/* iComment: Bugs or Bad Comments? */

Lin Tan[†], Ding Yuan[†], Gopal Krishna[†], and Yuanyuan Zhou[†‡]
[†]University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
[‡]CleanMake Co., Urbana, Illinois, USA
{lintan2, dyuan3, gkrishn2, yyzhou}@cs.uiuc.edu

ABSTRACT

Commenting source code has long been a common practice in software development. Compared to source code, comments are more *direct*, *descriptive* and *easy-to-understand*. Comments and source code provide relatively redundant and independent information regarding a program's semantic behavior. As software evolves, they can easily grow out-of-sync, indicating two problems: (1) bugs - the source code does not follow the assumptions and requirements specified by correct program comments; (2) bad comments - comments that are inconsistent with correct code, which can confuse and mislead programmers to introduce bugs in subsequent versions. Unfortunately, as most comments are written in natural language, no solution has been proposed to automatically analyze comments and detect inconsistencies between comments and source code.

This paper takes the *first* step in automatically analyzing comments written in natural language to extract implicit program rules and use these rules to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments. Our solution, *iComment,* combines Natural Language Processing (NLP), Machine Learning, Statistics and Program Analysis techniques to achieve these goals.

We evaluate iComment on four large code bases: Linux, Mozilla, Wine and Apache. Our experimental results show that iComment automatically extracts 1832 rules from comments with 90.8-100% accuracy and detects 60 comment-code inconsistencies, 33 new bugs and 27 bad comments, in the latest versions of the four programs. Nineteen of them (12 bugs and 7 bad comments) have already been confirmed by the corresponding developers while the others are currently being analyzed by the developers.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement —*Documentation*

General Terms

Algorithms, Documentation, Experimentation, Reliability

Keywords

comment analysis, natural language processing for software engineering, programming rules, and static analysis

1. INTRODUCTION

1.1 Motivation

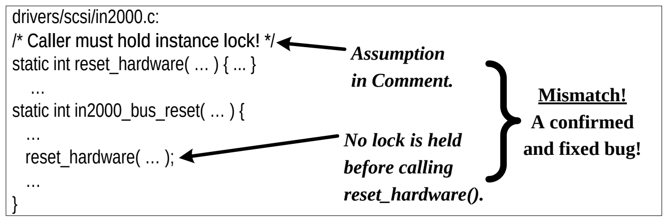
Despite costly efforts to improve software-development methodologies, software bugs in deployed code continue to thrive and contribute to a significant percentage of system failures and security vulnerabilities. Many software bugs are caused by a mismatch between programmers' intention and code's implementation. A mismatch would be developed due to miscommunication between 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. However, if such assumption is not specified clearly, other programmers can easily violate this assumption and introduce bugs. The problem above is further worsened by software evolution and growth, with programmers frequently joining and departing from the software development process.

To address the problem, comments became standard practice in software development to increase the readability of code and to express programmers' intention in a more explicit but less rigorous manner than source code. Comments are written by programmers in natural language to explain code segments and data structures, to specify assumptions, to record reminders, etc. that are often not expressed explicitly in source code. From our simple statistics, Linux contains about 1.0 million lines of comments for 5.0 million lines of source code, and Mozilla has 0.51 million lines of comments for 3.3 million lines of code, excluding copyright notices and blank lines. These results indicate the common usage of comments to improve software reliability and maintainability in large software.

Even though comments are less formal and precise than source code, comments have a unique advantage: comments are much more *direct, descriptive and easy-to-understand* than source code. In other words, many assumptions are specified directly and clearly in comments but are usually difficult to infer from source code. For example, the following comment from the latest Linux Kernel (kernel/irq/manage.c) clearly specifies that function `free_irq()` must not be called from interrupt context.

```
kernel/irq/manage.c:
/* This function must not be called from interrupt context */
void free_irq( ... ) { ... }
```

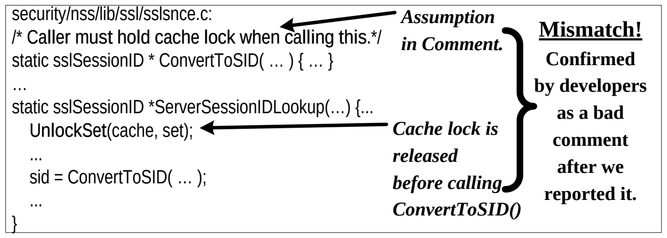
It is hard to infer this assumption from the source code, even with advanced techniques such as code mining or probabilistic rule inference [16, 24, 26] (more discussion in Section 8.2).



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

Assumption in Comment.
No lock is held before calling reset_hardware().
Mismatch! A confirmed and fixed bug!

Figure 1: A new bug detected by our tool in the latest version of Linux, which has been confirmed and fixed by the Linux developers.



```
security/nss/lib/ssl/sslsnce.c:
/* Caller must hold cache lock when calling this.*/
static sslSessionID * ConvertToSID( ... ) { ... }
...
static sslSessionID *ServerSessionIDLookup(...) {...
   UnlockSet(cache, set);
   ...
   sid = ConvertToSID( ... );
   ...
}
```

Assumption in Comment.
Cache lock is released before calling ConvertToSID()
Mismatch! Confirmed by developers as a bad comment after we reported it.

Figure 2: A new misleading bad comment detected by our tool in the *latest* version of Mozilla. It has been confirmed by the Mozilla developers, who replied us "I should have removed that comment about needing to hold the lock when calling ConvertToSID".

Comments and source code provide relatively **redundant and independent** information about a program's semantic behavior, creating a unique opportunity to compare the two to check for inconsistencies. As pointed out by a recent study [22] of the evolution of comments, when software evolves, it is common for comments and source code to be out-of-sync. An inconsistency between the two indicates either a bug or a bad comment, both of which have severe implication on software robustness and productivity:

(1) Bugs—source code does not follow correct comments. Such cases may be caused by time-constraints or other reasons, but a very likely reason is that some code and its associated comments are updated with a different assumption, while *some old code is not updated accordingly and still follows the old assumption*.

Figure 1 shows such a real world bug example from Linux Kernel 2.6.11. The comment above the implementation of function `reset_hardware()` *explicitly* states the requirement that the caller of this function *must* hold the instance lock. However, in the `in2000_bus_reset()` function body, the lock is not acquired before calling `reset_hardware()`, introducing a bug (it has been confirmed by the Linux developer as a true bug and has been fixed). In Section 7, we will show more *new* bug examples that our tool detected in the *latest versions* of large software including Linux.

(2) Bad comments that can later lead to bugs. It is common for developers to change code without updating comments accordingly as developers may not be motivated, may not have time or simply forget to do so. Furthermore, as opposed to source code that always goes through a series of software testing before release, *comments cannot be tested to see if they are still valid*. As a result, many comments can be out-of-date and incorrect. We refer to such comments as *bad comments*. Note that we do *not* consider comments with simple typographical errors or grammar errors as bad comments.

Figure 2 shows a bad comment example, automatically detected by our tool in the *latest* version of Mozilla and confirmed by the developers based on our report. The outdated comment, the caller *must* hold cache lock when calling function `ConvertToSID()`, does not match with the code that releases the lock before calling `ConvertToSID()`. Although such out-of-date or incorrect bad comments do not affect Mozilla's correctness, they can easily mislead programmers to introduce bugs later, as also acknowledged by several Mozilla developers after we reported such bad comments. In Section 7.2, we will show two real world bad comments in Mozilla that *have caused* new bugs in later versions.

The severity of bad comments is also realized by programmers to some degree. Very often some software patches only fix bad comments to avoid misleading programmers. We analyzed several bug databases and found that at least 62 bug reports in FreeBSD [4] are only about incorrect and confusing comments. For example, FreeBSD patch "kern/700" only modifies a comment in the file /sys/net/if.h. Similarly, the Mozilla patch for bug report 187257 in December 2002 only fixed a comment in file FixedTableLayoutStrategy.h.

The bug and bad comment examples above indicate that it is very important for programmers to maintain code-comment consistency; and it is also highly desirable to automatically detect bad comments so that they can be fixed before they mislead programmers and cause damages.

To the best of our knowledge, *no tool has ever been proposed to automatically analyze comments written in natural language and detect inconsistencies between comments and source code*. Almost all compilers and static analysis tools simply skip comments as if they do not exist, losing the opportunity to use comments to their maximum potential as well as to detect bad comments.

1.2 Challenges in Analyzing Comments

The reason for the almost non-existent work in comment analysis and comment-code inconsistency detection is that automatically analyzing comments is extremely difficult [44]. As comments are written in natural language, they are difficult to analyze and almost impossible to "understand" automatically, even with the most advanced natural language processing (NLP) techniques [27], which mostly focus on analyzing well written news articles from the Wall Street Journal or other rigorous corpora. To make things worse, unlike these news articles, comments are usually not well written and many of them are not grammatically correct. Moreover, many words in comments have different meanings from their real-world meanings. For example, words "buffer", "memory" and "lock" have program domain specific meanings that cannot be found in general dictionaries. Additionally, many comments are also mixed with program identifiers (variables, functions, etc.) that do not exist in any dictionary.

Despite the above fundamental challenges, it is highly desirable for a comment analysis and comment-code inconsistency detection tool to have the following properties: (1) **accuracy**: the analysis and inconsistency results need to be reasonably accurate. Too many false positives can greatly affect the usability of the tool; (2) **practicality**: the tool should be able to analyze real world comments from existing software (such as Linux) without requiring programmers to rewrite all comments; (3) **scalability**: the tool should be scalable to handle large software with multi-million lines of code and comments; (4) **generality**: the tool cannot be "hard-coded" to handle only a specific type of comments. (5) **minimum manual effort**: while it might be extremely difficult to eliminate programmers' involvement, the tool should operate as autonomously as possible.

1.3 Our Contributions

This paper makes the *first* step in automatically analyzing program comments written in natural language to extract programmers' assumptions and requirements (referred to as *rules*) and to automatically detect inconsistencies between comments and source code—indication of bugs or bad comments, both of which affect software quality now or later. Because it is virtually impossible to understand any arbitrary comment, our tool, called **iComment**, provides a general framework to analyze comments topic by topic, and has demonstrated its effectiveness by automatically analyzing lock-related and call-related comments to detect bugs and bad comments in large software. Since using NLP techniques alone cannot solve our problem, to address the fundamental challenges and provide the five desired properties listed earlier, iComment combines techniques from several areas including:

- Natural language processing techniques to tag each word as "verb", "noun", etc., parse a comment into main clauses, sub-clauses, etc., and label semantic roles such as subjects and objects so that later steps can focus on important words and clauses. Additionally, we use many language related features, such as "preposition leading", as features in our model deduction to extract rules from comments.
- Statistics techniques, specifically clustering and correlation analysis, to single out hot topics and correlated words from comments so that users can select hot comment topics and iComment can automatically extract comments related to the topic keyword specified by users.
- Machine learning techniques, specifically decision tree learning, to generate models from a small set of manually labeled comments from one software code base to automatically analyze other comments from the same or different software.
- Program analysis techniques to help filling rule parameters, detect inconsistencies between code and comments, rank rules and inconsistencies and prune false positives.

We have evaluated our iComment tool using four large open source software projects: one operating system (Linux), one server (Apache) and two desktop programs (Mozilla and Wine). With an average of 18% training data from each program, our tool can automatically extract a total of 1832 rules from comments in these four programs, with an accuracy higher than 90.8%.

Interestingly, our experimental results show that models trained from a small set of comments from one software system (Linux) can be used to analyze other software such as Mozilla, Wine, and Apache, with a reasonable (78.6-89.3%) accuracy. This result indicates that the training requirement of our tool is not stringent since we can always release our tool, together with the models trained from representative software, to scan other open source and commercial software with no further training required—reducing users' manual effort to minimum.

More importantly, our tool detected 60 comment-code inconsistencies, including 33 new bugs and 27 bad comments[1] related to the lock and call topics currently supported by iComment, from the latest versions of Linux, Mozilla, Wine, and Apache. Nineteen of these inconsistencies (12 bugs and 7 bad comments) have already been confirmed by the corresponding developers. From Linux alone, we have detected 30 new bugs and 21 bad comments, 14 of which are confirmed by the Linux developers. The other inconsistency errors are still being analyzed by them. Many of the bad comments can lead to bugs in future versions since they provide misleading information. iComment's inconsistency detection is also reasonably accurate with only 38.8% false positive rate. Section 7 shows more examples of detected new bugs and bad comments from Linux.

[1]Even though we use heuristics described in Section 4.4 to tell whether a detected inconsistency indicates a bug or a bad comment, we are not 100% sure, except for the 12 bugs and 7 bad comments that have already been confirmed by the developers. But in either case, there is an inconsistency between the code and the comment, so one of them is wrong.

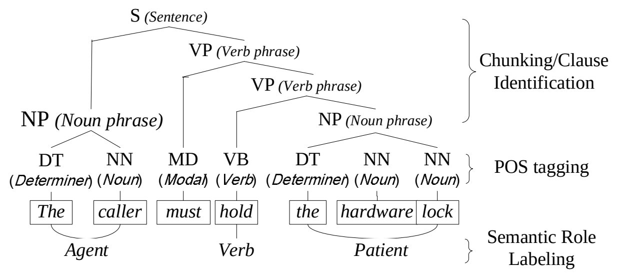


Figure 3: An example of POS tagging, chunking and clause identification, and semantic role labeling. The leaf nodes of this tree forms a sentence.

In summary, our paper makes the following contributions:

- The first work that analyzes comments written in natural language and extracts specifications automatically
- The first research effort that calls for attention on the damage caused by bad comments and the first tool to detect bad comments automatically
- The first work that uses comments written in natural language to automatically check code for bugs
- A practical and scalable tool that has detected many (60) new lock-related and call-related bugs and bad comments from the latest versions of large open source software including Linux and Mozilla without too much human involvement (especially since the models trained from one software system can be used for other software systems).

2. BACKGROUND

This section gives a brief background of natural language processing (NLP) techniques used in our work and the special challenges to apply NLP techniques for comment analysis.

Current NLP Capabilities. While recent NLP research has made impressive progress, it is still far from reaching its ultimate goal. Most NLP is still at the "processing" stage instead of the "understanding" stage [27]. As natural language is *ambiguous*, it is prohibitively expensive, if ever possible, to turn *arbitrary* sentences written in natural language to precise and unambiguous descriptions that computers can use to answer questions [27].

So far, word tagging, phrase and clause parsing, and semantic role labeling are three of the most mature NLP techniques, Figure 3 shows an example of these three techniques on a Linux comment.

(1) Word tagging: often called Part-Of-Speech (POS) tagging, identifies the Part-Of-Speech (e.g., Noun, Verb, etc.) of each word within the sentence. The basic approach is to train a classification model from some manually labeled dataset. The current state of the art can achieve labeling accuracy of more than 97% for similar news articles [2].

(2) Phrase and clause parsing: often referred as chunking and clause identification. Chunking is a technique to divide a text in syntactically correlated parts of words, i.e., *phrase* as in Noun phrase, Verb phrase, etc. Clause identification recognizes clauses which are word sequences with a subject and a predicate. These two techniques form a coherent partial syntax of a sentence. The current state of the art can achieve parsing precision and recall[2] of more than 90% for well written news articles [2].

(3) Semantic role labeling: it extracts all the semantic arguments of all the verbs within a sentence. Typically the semantic arguments include Agent, Patient, Instrument, etc. and also adjuncts such as Locative, Temporal, Manner, Cause, etc. The current state of the art can achieve labeling precision of more than 80% and recall of more than 70% [2].

In our work, we use word POS tagging to single out important words as well as to select features to help build classification models from training data to analyze comments. Phrase and clause parsing are used to extract important features in order to filter out noises. They are also used in combination with semantic role labeling to extract specific information (e.g., specific variable names) from the object (typically called "patient" using semantic terminology) of target verbs.

Special Challenges with Comment Analysis. As briefly discussed in Introduction, using NLP to analyze comments has special challenges, namely, (1) grammar and spelling errors are very common in program comments; (2) program identifiers are mixed with regular dictionary words; and (3) many words such as "buffer" and "memory" in comments have programming-domain specific meanings that are not reflected in any general dictionaries and synonym databases.

3. LIMITATIONS OF A "GREP"-LIKE METHOD

This section describes the limitations of an intuitive, "grep"-like method, which was used in our initial feasibility study of automatic comment analysis and motivated us to explore more advanced techniques to analyze comments.

An intuitive method is to perform keyword searches just like "grep-ing" over the source code repository for certain comments. For example, a programmer can first grep for comments that contain keyword "lock" to obtain all lock-containing comments. The results can then be fed into another keyword search for action keywords such as "acquire", "hold", or "release" or their variants like "acquired", "held" and "releasing". To differentiate negative rules from positive ones, the programmer can look for negation keywords such as "not", "n't", etc. to find comments that specify locking rules (i.e., must acquire a lock before certain operations).

While the method above is simple and can help narrow down the number of comments to examine, it heavily relies on programmers' manual effort: (1) to provide all keywords and their synonyms and variants (past tenses, plurals, abbreviations, etc.) in the search, (2) to examine the comments returned from the keyword search, and (3) to convert each comment to a rule manually. Moreover, when a programmer wants to look for other types of comments, for example, comments related to memory allocations, the programmer needs to repeat all the manual effort above all over again.

Table 1 shows the number of comments left after we apply the above series of keyword searches for lock-related comments in Linux and Mozilla. 720-1826 comments remain for programmers to manually examine and to convert to rules. While another round of search using other keywords may further reduce the number of comments, it can also filter out pertinent comments that contain lock-related rules.

Additionally, this method can be very inaccurate because it considers only the presence of a keyword, regardless of where in the comment the keyword appears, which will introduce both false positives and false negatives. For example, "returns -EBUSY if a lock is held" does not specify a locking rule since "if a lock is held" is a *condition* for the return value. Another 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 it does not specify a locking rule. Finally, a comment containing a negation keyword "not" does not necessarily imply the extracted rule is negative. "Lock L must be held before calling function F so that a data race will not occur", still expresses a positive rule.

[2]Recall is a standard statistical measure explained in Section 6.

Line Of Comments	Linux	Mozilla
Only *lock*	3981	1673
lock with other keywords	1826	720

Table 1: Number of comments after using keyword search for "lock", "acquire", "release", "hold", etc. and their variants.

4. iComment IDEA AND OVERVIEW

The goal of iComment is to automatically extract rules from comments and use these rules to detect code-comment inconsistencies. To achieve this goal, we need to address four main challenges: (1) What to extract? (2) How to extract? (3) How to check for inconsistencies between comments and code? (4) How to rank inconsistency results? This section presents our solutions to each challenge. Section 5 will describe each step of the comment analysis in more detail.

4.1 What to Extract?

Addressing this challenge requires considering two issues. The first is what type of information is useful to extract from comments. Typically there are two types of comments, one explains some code segment, and the other specifies programmers' assumptions and requirements. For example, Linux's comment "Find out where the IO space is" belongs to the first type, whereas "Caller must hold bond lock for write", belongs to the second type.

Clearly, it is less useful checking the first type of comments since they are usually consistent with the source code (since they are together). Even in the case of inconsistencies, they are less likely to mislead programmers and introduce bugs later. The second type is much more important—it clearly specifies certain assumptions and requirements that other programmers need to follow. For example, the second comment example given above requires all callers of the function to hold a lock before calling it. If such a comment is obsolete or incorrect, it can directly mislead programmers to introduce bugs. Therefore, our work focuses on the second type—*rule-containing comments*, comments that specify certain assumptions and requirements (referred to as *rules*). As shown in the next section, many important features are selected for this purpose, e.g., whether a comment contains any imperative words such as "must", "should", "need", "ought to", and many others.

In addition, as current NLP techniques are still primitive, it is prohibitively difficult to extract information from any arbitrary rule-containing comments. Therefore, our study (as a first step in this direction) targets for comments that are related to "hot" (common) and important topics. To find hot topics from a software code base or a set of them, we provide two *hot topic miners* that combine NLP's POS tagging technique with statistics techniques (described in detail in Section 5). Based on the hot topics or hot keywords extracted, the user or we can select those important ones to start comment analysis.

The second consideration in determining what to extract depends on what information can be checked against source code. Although both static and dynamic checking for software bugs have made impressive advances in recent years [16, 37], not any arbitrary rule can

ID	Rule Template
1/2	$\langle$ R $\rangle$ must (NOT) be claimed before entering $\langle$ F $\rangle$.
3/4	$\langle$ R $\rangle$ must (NOT) be claimed before leaving $\langle$ F $\rangle$.
5/6	$\langle R_A \rangle$ must (NOT) be claimed before $\langle R_B \rangle$.
7/8	$\langle$ R $\rangle$ must (NOT) be claimed in $\langle$ F $\rangle$.
9/10	$\langle F_A \rangle$ must (NOT) be called from $\langle F_B \rangle$.
11/12	$\langle F_A \rangle$ must (NOT) be called before $\langle F_B \rangle$.

Table 2: Examples of rule templates. R is a resource, e.g., a lock (or a buffer and a file descriptor), that a system can claim and release. F can be one function or a group of functions. Each row shows two rule templates, one positive and one negative.

be automatically checked easily. Therefore, we focus on extracting rules that can be checked against source code. Examples of such rules include "hold a lock l before calling function A ", "acquire a lock l in the function A", "allocate a buffer for b before entering function A", "call B before calling A", etc. We refer each type as a *rule template*, and refer lock *l*, function *A*, etc. as *rule parameters* to a rule template.

Since our work focuses more on comment analysis than on static rule checking, we select representative rules to demonstrate the idea, process and effects of our comment analysis. Currently, our iComment prototype supports 12 types of rules related to locking rules and calling rules as listed in Table 2. These rules are chosen based on our hot topic extraction from program comments and are so far not well addressed by previous work [16, 26] that can only check general rules like "releasing a lock after acquiring a lock" but not those software-specific rules as those listed in Table 2. As discussed in Section 8, we are in the process of extending our rule checkers to support other types of rules such as interrupt enabling/disabling, memory allocation, etc.

4.2 How to Extract?

Similar to general NLP research, extracting information from documents typically requires a model trained on a small set of manually labeled documents from representative document collections. The model can then analyze *other* documents in the same or other collections [27]. Our comment analysis follows this approach.

For each hot topic selected, the comment extraction process is divided into two stages (see Figure 4): (1) training stage; (2) comment analysis (rule generation) stage. The former builds the rule generation model and the latter uses the model to analyze comments. Each stage is further divided into several steps. The techniques used in each step are also listed in Figure 4.

By default, the training stage is done *in-house by an iComment builder (e.g., us)* to train the rule generator, a decision tree classifier, using a small set of randomly selected comments from representative software. In other words, we, as the iComment builder, first train iComment using a few manually-labeled (i.e., having the contained rules manually extracted) comments from representative software on some common topics such as lock-related, call-related, memory allocation-related, etc. After a user obtains iComment along with our trained rule generator, he/she can use it to analyze comments of similar topics from his/her software without any training required.

This is similar to most NLP research projects such as the famous Penn Tree Bank project [35], which release their tagging tools trained in-house with manually labeled sample datasets from some popular newspapers. The rationale is that models trained from some sample datasets should work reasonably well for documents of similar types.

Since developers share many common languages (wording, phrasing, etc.) about programming (After all, we are all trained from almost the same set of standard programming textbooks), we believe that, in-house training with a sample comment set covering a variety of developers will produce rule generators applicable to other software. *Interestingly, our correlation results on lock keywords shown in Table 3 in Section 5.3 with two different software, Linux and Mozilla, demonstrate that programmers do use similar words for the same topic. Also our cross-software training results (i.e., using models trained from one software to analyze comment in another) further confirm this hypothesis.*

Certainly, if a user desires higher analysis accuracy or wants to analyze a topic that is not currently supported by our rule generator, he/she can use iComment's training components shown in Figure 4 to train a rule generator specifically for his/her software on the selected topic.

The training stage consists of three steps: (i) TR-comment extraction, which extracts all topic-related, rule-containing comments, called *TR-Comments*, for a given topic keyword such as "lock" using NLP and statistics techniques; (ii) comment sampling, which provides a small set of randomly selected TR-comment samples for manual labeling; (iii) rule training that uses the manually labeled sample set to train the rule generator, which can then be used to analyze unlabeled comments from the same or different software.

The rule generation stage is straightforward. Based on the topic selected by the user, iComment uses the corresponding decision tree model to analyze all TR-comments from the target software by first mapping them to *rule templates*, and then uses semantic role labeling and program analysis to fill in the *rule parameters*. At the end, the rule generator produces all the rules whose confidence, produced by the decision tree model, is higher than certain a threshold so that the rule checker can use these rules to detect code-comment inconsistencies. Details about each training and extraction step are described in Section 5.

4.3 How to Check for Inconsistencies?

Based on rules extracted from comments, the rule checker of iComment analyzes the source code for mismatches: is a rule supported in all related source code, or is it violated in some code locations or paths? To achieve this goal, the rule checker performs a flow-sensitive and context-sensitive program analysis of the entire source code in a way similar to previous work [20, 26]. Before the checking process, for certain rules such as lock-related rules, similar to many previous rule checkers [20], the user needs to specify the actions (e.g., the lock function names) used in his/her software to acquire or release a lock. For example, Linux uses functions such as `spin_lock` and `spin_unlock` which can be easily found by looking at related header files. For many other rules such as call-related rules, no action specification is needed.

For each rule, the checker starts from every root node (e.g., main(), and exported functions) in the call graph to conduct a path-sensitive analysis to see if the rule is violated. For each path, a state machine corresponding to the rule is used to detect rule violations. The path analysis also traverses into function calls so it is context-sensitive. The path exploration ends when a state machine terminates, either reporting a pass or a violation. To examine the next path, instead of starting a new state machine and traversing from a root, the state machine and the path (both implemented using stacks) roll back to an earlier branch point and examine another unchecked branch edge just like a model checker [29]. This can avoid many redundant traversal steps.

Since the number of paths to be examined is exponential, we prune call graphs that are irrelevant to the parameters (e.g., function A, and lock l) specified in the rule. Similarly, we prune all basic blocks that are not related to these parameters. These two

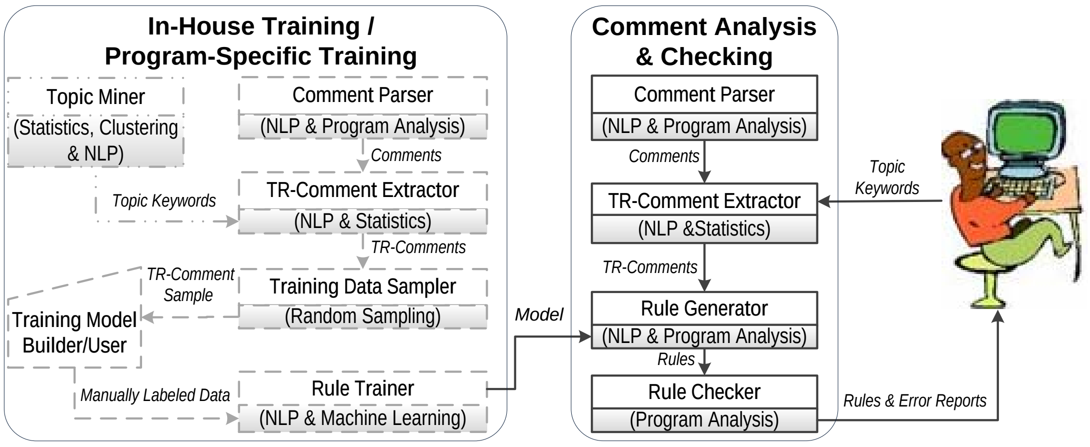


Figure 4: Above we see the iComment analysis process that extracts rules from comments and detects inconsistencies between comments and source code. Dashed lines and boxes indicate *optional* steps and data flows. The techniques used in each step are listed in () of each box. The rule training is done *in-house* by default with representative software, and optionally by users themselves if higher accuracy is desired. *TR-Comment* is potential Topic-related Rule-containing comments. The details of each step are presented in Section 5.

optimizations significantly reduce both the number of paths and the average length of paths being explored. Additionally, when the number of paths is still significantly high after the optimization above, for branch edges that are already explored via another path, we randomly sample some for further exploration.

To address the pointer aliasing problem, we combine flow insensitive pointer aliasing analysis [38] with local pointer analysis in the path currently being explored [13]. Currently, our aliasing analysis does not adequately handle items in arrays and pointer fields in structures, which introduces a few false positives in our inconsistency reporting (see Section 7).

Since this paper focuses more on comment analysis and the rule checking is similar to previous work [20, 26] except the false positive pruning and error ranking described below, we do not describe the details of the rule checking further.

4.4 How to Rank Results?

The goal of the rule checking process is not only to look for violations, but also to count the number of supports in source code (cases where the rule holds) for each rule inferred from comments. The numbers of rule violations and supports are used to calculate four values: (1) absolute number of supports, $numSupport$, (2) absolute number of violations, $numViolation$, (3) conditional support probability, $SP = \frac{numSupport}{(numSupport+numViolation)}$, (4) conditional violation probability, $VP = 1 - SP$.

These values are used for multiple purposes: determining whether violations indicate a bug or a bad comment, rule ranking, error ranking and false positive pruning. There can be many ways to rank results; we select one that is simple, intuitively sound, and also gives good empirical results.

In our result ranking, *we are more biased toward "bad comments" instead of bugs since code typically has gone through a series of testing whereas comments cannot be tested*. Therefore, the result ranking is as follows: if SP is above a certain threshold (e.g., 80%), the rule and the corresponding comment are likely to be correct. We use the number of support as the rule rank. In this case, any violation to the rule is likely to be a bug, with a confidence value of SP. We rank bugs based on the confidence. For bugs with the same confidence, those whose rules have higher $numSupport$ value are ranked higher.

For the remaining rules, their SPs are not above the specified threshold so we consider them as bad comment suspects. We rank them based on their VP. To avoid users examine all these suspects, we also use a cut-off threshold on the VP: if VP is not greater than a threshold (e.g., 75%), we do not report the comment as a bad comment. Of course, since this threshold is tunable, users can always set it lower (e.g., $1 - SP$) to get all bad comment suspects. In addition, reports without enough support, e.g., rules that are not practiced in the checked module, are automatically pruned by our checker.

While we try our best to rank the results based on the metrics described above, some reported bug suspects may be bad comments and vice-versa, because in many cases, it is hard to be sure whether a mismatch is a bug or a bad comment without developers' confirmation. However, *no matter whether they are bugs or bad comments, they are surely inconsistencies, which hurt software quality now or later*.

5. COMMENT ANALYSIS DETAILS

This section describes the method used in each step of the comment analysis, namely, the comment parser, the topic miner, the TR-comment extractor, the rule trainer and the rule generator.

5.1 Comment Parser

The process of comment parsing is shown in Figure 5. It first extracts all comments from a program and then breaks comments into sentences. This sentence separation task is non-trivial as it involves correctly interpreting abbreviates, decimal points, etc. Moreover, unique to program comments is that sentences can have "/" and "*" symbols embedded in one sentence. In addition, dot is frequently used between a structure and its member, e.g., Struct_A.Size, and should not be considered as sentence boundary. Furthermore, sometimes a sentence can end without any delimiter. Therefore, besides using regular delimiters, "!", "?", and ";", we use "." and spaces together as sentence delimiters instead of using "." alone. Additionally, we consider an empty line and end of comments as the end of a sentence.

Next, we use word splitters [9] to split each comment into words. We also break compound words such as interrupt_handler into two words, interrupt and handler, so that they can be tagged correctly as interrupt_handler is not in the dictionary. Afterward, we use Part-of-Speech (POS) tagging and semantic role labeling techniques [9,

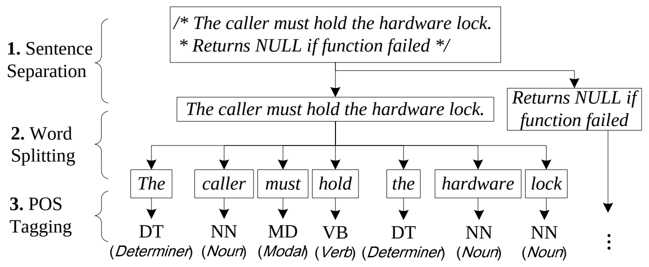


Figure 5: iComment Parser Process. The output of step 4, chunking and semantic role labeling (SRL), are not shown in this figure as they are already presented in Figure 3.

19, 32, 33] to tell whether each word in a sentence is a verb, a noun, etc., whether a clause is a main clause or a subclause, and what is the subject and the object, etc.

5.2 Topic Miner

To find hot topics in program comments, we provide two topic miners, *hot word miner* and *hot cluster miner*, that use NLP, clustering and simple statistics to automatically discover hot, popular topics from program comments. Both miners first use the NLP's POS tagging technique to filter out noisy words since words such as "we", "your" and "have" can prevent meaningful topic keywords from being mined. Words from subclauses also introduce noise. In addition, we need to filter explanation-based comments and concentrate on specification-based comments. Therefore, we consider only comment sentences with imperative words such as "should", "must", "need", "ought to", "have to", "remember", "make sure", and "be sure" and their variants.

After noisy words are filtered, the hot word miner uses simple word counting, i.e., counting the number of comments in which a word appears, to find popular nouns and verbs, which the user can use to determine hot topics.

The hot cluster miner is more sophisticated. Specifically, since many words are correlated with each other and are about the same topic, the hot cluster miner clusters correlated words together instead of using simple word count. For example, "lock", "acquire", "release", etc. are correlated words and are all about the same topic. For this purpose, we use mixture model clustering [45] that builds generative probabilistic mixture model to perform clustering because mixture model clustering is more expandable than k-mean based clustering techniques. Section 7.5 briefly summarizes the hot topics mined from the four evaluated open source software projects.

5.3 TR-Comment Extractor

Given a topic keyword (e.g., lock), the TR-comment extractor identifies all comments related to the selected topic, which will later be fed to the rule trainer and the rule generator. As mentioned earlier in Section 3, a comment that contains the topic keyword is not necessarily related to the topic. Therefore, we identify comments that contain not only the specified topic keyword (e.g., lock), but also at least one of the other words (such as acquire, release, hold, etc.) that are highly correlated to the topic keyword.

To achieve the functionality above, the TR-comment extractor first finds all words that are correlated to the specified topic keyword, i.e., words that appear frequently and mostly in the same comment with the specified topic keyword. For every word that has appeared in the same comment as the topic keyword at least once, we compute its correlation to the topic keyword using the *cosine* metric that is commonly used in statistics and data mining to measure the correlation of two items. The *cosine* of a topic word A and word B is calculated as

	Linux			Mozilla		
Rank	**Verb**	**Cosine**	**Freq**	**Verb**	**Cosine**	**Freq**
1	hold	0.182	598	hold	0.161	236
2	acquire	0.084	110	acquire	0.097	55
3	call	0.076	535	unlock	0.071	35
4	unlock	0.067	108	protect	0.065	45
5	protect	0.052	90	call	0.047	163
6	drop	0.047	113	enter	0.044	29
7	release	0.041	140	scope	0.041	41
8	contend	0.034	9	contend	0.034	2
9	sleep	0.032	72	wait	0.033	39
10	grab	0.031	49	release	0.030	55

Table 3: The top 10 words correlated to the topic keyword "lock". An interesting observation from the result is that the correlated words are very similar between two different programs, Linux and Mozilla. This indicates that programmers use similar words in comments for similar topics, providing a good evidence to explain the good results of cross-software training experiments (shown in Section 7.3.2) and our claim that our training process can be done in-house using representative software.

$$cosine(A, B) = \frac{P(A, B)}{\sqrt{P(A)P(B)}}$$

where $P(A, B)$ is the probability that word A and word B appear in the same comment, $P(A)$ and $P(B)$ are, respectively, the probabilities that word A and B appear in a comment sentence. We also tried other correlation metrics such as simple frequency count, LIFT, and Jaccard Coefficient, all of which are similar to or worse than cosine in terms of accuracy.

In addition to selecting a good correlation measure, we need to address another challenge — counting *different tenses* of a verb and *singular and plural forms* of nouns as the same word. We address this problem by automatically querying a dictionary.

After the above treatments, the TR-comment extractor selects the top n (default value is 10) words correlated to the topic keyword. Using these correlated words, we extract all comments that contain the topic keyword (e.g., lock) and also at least one of the n correlated words (e.g., hold, acquire, etc). Doing so allows us to filter out topic-unrelated comments such as a comment from Mozilla *"file locking error"* because they do not contain any word correlated to "lock".

Table 3 shows the top 10 ranked correlated verbs for lock-related comments in Linux and Mozilla. Compared to the simple word frequency measure, the correlation metric, cosine, is much better since some of the words are not frequently used but they are almost always used in the same comment with word "lock".

5.4 Rule Trainer

As described in Section 4, the goal of the rule trainer is to use a small set of manually labeled (manually mapped to a rule template) TR-Comments from some representative software to generate a model to analyze unlabeled TR-Comments from the same or different software. We use a decision tree classifier as our model to map a comment to a rule template. In data mining and machine learning, a *decision tree* (also referred as a classification tree or a reduction tree), is a predictive model; that is, a mapping from observations about an item to conclusions about its target value. In these tree structures, leaves represent classifications and branches represent conjunctions of features that lead to those classifications [28]. The machine learning technique for inducing a decision tree from training data is called *decision tree learning*.

We use a standard off-the-shelf decision tree learning algorithm called C4.5 Revision 8 [34], implemented in the software package

Weka [41], that does over-fitting pruning to prevent the tree from being over-fitted just for the training set. *This technique makes the tree general to unlabeled data and can tolerate mistakenly labeled training data.* Figure 6 shows the top of the decision tree model *trained* from a small set of manually labeled lock-related Linux comments.

Feature selection is an important factor for the accuracy of the decision tree learning algorithm. Essentially, features are used in a decision tree classifier to determine how to traverse the decision tree for a given input. The feature pool used in iComment's rule trainer can be divided into two categories. The first feature category is typical text classification features that are widely used in text classification studies [40]. The second feature category contains features that are unique to comment analysis but are general to different comment topics, different rule templates and different software.

In the following, we briefly describe the rationale for selecting some of the important features.

(1) Comment Scope: In program analysis, variable and function scopes are important concepts. Similarly, comment scope plays an important role in our comment analysis. We define two types of comment scopes, i.e., global scope and local scope. If a comment is written outside a function, we define its scope as global. If a comment is written within the body of a function, its scope is local to that function. The rationale for choosing this feature is that comments of global scope usually indicate rules related to the use of the function, while comments of local scope usually imply rules about the implementation of the function.

(2) Conditional Rules: Three conditional rules related features (Preposition Leading, Contain Whether-Prep, W-Tagged) indicate whether a comment contains any preposition or condition. These three features help determine if a comment is conditional. The rule templates supported by our rule checker currently do not support conditional rules.

(3) Modal Word Class: This feature indicates whether a comment contains a word in a Modal Word Class. There are three modal word classes: (i) *Imperative Class*, containing "must", "should", "will", "ought", "need", etc. (ii) *Possible Class*, containing "can", "could", "may", "might", etc. (iii) *Other Class*, not containing any of the words above. The reason for choosing this feature is that imperative modal words are more likely to imply rules, while words like *may* and *could* usually do not indicate rules.

(5) Application Scope: Application scope is decided by whether a comment expresses a pre-condition, post-condition or an in-function-condition of a routine: (i) *Prior Routine*: the condition should be satisfied before calling a routine, just like a Linux's comment "The queue lock with interrupts disabled must be held on entry to this function". (ii) *Post Routine*: the condition should be satisfied after a routine returns, e.g., a Linux's comment "This function returns with the `chip->mutex` lock held" . (iii) *In Routine*: the condition should be satisfied within a routine, as in a Linux's comment "As the `kmap_lock` is held here, no need for the wait-queue-head's lock". Currently, we simply use keywords to determine the application scope of a comment. Specifically, we use keyword set *(entry, call)* for *Prior Routine*, *(exit, complete, close, return)* for *Post Routine*, and comments without any word in these two sets are considered having value *In Routine* for this feature.

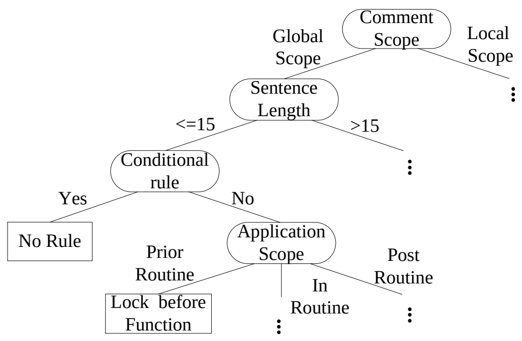


Figure 6: The top of the decision tree model built automatically by iComment trainer from Linux's lock-related comments.

5.5 Rule Generator

The functionality of the rule generator is to generate a concrete rule from an unlabeled TR-comment. Rule generation includes two steps: first it uses the trained decision tree classifier to map the TR-comment into a rule template, and then fills the parameters of the rule template. The first step is straightforward: once the decision tree is trained, it automatically "classifies" a comment to a rule template.

The second step needs to find the function names, variable names, etc. that are required to fit into the rule template. Specifically, the rule generator obtains the following information using simple program analysis and heuristics:

What is the function name? For function-related rules (such as rule template 1, 2, 9 and 10 listed in Table 2), we need to extract the function name for each rule. For a comment written outside any function, the function name can be found right after the comment by using a source code parser. For a comment written within a function, the function name can be found from the function declaration before the comment using a source code parser. For templates 9 and 10, the second function name, F_B usually can be found in the comment itself after certain words such as "from".

What is the variable name? The variable name (e.g., a lock name) of a rule is usually the object/patient of the action verb in the main clause of the corresponding comment. Therefore, we can automatically extract it after applying NLP's Semantic Role Labeling technique (described in Section 2).

Is a rule positive or negative? By identifying the action verb and negation words, such as "not", in the main clause, we can determine whether the rule is positive. By default, a rule is positive. We consider a rule as a negative rule only if the comment sentence contains a negation word, i.e., *barely, hardly, neither, no, none, not, nothing, nobody, n't, scarcely*, and *without*, and the negation word appears in the same clause as the action verb.

After the rules are generated and rule parameters are filled, rules with high confidence (computed by the decision tree classifier) are provided to the rule checker (described in Section 4.3) to check for comment-code inconsistencies.

6. EXPERIMENTAL METHODOLOGY

Evaluated Software. To demonstrate the effectiveness of iComment, we analyze comments of two representative and important topics, lock-related and call-related, in *four* large open source software projects, Linux, Mozilla, Wine and Apache as listed in Table 4.

We perform two types of experiments to examine the accuracy of our rule generation from comments.

(1) Software-specific training. In this set of experiments, we use models trained from each software system to analyze comments from the *same* software. Specifically, for large software such as Linux and Mozilla, we randomly sample 20% TR-Comments and

Software	LOC	LOM	Language	Description
Linux	5.0M	1.0M	C	Operating System
Mozilla	3.3M	.51M	C&C++	Browser Suite
Wine	1.5M	.22M	C	Program to Run WinApp on Unix
Apache	.27M	.057M	C	Web Server

Table 4: Evaluated software. LOC is lines of code and LOM is lines of comments. We count "lines of code" as the entire size of the program, including comments. All numbers are counted with copyright notices and blank lines excluded. WinApp means Windows applications. We use the latest versions of these software systems.

Software	Lock related		Call related	
	Total #TR-Comments	Training # (%)	Total #TR-Comments	Training # (%)
Linux	2070	242 (12%)	812	119 (15%)
Mozilla	720	65 (9%)	284	40 (14)%
Wine	91	27 (30%)	92	22 (24%)
Apache	129	28 (22%)	22	–
Total	3010	362 (18%)	1210	181 (18%)

Table 5: Training data (randomly selected) sizes used in software-specific training. TR-comments are extracted by using NLP and statistics techniques as shown in Section 5.3. As there are only 22 call-related TR-comments in Apache, we manually label all instead of using machine learning. The average training data percentage is computed as the average of all percentages, not the percentage of the total (which is actually smaller).

then manually label these comments. For small software such as Wine and Apache, we manually label a higher percentage (20-30%) of TR-Comments since they contain much fewer TR-Comments (fewer than 150).

Similar to previous studies using machine learning techniques [28, 43], we divide manually labeled comments into two groups, one used for training, *the other used for testing to automatically measure the accuracy of our trained rule generators.* The training data size for each program is listed in Table 5, while the default test set size is 5% of all TR-comments. To understand the sensitivity of the training data set size, we also evaluate its effect on accuracy by changing the training data set size from 2% to 15% of the total TR-Comments.

Note that our rule generator is also applied on the remaining 80% *unlabeled* TR-Comments to extract rules, all of which are then used by the rule checker to detect comment-code inconsistencies. Actually as shown in the next sections, majority of the extracted rules come from the *unlabeled* TR-comments that are not used in our training.

(2) Cross-software training. To demonstrate that our rule generator trained with sample TR-Comments from one piece of software can also be used to analyze comments in other software, we conduct a series of experiments using cross-software training. Specifically, we use the rule generator trained from Linux's comment samples (the same size as listed in Table 5) to analyze comments in the other three software systems, Mozilla, Wine and Apache. We also combine Linux and Mozilla's training data to build another rule generator to analyze comments in Wine and Apache.

Evaluation Measures. To evaluate the accuracy of our rule generation, we use widely used standard measures. For the overall measure, we use three metrics: *Accuracy Percentage, Kappa*, and *Macro-F Score*. Accuracy Percentage (AP) measures the overall percentage of our classification accuracy, it is simply defined as:

$$\text{AP} = \frac{\text{Total Number of Correctly Labeled Comments}}{\text{Total Number of Comments Given for Labeling}}$$

Kappa (κ) is a statistical measure of inter-rater reliability. It measures the agreement between two raters each of which classifies N items into C mutually exclusive categories. We use it to measure the agreement between a rater produced by iComment and the oracle rater that labels all comments correctly. Kappa is defined as: $\kappa = \frac{pr(a)-pr(e)}{1-pr(e)}$, where $pr(a)$ is the percentage of correctly labeled comments, and $pr(e)$ is the percentage of correctly labeled comments due to pure chance.

Macro-F Score is the mean of the *F-scores* of different categories, where F-score is a combination metric of *precision* and *recall*. Precision is defined as ($P = \frac{T_+}{T_+ + F_+}$), recall is defined as ($R = \frac{T_+}{T_+ + F_-}$), and F-score is defined as ($F1 = \frac{2PR}{P+R}$), where T_+, T_-, F_+ and F_- are, respectively, true positives, true negatives, false positives and false negatives.

As shown earlier in this section, AP, as well as other measures, is automatically measured on the manually labeled 5% test data set, which are not used for training.

7. EXPERIMENTAL RESULTS

7.1 Overall Results

Table 6 shows the overall comment analysis and inconsistency detection results of iComment. Our tool extracts a total of 1832 rules from comments in the four evaluated programs related to the two topics: lock-related and call-related. The accuracy of our rule extraction is reasonable (90.8-100%) (measured automatically on the manually labeled test set). The detailed accuracy results will be presented later in Section 7.3.1. As expected, the number of rules extracted from each software is positively correlated to the software size.

Using the rules automatically inferred from comments (without any manual examination), iComment's rule checker reported a total of 98 comment-code inconsistencies, 60 of them are true inconsistencies (bugs or bad comments), including 33 new bugs and 27 bad comments from the latest versions of our evaluated software with 12 bugs and 7 bad comments already confirmed by the corresponding developers. We count bugs based on their fixes. Therefore, even if a piece of code violates multiple different comments, but if it requires only one fix, we count it as *one* bug.

Many (at least 37) inconsistencies are very hard to detect by inferring from source code alone (even with the most sophisticated techniques) because different parts of the source code are actually consistent with each other. But the code does not match with the comments. A more detailed discussion can be found in Section 8.

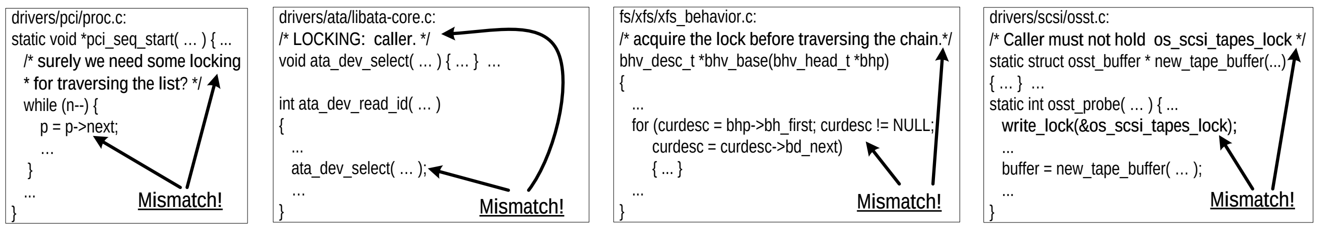
The above results clearly indicate that comments provide useful, redundant and independent information to check against source code for bugs and bad comments, both of which affect programs' robustness now or later.

We manually examine each reported inconsistency and the related source code to decide if the report is a true inconsistency, which allows us to count the false positive rate (38.8%). Additionally, we try our best to determine if a true inconsistency is a bug or a bad comment, which is much more difficult. Therefore, we also submit the reports to let the developers judge if an inconsistency is a bug or a bad comment, which both hurt software quality.

Although iComment has a reasonable overall false positive rate (38.8%), there is still space to further improve the accuracy. Many of the false positives are caused by the following two reasons and can be eliminated or reduced. First, since our rule checker does not adequately handle aliasing for array elements, pointers in structures and function pointers, they introduce some false positives. Reducing these false positives requires enhancing the pointer alias-

Software	Total					Lock-Related			Call-Related		
	Inconsistencies	Bugs	BadCom	FP	Rules	Bugs	BadCom	Rules	Bugs	BadCom	Rules
Linux	**51 (14)**	30 (11)	21 (3)	32	1209	29(10)	10(2)	990	1(1)	11(1)	219
Mozilla	**6 (5)**	2 (1)	4 (4)	3	410	1(1)	2(2)	277	1	2(2)	133
Wine	**2**	1	1	3	149	1	0	80	0	1	69
Apache	**1**	0	1	0	64	0	0	62	0	1	2
Total	**60 (19)**	33 (12)	27 (7)	38	1832	31(11)	12(4)	1409	2(1)	15(3)	423

Table 6: Overall results. Inconsistencies is the total number of validated code-comments mismatches. BadCom is the number of bad comments. FP is the number of false positives, which is not included in the number of inconsistencies. Numbers in "()" are the number of inconsistencies (bugs and bad comments) confirmed by developers. The other 41 inconsistencies are reported, but still waiting for confirmation from developers.


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

int ata_dev_read_id( ... )
{
  ...
  ata_dev_select( ... );
  ...
}
```
```
fs/xfs/xfs_behavior.c:
/* acquire the lock before traversing the chain.*/
bhv_desc_t *bhv_base(bhv_head_t *bhp)
{
  ...
  for (curdesc = bhp->bh_first; curdesc != NULL;
       curdesc = curdesc->bd_next)
       { ... }
  ...
}
```
```
drivers/scsi/osst.c:
/* Caller must not hold  os_scsi_tapes_lock */
static struct osst_buffer * new_tape_buffer(...)
{ ... }  ...
static int osst_probe( ... ) { ...
  write_lock(&os_scsi_tapes_lock);
  ...
  buffer = new_tape_buffer( ... );
  ...
}
```

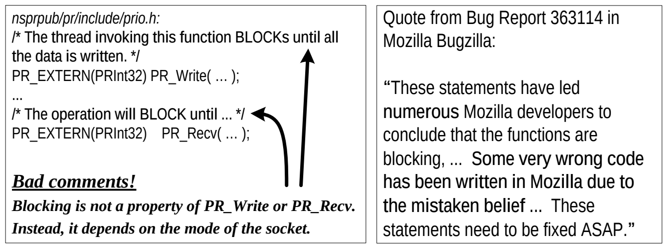
(a) The comment says a lock is needed when the list is traversed. But there is no lock acquisition in the code.

(b) Developers replied to us **"there is a race there. The direct call to `ata_dev_select()` in `ata_dev_read_id()` is there for mostly historical reasons."**

(c) Developers confirmed **"that comment is a left over from the original version ... It should probably be removed as it is mis-leading."**

(d) After we reported this inconsistency, the developers confirmed that the comment above `new_tape_buffer()` is wrong.

Figure 7: Four detected inconsistency examples - two *new* bugs and two bad comments in the latest version of Linux. All of them are recently confirmed by Linux developers.


```
nsprpub/pr/include/prio.h:
/* The thread invoking this function BLOCKs until all
the data is written. */
PR_EXTERN(PRInt32) PR_Write( ... );
...
/* The operation will BLOCK until ... */
PR_EXTERN(PRInt32)    PR_Recv( ... );
```
Bad comments!

Blocking is not a property of PR_Write or PR_Recv. Instead, it depends on the mode of the socket.

Quote from Bug Report 363114 in Mozilla Bugzilla:

"These statements have led numerous Mozilla developers to conclude that the functions are blocking, ... Some very wrong code has been written in Mozilla due to the mistaken belief ... These statements need to be fixed ASAP."

Figure 8: Two bad comments in Mozilla that caused many new bugs, including #355409 in Mozilla. Some wrong code was written because the programmers were misled by these bad comments, as acknowledged in Mozilla's Bugzilla bug report shown on the right.

ing analysis in our rule checker. The other cause of false positives is incorrect rules produced by the rule generator from comments. More training data, better feature selection and a better classification algorithm can reduce the number of such false positives.

7.2 Bugs and Bad Comments Examples

Besides the bugs and bad comment examples shown in Introduction, here we show four more examples of the detected new bugs and bad comments that are confirmed by the corresponding developers, and two manually discovered bug-causing bad comments (bad comments that have caused new bugs).

Bug Examples: Figure 7(a) presents a bug confirmed and currently fixed by Linux developers. Before the while loop accessing a linked list pointed by `p`, "surely we need some locking for traversing the list", as specified by the comment. However, no locking is used in function `pci_seq_start()` for accessing the list. iComment can automatically convert this comment to rule "a lock must be claimed in `pci_seq_start()`", which is an approximation of the meaning of the comment. By comparing the code and the rule, we can detect this bug.

Another bug detected in the latest Linux version and also recently confirmed by the Linux developers is shown in Figure 7(b). The comment indicates that the caller of `ata_dev_select()` should be responsible for locking, but in the code no lock is acquired before calling it. iComment automatically maps the comment to the rule "a lock must be claimed before entering function `ata_dev_select()`" and automatically compares the rule against the code to detect the bug. After we reported this mismatch, Linux developers confirmed this is a data race bug.

Bad Comment Examples: Figure 7(c) shows a bad comment example detected by iComment in the latest version of Linux. The comment says that the lock should be acquired in the function before accessing the chain. However, the code does not follow the comment. After we reported this inconsistency, *the Linux developers replied to us that the comment is a leftover from the original version and this comment should be fixed or removed since it can mislead other programmers to introduce bugs later.*

Figure 7(d) shows another bad comment example detected by us. According to the comment, lock `os_scsi_tapes_lock` *must not* be acquired before calling function `new_tape_buffer()`. However, the code calls the function with the lock held. After the inconsistency was reported, Linux developers confirmed "the comment over `new_tape_buffer()` is wrong."

Figure 8 presents two bad comments from a series of eight bad comments in Mozilla that has caused many new bugs, including the one in the bug report #355409 of Mozilla's Bugzilla. As pointed out by the developer in his/her post, although whether `PR_Write` and `PR_Recv` are blocking depends on the blocking property of the sockets these two functions are used on, many developers mistakenly conclude the functions are always blocking after reading the bad comments. These bad comments negatively affect software reliability and software development effectiveness, therefore it is important to detect them and fix them promptly. Besides `PR_Write` and `PR_Recv`, six other functions, namely, `PR_Read`, `PR_Writev`, `PR_Send`, `PR_RecvFrom`, `PR_SendTo`, and `PR_AcceptRead`, have similar false comments. We found these bad comments by manually examining Mozilla's Bugzilla database. Detecting them automatically remains as our future work.

7.3 Rule Generation Accuracy and Sensitivity

This section presents our rule generation accuracy results and training sensitivity results, for both software-specific training and cross-software training.

7.3.1 Software-Specific Training

Table 7 shows the overall rule generation accuracy using Software-specific training. We can achieve higher than 90% Accuracy Percentage (AP) and higher than 80% $Kappa$ for all software. As expected, the accuracy increases as the training data size increases. Figure 9 shows the curves for analyzing lock-related comments in Linux and Mozilla. The curves for the other software and call-related comments are similar.

Measures	**Linux**	**Mozilla**	**Wine**	**Apache**
AP	90.8%	91.3%	96.4%	100%
Kappa	0.85	0.87	0.94	1
M-F	0.89	0.93	0.95	0.67

Table 7: Software-specific training accuracy for lock-related rules. The results for call-related rules are similar. M-F is Macro-F score.

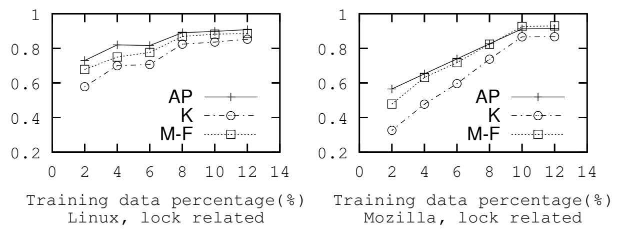


Figure 9: Training sensitivity. AP denotes Accuracy Percentage, K denotes Kappa and M-F denotes Macro-F score.

7.3.2 Cross-Software Training

Table 8 shows that cross-software training can achieve reasonable accuracy in comment analysis. Using Linux alone for training, we can achieve accuracy of 78.6% to 83.3% on the other three applications. Adding Mozilla's training data can increase the accuracy from 78.6-83.3% to 88.9-89.3%. Such results are not a big surprise since programmers tend to use similar words to describe similar topics as demonstrated in our correlation analysis results shown in Table 3 in Section 5.3. These results indicate that the training task can be done in-house with representative software before releasing iComment to users to analyze comments in their software.

Training Software	**Mozilla**		**Wine**		**Apache**	
	AP	**M-F**	**AP**	**M-F**	**AP**	**M-F**
Linux	81.5%	0.81	78.6%	0.78	83.3%	0.73
Linux+Mozilla	-	-	89.3%	0.87	88.9%	0.92

Table 8: Cross-software training accuracy (M-F is Micro-F score): We use the model trained from sample comments in one or two software to analyze the other software. The accuracy in other metrics is similar.

7.4 Comment Analysis Time

iComment can automatically analyze and extract rules from large code bases with millions of lines of code and comments in about one and half hours. Also as shown in Table 9, it takes about 59 minutes to analyze Linux lock-related comments and to check for comments-code inconsistencies. For Mozilla, Wine and Apache, the analysis times are 24 minutes, 12 minutes and 1 minute, respectively.

To build the models for the four pieces of software, we manually labeled 362 lock-related comments, and 181 call-related comments. It took us about 137 minutes to label all of them, which enabled iComment to automatically label 4220 comments and detect 60 inconsistencies. As we demonstrated in the previous section that cross-software rule generation produces decent accuracies, models built using these manually labeled comments can be used for labeling comments in other software.

Software	All comments	Lock-related comments				
	Parser	Extractor	Sampler	Trainer	Generator	Checker
Linux	31m36s	2m1s	0.8s	0.17s	3.5s	56m51s
Mozilla	30m20s	1m37s	0.3s	0.03s	1.1s	22m28s
Wine	10m	29s	0.3s	0.02s	0.8s	12m24s
Apache	2m12s	8s	0.3s	0	0.2s	0.4s

Table 9: Time for analyzing and checking lock-related comments. The analysis and checking time for call-related comments is in the same order. The parser time is for analyzing all comments, regardless of topics, in a piece of software. "m" stands for minute, and "s" denotes second.

7.5 Hot Topics

Table 10 presents the hot keyword mining results of selected Linux modules and Mozilla. Table 11 and Table 12 show the hot cluster mining results of the Linux kernel module and Mozilla respectively. In general, both lock and call rank high on the hot keyword miner and the hot cluster miner on comments from all evaluated software. For example, our hot keyword miner shows that "lock" is the most ranked word in the kernel module and is ranked second in the memory management (mm) module. According to our hot cluster miner, kernel module and Mozilla contain up to 5 (out of 10) clusters having "lock" in their topic keywords.

Rank	kernel	mm	drivers	Mozilla
1	**lock**	page	receiv	**call**
2	**call**	**lock**	copi	check
3	thread	**call**	gener	set
4	task	**caller**	licens	return
5	signal	cach	gnu	file
6	**held**	memori	public	data
7	copi	check	**call**	**function**
8	timer	list	check	case

Table 10: Hot keyword mining results. The parser uses stemming techniques to use the root of words to represent a group of words that share the root. For example, devic represents device, devices, etc.

Similarly, keywords related to function calls also appear in a significant portion of comments. "Call" is the most ranked word in Mozilla and among the top 7 for all three Linux modules. Our hot cluster miner shows that the Linux kernel module and Mozilla have at least one "call" cluster.

In addition to lock and call, many other topics are common and can potentially be analyzed by iComment for inconsistency detection. While some of these topics are general to all software, such as memory allocation and the two topics discussed above, different programs have their specific hot topics. For example, interrupt is a hot topic in comments from the Linux kernel module, whereas "error", "return", and "check" are hot topics in Mozilla. Moreover, a substantial percentage of comments in the kernel module contain keywords "thread", "task" or "signal", whereas many comments in the memory management module contain keywords "page", "cache", or "memory".

%	Key 1	Key 2	Key 3	Key 4	Key 5
14.6%	**call**	**held**	**lock**	read	sem
11.1%	check	runqueu	alloc	process	sure
11.1%	sure	thread	signal	up	wake
10.8%	task	**caller**	set	**lock**	time
10.0%	write	swap	**lock**	save	etc
9.7%	interrupt	**lock**	out	disabl	list
9.2%	**lock**	releas	return	move	tabl
8.9%	copi	gener	receiv	along	licens
8.1%	schedul	clock	thing	restart	timer
6.5%	cpu	timer	structur	handler	irq

Table 11: Hot cluster mining results of the Linux kernel module. % is the percentage of comments that belong to each cluster.

%	Key 1	Key 2	Key 3	Key 4	Key 5
12.8%	sure	check	return	error	chang
10.5%	rememb	**function**	data	kei	type
10.3%	**call**	thread	add	**lock**	match
9.8%	up	list	string	name	end
9.7%	out	go	note	befor	find
9.5%	set	frame	content	delet	bit
9.5%	file	first	into	buffer	line
9.5%	case	code	new	mai	size
9.3%	tabl	same	point	select	cach
9.2%	creat	window	copi	messag	happen

Table 12: Hot cluster mining results of Mozilla. % is the percentage of comments that belong to each cluster.

8. DISCUSSION

8.1 What other comment topics can iComment analyze and check?

Besides the lock-related and call-related topics, there are definitely many other hot topics, some shown in Section 7.5, that iComment can check in our future work. For example, interrupt is a hot topic in Linux, accounting for 9.7% or more in different modules. Many interrupt-related comments can be easily found in Linux, such as "interrupts must be unmasked", "this function must not be called from interrupt or completion handler", etc. The only extra requirement to analyze this topic is to manually label some comment samples and extend the rule checker to check these rules. Another hot topic is memory allocation/deallocation such as "allocating a certain buffer before calling some function".

8.2 Can iComment detect inconsistencies that cannot be detected by previous work?

Many bug detection tools [16, 17, 24, 26] have been proposed to extract rules or invariants from source code or execution traces to detect bugs. For example, recent work [24] uses probability and logic to automatically infer specifications from source code to detect bugs.

While previous work in rule extraction can automatically infer some programming rules from source code and detect inconsistencies among source code, our approach provides a complementary capability in the following aspects.

First, in addition to bugs, our approach can be used to detect bad comments that can mislead programmers to introduce new bugs in subsequent versions. None of the previous work has such capability. Therefore, none of the 27 bad comments detected by iComment, including the 3 examples shown in Figure 2, Figure 7(c), and Figure 7(d), can be identified by previous work.

Second, previous work looks for inconsistencies among source code whereas ours examines inconsistencies between code and comments. As a result, if different parts of source code are consistent with each other, but they do not match with the correct comment, then iComment can detect such bugs, but it will be hard for previous work that uses source code alone to detect. For example, if the callers of a function are supposed to acquire a lock, but all of the callers do not acquire the lock, then all these callers are consistent among themselves, which makes it extremely difficult for the previous work to detect the bugs. With the help of the comments that documented the correct behavior, we can find that the comment is not consistent with the code to detect these bugs.

Third, for many types of rules, including lock-related and call-related, if the code does not have statistical support, it will be very difficult for previous work to infer such rules. Several bugs iComment detected belong to this category, including the two bugs shown in Figure 1 and Figure 7(a). For example, the list is traversed there in the function only *once* as shown in Figure 7(a). Figure 1 shows that a lock must be acquired before calling function `reset_hardware()`, but there are only 2 places in Linux where `reset_hardware()` is called. Therefore, it is very difficult to infer these rules from only the source code. But such rules are usually specified by the programmer in comments, so they can be extracted and compared against the source code for any inconsistency.

Fourth, many rules, such as "a function can only be called from function A" virtually cannot be inferred from source code alone, because if a function is called from A 1000 times, and is called from B once, we cannot infer that the function must be called from A only.

Therefore, many (at least 37) of the inconsistencies iComment detected cannot or are very difficult to be detected by previous work using only source code. These inconsistencies include all of the 27 bad comments and at least 10 of the bugs.

On the other hand, the bug example detected by iComment shown in Figure 7(b) may potentially be detected by previous work because function `ata_dev_select()` is called from several places in the program and many of the places acquire the lock before calling `ata_dev_select()`.

8.3 How much manual effort is needed from the user?

iComment requires the following three different levels of human involvement. First, by default (if the topic is already trained in-house by us using representative software), iComment requires *minimal* human involvement: the user only needs to select a topic and provide the software-specific actions (e.g., the lock function names) for the rule checker to detect code-comment inconsistencies. The latter is common to most existing rule checkers [20], and are not specific to iComment. Second, if the user would like to improve the rule generation accuracy, he/she can manually label some sample TR-Comments from his/her program and then use these samples to retrain our decision tree classifier. Third, if the user wants to analyze comments on a topic currently not covered by iComment, in addition to training, the user needs to define his/her rule templates and then extend the rule checker in a way similar to the Stanford Meta Compiler by Engler et al. [20] to check these rules against the source code.

8.4 Can we use other rule checking methods?

Although iComment uses static checking to detect inconsistencies, it is also quite conceivable that rules extracted from comments can also be checked dynamically by running the program.

9. RELATED WORK

We briefly describe closely related work.

Extracting rules and invariants from source code and execution behaviors. Previous work [16, 17, 24, 26] extracts rules or invariants from source code or execution traces to detect bugs. Different from these studies, our work is the first (to the best of our knowledge) to extract program rules from *comments*. Our work well complements these previous approaches because, compared to source code, comments are much more explicit and descriptive, directly reflecting programmers' assumptions and intentions. Moreover, our work can also detect bad comments, which can mislead programmers to introduce bugs later (more discussion in Section 8.2).

Empirical study of comments. Several empirical studies in the software engineering field have studied the conventional usage of comments, the evolution of comments and the challenges of automatically understanding them. Woodfield, Dunsmore and Shen [42] 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 [22] studied the trend of the percentage of commented functions in PostgreSQL. Recent work from Ying, Wright and Abrams [44] shows that comments are very challenging to analyze automatically because they have ambiguous context and scope. None of them propose any solution to automatically analyze comments or to automatically detect comment-code inconsistencies.

Annotation language. Annotation languages [5, 7, 8, 11, 14, 18, 21, 30, 31, 36, 39] are proposed for developers to comment source code using a formal language to specify some special information such as type safety [46], information flow [30], performance expectations [31], etc. Although these annotation languages can be easily analyzed by compiler, they have their limitations. Compared to natural language, these languages are not as expressive or as flexible. They are mostly used to express only simple assumptions such as buffer length, data types, etc. Additionally, as it requires programmers to learn a new language to be able to use them, they are so far not widely adopted. Furthermore, millions of lines of comments written in natural language already exist in legacy code. Therefore, while we should encourage programmers to use annotation language, it is also highly desirable to seek alternative or complimentary solutions that can automatically analyze comments written in natural language. Our work well compliments the approach above and rules inferred by our iComment can be used to *automatically* annotate programs to reduce programmers' manual effort.

Automatic document generation from comments. Many comment style specification and tools are proposed and widely used to automatically build documentation from comments, e.g., JavaDoc [6], RDoc [10], Doxygen [3] and C#'s XML Comments [1]. Since they restrict only the format but still allow programmers to use natural language for the content (i.e., they are semi-structured like web pages), automatically "understanding" and analyzing these comments still suffer from challenges similar to analyzing unstructured comments.

Software debugging. Many debugging tools (for performance or correctness) [12, 15, 23, 25, 37] are proposed to detect various types of software bugs. iComment compliments research in this direction since we extract rules from comments and then detect inconsistencies between code and comments, indication of bugs or bad comments, both of which affect software robustness now or later.

10. CONCLUSIONS AND FUTURE WORK

This paper makes the first attempt to automatically analyze comments written in natural language and detect comment-code inconsistencies. Our tool iComment has inferred 1832 rules with high accuracy and also has detected 60 comment-code inconsistencies (33 new bugs and 27 bad comments), with 19 of them confirmed by the developers in the *latest* versions of Linux, Mozilla, Wine and Apache. These results demonstrate the effectiveness of iComment, which is a first step with promising results to inspire and motivate more research work in this direction.

Even though our work opens a brand new direction in automatically analyzing comments and documents written in natural language to improve software system robustness and reliability, there is still much room for further improvement as we discussed in Section 8, namely analyzing comments of other topics, using dynamic checking to detect errors, improving comment analysis accuracy, and evaluating other software. We hope that our work, in particular the problem of bad comments, can motivate ideas on designing some easy-to-learn and flexible comment languages that can be easily analyzed and checked against code for inconsistencies.

Finally, it also quite conceivable that some of our ideas and experience can be borrowed to automatically analyze other system documents written in natural languages, such as user manuals and user error reports, to extract information for many other purposes such as automatically tuning system performance, trouble-shooting system configurations and enhancing system security in addition to software reliability.

11. ACKNOWLEDGMENTS

We greatly appreciate our shepherd, Stefan Savage, for his invaluable feedback, precious time, and the anonymous reviewers for their insightful comments. We thank Dan Roth, Nick Rizzolo, Mark Sammons, the Cognitive Computation Group, Chengxiang Zhai, and Xuanhui Wang, for their tools, feedback and suggestions on natural language processing and machine learning related issues. We are also grateful to Tony Bergstrom, Matthew Yapchaian, and Chengdu Huang for their help on proofreading our paper. This research is supported by NSF CCF-0325603, NSF CNS-0347854 (NSF Career Award), NSF CNS-0615372 (NSF Delta Execution), DOE DE-FG02-05ER25688 (DOE Early Career Award), Intel gift grants, and Microsoft gift grants. Lin Tan was awarded an SOSP student travel scholarship, supported by the National Science Foundation, to present this paper at the conference. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of NSF, DOE, Intel or Microsoft.

12. REFERENCES

[1] C# XML comments let you build documentation directly from your Visual Studio .NET source files. http://msdn.microsoft.com/msdnmag/issues/02/06/XMLC/.

[2] CoNLL-2000 shared task web page – with data, software and systems' outputs availble. http://www.cnts.ua.ac.be/conll/.

[3] Doxygen - source code documentation generator tool. http://www.stack.nl/~dimitri/doxygen/.

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

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

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

[7] Lock_Lint - Static data race and deadlock detection tool for C. http://developers.sun.com/sunstudio/articles/locklint.html.

[8] MSDN run-time library reference – SAL annotations. http://msdn2.microsoft.com/en-us/library/ms235402.aspx.

[9] NLP tools. http://l2r.cs.uiuc.edu/~cogcomp/tools.php.

[10] RDoc - documentation from Ruby source files. http://rdoc.sourceforge.net/.

[11] Sparse - A semantic parser for C. http://www.kernel.org/pub/software/devel/sparse/.

[12] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[13] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[14] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking, SRC research report 159. ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-159.ps.

[15] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[16] 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 *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[17] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

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

[19] Y. Even-Zohar and D. Roth. A sequential model for multi class classification. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing*, 2001.

[20] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*.

[21] W. E. Howden. Comments analysis and programming errors. *IEEE Transactions on Software Engineering*, 1990.

[22] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in PostgreSQL. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*.

[23] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference*, 2005.

[24] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. R. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation*, 2006.

[25] T. Li, C. Ellis, A. Lebeck, and D. Sorin. On-demand and semantic-free dynamic deadlock detection with speculative execution. In *USENIX Annual Technical Conference*, 2005.

[26] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[27] C. D. Manning and H. Schütze. *Foundations Of Statistical Natural Language Processing*. The MIT Press, 2001.

[28] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[29] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedingts of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[30] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 2000.

[31] S. E. Perl and W. E. Weihl. Performance assertion checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993.

[32] V. Punyakanok and D. Roth. The use of classifiers in sequential inference. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*, 2001.

[33] V. Punyakanok, D. Roth, and W. Yih. The necessity of syntactic parsing for semantic role labeling. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2005.

[34] R. J. Quilan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[35] A. Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 1996.

[36] K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual, SRC technical note 2000-002. http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-2000-002.html.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 1997.

[38] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[39] N. Sterling. WARLOCK - A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[40] S. Teufel and M. Moens. Summarizing scientific articles – experiments with relevance and rhetorical status. *Computational Linguistics*, 2002.

[41] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques (2nd Ed.)*. Morgan Kaufmann, 2005.

[42] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[43] A. Yaar, A. Perrig, and D. X. Song. Pi: A path identification mechanism to defend against DDoS attack. In *IEEE Symposium on Security and Privacy*, 2003.

[44] A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: An exploration of eclipse task comments and their implication to repository mining. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*.

[45] C. Zhai, A. Velivelli, and B. Yu. A cross-collection mixture model for comparative text mining. In *Proceedings of the 2004 ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*.

[46] 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 *Proceedings of the 7th Symposium on Operating System Design and Implementation*, 2006.