HotComments: How to Make Program Comments More Useful?

Lin Tan, Ding Yuan and Yuanyuan Zhou
Department of Computer Science, University of Illinois at Urbana-Champaign
{*lintan2, dyuan3, yyzhou*}*@cs.uiuc.edu*

Abstract

Program comments have long been used as a common practice for improving inter-programmer communication and code readability, by explicitly specifying programmers' intentions and assumptions. Unfortunately, comments are not used to their maximum potential, as since most comments are written in natural language, it is very difficult to automatically analyze them. Furthermore, unlike source code, comments cannot be tested. As a result, incorrect or obsolete comments can mislead programmers and introduce new bugs later.

This position paper takes an initiative to investigate how to explore comments beyond their current usage. Specifically, we study the *feasibility* and *benefits* of automatically analyzing comments to detect software bugs and bad comments. Our feasibility and benefit analysis is conducted from three aspects using Linux as a demonstration case. First, we study comments' characteristics and found that a significant percentage of comments are about "hot topics" such as synchronization and memory allocation, indicating that the comment analysis may first focus on hot topics instead of trying to "understand" any arbitrary comments. Second, we conduct a preliminary analysis that uses heuristics (i.e. keyword searches) with the assistance of natural language processing techniques to extract information from lock-related comments and then check against source code for inconsistencies. Our preliminary method has found 12 new bugs in the latest version of Linux with 2 already confirmed by the Linux Kernel developers. Third, we examine several open source bug databases and find that bad or inconsistent comments have introduced bugs, indicating the importance of maintaining comments and detecting inconsistent comments.

1 Introduction

1.1 Motivation

Despite costly efforts to improve software-development methodologies, software bugs in deployed code continue to thrive, contributing to a significant percentage of system failures and security vulnerabilities. Many software bugs are caused by *miscommunication* among programmers, misunderstanding of software components, and careless programming. For example, one programmer who implements function `Foo()` may assume that the caller of `Foo` holds a lock or allocates a buffer before calling `Foo`. However, if such assumptions are not clearly specified, other programmers can easily violate them, introducing bugs.

The problem above is further worsened by software evolution and growth. Typically, industrial and open source software are written by numerous developers over long periods of time, e.g. more than 10 years, with programmers frequently joining and departing the software development process. As a result, miscommunication and misunderstanding become increasingly severe, significantly affecting software quality and productivity.

To address the problem, comments have been used as a standard practice in software development to increase the readability of code by expressing programmers' intentions in a more *direct, explicit, and easy-to-understand*, but less precise (i.e. ambiguous) way. Comments are written in natural language to explain code segments, to specify assumptions, to record reminders, etc., that are often not expressed explicitly in source code. For example, the function `do_acct_process()` in Linux Kernel 2.6.20 assumes that it is only called from `do_exit()`; otherwise it may lead to failure. Fortunately, this assumption is stated in the source code comments, so other programmers are less likely to violate this assumption. Similarly, the comment above function `reset_hardware()` states that the caller must hold the instance lock before calling `reset_hardware()`. Such comments are very common in software including Linux and Mozilla (as shown later in this paper).

Though comments contain valuable information, including programmers' assumptions and intentions, they are not used to their maximum potential. Even though they significantly increase software readability and improve communication among programmers, they have not been examined by compilers or program analysis tools, such as debugging tools. Almost all compilers and program analysis tools simply skip the comments and parse only the source code.

If compilers and static analysis tools could automatically extract information such as programmers' assumptions described above, the extracted information could be used to check source code for potential bugs. For example, if `do_acct_process()` is called from a function other than `do_exit()`, or if the instance lock is not acquired before calling `reset_hardware()`, it may indicate a bug. The compilers and static analysis tools could detect such bugs automatically by comparing the source code and the assumptions extracted from comments if they could automatically extract such assumptions.

While comments can help programmers understand source code and specify programmers' assumptions and intentions in an explicit way, **bad or obsolete comments** can negatively affect software quality by increasing the chance of misunderstanding among programmers. In practice, as software evolves, programmers often forget to keep comments up to date. These obsolete comments, no longer consistent with the source code, provide confusing, misleading and even incorrect information to other programmers, which can easily introduce new bugs later. Unlike source code that can be tested via various in-house testing tools, comments can not be tested by current tools. Therefore, if comments could be automatically analyzed and checked against source code for inconsistencies, such bad comments may be identified to avoid introducing new bugs.

Unfortunately, automatically extracting information from comments is very challenging [20] because comments are written in natural language, may not even be grammatically correct, and are a mixture of natural language phrases and program identifiers. Moreover, many phrases in software have different meanings from natural language. For example, the word "pointer" in software is associated with "memory" and "buffer". While natural language processing (NLP) techniques have made impressive progress over the years, they are still limited to certain basic functionalities and mostly focus on well written documents such as the Wall Street Journal or other rigorous news corpus. Therefore, to automatically *understand* comments, it would require combining NLP with other techniques such as program analysis, statistics, and even domain-specific heuristics.

1.2 Contributions

This position paper takes the first initiative to study the *feasibility* and *benefits* of automatically analyzing program comments to detect software bugs and bad comments. Our feasibility and benefit analysis is conducted from three aspects using Linux as a demonstration case:

- **Are there hot topics in comments?** To analyze comments, we first need to understand their characteristics. Our results show that, while different Linux modules have different hot topics, they also share common topics such as synchronization and memory allocation. Specifically, 1.2-12.0% of comments (in different modules) in Linux are related to locks, and a substantial percentage (3.8-17.0%) of comments are about memory allocation. These results indicate that instead of aiming for the prohibitively challenging task of understanding any arbitrary comments, we can focus our comment analysis on automatically extracting information related to only hot topics.
- **Are comments useful for detecting bugs?** Comments provide an *independent* and more explicit source of information compared to source code, and this information can be used to perform sanity (consistency) checks against source code for potential bugs or bad comments. As a proof of concept, we conduct a preliminary analysis that uses heuristics (i.e. keyword searches) with the assistance of basic natural language processing techniques to extract information from lock-related comments. We choose the lock topic as our demonstration case as it is one of the major topics in comments, and synchronization bugs can cause severe damage that is difficult to detect. In our preliminary results, comments helped us find 12 new bugs in Linux, with 2 confirmed by the Linux developers, demonstrating some promising results to motivate the research direction of automatic comment analysis.
- **Are bad comments causing harm in practice?** While it is conceivable that bad comments can mislead programmers, have they introduced new bugs in practice? Our preliminary bug reports examination finds that inconsistent comments did introduce new bugs in real world software such as Mozilla. This indicates that it is important for programmers to maintain comments and keep them up to date and consistent with code; and it is also highly desirable to automatically detect bad and inconsistent comments to avoid misleading programmers and introducing new bugs later. Moreover, we analyze several bug databases and find that at least 62 bug reports in FreeBSD [3] are about incorrect and confusing comments, implying that some programmers have already realized the harm that can be caused by bad comments.

2 Hot Topics of Comments

To find out whether program comments have "hot topics", we conduct a simple keyword frequency study on Linux's comments. In our analysis, *a comment* is defined as one comment sentence.

Table 1 shows the most frequently used keywords, *hot keywords*, in comments from five major Linux modules. As expected, many hot topics are module specific. For example, a substantial percentage of comments in the kernel modules contain keywords "signal", "thread", or "cpu", whereas many comments in the memory management module contain keywords "page" or "cache".

Interestingly, while different Linux modules have their own hot keywords, they share some common hot keywords such as "lock", "alloc" and "free". For example,

kernel			mm			arch			drivers			fs		
keyword	**%**	**Freq**	**keyword**	**%**	**Freq**	**keyword**	**%**	**Freq**	**keyword**	**%**	**Freq**	**keyword**	**%**	**Freq**
signal	8.1%	109	page	30.6%	331	bit	5.4%	2729	device	4.1%	6828	block	4.4%	3174
thread	7.7%	104	cache	11.2%	121	interrupt	5.2%	2650	data	3.7%	6129	inode	3.4%	2407
cpu	7.0%	94	map	10.3%	111	register	4.2%	2121	interrupt	3.6%	6093	file	2.9%	2069
process	6.8%	92	memory	9.8%	106	address	3.3%	1661	bit	3.1%	5260	buffer	2.3%	1680
kernel	6.8%	91	*alloc*	9.6%	104	copyright	3.2%	1616	register	3.1%	5163	page	2.1%	1470
lock	6.2%	83	***lock***	7.4%	80	pci	3.1%	1561	command	3.0%	5023	*alloc*	2.0%	1413
task	5.9%	79	*free*	7.4%	80	kernel	2.9%	1489	buffer	3.0%	4989	*free*	1.8%	1322
timer	4.7%	63	swap	6.8%	73	irq	2.9%	1452	driver	2.9%	4836	directori	1.5%	1108
check	4.5%	61	start	6.2%	67	...	...	...	...	...	...	***lock***	1.5%	1082
time	4.4%	59	mm	5.6%	60	***lock***	0.8%	384	***lock***	0.8%	1385	pointer	1.5%	1055

Table 1: Keyword frequency in Linux. % is calculated as the number of comments in a module containing the keyword over the total number of comments in the module. Freq is keyword frequency.

kernel		mm		arch		drivers		fs	
%	**Freq**	**%**	**Freq**	**%**	**Freq**	**%**	**Freq**	**%**	**Freq**
10.0%	135	12.0%	130	1.2%	600	1.2%	1983	2.3%	1667

Table 2: Lock-related keyword frequency in Linux. Noise such as “block” and“clock” is excluded.

0.8% to 7.4% of the comments in Linux, a total of 3014 comments, contain the word “lock”. This is probably because often Linux code is reentrant and thereby requires locks to protect accesses to shared variables. As synchronization-related code is usually complicated and tricky with many assumptions, programmers commonly use comments to make the synchronization assumptions and intentions explicit.

Different keywords, e.g. lock, unlock, spinlock, and rwlock, are all about locks; however, they are considered separate keywords. Therefore, we improved our keyword rank techniques to find lock-related comments. We replace all lock-related keywords with “lock” and then count the total number of comments that contain “lock”. The results are shown in Table 2. The percentage of comments that contain “lock” is then increased to 1.2-12.0%.

Similarly, keywords related to memory allocation and deallocation also appear in a significant portion of comments, 3.8% and 17.0% in the *fs* module and the *mm* module, respectively. This is because memory management is another important topic that requires developers to communicate with each other. Miscommunication can easily lead to memory related bugs, which can be exploited by malicious users to launch security attacks.

While so far we have studied comments only from Linux code, we believe that our results represent comments of most system software including operating system code and server code, because synchronization and memory allocation/deallocation are important yet error-prone and confusing issues for such software.

3 Comment Analysis

As a proof of concept, we conduct a preliminary study that combines natural language processing techniques and topic-specific heuristics to analyze synchronization-related comments in Linux and use the extracted information to detect comment-code inconsistencies. As the goal of this position paper is merely to motivate the research of automatic comment analysis by demonstrating its feasibility and potential benefits, the comment analysis in this paper is heuristic-based and cannot be used to extract comments of arbitrary topic—achieving such goal remains as our immediate future work.

3.1 Analysis Goals

As a feasibility study to demonstrate the benefit potential, the analysis in our preliminary study focuses on extracting lock-related programming rules. Specifically, the goal of our analysis is to extract lock-related information (referred to as “rules” in this paper) according to the eight templates listed in Table 3. These templates are designed based on our manual examination of comment samples from Linux. Some comments have positive forms such as “the lock must be held here”, whereas some others are negative such as “the lock must not be held here”. Therefore, the automatic comment analysis needs to differentiate negative and positive forms. Otherwise, it will badly mislead the sanity checks.

In addition to determining to which template a lock-related comment belongs, we need to find the specific parameter values, i.e. which lock is needed.

ID	Rule Template
1/2	L must (NOT) be held [for V] before entering F.
3/4	L must (NOT) be held [for V] before leaving F.
5/6	L_A must (NOT) be held [for V] before L_B.
7/8	L must (NOT) be held here.

Table 3: Example rule templates. Each row shows two rule templates, one positive and one negative. L denotes a lock. F is a function. V means a variable. Brackets ([]) denote optional parameters.

3.2 Analysis Process

To automatically understand what type of lock-related rule a comment contains is a challenging task. The reason is that the same rule can be expressed in many different ways. For example, the rule *"Lock L must be held before entering function F"* can be paraphrased in many ways, such as (selected from comments in Linux): (1) *"We need to acquire the write IRQ lock before calling ep_unlink()"*; (2) *"The queue lock with interrupts disabled must be held on entry to this function"*; (3) *"Caller must hold bond lock for write."* Therefore, to analyze comments, we need to handle various expressing forms.

A Simple Method. A simple method is to use some heuristics such as "grep" to search for certain keywords in comments. For example, we can first grep for comments that contain keyword "lock" to obtain all lock-related comments. We then look for action keywords "acquire", "hold", or "release" or their variants such as "acquired", "held" and "releasing". Afterward, we look for negation keywords such as "not", "n't", etc. to differentiate negative rules from positive ones.

While the method is simple and can narrow down the number of comments for manual examination, it is very inaccurate because it considers only the presence of a keyword, regardless where in the comment the keyword appears. The simple approach will make *mistakes* in at least the following three cases. First, if the action keyword is not in the main clause, the sentence may not contain an expected rule. For example, comment "returns -EBUSY if locked" from Linux does not specify a locking rule since "if locked" is a *condition* for the return value. Second, if the object of the action verb is not a lock, maybe no locking rule is contained. For example, a comment from Linux "lockd_up is waiting for us to startup, so will be holding a reference to this module, ..." contains "lock" and "hold", but the object of "hold" is not a lock, and no expected rule is contained. Third, a comment containing the keyword "not" does not necessarily imply the extracted rule is negative. For instance, "Lock L must be held before calling function F so that a data race will not occur", still expresses a positive rule.

Our Preliminary Method. To accurately analyze comments for lock-related rules, we extend the simple methods above with systematic natural language processing (NLP) techniques to analyze comment structures and word types.

We first break each comment into sentences, which is non-trivial as it involves correctly interpreting abbreviations, decimal points, etc. Moreover, unique to program comments is that sentences can have '*', '/' and '.' symbols embedded in one sentence. Furthermore, sometimes a sentence can end without any delimiter. Therefore, besides using the regular delimiters, '!', '?', and ';', we use '.' and spaces together as sentence delimiters instead of using '.' alone. Additionally, we consider an empty line and the end of a comment as the end of a sentence.

Next, we use a modified version of word splitters [7] to break a sentence into words. We then use Part-of-Speech (POS) tagging and Semantic Role Labeling techniques [7] to tell whether a word in a sentence is a verb, a noun, etc., to distinguish main clauses from sub clauses, and to tell subjects from objects.

Then we apply keyword searches on selected components of each comment. Specifically, we first search for keyword "lock" in the *main clause* to filter out those lock-unrelated comments. Then we check whether the keyword "lock" serves as the *object* of the verb or the *subject* in the main clause, and whether the *verb* of the main clause is "hold", "acquire", "release", or their variants. By applying these searches on the most relevant components, we can determine whether the comment contains a lock-related rule or not.

Finally, we determine the following information to generate the rule in one of the forms presented in Table 3.

Is the rule specific to a function? If we see words such as "call" or "enter function" in a sentence, then it is highly likely that the rule contained in the target comment is specific to a function associated with the comment (Template 1 - 4 in Table 3). In this case, we can automatically extract the function name from the source code. The intuition here is that a comment about a function is usually inserted at the beginning of the function. Therefore, a simple static analysis can easily find the name of the function defined right after the comment.

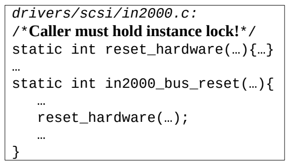
What is the lock name? The lock name of a rule is usually the object of the verb in the main clause, which is often explicitly stated in comments. Therefore, we can automatically extract it as our NLP tools can tell which word is the object.

Is the rule positive or negative? By identifying the verb and negation words, such as "not", we can determine whether the rule is positive (template 1, 3, 5, or 7) or negative (template 2, 4, 6, or 8). For example, a main clause containing verb "hold" without any negation word is likely to be positive, whereas a main clause containing verb "hold" with a negation word is likely to be negative.

Our analysis algorithm is still primitive and is now designed for lock-related comments, and we are in the process of improving its accuracy and flexibility to analyze comments of any topic selected by users.

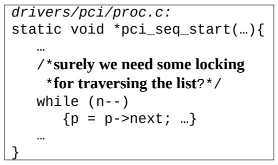
3.3 Inconsistency Detection

After we extract the lock related rules, we scan the source code to detect comment-code inconsistencies. Our analysis is flow-sensitive and context-sensitive. For example, if a rule extracted from comments says that a lock L should be held before calling function F, our checker



```
drivers/scsi/in2000.c:
/*Caller must hold instance lock!*/
static int reset_hardware(…){…}
…
static int in2000_bus_reset(…){
   …
   reset_hardware(…);
   …
}
```

(a) The comment says that `reset_hardware()` must be called with the instance lock held, but no lock is acquired before calling it in the code.



```
drivers/pci/proc.c:
static void *pci_seq_start(…){
   …
   /*surely we need some locking
    *for traversing the list?*/
   while (n--)
      {p = p->next; …}
   …
}
```

(b) The comment states that a lock is needed when the list is traversed. But there is no lock acquisition in the code.

Figure 1: Two confirmed bug examples in Linux.

performs a static analysis from every root (without any caller, e.g. `main()`) of the call-graph to explore every path that may lead to function F to see if it acquires lock L before calling F. To improve efficiency, we first prune the control flow graph and call graph and only keep those nodes that are related to lock acquire, L, or function F. We also perform simple points-to analysis to handle variables that may be aliases of L. The details of our rule checkers are similar to [13, 16].

Although we use static checking to detect bugs, it is quite conceivable that rules extracted from comments can be checked dynamically by running the program.

3.4 Results: New Bugs Detected

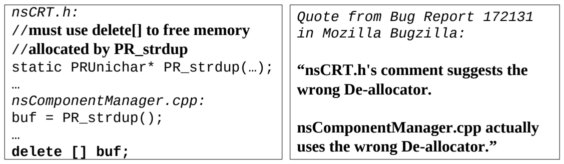
We conducted a preliminary evaluation of the analysis on Linux, which automatically extracted 538 lock-related rules from five Linux modules shown below.

kernel	mm	arch	drivers	fs
29	16	50	263	180

We detected 12 bugs in Linux with 2 confirmed by developers by using 137 rules extracted from comments, and we are working on checking the rest of the rules. Figure 1(a) shows a confirmed bug in Linux. The comment above `reset_hardware()` states that the caller must hold the instance lock, but the lock is not acquired when the function is called from `in2000_bus_reset()`, which can cause data races. This bug was fixed in Linux by adding `spin_lock_irqsave(instance->host_lock, flags)`. As shown in Figure 1(b), a comment says a lock is needed to traverse the list. However, no lock is used for accessing the list in the code. This bug was fixed by adding proper locking for accessing the list.

4 Bad Comments

If we can automatically detect bad comments, we can help prevent these bad comments from confusing and misleading programmers, who consequently introduce bugs. In this feasibility study, we manually study bug reports from several Bugzilla databases to find out if bad comments have caused new bugs.



```
nsCRT.h:
//must use delete[] to free memory
//allocated by PR_strdup
static PRUnichar* PR_strdup(…);
…
nsComponentManager.cpp:
buf = PR_strdup();
…
delete [] buf;
```

Quote from Bug Report 172131 in Mozilla Bugzilla:

"nsCRT.h's comment suggests the wrong De-allocator.

nsComponentManager.cpp actually uses the wrong De-allocator."

Figure 2: Bad comments that caused a new bug in Mozilla. Code is modified to simplify illustration.

Such a real world bug example from Mozilla (Revision 1.213 of nsComponentManager.cpp) is shown in Figure 2. This bug was introduced because the programmer read and followed an incorrect comment, as indicated by the description in the Bugzilla bug report: *"nsCRT.h's comment suggests the wrong De-allocator. nsComponentManager.cpp actually uses the wrong De-allocator"*. Misled by the incorrect comment, "must use `delete[]` to free the memory", a programmer used `delete[]` to free the memory pointed by `buf`, resulting in a bug as reported to Mozilla's Bugzilla database [6]. In a later version (Revision 1.214 of nsComponentManager.cpp), this bug was fixed by replacing `delete[] buf` with `PR_free(buf)`. The incorrect comment has also been fixed accordingly (in file nsCRT.h).

Moreover, we found that at least 62 bug reports in FreeBSD [3] are about incorrect and confusing comments, indicating that some programmers have realized the importance of keeping comments updated.

5 Related Work

Extracting rules and invariants from source code. Many bug detection tools [10, 11, 16] have been proposed to extract rules or invariants from source code or execution traces to detect bugs. Unlike these tools, our study automatically extracts programming rules from *comments*. Our approach also allows the detection of bad comments that can introduce bugs later.

Empirical study of comments. Woodfield, Dunsmore and Shen [19] conducted a user study on forty-eight experienced programmers and showed that code with comments is likely to be better understood by programmers. Jiang and Hassan [15] studied the trend of the percentage of commented functions in PostgreSQL. Recent work from Ying, Wright and Abrams [20] shows that comments are very challenging to analyze automatically because they have ambiguous context and scope. None of these propose any solution to automatically analyze comments or detect comment-code inconsistencies.

Annotation language. Annotation languages [4, 9, 12, 14, 21] are proposed for developers to comment source code using a formal language to specify special information such as type safety [21]. Previous work titled "comment analysis" [14] automatically detects bugs caused by wrong assumptions made by programmers. However, what they refer to as "comments" are essentially annotations written in a formal annotation language, not comments written in natural language that are used in most existing software and are analyzed in our work.

While these annotation languages can be easily analyzed by a compiler, they have their own limitations. First, these annotation languages are not as expressive or flexible as natural language, often only expressing simple assumptions such as buffer lengths and data types. Additionally, they are not widely adopted because developers are usually reluctant to learn a new language. Finally, millions of lines of comments written in natural language already exist in legacy code. Due to all these reasons, our approach well compliments the annotation language approach since we analyze general comments written in natural language. Rules inferred by our approach from comments can also be used to *automatically* annotate programs to reduce manual effort.

Automatic document generation from comments. Many comment style specification tools are proposed and are widely used to automatically build documentation from comments [1, 2, 5, 8]. Since these specification tools restrict only the format but still allows programmers to use natural language for the content (i.e. they are semi-structured like web pages), automatically "understanding" or analyzing these comments still suffers from similar challenges to analyzing unstructured comments.

Comment and document analysis for software reuse. Matwin and Ahmad [18] used natural language processing techniques to extract noun phrases from program comments in LINPACK (a linear algebra package) to build a function database so that programmers can search the database to find routines for software reuse. Another study [17] built a code library by applying information retrieval techniques on documents and comments. But none of these work attempts to "understand" the information contained in comments to automatically checked against code for inconsistencies.

6 Conclusions and Future Work

In this paper, we study the feasibility and benefits of automatically analyzing comments to detect software bugs and bad comments. Our preliminary results with real world bugs and bad comment examples have demonstrated the benefits of such new research initiative. We are in the process of continuing exploring this idea in several ways. First, we are improving the accuracy and *generality* of our comment analysis algorithm. Second, we are applying our algorithm to extract other types of rules such as memory-related rules, to detect other types of bugs, and to detect bad comments. Third, we are studying the characteristics of comments from other software to validate that our observations from Linux comments are representative. So far, our examinations of Mozilla and Apache have shown results similar to Linux.

7 Acknowledgments

We thank the anonymous reviewers for their useful feedback and the Opera group for useful discussions and paper proofreading. This research is supported by NSF CNS-0347854 (career award), NSF CCR-0325603 grant, DOE DE-FG02-05ER25688, and Intel gift grant.

References

[1] C# XML comments. http://msdn.microsoft.com/msdnmag/issues/02/06/XMLC/.

[2] Doxygen. http://www.stack.nl/˜dimitri/doxygen/.

[3] FreeBSD problem report database. http://www.freebsd.org/support/bugreports.html.

[4] Java annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[5] Javadoc tool. http://java.sun.com/j2se/javadoc/.

[6] Mozilla Bugzilla database. https://bugzilla.mozilla.org/.

[7] NLP tools. http://l2r.cs.uiuc.edu/˜cogcomp/tools.php.

[8] RDoc. http://rdoc.sourceforge.net/.

[9] SAL annotations. http://msdn2.microsoft.com/en-us/library/ms235402.aspx.

[10] D. R. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP '01*.

[11] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE'00*.

[12] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 2002.

[13] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *PLDI'02*.

[14] W. E. Howden. Comments analysis and programming errors. *IEEE Trans. Softw. Eng.*, 1990.

[15] Z. Jiang and A. Hassan. Examining the evolution of code comments in PostgreSQL. In *MSR '06*.

[16] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE'05*.

[17] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 1991.

[18] S. Matwin and A. Ahmad. Reuse of modular software with automated comment analysis. In *ICSM '94*.

[19] S. Woodfield, H. Dunsmore, and V. Shen. The effect of modularization and comments on program comprehension. In *ICSE'81*.

[20] A. Ying, J. Wright, and S. Abrams. Source code that talks: An exploration of Eclipse task comments and their implication to repository mining. In *MSR'05*.

[21] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI'06*.