, 27, 35]. Instead of separating concerns, with preprocessors, developers often implement concerns with many small annotated code fragments scattered across the code base. There are two common suggestions to deal with this situation: The first is to *refactor concerns* and replace conditional compilation my means of contemporary language concepts that support crosscutting implementations, such as aspects, mixin layers, or feature modules; for example, a large body of research addresses the potential of refactoring `#ifdef` directives into aspects [1, 5, 6, 27, 32]. The second suggestion—called *concern management* or *virtual separation of concerns*—is to keep but explicitly manage scattered preprocessor implementations, often with additional tool support, for example, in the form of views on selected concerns, visualizations, and preprocessor-aware type systems [11, 15, 16, 17, 21, 30, 34].

Both approaches, concern refactoring and concern management, rely on an integrated analysis of the source-code structure and the effect of preprocessor directives. However, parsing and analyzing the unprocessed representation of the source code (pre-cpp) is known to be hard, because the preprocessor *cpp* is token-based and as such oblivious to the underlying source-code structure. Developers may annotate

arbitrary tokens, such as a single closing bracket. Although this is not a problem for tools that analyze a single preprocessed variant of the source code (post-cpp), such as a compiler or many static analysis tools, pre-cpp tools have difficulties handling arbitrary text-based annotations. These difficulties are widely acknowledged in prior research on code refactoring [12, 13, 14, 36, 38], transformation [1, 4, 22], slicing [37], or product-line aware analysis tools [3, 20, 23, 31]. Despite significant research effort (e.g., [12, 14, 36, 38]) and significant improvements, refactoring engines of IDEs such as Eclipse and Visual Studio still struggle with certain kinds of annotation in pre-cpp code.

A typical approach that has been used to analyze pre-cpp code in the past (sometimes explicitly, but mostly implicitly) is to handle not all, but only a subset of annotations, which we call *disciplined annotations*. Disciplined annotations are annotations on certain syntactic *code structures*, such as entire functions and statements, whereas we call annotations of individual tokens or brackets that do not align with underlying code structure *undisciplined annotations*. Restricting developers to disciplined annotations makes it much easier to build proper tools and to ensure correctness and completeness of the mechanisms involved. For example, we can parse source code including disciplined annotations in one step and propagate annotations to elements of an abstract syntax tree that we use as input for concern refactoring or concern management tools [22]. As another example, when developers annotate only entire functions, refactoring tools can safely transform function bodies or rename all alternative implementations of a function without loss of variability information. (As a side effect, disciplined annotations also prevent tedious syntax errors such as annotating only the closing but not the opening bracket of a function.)

A tool based only on disciplined annotations will not understand all existing source code, so developers may need to prepare the source code and bring some annotations into a disciplined form first. Some researchers claim that the restriction to disciplined annotations is not a significant limitation in practice and that source-code preparation is straightforward and quick to apply [4]. However, we are not aware of any empirical evidence for such claims. Although the preprocessor has been the topic in several studies [8, 26, 35], the issue of discipline has not been investigated empirically.

We make the following contributions:

- We describe (different forms of) disciplined and undisciplined annotations.
- We analyze 40 software projects with over 30 million lines of C code regarding the discipline of their annotations.
- We classify undisciplined annotations and present numbers of their occurrence.
- We present transformations for undisciplined annotations to make them disciplined and discuss implications for tool builders.
- We give recommendations for a common definition of disciplined annotations on which tool builders can rely.

This work differs from our previous work [26] in that we focus here on the discipline of preprocessor use, whereas, earlier, we analyzed *cpp*'s variability mechanisms regarding their potential to implement product lines. Here, we look at the preprocessor from the tool builder's perspective and discuss parsing and handling of potential ill-formed `#ifdef` annotations, whereas earlier we analyzed the programmers need for variability on the source-code level and the scattered nature of features implemented with conditional compilation. In previous work, we took side with the software engineer in terms of program understanding and alternative product-line implementation techniques, such as aspects or feature modules; here we take side with tool builders.

2. DISCIPLINED ANNOTATIONS

Before we look at the use of *cpp* in practice, we provide an overview of its conditional-inclusion capabilities and classify annotations into disciplined and undisciplined annotations. We illustrate our explanations with code excerpts of the open-source text editor *vim*.

2.1 Conditional Inclusion

Conditional inclusion is one of three capabilities to implement variable software with the preprocessor. Programmers annotate conditional parts of the source code with preprocessor directives such as `#ifdef` (for brevity, we refer to all conditional inclusion macros `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` as `#ifdef`s), which control lexical program transformations. The inclusion of conditional code depends on the evaluation of the condition that belongs to an `#ifdef`. The condition consists of one or more integer constants that are defined prior to the `#ifdef`. Figure 1 shows an example of conditional inclusion (Fig. 1 a) and the result of applying *cpp* to the source code (Fig. 1 b). Here, the integer constant `FEAT_NETBEANS_INTG` controls the inclusion of `struct signlist *prev;` (Line 13). A single `#ifdef` controls the inclusion of all subsequent lines until the next `#ifdef` occurs. This way, a programmer specifies which source code is optional. For implementing alternative source-code fragments, a programmer uses a combination of `#if`, `#elif`, and `#else`. In prior work we found that programmers use conditional inclusion frequently to implement variable source code that crosscuts the mandatory part of the software system [26].

Based on the values of integer constants, different *variants* can be generated. In our example, three variants are possible: with `struct signlist` (Lines 7 to 15) and/or `struct signlist *prev;` (Line 13) and without both elements. Because *cpp* works on the basis of tokens of the target language, it is language independent.[1] That is, programmers may annotate arbitrary source-code fragments. A key observation is that annotations may be used in a disciplined or undisciplined way.

2.2 Tool Builders' Requirements

Before we describe what we consider as disciplined annotations, we describe what form of annotations tool builders

[1]To be specific, *cpp* does not care about the underlying language and can thus be combined with any language. It even provides the command-line flag `-P` when running the preprocessor with non-C input to suppress `#line` directives. However, preprocessor directives may break existing tool support for unprocessed code. For example, a Java editor does not understand the preprocessor directives. As a consequence, several preprocessors have been developed that can define preprocessor directives inside comments, such as *Antenna* (`antenna.sourceforge.net`) and *Munge* (`weblogs.java.net/blog/tball/archive/munge/Munge.java`) for Java, or that have a configurable syntax, such as the preprocessor facilities provided by the commercial product-line tools *pure::variants* (`www.pure-systems.com`) and *Gears* (`www.biglever.com`).

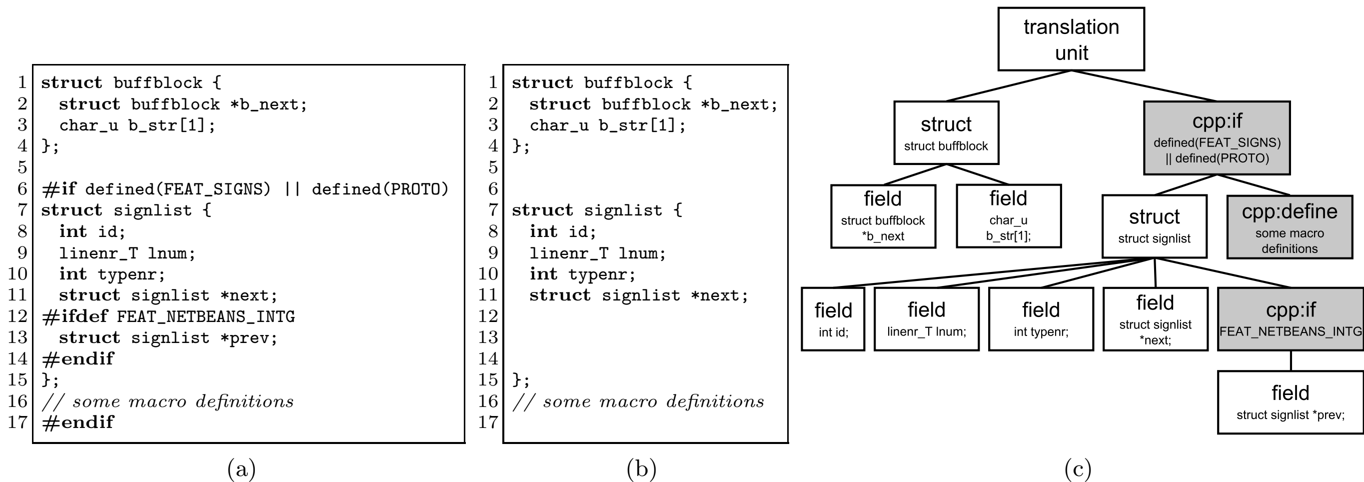

Figure 1: Example of conditional inclusion (a), the result after applying *cpp* (b), and the AST (c)

can or want to handle. For illustration, we use two scenarios (concern refactoring and concern management) that are common in the research area of crosscutting concerns and discuss the technical influence of preprocessor annotations on both scenarios.

Concern refactoring. A common idea is to refactor scattered `#ifdef`-based implementations into cohesive implementations such as aspects, if possible, automatically. After refactoring, variability is implemented by selecting which aspects to weave into the program instead of running a textual preprocessor. For example, Lohmann et al. refactored `#ifdef` annotations in an operating systems kernel [27], and Adams et al. identified common patterns of conditional inclusion that can be refactored into aspects [1]. The authors of both papers described that although they could refactor most annotations, they had to rely on heuristics and could not refactor all code. However, the authors also stated that handling of variable source-code fragments is a major problem when dealing with unprocessed source code (pre-cpp). To automate such refactoring, tools require a parse tree or abstract syntax tree (AST) with variability information, often enriched with information on types for analysis or transformation. It must be possible to identify variable code as compared to the base code. Hence, it is preferable that `#ifdef` annotations map to complete subtrees in the AST representation, such that reasoning about the source-code structure and its annotations is possible on the basis of a uniform representation.

Concern management. As an alternative strategy for refactoring, some researchers propose to keep scattered implementations of `#ifdef`s, but manage them with tool support. A typical strategy is to generate views on the source code. For example, a view on a feature X shows all code fragments annotated with `#ifdef X` together with the feature's context, but hides all remaining code [2, 16, 17, 21]. A developer can inspect (and often also edit) such view similar to a cohesive aspect implementation, but the view is generated on demand based on a scattered implementation. Henceforth, we call this approach *virtual separation of concerns* [21]. Beside visualizations, additional tool support may also include advanced error checking, such as syntactical correctness and consistency of views, or even that all valid feature combinations result in well-typed programs [20, 21].

Although some tool support for concern management is possible at a textual level, a common abstract syntax tree that includes information about features is helpful to reason about the code and provide navigation support. For example, to be useful, views must include some context information, such as the class or function the code fragment is defined in.[2] To determine the context for a view, an abstract syntax tree provides more accurate information than grep-like heuristics such as "show the 3 lines before and after the code fragment". In addition, advanced features such as variability-aware type checking strongly depend on an abstract syntax tree as well.

Interestingly, concern refactoring and concern management are two approaches popular in the research community on code-clone detection, in which different groups advocate either code-clone removal by refactorings or code-clone management with tool support [33].

An AST with variability information. Many tools need to build a parse tree or abstract syntax tree that contains also variability information from the annotations. In a uniform representation `#ifdef` annotations have to map to complete subtrees in the parse tree or abstract syntax tree to enable the combination of the tree information with the variability information. Consider the code fragment in Figure 1 a and the corresponding AST in Figure 1 c. Annotations are represented by nodes in the AST: The annotation of `struct signlist` in Line 7 has its corresponding `struct` subtree in the AST; the same for `struct signlist *prev` in Line 13. Note that the resulting AST in Figure 1 c contains the entire information for all three variants, so we do not need to generate all variants.

Unfortunately, annotations do not always align with the underlying source-code structure. Figure 2 shows an example, in which an annotation of optional code is split up into

[2]Presenting a single statement without further context would probably not be very helpful in understanding the implementation of a feature; similar context is available in aspects by repeating class and function names in interfaces or pointcuts.

two single annotations (Line 6 and 15); the end of the block `USE_ISPTS_FLAG` in Line 6 is in a different block (Line 15). This annotation cannot be mapped to a node or subtree in the AST and we consider it as undisciplined, as we explain in Section 2.3.

```
 1 #if defined(__GLIBC__)
 2 // additional lines of code
 3 #elif defined(__MVS__)
 4 result = pty_search(pty);
 5 #else
 6 #ifdef USE_ISPTS_FLAG
 7   if (result) {
 8 #endif
 9   result = ((*pty = open("/dev/ptmx", O_RDWR)) < 0);
10 #endif
11 #if defined(SVR4) || defined(__SCO__) || \
12     defined(USE_ISPTS_FLAG)
13   if (!result)
14     strcpy(ttydev, ptsname(*pty));
15 #ifdef USE_ISPTS_FLAG
16   IsPts = !result;
17 }
18 #endif
19 #endif
```

Figure 2: Example of an undisciplined annotation in xterm

A clean mapping from annotations to AST elements has the following advantages:

- The AST representation contains all information of a program including its variability. This simplifies tools for concern refactoring, concern management, and many others, because heuristics are not necessary when dealing with the source code and programmers can safely analyze and transform the AST. We do not need to preprocess the code first, transform it, and then revert the preprocessing step. For example, when we refactor code by extracting an annotated sequence of statements into an advice declaration, we can make sure that we move them correctly without loss of variability information.
- The mapping of `#ifdef` annotations to AST elements ensures the absence of syntax errors. When we allow only annotations on structural elements, but not on arbitrary tokens such as brackets, the removal of `#ifdef`s cannot introduce syntax errors [23].
- Based on an AST, we can reason about types or control flow or perform other static analyses and model checking [7], always including variability information. These semantic analyses become more complex due to annotations (for example, a variable can have different types depending on the selection of preprocessor constants), but it still can be performed on the entire variable code base in a single step, instead of generating all variants upfront. For example, in [20] we have built a type system that compares annotations between function declarations and function calls, which depends on variability annotations in a single AST.

2.3 Defining Disciplined Annotations

Based on the idea of mapping annotations to elements of the underlying source-code structure, we propose a definition of disciplined annotations. Actually, it is quite difficult to find a common definition, because different tools may have different requirements (e.g., some can handle annotated function parameters, others do not). Here, we put forward a conservative definition.

> **Definition Disciplined Annotations:** In C, annotations on one or a sequence of *entire functions* and *type definitions* (e.g., `struct`) are disciplined. Furthermore, annotations on one or a sequence of *entire statements* and annotations on *elements inside type definitions* are disciplined. *All other annotations are undisciplined.*

We chose this definition, because we can map functions, type definitions, and statements straightforwardly to subtrees in the AST. Furthermore, we can exploit this definition when extending an existing C grammar in Section 2.4. In Section 4.1, we discuss our definition of disciplined annotations based on an empirical evaluation of the annotation discipline in 40 software projects.

In Figure 4, we show some examples of disciplined annotations taken from the text editor *vim*: an annotation on an entire function (Fig. 4 a), an annotation on an entire statement including a nested substatement (Fig. 4 b), and an annotation on a field inside a struct (Fig. 4 c).

One may argue that our definition is too strict. We could also allow annotations at expression level (as in Fig. 5 d), on parameters (as in Fig. 5 g), on case blocks in a switch statement (as in Fig. 5 h), or on the else branch of an if statement (as in Fig. 5 b). In all these cases, we can map the annotation to subtrees of the AST. Actually, we can even map partial annotations on if, for, or while statements that do not include the nested body as in Figure 5 a. In these cases we would map the annotation to an individual AST element and not to an entire subtree (which we discussed as *wrappers* in [23]). All these fine-grained annotations (and several more) would be possible to interpret as disciplined, but then the tools that work on the resulting AST will be more complex. Some tools benefit from disallowing annotations on expressions or parameters, because this way they have to consider fewer annotated code fragments and thus fewer transformation patterns [22]. One goal of our analysis is to find out whether our conservative definition of disciplined annotations is sufficient in practice or whether some or all fine-grained annotations should be considered as disciplined as well, because software engineers use them frequently.

There are annotations that we can classify as undisciplined without any doubt. These are ill-formed annotations in which already the number of `#ifdef` and `#endif` statements does not match and, annotations that can produce syntax errors when removed (such as an annotation of an opening bracket without an annotation on the corresponding closing bracket). Figure 2 shows an example of an ill-formed annotation. (If the flags `__GLIBC__` and `USE_ISPTS_FLAG` are selected, the resulting code will contain an syntax error.)

2.4 Parsing Disciplined Annotations

An AST representation of unprocessed code requires a mapping of `#ifdef` annotations to the program structure. To this end, we need a parsing step that can at best parse the entire unprocessed source code (pre-cpp) in a single step. A straightforward approach is to introduce preprocessor directives into the grammar of the host language. Figure 3 shows an excerpt of a cpp-extended C grammar. The grammar supports optional or alternative function definitions with *cpp* directives (additional productions for annotating functions

highlighted). Note that, due to *cpp*'s unlimited annotation capabilities it is difficult to write a preprocessor-aware grammar that covers all possible annotations and considered impossible by some researchers (e.g., [29]). But when we enforce disciplined annotations, this approach becomes practical. When preprocessor directives are already part of the grammar, and hence recognized by the parser, we can assign parsed annotations directly to code fragments of the AST.

```
1  translation_unit
2    : external_declaration
3    | translation_unit external_declaration
4    ;
5  external_declaration
6    : function_definition
7    | '#' 'if' cppexp nl function_definition nl cppthenfunc
8    | declaration
9    ;
10 cppthenfunc
11   : '#' 'endif' nl
12   | '#' 'else' nl function_definition nl '#' 'endif' nl
13   | '#' 'elseif' cppexp nl function_definition nl cppthenfunc
14   ;
15 function_definition ...
```

Figure 3: Excerpt of an cpp-extended ISO/IEC 9899 lexical C grammar; rules for preprocessor directives are in Line 7 and Lines 10 to 14; `cppexp` is the condition; `nl` is a newline; `cppthenfunc` represents the `#endif` or alternative function definitions

3. EMPIRICAL STUDY

Next, we analyze how annotations are used in practice and whether enforcing disciplined annotations would be a feasible endeavor.

3.1 Hypothesis

Before we present and discuss the results of the analysis, we formulate our hypothesis regarding the discipline of preprocessor annotations. *We expect that the majority of* **`#ifdefs`** *used in C programs is disciplined.* We have three reasons for this hypothesis:

1. Developers prefer disciplined annotations and consider undisciplined annotations as hard to read. For example, Baxter and Mehlich report of a project that contained some undisciplined annotations: *"The reaction of most staff to this kind of trick is first, horror, and then second, to insist on removing the trick from the source"* [4].
2. Some software projects have *coding guidelines* that state how to use the preprocessor. They typically suggest disciplined over undisciplined annotations. For example, in Linux kernel development, guidelines state that programmers shall annotate entire functions instead of arbitrary source-code fragments: *"Code cluttered with ifdefs is difficult to read and maintain. Don't do it. Instead, put your ifdefs in a header, and conditionally define static inline functions, or macros, which are used in the code."* [3]
3. Disciplined annotations are sufficient for most problems in software development. Arbitrary undisciplined annotations are simply not necessary in most cases. Even though variability involves changes at subfunction level, it is questionable whether annotations at the level of expressions or parameters outweigh the problems they introduce.

Although these reasons are backed by anecdotal reports from practice, we are unaware of substantial empirical evidence. Therefore, we empirically analyze 40 C projects regarding the discipline of the preprocessor use to confirm or reject our hypothesis.

3.2 Sample Projects & Collecting Data

We selected 40 software projects to get a comprehensive overview of the discipline of preprocessor use. We had three criteria for the selection. First, the software projects must have a large developer base. Second, the selection must contain systems from different domains and of different sizes. Third, the primary programming language must be C. In Table 2, we provide information on the selected projects. Together these systems contain 30 255 220 lines of C code (normalized by pretty printing and eliminating empty lines and comments) and contain 330 017 annotations. Before we analyzed the software projects, we prepared the source code by eliminating include guards.[4]

Our analysis requires representing source code as an AST. We use the tool *src2srcml*[5] for this task. It parses the unprocessed C code and generates an XML document. The XML representation includes both plain C code and preprocessor directives. Based on this coherent representation, we visit all annotations and determine whether they align with functions, types, and statements, as defined above.[6] For each project, we classify the annotations and count the number of disciplined and undisciplined annotations. We omitted 71 of 104 020 files during the analysis (0.7 % of all files analyzed), because either *src2srcml* could not correctly parse these files or they contained incomplete annotations such as an `#ifdef` without the corresponding `#endif`. Without a proper AST representation, we were not able to classify the annotations in these files.[7]

To discuss whether our conservative definition of disciplined annotations is too strict or whether even a restriction to annotations on functions would be feasible, we took a closer look at the different kinds of disciplined and undisciplined annotations. Our analysis tool distinguishes between the following kinds of disciplined annotations:

FT: Annotations on one or multiple functions or type definitions (Fig. 4 a).

SF: Annotations on one or multiple statements inside a function or on fields inside a type definition (Fig. 4 b and 4 c).

As described in Section 2.3, there is a gray zone of annotations that we consider as undisciplined, but which could be mapped to AST elements with some effort. Hence, we

[3] see `/Documentation/SubmittingPatches` in the Linux source

[4] An include guard is a well-known pattern of conditional inclusion that does not represent any functional aspect. It annotates one or more function or type definitions, which is per definition disciplined. Including include guards in our statistics would bias the results toward disciplined annotations.

[5] `http://www.sdml.info/projects/srcml/`

[6] Note that we split alternative annotations such as `#ifdef-#else-#endif` into two annotations, one from `#ifdef` to `#else` and one from `#else` to `#endif`, because they enframe different code fragments and have to be analyzed separately.

[7] Our analysis tool and the comprehensive data is available at the project's website `http://fosd.de/cppstats/`.

```
1 #if defined(__MORPHOS__) && \
2     defined(__libnix__)
3 extern unsigned long *__stdfiledes;
4
5 static unsigned long
6     fdtofh(int filedescriptor) {
7   return __stdfiledes[filedescriptor];
8 }
9 #endif
```

(a) compilation unit

```
1 void tcl_end() {
2 #ifdef DYNAMIC_TCL
3   if (hTclLib) {
4     FreeLibrary(hTclLib);
5     hTclLib = NULL;
6   }
7 #endif
8 }
```

(b) sub-function level

```
1 typedef struct {
2   typebuf_T save_typebuf;
3   int typebuf_valid;
4   struct buffheader save_stuffbuff;
5 #if USE_INPUT_BUF
6   char_u *save_inputbuf;
7 #endif
8 } tasave_T;
```

(c) sub-type level

Figure 4: Examples of disciplined annotations in *vim*

```
1 #ifdef RISCOS
2   if ((s = vim_strrchr(result, '/'))
3       != NULL && s >= gettail(result))
4 #else
5   if ((s = vim_strrchr(result, '.'))
6       != NULL && s >= gettail(result))
7 #endif
```

(a) if

```
1 #ifdef FEAT_FIND_ID
2   else if (*e_cpt == 'i')
3     type = CTRL_X_PATH_PATTERNS;
4   else if (*e_cpt == 'd')
5     type = CTRL_X_PATH_DEFINES;
6 #endif
```

(b) else-if block

```
1   int n = NUM2INT(num);
2 #ifndef FEAT_WINDOWS
3   w = curwin;
4 #else
5   for (w = firstwin; w != NULL;
6       w = w->w_next, --n)
7 #endif
8     if (n == 0)
9       return window_new(w);
```

(c) for wrapper

```
1   if (char2cells(c) == 1
2 #if defined(FEAT_CRYPT) || \
3     defined(FEAT_EVAL)
4       && cmdline == 0
5 #endif
6   )
```

(d) expression

```
1   if (!ruby_initialized) {
2 #ifdef DYNAMIC_RUBY
3     if (ruby_enabled(TRUE)) {
4 #endif
5       ruby_init();
```

(e) ill-formed annotation

```
1 #if defined(FEAT_SEARCHPATH) || \
2     defined(FEAT_BROWSE)
3 theend:
4   vim_free(fname);
5 #endif
```

(f) goto

```
1   need_redraw =
2       check_timestamps(
3 #ifdef FEAT_GUI
4       gui.in_use
5 #else
6       FALSE
7 #endif
8   );
```

(g) parameter

```
1 #ifdef FEAT_CLIENTSERVER
2   case SPEC_CLIENT:
3     sprintf((char *)strbuf,
4       PRINTF_HEX_LONG_U,
5     (long_u)clientWindow);
6   result = strbuf;
7   break;
8 #endif
```

(h) case block

```
1     for ( ; mp != NULL;
2 #ifdef FEAT_LOCALMAP
3         mp->m_next == NULL ?
4         (mp = mp2, mp2 = NULL) :
5 #endif
6         (mp = mp->m_next)) {
```

(i) expression

```
1  int put_eol(fd)
2      FILE *fd;
3  {
4    if (
5  #ifdef USE_CRNL
6      (
7  #ifdef MKSESSION_NL
8        !mksession_nl &&
9  #endif
10       (putc('\r', fc) < 0)) ||
11 #endif
12       (putc('\n', fd) < 0))
13     return FAIL;
14   return OK;
15 }
```

(j) nested `#ifdefs`

```
1 #ifdef FEAT_MBYTE
2   int props;
3   p_encoding = enc_skip(p_enc);
4   props = enc_canon_props(p_encoding);
5   if (!(props & ENC_8BIT)
6       || !prt_find_resource((char *)p_encoding,
7       &res_encoding))
8 #endif
9 {
```

(k) combination of sub-function and if

Figure 5: Examples of undisciplined annotations in *vim*

search for certain patterns of undisciplined annotations, such as annotations on parameters, case blocks, and expressions, and determine how often they occur. If these patterns occur frequently, we may consider defining them as disciplined as well. The patterns we consider are the following:[8]

IF: Partial annotations of an if statement, e.g., an annotation of the if condition or the if-then branch without the corresponding else branch (Fig. 5 a).

CA: Annotations on a case statement in which only a case block is annotated (Fig. 5 h).

EI: Annotations on an else-if branch inside an if-then-else cascade (Fig. 5 b).

PA: Annotations on a parameter of a function declaration or a function call (Fig. 5 g).

EX: Annotations on well-formed parts of expressions (Fig. 5 d, Fig. 5 i, and Fig. 5 j).

3.3 Results

In Table 2, we list the results of our analysis: we present the number of occurrences of disciplined and undisciplined preprocessor uses for each of the 40 projects. The key result is that 84.4 ± 6.4 % of all annotations are in a disciplined form. Disciplined preprocessor use in the software projects ranges from 69.7 % in *subversion* to 100.0 % in *mpsolve*. Except for *mpsolve*, we always found some undisciplined annotations.

Looking closer at the disciplined annotations, we found that 27.1 ± 11.9 % of all annotations wrap entire functions and type definitions and 57.2 ± 11.2 % wrap statements or fields.

The most common pattern of undisciplined annotations is annotations on case blocks (CA; 4.1 ± 3.3 %). Except for *irssi* and *mpsolve*, they occur in every project. Partial annotations on if statements (IF; 2.4 ± 2.0 %) are less frequent, but still occur several times in every project except *lighttpd* and *mpsolve*. Annotations on else-if (EI; 0.3 ± 0.4 %), parameters (PA; 0.3 ± 0.4 %), and expressions (EX; 0.7 ± 1.0 %) occur infrequently and only in some projects. There are only few projects with an exceptionally high number of occurrences of such patterns (up to 5.5 %), for example, *gcc*, *sendmail*, or *vim*.

We were not able to identify the remaining annotations automatically (7.7 ± 4.9 %; range from 0.0 to 28.2 %). Among those are ill-formed annotations and infrequent patterns (or combinations of identified patterns), such as in Figure 5 c, 5 e, 5 f, and 5 k.

To have a closer look at the unclassified annotations, we manually analyzed all unclassified annotations in 10 % of all projects.[9] In Table 1, we show the number of unclassified annotations that we identified as disciplined, undisciplined, and ill-formed. The reason for *src2srcml*'s inability to identify these few patterns is that it is based on heuristics. We discuss this limitation as a threat to validity in Section 4.3. However, the enormous size of the data set (Table 2) amortizes the influence of a few false negatives (i.e., not as disciplined classified annotations) in unclassified patterns.

name	disciplined	undisciplined	ill-formed
cherokee	4	20	6
irssi	0	11	0
lighttpd	1	21	0
xterm	0	39	8

Table 1: Results of a manual inspection of unclassified annotations in four projects

4. INTERPRETATION AND DISCUSSION

The results raise a number of questions. If 84 % of all annotations are disciplined, how do we handle the remaining 16 %? Should we include further annotation patterns to be disciplined or should we transform undisciplined into disciplined annotations?

4.1 Toward a Common Definition of Disciplined Annotations

Transforming undisciplined annotations into disciplined annotations requires a certain effort. Disciplined annotations are most useful, when the community can agree on a common definition to establish a solid foundation for the development of inter-operable tools.

In Section 2.3, we proposed a conservative definition of disciplined annotations that is easy to reason about for tools. We already noted that several other annotation patterns could be regarded as disciplined at the expense of more complex tool implementation (for all tools), but at the benefit that less effort is necessary to transform legacy code into disciplined annotations. Here, we come back to this issue and initiate a discussion about the suitability of our definition.

First, our results show that, with our conservative definition, 84 % of all annotations are disciplined. Making the definition stricter is not feasible, because this way we can cover less annotations without any further benefit. For example, when considering only annotations on function and type definitions disciplined, we can cover only 27 % of all annotations; annotations on statements and fields would be considered undisciplined, even though they are easy to handle by tools.

Second, the most common pattern of undisciplined annotations that we recognized were annotations on case blocks inside switch statements as in Figure 5 h (4.1 ± 3.3 %). Such annotations occur in every sample project except *mpsolve* and *irssi*. They can easily be mapped to AST subtrees, but (similar to or even worse than if-else chains) they are surprisingly difficult to handle due to the complicated control flow (in particular, in the presence of break statements).

Third, the next common pattern of undisciplined annotations (2.4 ± 2.0 %) are partial annotations of if statements, in which only parts of an if statement are annotated (e.g., only the condition or the if-then branch without the alternative else) as in Figure 5 a. Although this pattern occurs quite frequently in some projects (and at least once in every project except for *mpsolve* and *lighttpd*), handling such annotations is difficult, since we cannot map the annotations to an entire AST subtree, but only to an individual AST

[8] The identification of patterns that might be considered disciplined was an iterative process. We started with patterns we were familiar with and iteratively added additional patterns that we found during manual inspection of undisciplined annotations in the analysis results.

[9] Due to the large number of remaining unclassified annotations in larger software projects, such as freebsd or opensolaris, we selected some smaller projects to get an intuition of the kind of unclassified annotations. In terms of lines of code, the selected projects cover 1 % of all systems.

element without its children.[10] Aiming at an inter-operable infrastructure for many tools, we suggest not regarding such annotations as disciplined, but to refactor the source code (Sec. 4.2).

Fourth, the remaining identified patterns of annotated if-then-else chains ($0.3 \pm 0.4\,\%$), parameters ($0.3 \pm 0.4\,\%$), and expressions ($0.7 \pm 1.0\,\%$) occur infrequently (and not in all projects). Due to the rare occurrences of such patterns we consider refactoring the source code as the better and more inter-operable solution.

Finally, there are several annotations that do not fit into either of our patterns ($7.7 \pm 4.9\,\%$). The individual pattern (or combination of patterns) behind these annotations occurs so infrequently that we can safely discard them as undisciplined. Overall, we interpret the results of our analysis as a confirmation of our initial conservative definition of disciplined annotations.

4.2 Handling the Remaining 16 Percent

Tools aiming at disciplined annotations are able to handle most `#ifdef`s (84 %). This may be sufficient for simple analyses such as the measurement of source-code complexity metrics or roughly estimating the potential for refactorings as done by Adams et al. (their tool simply ignored all annotations it did not understand) [1]. However, for some tools a single undisciplined annotation may render the tool unsafe or useless; for example, concern refactoring tools may fail or even produce incorrect results, or concern management tools may show inconsistent views, because they could not parse certain files or code fragments. Taking into account that, except for one project, all projects contain at least a few undisciplined annotations, this is a serious issue that deserves attention.

For handling the remaining 16 percent, we see two possibilities. First, we can introduce more sophisticated tools that use heuristics to accept more annotations than we currently classify as undisciplined. Still, since arbitrary undisciplined annotations may occur in a software system, the effort to write such tools is extraordinary. Two examples of such tools are Garrido's refactoring tool *CRefactory* [12] or Padioleau's preprocessor-aware parser *Yacfe* [29]. But, because both tools use heuristics, the outcome of these tools may not be 100 % correct.

Second, we can enforce disciplined annotations and require the developer to transform all remaining undisciplined annotations into disciplined annotations. To make this approach practical, tool support is necessary for the automation of the transformation task. Enforcing disciplined annotations simplifies the development of tools significantly and hence can foster a community of tool developers for pre-cpp code. We discuss the transformation of undisciplined annotations and the effort of such transformation next.

Transformation of Undisciplined Annotations. To transform undisciplined annotations into disciplined annotations, `#ifdef`s need to be expanded until they wrap entire function or type definitions, statements, or fields. Except for ill-formed annotations, undisciplined annotations can always be expanded into disciplined annotations. A brute-force algorithm that works in every case is to replicate the source code for every possible combination of `#ifdef`s and to annotate the entire replicated code fragment.[11] For example, Figure 6 a shows an example of an undisciplined parameter annotation with two possible variants. We can replicate the code fragment (one version in which FEAT_GUI is defined and one version in which it is not defined) and annotate the entire statement in a disciplined form, as shown in Figure 6 c. In the worst case, we would have to replicate the entire file multiple times and annotate the file's content. However, in most cases, more sophisticated expansions at the level of statements or functions are appropriate.

However, an automatic expansion is not always that easy because of nested `#ifdef`s and scattered annotations. First, we found some annotations that were ill-formed and that do not map to nodes or subtrees in the AST or even a parse tree (Fig. 2). We argue that a programmer may expand these annotations manually after a tool automatically identifies them as undisciplined. Second, consider the nested `#ifdef` example in Figure 5 j. The `#ifdef` MKSESSION_NL is embedded in the `#ifdef` USE_CRNL, and the preprocessor evaluates it only in case USE_CRNL evaluates to true. The expansion of this nesting leads to three alternatives: without USE_CRNL and MKSESSION_NL, with USE_CRNL and without MKSESSION_NL, and with both. This example may be simple, but it has been shown that nesting occurs frequently and a nesting depth beyond 2 is quite common [26]. Expanding nested `#ifdef`s may lead to an exponential number of code clones as a result of the combinatorial explosion of all annotations involved. It is questionable whether it is feasible to expand deeply nested `#ifdef`s in favor of source-code refactoring. However, because we limit the expansion to statements and functions, we believe that the expansion approach does not lead to combinatorial explosion and the expanded output is manageable by concern management tools.

Although a brute-force expansion is possible, developers can often write more elegant annotations manually. For example, for annotations on parameters or fragments of expressions as in Figures 5 d, 5 i, 5 j, and 6 a, we would typically rather introduce a variable and one or more annotated assignment statements, as illustrated in Figure 6 b. Hence, we recommend a semi-automatic process instead of a fully automatic transformation. A tool locates undisciplined annotations and proposes expansions, but the developer can provide better implementations. More examples of automated and manual expansions of undisciplined `#ifdef` annotations (as in Fig. 5) are available at the project's website `http://fosd.de/cppstats/`.

Expressiveness, Replication, and Readability. There is a *trade-off between expressiveness, code replication and readability.* Expressiveness denotes the ability and flexibility to make changes to the source code. An expressive preprocessor is able to handle fine-grained annotations and thus minimizes code replication. However, expressive preprocessors such as *cpp* are criticized for making code difficult to read and understand [4]. Limiting the expressiveness of the preprocessor, for example, to disciplined annotations, requires a certain amount of code replication, as we have explained.

[10] Partial annotations complicate tool support, because depending on the evaluation of `#ifdef` conditions the child element belongs to two different AST elements. For example, the if statement (Line 8) in Figure 5 c belongs either to the for loop (Line 5) or the function (not printed there) directly.

[11] Note that the use of `#include`, `#define`, and `#undef` macros does not hinder the expansion, because we can replicate these macros as well.

```
1   need_redraw =
2       check_timestamps(
3 #ifdef FEAT_GUI
4       gui.in_use
5 #else
6       FALSE
7 #endif
8   );
```

(a)

```
1   int visible = FALSE;
2 #ifdef FEAT_GUI
3   visible = gui.in_use;
4 #endif
5   need_redraw =
6       check_timestamps(visible);
```

(b)

```
1 #ifdef FEAT_GUI
2   need_redraw = check_timestamps(gui.in_use);
3 #else
4   need_redraw = check_timestamps(FALSE);
5 #endif
```

(c)

Figure 6: Transformation of an undisciplined parameter annotation (a); possible disciplined annotation provided by a programmer (b); automatic transformation by expansion (c)

Furthermore, it improves readability in that annotations are only mixed with the base code at a predefined level.

Code replication is related to code cloning. Although some researchers consider code clones harmful (e.g., [18, 33]), we and others [19] argue that clones, as a result of expansion, are manageable. One reason is that the alternatives containing the replicated code are side-by-side and therefore easy to track for a programmer. Another reason is that expanded code is limited to function or type definitions and statements, which we consider as disciplined annotations. Note that different patterns of undisciplined annotations may cause different amounts of code replication, which is interesting to explore in further work.

4.3 Threats to Validity

Next, we describe two threats to validity that are crucial for our empirical analysis: threats to internal validity (relation of the observed output to the input) and threats to external validity (generalization of findings).

Threats to Internal Validity. The selection of sample software projects is crucial for empirical analyses, because a biased selection leads to biased results. To control this confounding variable, we analyzed a large number of software projects of different domains and different sizes.

Not all annotation patterns have a functional aspect. In this work we omit the include-guard pattern. An include guard prevents the multiple inclusion of a header file during the compilation process and is per se disciplined, as it annotates the whole content of a header file. Including include guards in our statistics would bias our results towards disciplined annotations, so we excluded them.

The tool *src2srcml*, which we use for our analysis, is partially based on heuristics that are used to infer an abstract syntax tree from unpreprocessed source code. From our experiences, *src2srcml*'s heuristic approach fails from time to time, especially when it comes to ill-formed annotations. That is, it may classify a few annotations incorrectly. Note that we face an instance of the chicken-or-the-egg problem here! For a completely correct classification, we would have to use a tool that can handle all kinds of annotations, which is not possible, as explained before. Alternatively, we could use only disciplined annotations, which is exactly the view we want to support with our analysis.

Threats to External Validity. Using a large sample size we believe that our results are representative for the preprocessor usage in C software projects. We cannot generalize our results to other programming languages that also make use of the preprocessor, such as C++. This is because C++, for instance, provides additional capabilities for expressing variable source code (e.g., template metaprogramming) and, to this end, programmers may use *cpp* differently.

5. RELATED WORK

Our discussion of disciplined annotations is influenced by prior work on preprocessor-aware tool support – especially from preprocessor-aware refactorings – and analyses of preprocessor usage. Two *cpp*-aware refactoring tools for C are *Xrefactory* [38] and *CRefactory* [13, 14]. Internally, both tools automatically expand macros (using textual substitutions) and undisciplined annotations to disciplined annotations. *Xrefactory* uses a brute-force expansion at the level of files (which leads to massive replication), whereas *CRefactory* uses a very sophisticated expansion mechanism at finer granularity, but is tailored to refactoring and also based on heuristics.

Regarding refactorings from `#ifdef`-based code to aspects, prior work was mostly conceptual or manual, because of the complexity of parsing pre-cpp code. For example, Adams et al. analyzed the feasibility of refactoring, but did not actually execute refactorings [1]. Lohmann et al. refactored `#ifdef` annotations in an operating systems kernel, but did not automate the refactoring [27]. In prior work, we automated and formalized refactorings from annotated Java code to mixin-style feature modules (which can be adapted to AspectJ as a target language as well), but strictly relied on disciplined annotations [22].

Baxter and Mehlich proposed *DMS*, a source-code transformation system for C and C++ [4]. The authors use *cpp*-aware grammars that are able to capture a subset of possible `#ifdef` annotations for both languages (as discussed in Sec. 2.4). To resolve interactions of macro substitution and file inclusion with conditional compilation, the authors use specialized heuristics in *DMS*. Baxter and Mehlich provide anecdotal evidence that their tool can parse 85 % of industrial pre-cpp code and that rewriting undisciplined annotations in a 50 000 lines of code project can be done in "an afternoon". Our empirical analysis of a wide range of software projects confirms their findings.

The problems with arbitrary preprocessor transformation inspired other researchers to create their own disciplined macro language [9, 25, 28, 39]. Erwig and Walkingshaw proposed *choice calculus*, a common language for software variation management [9]. The theoretical foundation of *choice calculus* is similar to our definition of disciplined annotations; both approaches allow only the variation of subelements in AST-like structures. Considering macro expansion, there has been significant effort to introduce Lisp-style syntax macros that operate on ASTs instead of relying on token substitution. Syntax macros are a disciplined form of macros and are complementary to disciplined annotations for conditional inclusion. In this context, especially *ASTEC* [28] is interesting, because it also covers both syntax macros and

conditional inclusion in a disciplined form (it allows annotations only on declarations, statements, and expressions).

The use of *ASTEC* requires a one-time transformation of all *cpp* directives into the *ASTEC* macro language. The authors evaluated their approach on four different programs with the result that most `#ifdef` annotations could be transformed automatically. Our work differs from *ASTEC* and other syntax-macro systems in that we stick with the lexical preprocessor *cpp* (we only restrict its use) to avoid changing or adapting existing tool support. Additionally, our analysis gives a more comprehensive overview of `#ifdef` annotations, because we analyzed a substantial code base. Unclassified, undisciplined annotations make up at max 5.8 % during the evaluation of *ASTEC*. We found that some software projects contain up to 28.2 % unclassified, undisciplined annotations (for example, the Linux kernel used for the *ASTEC* evaluation seems to be a favorable case compared to the other systems in our analysis, presumably due to its coding guidelines mentioned in Sec. 3.1).[12] Overall, our work confirms the findings in *ASTEC* and we strongly agree that disciplined annotations are practical for implementing variable source code.

Some programming languages (e.g., Java, D, and Visual Basic) have a "static if" in their syntax that is evaluated during compiler invocation. The "static if" is limited to entire statements and therefore represents a small but strict subset of disciplined annotations according to our definition.

Finally, numerous analyses of preprocessor usage have been conducted by several researchers (e.g., [8, 26]). For example, Ernst et al. analyzed the preprocessor usage covering mainly macro definitions using `#define` directives [8]. While `#define` directives are also important for tool support, conclusions based on their usage give only a limited view as `#ifdef` usage also effects tool support. Our findings complement the understanding of preprocessor usage for tool support.

In prior work, we have analyzed the preprocessor usage from the perspective of the software engineer [26]. We looked at how software engineers implement variability using the preprocessor (e.g., where do `#ifdef` annotations occur in the code and how complex are those annotations). In contrast, in this work, we aim at tool support for source code that contains preprocessor annotations and discuss the notion of disciplined annotations, which we did not consider earlier [26].

6. CONCLUSION

We have empirically analyzed the discipline of *cpp* preprocessor annotations in 40 software projects with 30 million lines of code. We found that 84 % of `#ifdef` of all annotations are disciplined – they respect the underlying source-code structure by annotating only entire function or type definitions, statements, and fields. It seems that software engineers are aware of the problems of undisciplined annotations and deliberately limit their use.

We demonstrated that disciplined annotations bear the potential to significantly improve the situation for tool developers that aim at unprocessed source code. This way tools for concern refactoring, concern management, as well as variability-aware parsers, refactoring engines, static analyzers, model checkers, and type checkers become feasible in the first place for a vast amount of industrial C code. Since undisciplined annotations occur in nearly all analyzed programs, we propose to transform them toward disciplined annotations by means of an semi-automatic transformation. In this context, we identified a fundamental trade-off between expressiveness, replication, and readability. We argue that, to take advantage of disciplined annotations, it is feasible to accept certain kinds of and a certain amount of code replication in favor of a better tool support and a better readability of code.

In ongoing work, we develop a tool infrastructure to guide the semi-automatic transformation process from undisciplined to disciplined annotations.

[12] We assume that the authors classification of imperfect `#ifdef` annotation is comparable to our classification of unclassified annotations.

Acknowledgments

We thank Andy Kenner and Christopher Resch for fruitful discussions on (un)disciplined annotations. Furthermore, we thank Christian Lengauer for helpful comments on earlier drafts of this paper. Apel's work is supported by the DFG projects #AP 206/2-1 and #AP 206/4-1. Kästner's work is supported in part by ERC (#203099).

7. REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. Hassan. Can we Refactor Conditional Compilation into Aspects? In *Proc. of the Int'l Conf. on Aspect-Oriented Software Development (AOSD)*, pages 243–254. ACM Press, 2009.

[2] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Trans. on Softw. Eng. (TSE)*, 28(7):625–637, 2002.

[3] L. Aversano, L. Di Penta, and I. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proc. of the Int'l Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 83–92. IEEE CS, 2002.

[4] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. of the Working Conf. on Reverse Engineering (WCRE)*, pages 281–290. IEEE CS, 2001.

[5] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple Crosscutting Concerns Are Not So Simple: Analysing Variability in Large-Scale Idioms-Based Implementations. In *Proc. of the Int'l Conf. on Aspect-Oriented Software Development (AOSD)*, pages 199–211. ACM Press, 2007.

[6] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating Idiomatic Crosscutting Concerns. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, pages 37–46. IEEE CS, 2005.

[7] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pages 335–344. ACM Press, 2010.

[8] M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. on Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.

[9] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Softw. Eng. and Meth. (TOSEM)*, 2010. to appear.

[10] J.-M. Favre. The CPP Paradox. In *Proc. of the Europ. Workshop on Software Maintenance*, 1995. `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.9464`.

[11] J.-M. Favre. Understanding-In-The-Large. In *Proc. of the Int'l Workshop on Program Comprehension (IWPC)*, pages 29–38. IEEE CS, 1997.

[12] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois, 2005.

[13] A. Garrido and R. Johnson. Refactoring C with Conditional Compilation. In *Proc. of the Int'l Conf. on Automated Software Engineering (ASE)*, pages 323–326. IEEE CS, 2003.

[14] A. Garrido and R. Johnson. Analyzing Multiple Configurations of a C Program. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, pages 379–388. IEEE CS, 2005.

[15] W. Griswold, J. Yuan, and Y. Kato. Exploiting the Map Metaphor in a Tool for Software Evolution. In *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pages 265–274. IEEE CS, 2001.

[16] F. Heidenreich, I. Savga, and C. Wende. On Controlled Visualizations in Software Product Line Engineering. In *Proc. of the SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, pages 303–313. Lero International Science Centre, 2008.

[17] D. Janzen and K. De Volder. Programming with Crosscutting Effective Views. In *Proc. of the Europ. Conf. on Object-Oriented Programming (ECOOP)*, pages 195–218. Springer-Verlag, 2004.

[18] E. Jürgens, F. Deissenböck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pages 485–495. IEEE CS, 2009.

[19] C. Kapser and M. Godfrey. "Cloning Considered Harmful" Considered Harmful. *Empirical Software Engineering*, 13(6):645–692, 2008.

[20] C. Kästner and S. Apel. Type-Checking Software Product Lines – A Formal Approach. In *Proc. of the Int'l Conf. on Automated Software Engineering (ASE)*, pages 258–267. IEEE CS, 2008.

[21] C. Kästner and S. Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, 2009.

[22] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. of the Int'l Conf. on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009.

[23] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. of the Int'l Conf. Objects, Models, Components, Patterns (TOOLS Europe)*, pages 174–194. Springer-Verlag, 2009.

[24] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[25] B. Leavenworth. Syntax Macros and Extented Translation. *Communications of the ACM (CACM)*, 9(11):790–793, 1966.

[26] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pages 105–114. ACM Press, 2010.

[27] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proc. of the EuroSys Conf.*, pages 191–204. ACM Press, 2006.

[28] B. McCloskey and E. Brewer. ASTEC: A New Approach to Refactoring C. In *Proc. of the Europ. Software Engineering Conf. and of the Int'l Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 21–30. ACM Press, 2005.

[29] Y. Padioleau. Parsing C/C++ Code without Pre-processing. In *Proc. of the Int'l Conf. on Compiler Construction (CC)*, pages 109–125. Springer-Verlag, 2009.

[30] T. Pearse and P. Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, pages 270–277. IEEE CS, 1997.

[31] H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proc. of the Int'l Conf. on Automated Software Engineering (ASE)*, pages 347–350. IEEE CS, 2008.

[32] A. Reynolds, M. Fiuczynski, and R. Grimm. On the Feasibility of an AOSD Approach to Linux Kernel Extensions. In *Proc. of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 1–7. ACM Press, 2008.

[33] C. Roy and J. Coardy. A Survey on Software Clone Detection Research. Technical Report 2007-541, Queen's University at Kingston, 2007.

[34] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A Tool for Navigating Highly Configurable System Software. In *Proc. of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, page 9. ACM Press, 2007.

[35] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *Proc. of the USENIX Technical Conf.*, pages 185–197. USENIX Association Berkeley, 1992.

[36] D. Spinellis. Global Analysis and Transformations in Preprocessed Languages. *IEEE Trans. on Softw. Eng. (TSE)*, 29(11):1019–1030, 2003.

[37] L. Vidács, A. Beszédes, and T. Gyimóthy. Combining Preprocessor Slicing with C/C++ Language Slicing. *Science of Computer Programming (SCP)*, 74(7):399–413, 2009.

[38] M. Vittek. Refactoring Browser with Preprocessor. In *Proc. of the Europ. Conf. on Software Maintenance and Reengineering (CSMR)*, pages 101–110. IEEE CS, 2003.

[39] D. Weise and R. Crew. Programmable Syntax Macros. In *Proc. of the Int'l Conf. on Programming Language Design and Implementation (PLDI)*, pages 156–165. ACM Press, 1993.

name	version	domain	# LOC	# AA	disciplined		undisciplined						
					% FT	% SF	% IF	% CA	% EI	% PA	% EX	% NC	% TD % TU % NC
apache[1]	2.2.11	Web server	212 493	3 543	17.6	62.9	2.2	5.0	0.7	0.2	0.7	10.7	
berkeley db[1]	4.7.25	database system	187 673	3 167	13.7	74.4	0.8	4.1	0.0	0.3	0.6	6.3	
cherokee[1]	0.99.11	Web server	51 958	659	25.2	52.2	6.4	11.2	0.3	0.0	0.2	4.6	
clamav[1]	0.94.2	antivirus program	75 486	1 022	20.7	61.8	3.2	2.5	0.1	0.0	0.0	11.6	
dia[1]	0.96.1	diagramming software	129 318	447	17.2	70.9	0.5	1.1	0.0	1.1	0.0	9.2	
emacs[1]	22.3	text editor	237 044	5 556	26.0	61.4	2.8	2.2	0.6	0.3	1.8	4.9	
freebsd[1]	7.1	operating system	5 935 445	76 093	30.0	54.7	2.2	6.3	0.4	0.4	0.8	5.4	
gcc[1]	4.3.3	compiler framework	1 639 545	14 577	34.5	47.8	1.8	2.3	0.3	0.2	2.4	10.9	
ghostscript[1]	8.62.0	postscript interpreter	443 429	2 973	36.6	52.9	1.0	2.9	0.2	0.3	0.3	5.9	
gimp[1]	2.6.4	graphics editor	590 722	1 551	33.8	56.7	0.8	2.5	0.2	0.1	0.0	5.9	
glibc[1]	2.9	programming library	747 415	11 836	40.0	44.3	2.7	5.0	0.2	0.2	0.6	11.1	
gnumeric[1]	1.9.5	spreadsheet application	256 311	1 697	12.0	71.7	2.4	1.2	0.4	0.8	0.5	11.1	
gnuplot[1]	4.2.5	plotting tool	76 185	1 951	30.2	48.9	4.0	8.1	0.4	0.1	0.9	7.5	
irssi[1]	0.8.13	IRC client	49 833	150	15.3	70.7	6.7	0.0	0.0	0.0	0.0	7.3	
libxml2[1]	2.7.3	XML library	210 901	9 250	68.9	24.6	0.9	2.3	0.7	0.0	0.3	2.4	
lighttpd[1]	1.4.22	Web server	39 037	666	16.5	74.8	0.0	5.3	0.0	0.0	0.2	3.3	
linux[1]	2.6.28.7	operating system	5 977 732	43 259	36.4	55.9	1.1	3.5	0.2	0.2	0.2	2.6	
lynx[1]	2.8.6	Web browser	117 701	3 593	17.7	61.8	5.4	3.9	0.3	0.2	1.8	9.1	
minix[1]	3.1.1	operating system	64 235	879	33.0	59.5	0.7	1.7	0.1	0.0	0.1	4.9	
mplayer[1]	1.0rc2	media player	600 428	6 230	22.4	60.4	2.7	5.2	0.1	0.4	0.3	8.5	
mpsolve[2]	2.2	mathematical software	10 191	30	53.3	46.7	0.0	0.0	0.0	0.0	0.0	0.0	
openldap[1]	2.4.16	LDAP directory service	246 385	2 729	22.4	65.6	2.1	5.0	0.0	0.2	0.9	3.9	
opensolaris[3]	(2009-05-08)	operating system	8 224 152	77 832	19.7	56.0	1.6	3.4	0.2	0.3	0.3	18.4	
openvpn[1]	2.0.9	seftrity application	38 568	905	29.6	62.9	1.7	0.6	0.7	1.4	0.6	2.7	
parrot[1]	0.9.1	virtual machine	103 790	1 520	39.9	50.1	0.5	3.2	0.1	0.0	0.1	6.2	
php[1]	5.2.8	program interpreter	573 469	7 390	28.8	54.1	2.2	7.6	0.1	0.3	0.5	6.4	
pidgin[1]	2.4.0	instant messenger	269 146	1 871	17.7	68.4	0.8	0.9	0.1	1.5	0.5	10.1	
postgresql[1]	(2009-05-08)	database system	453 890	2 918	22.5	57.5	1.6	5.4	0.4	0.3	0.6	11.7	
privoxy[1]	3.0.12	proxy server	24 037	663	20.5	53.7	4.5	8.6	0.0	0.0	1.1	11.6	
python[1]	2.6.1	program interpreter	374 117	8 650	12.6	76.8	0.8	2.8	0.6	0.8	0.3	5.3	
sendmail[1]	8.14.2	mail transfer agent	83 644	2 729	18.9	55.8	3.0	12.5	1.4	0.4	2.1	5.8	
sqlite[1]	3.6.10	database system	94 418	1 459	31.4	56.8	2.8	3.4	0.1	0.1	0.4	5.1	
subversion[1]	1.5.1	revision control system	509 193	2 815	30.3	39.4	1.7	0.1	0.0	0.1	0.1	28.2	
sylpheed[1]	2.6.0	e-mail client	101 418	899	28.1	55.2	1.2	1.2	0.6	0.9	0.4	12.4	
tcl[1]	8.5.7	program interpreter	135 079	2 704	52.6	28.3	1.0	14.4	0.0	0.1	0.6	3.1	
vim[1]	7.2	text editor	225 807	11 529	14.3	58.3	7.9	2.8	1.4	0.8	5.5	9.1	
xfig[1]	3.2.5	vector graphics editor	72 583	366	26.8	53.8	7.4	2.7	1.1	0.3	1.4	6.6	
xine-lib[1]	1.1.16.2	media library	494 543	4 798	23.5	61.7	1.2	3.3	0.1	0.3	0.2	9.8	
xorg-server[4]	1.5.1	X server	528 313	7 342	24.6	59.4	3.0	5.1	0.4	0.2	1.1	6.3	
xterm[1]	2.4.3	terminal emulator	49 586	1 769	23.1	63.5	4.4	4.9	0.2	0.2	1.0	2.7	
avg ± stdev					84.4±6.4%		7.9±4.7%					7.7±4.9%	

[1] http://freshmeat.net/ [2] http://www.dm.unipi.it/cluster-pages/mpsolve/ [3] http://opensolaris.org/os/ [4] http://x.org/

LOC: lines of code (normalized by pretty printing and eliminating empty lines and comments); **AA**: all annotations; **TD**: disciplined annotations; **FT**: function and type definitions; **SF**: statement or field; **TU**: undisciplined annotations; **IF**: `IF` wrapper; **CA**: conditional case; **EI**: conditional `else if`; **PA**: parameter; **EX**: expression; **NC**: not classified;

Table 2: Data obtained from the analysis