aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs

Lin Tan
University of Waterloo
200 University Avenue West
Waterloo, ON N2L3G1, Canada
lintan@uwaterloo.ca

Yuanyuan Zhou
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093, USA
yyzhou@cs.ucsd.edu

Yoann Padioleau
Facebook, Inc.
1601 California Avenue
Palo Alto, CA 94304, USA
yoann.padioleau@facebook.com

ABSTRACT

Concurrency bugs in an operating system (OS) are detrimental as they can cause the OS to fail and affect all applications running on top of the OS. Detecting OS concurrency bugs is challenging due to the complexity of the OS synchronization, particularly with the presence of the OS specific interrupt context. Existing dynamic concurrency bug detection techniques are designed for user level applications and cannot be applied to operating systems.

To detect OS concurrency bugs, we proposed a new type of annotations – interrupt related annotations – and generated 96,821 such annotations for the Linux kernel with little manual effort. These annotations have been used to automatically detect 9 real OS concurrency bugs (7 of which were previously unknown). Two of the key techniques that make the above contributions possible are: (1) using a *hybrid* approach to extract annotations from both code and comments written in natural language to achieve better coverage and accuracy in annotation extraction and bug detection; and (2) automatically *propagating annotations* to caller functions to improve annotating and bug detection. These two techniques are general and can be applied to non-OS code, code written in other programming languages such as Java, and for extracting other types of specifications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.4.1 [**Operating systems**]: Process Management—*Concurrency, Deadlock*

General Terms

Documentation, Experimentation, Languages, Reliability

Keywords

Concurrency bug detection, Annotation languages, Interrupts, Operating systems, Static analysis

1. INTRODUCTION

Concurrency bugs are inevitable in multi-threaded programs as concurrency is inherently complex and programmers are trained to think sequentially when coding. Concurrency bugs not only cause severe damage [30, 55], but also are hard to detect due to their non-deterministic nature. The severity of the concurrency bugs increases as the popularity of multicore hardware makes multi-threaded programs more pervasive. Concurrency bugs in an operating system (OS) are particularly detrimental because OS failures caused by concurrency bugs, e.g., hangs and crashes, can bring down all software running on top of the OS. As operating systems inherently have to deal with concurrent activities and shared resources, they have a much higher percentage of concurrency bugs than application software [59]. A recent study shows that 19% of OS driver bugs are concurrency bugs [52].

Detecting concurrency bugs in an operating system is not well addressed and is particularly challenging mainly for two reasons:

- The OS specific interrupt context makes OS concurrency extremely complex and it has challenged the OS community for decades [11]. Figure 1 shows that a thread T holding lock L is interrupted by an interrupt handler T_H (referred to as *in interrupt context*), which needs to acquire the same lock. Since the handler T_H has preempted the thread T, T would not be rescheduled until the interrupt handler T_H finishes [10]. The interrupt handler T_H cannot finish its execution because it waits for the lock L that the thread T holds. Such a real deadlock bug has been found in the Linux kernel 2.6.12: in the function `do_entInt` in *arch/alpha/kernel/irq_alpha.c*, the developers forgot to call `local_irq_disable` to disable interrupts to prevent an interrupt handler from contending with other threads on the same lock (more details appear in Figure 5 and Section 2.5).

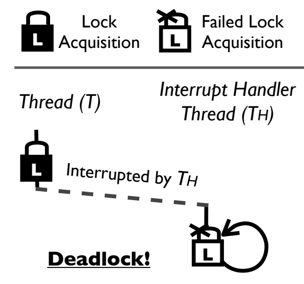

Figure 1: A deadlock. Since the interrupt handler T_H interrupted the thread T, thread T cannot be scheduled until T_H finishes; T_H cannot finish because it is waiting for T to release the lock L.

It is difficult for developers to avoid such interrupt related bugs. Typically, an OS has tens of thousands of device drivers [40] and interrupt handlers written by thousands of developers across several decades [46], with new interrupt handlers being added constantly. Additionally, interrupts can happen anytime during

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...$10.00.

linux/kernel/time/tick-oneshot.c:

```
/* ... Called with interrupts disabled. */
int tick_init_highres(void) { ... }
```

(a) The Original Version

```
/* ... Called with interrupts disabled.  */
int /*@IRQ(0, X)*/ tick_init_highres(void) { ... }
```

(b) The Annotated Version

Figure 2: Converting a comment in the Linux kernel into an annotation

the execution of a thread, creating numerous possible interleaving combinations. Therefore, it is difficult for a kernel developer to reason about the *numerous* possible interleaving combinations between a thread and the *large* amount of *evolving* interrupt handlers. Further, as interrupts are uncommon events ("anomalies"), they are likely to be neglected by developers, because developers are less good at enumerating and correctly implementing all possible anomalies than making the normal/main flow correct.

- The complexity of an operating system, the difficulty of instrumenting an OS and the large amount of drivers make bug detection for operating systems particularly difficult and cumbersome. Existing concurrency bug detection tools are mainly built for user level applications, and have not demonstrated their effectiveness of detecting concurrency bugs in operating system code. The complexity and sheer size of an operating system can incur a prohibitively high run-time overhead and instrumentation difficulties on dynamic approaches, which is undesirable.

1.1 State of Art

In order to detect concurrency bugs, many techniques have been proposed [5, 8, 17, 20, 25, 35, 36, 45, 47, 51, 54, 63, 65, 67, 68]. There are two main limitations of these techniques: (1) these tools do not consider the interaction with the OS specific interrupt handlers, thus missing the opportunity to detect related bugs; and (2) most of the effective concurrency bug detection tools are dynamic tools that are designed for user level applications. Detailed comparison with prior concurrency bug detection work is discussed in Section 6.2.

To address the two limitations above, *static approaches with interrupts in mind* would be a great solution for tackling operating system concurrency bugs. Toward using static approaches, many annotation languages [9, 15, 38, 39, 62, 66] are proposed to allow programmers to formally express their intentions and assumptions, e.g., where a lock is needed, etc. These annotations not only can be checked against code to detect software bugs, but also can *prevent* developers from introducing new bugs by making the intentions and assumptions explicit.

These annotation languages have made significant impact. For example, Microsoft's SAL annotations [38] helped to detect more than 1,000 potential security vulnerabilities in Windows code [4]. Seeing the success of SAL, Microsoft recently proposed new annotations including concurrency related annotations [4]. In addition, several other annotation languages, including Sparse [62] from the Linux kernel community, Sun's Lock_Lint [39], and SharC [3], express concurrency related concerns.

However, none of the annotation languages above fully express the concurrency assumptions that an OS needs, e.g., interrupt related assumptions. Ideally, we want to know the *preconditions* and *postconditions* regarding interrupts of every function. In other words, we want to know *whether interrupts should be disabled or enabled upon entry of a function, and whether the interrupts should be disabled or enabled upon exit of the function*. Considering that software can contain hundreds of thousands of functions, it is tedious and time-consuming to manually annotate all of these functions. The significant amount of effort involved in annotating programs can greatly limit the impact of annotation languages [16]. Therefore, while we should definitely encourage developers to write annotations, it would be *desirable to provide support for annotating new and legacy code*.

1.2 Idea and Contributions

Fortunately, it is feasible to extract interrupt related preconditions and postconditions from both source code and comments written in natural language. We will use examples to explain how we extract postconditions and preconditions in this section. The detailed extraction techniques are described in Section 2.3. For *postconditions*, if we know that `local_irq_disable` disables interrupts, we can infer that all functions that call `local_irq_disable` but not any interrupt-enabling-function also disable interrupts.

Preconditions can be inferred in multiple ways. It has been a common practice for programmers to write comments to directly express their intentions and assumptions [46, 60]. For example, the comment in Figure 2(a) states that interrupts must be disabled before calling `tick_init_highres`. This comment can potentially be expressed as annotation `/* @IRQ (0, X) */`, where `0` indicates that interrupts must be disabled before calling the function, and `X` indicates that interrupts can be either disabled or enabled upon exit of this function (Figure 2(b)). Section 2.4 describes how the postcondition, `X`, can be refined during the annotation propagation process.

Additionally, programmers often write code assertions such as `BUG_ON(!irqs_disabled())` to print an error message if interrupts are not disabled, indicating that they assume that interrupts must be disabled already. The function `run_posix_cpu_timers`, for instance, starts with `BUG_ON(!irqs_disabled())`, indicating that this function must be called with interrupts disabled. Although such dynamic assertions can help detect bugs, they are limited because (1) they require bugs to manifest in order to detect them, which is difficult for OS concurrency bugs; and (2) these assertions incur high runtime overhead, therefore are often disabled for production runs for better performance. If we could convert such assertions into annotations and check the annotations statically, such a static approach can complement dynamic assertions [69] to detect more bugs with no run-time overhead. For example, we add annotation `/* @IRQ (0, X) */` to function `run_posix_cpu_timers`, which enables us to detect the Linux kernel bug discussed earlier. Furthermore, we may infer interrupt related preconditions from source code by using statistical approaches similar to prior work [13, 31, 33].

We propose converting programmers' intentions inferred from the comments and code they write into formal annotations and use these annotations to detect interrupt related OS concurrency bugs. Two key techniques helped us generate annotations for all functions for effective bug detection: *hybrid annotation extraction from both comments and code* and *annotation propagation*. First, we infer annotations solely from comments. We then infer annotations from code only, e.g., from code assertions such as `BUG_ON`. Finally, we combine the annotations inferred from both comments and code to detect bugs. We also demonstrate that the annotations inferred from

comments and code complement each other. By combining them, we achieve better coverage and accuracy in annotation extraction, which help us detect more bugs more accurately. For effective bug detection, we also automatically propagate annotations from callee functions to caller functions when necessary.

Although this paper focuses on generating interrupt related annotations and detecting OS concurrency bugs, *the hybrid approach of extracting specifications from both comments and code can be applied to non-OS code, code written in other programming languages such as Java, and for extracting other types of specifications*. While this paper focuses on leveraging the extracted annotations to detect bugs, the annotations can be used for many other purposes, such as helping developers avoid bugs.

In total, we generate 96,821 interrupt related annotations from the Linux kernel, which are automatically propagated from a total of 245 *seed* annotations. These seed annotations are inferred directly from comments and code assertions with little manual effort (226 of which are from comments, and 24 of which are from code assertions). Only 5 of the seed annotations can be extracted from both comments and code assertions, meaning that the majority (221 from comments and 19 from code assertions) can only be extracted from one source. The result indicates that it is beneficial to infer annotations from both sources. We have used these annotations to detect 9 real bugs (7 of which were previously unknown), which are more than we could have detected by using annotations extracted from code alone or using annotations extracted from comments alone.

This work, *aComment*, makes the following contributions:

- Proposed a new type of annotations – interrupt related annotations – and generated 96,821 such annotations for the Linux kernel with little manual effort.
- Leveraged the new annotations to automatically detect bugs caused by the complex OS synchronization related to interrupt context.
- Applied a *general, hybrid* approach to extract specifications from both code and comments. Note that our prior work iComment [60] only extracts rules from comments (a background of iComment and a detailed comparison with iComment appear in Section 6.3).
- Used the interrupt related annotations to compare the coverage of code and comments regarding annotation extraction and bug detection.

2. DESIGN OF aComment

To help prevent and detect software bugs, our ideal goal is to annotate all functions with interrupt related annotations and use these annotations to check for bugs. However, this process is tedious and time-consuming; aComment automates this process. There are three steps in annotating all functions and using them for bug detection: (1) designing expressive and easy-to-use annotation languages, (2) converting comments and code into formal annotations, including propagating annotations to the callers of a function when necessary, and (3) verifying that these annotations are followed by the code.

Section 2.1 presents the three major challenges of aComment and an overview of our solutions. Section 2.2-2.4 describe how we perform the three steps described above respectively and how we address the three major challenges.

2.1 Challenges and An Overview of Our Solutions

This section discusses the three major challenges in annotating all functions and using the annotations for bug detection.

Converting comments and code into annotations.

Although it is promising to extract annotations from comments and code, it is quite challenging. First, comments are ambiguous and written in free form; developers can express the same meaning using different words, phrases, sentence structures, etc. It is difficult to automatically and precisely analyze comments to extract the correct annotations from them. Furthermore, we want the annotations generated by our aComment tool to be accurate, as these annotations are intended to be added back to the source code to help developers better understand the program to prevent them from introducing new bugs.

To address the challenges above, we improve our comment parser used in iComment [60], design new heuristics for extracting annotations, and manually verify all generated annotations. For each generated annotation, our analyzer shows the original comment and the surrounding code to allow a user to either accept, reject, or modify the extracted annotations (e.g., flip the annotation, change the function name, etc.). The 245 manually verified correct annotations can help us detect bugs, as well as guide the developers to prevent the introduction of new bugs. To extract annotations from code assertions (briefly described earlier in Section 1.2, and elaborated later in this Section), we built a scalable static analysis tool.

Dealing with scarceness of annotations.

Since not all functions have comments or code assertions stating their preconditions and postconditions, one cannot annotate all functions in a piece of software simply by extracting annotations from comments and code assertions.

To annotate all functions, we need to have the ability to propagate annotations of a function to the function's callers. For example, if we know that function `local_irq_enable` enables all local interrupts and that a direct caller of it does not disable interrupts, we can infer that the postcondition of its caller assumes that interrupts are enabled. It is tedious and time-consuming to manually perform this analysis as software contains tens of thousands of functions and their interaction is complex. It is challenging to make this process efficient and scalable. We propose a *bottom-up summary-based* annotation propagation technique to automate this process, which avoids analyzing a function repetitively (Details in Section 2.4).

Handling interrupt restoring functions.

It is insufficient for aComment to consider only interrupt disabling functions (e.g., `local_irq_disabled`) and interrupt enabling functions (e.g., `local_irq_enable`), as some functions (called interrupt restoring functions, e.g., `local_irq_restore`) restore a previously-saved interrupt state. We cannot treat such restoring functions as a simple interrupt disabling function or an interrupt enabling function. Thus, they have to be specially treated.

2.2 Annotation Language Design

As we are concerned with the OS synchronization in the special interrupt context, we design annotations in the following format: `@IRQ(Precondition, Postcondition)`, where `Precondition` and `Postcondition` can have one of the 4 values, i.e., `0`, `1`, `X` and `P`. The meanings of each of the 4 values are shown in Table 1(a). Value `P` indicates that a function, e.g., `local_irq_restore`, restores the saved interrupt state. We use `(X, P)` to indicate functions that restore a saved interrupt state, and all other 6 annotations that contain a value `P` is not accepted. Therefore, although there are 16 possible annotations, only 10 of them are accepted by aComment as shown in Table 1(b), and the rest of the 6 should not appear in aComment.

Value	Meaning
0	Interrupts are disabled.
1	Interrupts are enabled.
X	Don't-care: Interrupts are either disabled or enabled.
P	Interrupts are restored to the saved interrupt state.

(a) The meaning of the 4 annotation values

@IRQ (Pre, Post)	Meaning
@IRQ (0, 0)	Interrupts are disabled on entry and remain disabled on exit.
@IRQ (0, 1)	Interrupts are disabled on entry but are enabled on exit.
@IRQ (1, 0)	Interrupts are enabled on entry but are disabled on exit.
@IRQ (1, 1)	Interrupts are enabled on entry and remain enabled on exit.
@IRQ (X, X)	Either @IRQ (0, 0) or @IRQ (1, 1)
@IRQ (X, 0)	Don't-care on entry and interrupts are disabled on exit.
@IRQ (X, 1)	Don't-care on entry and interrupts are enabled on exit.
@IRQ (0, X)	Interrupts are disabled on entry and don't-care on exit.
@IRQ (1, X)	Interrupts are enabled on entry and don't-care on exit.
@IRQ (X, P)	Don't-care on entry and interrupts are restored to the saved state on exit.

(b) All valid annotations. 'Pre' stands for preconditions and 'Post' denotes postconditions.

Table 1: Proposed Annotations.

Software	#Sentence	#IRQSent
Linux	1,024,624	23,662
FreeBSD	420,013	11,117
NetBSD	680,650	23,942
OpenSolaris	535,073	8,074
Total	2,660,360	66,795

Table 2: Extracting annotations from comments is challenging. #Sentence is the total number of comment sentences. #IRQSent is the total number of comment sentences that contain the keyword 'interrupt' (case insensitive).

ID	Heuristic
1	<call> & <with> & <interrupt> (ordered)
2	<before> & <disable/enable> & <interrupt> (orderless)
3	<assume> & <disable/enable> & <interrupt> (orderless)

Table 3: Main heuristics used to extract annotations from comments. Names in <> are variables (defined in Table 4) which can expand into multiple words and their variants. The first heuristic requires the three variables to appear in the specified order while the other two do not.

Typically, there are two ways to incorporate annotations in software: (1) adding annotations in comments so that they are backward compatible, or (2) introducing new language keywords, which can ensure that the annotations evolve with code but is not backward compatible. Either would work for aComment. We choose the first approach for backward compatibility.

2.3 Annotation Extraction

This section describes how we extract preconditions from the comments and source code (more specifically, code assertions). We call these directly extracted annotations *seed annotations*. Section 2.4 presents how to propagation these seed annotations to their caller functions when necessary, and postconditions are determined during this propagation process.

Extracting Annotations from Comments.

Annotation extraction from comments consists of two steps: (1) *comment extraction*: extracting *annotation containing comments*, which are comments that contain interrupt-related preconditions (defined in Section 2.2) and (2) *annotation generation:* converting these comments into annotations. Postconditions are inferred during the propagation process, which is presented later in Section 2.4.

Comment Extraction: We improve the comment parser from iComment [60], use it to extract all comments from a given program, and break these comments into sentences.

What does not work? We extracted comment sentences that contain word "interrupt" regardless of cases. Table 2 shows that there are on the order of 10,000 such comments. A cursory examination found that less than 5% of the comments contain the kind of annotations we want to extract (defined in Section 2.2), which is consistent with our comment characteristics study results [46]. Therefore, it is inefficient to manually read all of these comments to extract seed annotations. Our prior work iComment [60] used machine learning techniques to automatically analyze several thousands of lock-related comments to extract programming rules. However, we cannot directly apply the techniques used in iComment [60] because around 25% of the thousands of lock-related comments contain rules, but less than 5% of the over 10,000 interrupt related comments contain rules/annotations. The same techniques used in iComment [60] would produce much less accurate results for interrupt related comments. Additionally, aComment requires higher analysis accuracy as the extracted annotations are intended to be added back to the source code to improve program comprehension and prevent the introduction of new bugs. We do not want to add wrong annotations to mislead developers.

Therefore, we combine simple program analysis with effective heuristics to extract annotations and manually verified all of the annotations. The heuristics are shown in Table 3 where each *variable* can be expanded into multiple words and their variants as shown in Table 4. We tried our best to include as many paraphrases and variants. For example, in addition to "disable", we used "turn off", "block", "lock out", and their variants such as "disables", "disabling", "disabled", "turning off", "turned off", "turns off", etc. In the future, we can leverage advanced natural language processing techniques [19, 32] to automatically discover paraphrases. Additionally, we filter out comments that contain words such as "may" and "might". Furthermore, as aComment only extracts function preconditions and postconditions, we only consider comments that are before a function body, a function declaration, or a function call. As we aim for high precision, i.e., more extracted annotations are accurate and correct, the heuristics above are biased to find more comments that are likely to contain annotations at the cost of missing some annotation containing comments.

Annotation Generation: After aComment extracts the comments that contain the interrupt related preconditions, aComment needs to decide if the precondition is `0` or `1` and extract the name of the function associated with the annotation. aComment obtains the information above using simple program analysis and heuristics:

Is the precondition `0` or `1`? By identifying the verbs (e.g., "disable" and "enable") and negation words (e.g., "not"), we can determine the precondition. For example, "disable" is mapped to `0`, and a negation word "not" flips the precondition once to `1`.

What is the function name? Given an annotation containing comment, aComment can extract the function name by analyzing the

Variable	Definition
<call>	Word “call”, its variants such as calls, called, and calling
<before>	“caller”, “before”, “on entry”, “upon entry”, and their variants
<disable/enable>	“disable”, “enable”, “turn on”, “turn off”, “block”, “lock out”, and their variants
<interrupt>	“interrupt”, “irq”, and their variants

Table 4: **Definitions of the variables in Table 3**

linux/kernel/posix-cpu-timers.c:

```
1  static void vmi_timer_set_mode(enum clock_event_mode mode,
2                    struct clock_event_device *evt) {
3        cycle_t now, cycles_per_hz;
4        BUG_ON(!irqs_disabled());
5        ...
6  }
```

Figure 3: **Code assertion example**

linux/kernel/timer.c:

```
1   void update_process_times(int user_tick)
2   {                                                          {
3       struct task_struct *p = current; // calls get_current()   (X, X)
4       int cpu = smp_processor_id(); // not a function call
5
6       account_process_tick(p, user_tick);                     (0, 0)
7       run_local_timers();                                     (0, 0)
8       if (rcu_pending(cpu))                                   (X, X)
9           rcu_check_callbacks(cpu, user_tick);                (X, X)
10      scheduler_tick();                                       (0, 0)
11      run_posix_cpu_timers(p);                                (0, 0)
12  }                                                          }
```

Figure 4: **The above annotation propagation example for the Linux kernel illustrates, that we can infer that the annotation for function** `update_process_times` **is** `@IRQ (0,0)` **based on its callee functions' annotations (shown on the right).**

code segment below the comment, e.g., a function definition or a function call statement.

For each generated annotation, our analyzer shows the original comment and the surrounding code to allow a user to either accept, reject, or modify the annotation (e.g., flip the annotation, change the function name, etc.).

As shown later in Table 8, the heuristics are effective in dramatically reducing the number of comments we need to verify. We only needed to read 682 of the 66,795 comments to verify a total of 355 accurate seed annotations from the four operating system code bases. These manually verified accurate 355 interrupt related annotations can not only help detect bugs, but also help developers prevent the introduction of bugs.

Extracting Annotations from Code.

As many functions do not have comments explaining their preconditions and postconditions, we need to extract more seed annotations by learning from the source code. We observed that source code typically contains assertions to indicate that a function must be called with interrupts disabled or enabled. For example, Figure 3 shows that function `vmi_timer_set_mode` calls assertion code `BUG_ON(!irqs_disabled())` as its first statement to indicate that interrupts should have already been disabled before calling `vmi_timer_set_mode`. At runtime, if interrupts are not disabled before calling `vmi_timer_set_mode`, error messages will be printed by the kernel. While such assertions can help detect bugs to some extent, they are limited as mentioned in Section 1.2: (1) debugging macros such as `BUG_ON` are disabled by default mainly due to the high runtime overhead; and (2) such a dynamic approach can only detect manifested bugs, and the manifestation of concurrency bugs in an OS is extremely difficult. If we can convert such assertions into annotations and check if the code conforms to the annotations statically, we could detect bugs that cannot be detected by these dynamic assertions.

Therefore, we use simple static analysis to extract annotations from these assertion macros. We analyze the direct callers of `BUG_ON(!irqs_disabled())` and `BUG_ON(irqs_disabled())` to see if these functions are intended to be called with interrupt disabled or enabled.

2.4 Annotation Propagation

Our goal is to annotate all functions based on the seed annotations extracted from comments and code. If we know the annotations of all the callee functions of function `Foo`, then we can track the interrupt related *state* to automatically generate the annotation for function `Foo`. Taking the code in Figure 4 as an example, we can infer that the annotation for function `update_process_times` is `@IRQ (0, 0)`. Specifically, we start with the precondition of the first callee function (Line 3), which is `X`. When we see Line 6, we can infer that the interrupts must be disabled after Line 3. As the annotation for Line 3 is `(X, X)` (recall that this means either `(0, 0)` or `(1, 1)`), we know the interrupt state before Line 3 is `0`, which is the precondition of `update_process_times`. Similarly, we can infer that the interrupts should remain disabled on exit of function `update_process_times`. Therefore, both the precondition and postcondition of function `update_process_times` must be `0`. Note that we cannot update the annotation of the callee function `get_current` from `(X, X)` to `(0, 0)` because it is possible that when called from a different location, the states before and after calling `get_current` are both `1` (interrupts are enabled), meaning that its annotation can be either `(0, 0)` or `(1, 1)`.

The next question is how to obtain the annotations of all the callee functions in the first place. This task is performed in a bottom-up fashion in two steps, so that all the callee functions are annotated before their callers. In the first step, *the initialization step*, we assign the seed annotations to their corresponding functions, called *seed functions*. In the second step, *the propagation step*, we propagate the seed annotations from the seed functions to their callers repeatedly in a bottom-up manner.

Step I: Initialization: We add the seed annotations extracted from comments and assertions to their corresponding functions. In addition, we annotate functions that *directly* disable, enable, or restore interrupts, e.g., `local_irq_disable` for the Linux kernel, called *IRQ functions*. A kernel usually has a very small number of IRQ functions. For example, we only need to annotate 8 IRQ functions for the Linux kernel (Section 3). We annotate these IRQ functions with `(X, 0)` if they disable interrupts, with `(X, 1)` if they enable interrupts, and with `(X, P)` if they restore a saved interrupt state. We then find all unannotated functions that do not have any callees, and annotate them with `(X, X)`, meaning that

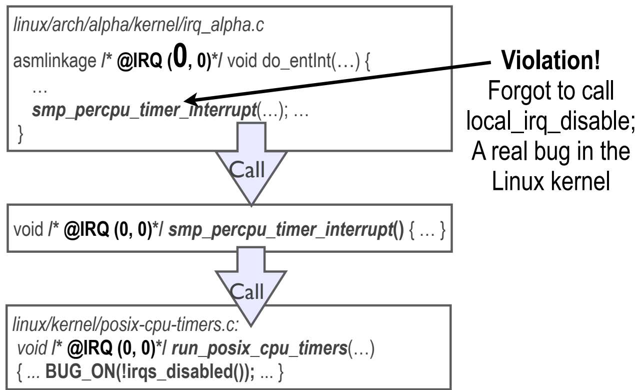

Figure 5: This is a real bug in the Linux kernel 2.6.12, which is described in Introduction. A "call" arrow denotes a direct call or an indirect call of a function.

these functions can be called with interrupts disabled or enabled, but the postconditions should be the same as their preconditions.

The reason that we use `(X, 0)` instead of `(1, 0)` as the annotation for an interrupt disabling function is, that these interrupt disabling functions simply clear the interrupt state, and do not assume interrupts enabled on entry. Therefore, interrupt disabling functions can be nested, e.g., it is legitimate to call `local_irq_disable` twice consecutively. The same is true for interrupt enabling functions. If a user of aComment wants to flag such nested usage as a warning, he or she can set the annotations for interrupt disabling functions as `(1, 0)` instead. Our tool supports both options. Without losing generality, we will use `(X, 1)` for explanation and results presentation in the rest of this paper.

Step II: Propagation: Our propagation analysis starts from the bottom of a call graph to find all functions whose callee functions are all annotated, and automatically infers the annotations for them. This propagation process is repeated until all functions are annotated. If a call graph contains no cycles, it is guaranteed that all functions in the call graph will be annotated. In case of a cycle (i.e., recursive function calls), we follow the cycle until the annotations stabilize.

If an interrupt restoring function is encountered, we simply restore the state to `X`. A more precise analysis would restore to the saved interrupt state (i.e., a parameter of the interrupt restoring function). However, this is more expensive as it requires context sensitivity. Because developers choose to save the interrupt state, it generally indicates that the saved state can be `0` or `1`; therefore, `X` is a better choice than either `0` or `1`.

Let us still use the example in Figure 4 to explain the propagation process. After several rounds of propagation from the bottom of the call graphs, all of function `update_process_times`'s callees are annotated as Figure 4 shows. Therefore, we can infer that the annotation for `update_process_times` is `(0, 0)`.

2.5 Annotation Checking and Bug Detection

Bugs are detected during the propagation process described above. There are two types of violations, i.e., *root function violations* and *unsatisfiable violations*. If a root function's precondition is not `X`, it is considered a *root function violation*, where a root function is a function that does not have a caller (e.g., in the kernel). A function may not have any caller in the kernel because the function is intended to be called by user level functions. As root functions' callers are outside the kernel, if the precondition is `0` or `1`, it can not be guaranteed, indicating a bug. Therefore, although aComment did not analyze the user level code, aComment can detect bugs caused by the interaction between the user level code and the kernel code. Take the bug described in the Introduction as an example (Figure 5), our aComment tool reports a violation when it propagates annotation `(0, 0)` from function `run_posix_cpu_times` to function `smp_percpu_timer_interrupt`, and eventually to the root function `do_entInt`. This is a real bug as confirmed by the Linux kernel developers. The function `local_irq_disable` should be called before calling `smp_percpu_timer_interrupt` to disable interrupts to ensure the precondition `0` of function `smp_percpu_timer_interrupt`. Section 4 shows more bugs detected by our aComment tool.

Software	LOC	#Sentence
Linux	5.2M	1,024,624
FreeBSD	2.4M	420,013
NetBSD	3.3M	680,650
OpenSolaris	3.7M	535,073

Table 5: Operating systems evaluated by aComment. LOC is the total number lines of code (including comments), with blank lines excluded. #Sentence is the total number of comment sentences.

If preconditions and postconditions conflict with each other, our aComment tool reports them as *unsatisfiable violations*. For example, if function A (with annotation `(X, 1)`) is invoked immediately before function B (with annotation `(0, 0)`), we know it is not satisfiable because the interrupts are enabled after calling A, but they should be disabled before calling B.

The reported bugs are ranked according to their confidence scores, which are affected by several factors including *seed annotation confidence* and *violation confidence*. As code is generally more reliable than comments, seed annotations extracted from code assertions are considered more accurate than seed annotations inferred from comments. Therefore, bugs violating seed annotations extracted from code assertions are given higher confidence scores. Unsatisfiable violations are given higher confidence scores than root function violations, because aComment may miss some of the callers of the root functions due to static analysis imprecision (details in Section 4.1). If the confidence score of a bug is lower than an adjustable threshold, the bug is not reported to the user. The user of aComment can always set the threshold to be 0 to retrieve all the potential bugs.

2.6 Static Analysis

As the used static analysis techniques are not our major contribution, we only briefly describe them. We extend our static analysis tool from iComment [60] to extract annotations from assertions, propagate annotations, and detect OS concurrency bugs. The analysis is inter-procedural, flow-insensitive, and summary-based.

3. EXPERIMENTAL METHODS

Our aComment tool automatically propagates annotations starting from a few *IRQ annotations*, i.e., annotations for functions that directly disable, enable or restore interrupts. For the Linux kernel, we manually identified 4 interrupt disabling functions, 2 interrupt enabling functions, and 2 interrupt restoring functions. aComment takes the 8 IRQ annotations as input, and automatically propagates them to all other functions, a total of 96,821 annotations. One can update these IRQ annotations easily if the code changes or if we want to analyze a different code base.

Source	Seed	SeedChecked	TrueBugs	FalsePositives
Comment	226	119	7	2
Assertion	24	17	3	1
Total	245	133	9	3

Table 6: aComment generated 96,821 annotations for the Linux kernel and detected 9 true bugs. The 'Total' row is not the sum of the two rows above, because seed annotatios extracted from assertions overlap with seed annotations extracted from comments, causing the detected bugs to overlap as well.

Evaluated Software.

We evaluated our aComment tool on the latest versions of the Linux kernel. In addition, we extracted annotations from the comments of three other large kernel code bases, i.e., FreeBSD, NetBSD and OpenSolaris (Table 5). All of the four OSs are written in the C programming language, which is the dominant programming language for writing operating systems. However, our hybrid annotation extraction, annotation propagation, and bug detection techniques are general and can be applied to code written in other programming languages such as Java.

4. RESULTS

4.1 Overall Results

Table 6 shows the overall annotation extraction, annotation propagation, and bug detection results of aComment. The 'Total' row is not the sum of the two rows above, because seed annotations extracted from assertions overlaps with seed annotations extracted from comments, causing the detected bugs to overlap as well.

aComment generates 96,821 interrupt related annotations by analyzing comments and code from the Linux kernel. In total, 245 seed annotations are inferred from comments and code assertions, 226 of which are from comments, and 24 of which are from code assertions. Only 5 of them can be extracted from both comments and code assertions, meaning that the majority (221 from comments and 19 from code assertions) can only be extracted from one of the two sources. This result indicates that it is beneficial to infer annotations from both sources. Our hybrid approach greatly increases the number of annotations that can be extracted, which helps detect more bugs (Table 6, Column 'TrueBugs') more accurately. All the 245 seed annotations are manually verified as correct.

Using the annotations, our checker reports 12 bugs, 9 of which are true bugs (7 of which were previously unknown) from the latest versions of the Linux kernel. These bugs are not only *important* (they are in the core kernel modules; they can crash, hang, or corrupt the OS; and they can affect all applications running on top of the OS), but also *hard-to-detect* (due to their non-deterministic nature and the complex interaction with interrupts).

Table 7 presents the annotation distribution in the Linux kernel, which shows that 25,509 functions' annotations are not simply `(X, X)`; it would be tedious and time-consuming to manually specify them all. The result demonstrates that our propagation and extraction techniques are effective in annotating all functions from a handful of IRQ annotations.

Table 6 also shows that only a portion (133) of the inferred seed annotations were used for bug detection because we used a typical x86 Linux kernel compilation, meaning that some of kernel code was not compiled and cannot be analyzed. Some of the kernel code was not compiled because it depends on a particular architecture or a particular driver. To actively compile a maximum amount of code remains as our future work.

Annotation	No.
(0, 0)	404
(0, 1)	89
(1, 0)	873
(1, 1)	4,539
(X, 0)	2,046
(X, 1)	17,470
(X, X)	71,312
(X, P)	2
(0, X)	8
(1, X)	78
Total	96,821

Table 7: Distribution of Linux annotations

.

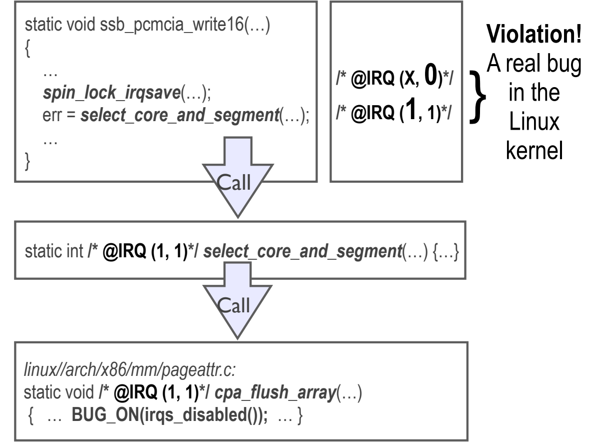

Figure 6: A real bug detected by aComment in the Linux kernel. A "call" arrow denotes a direct call or an indirect call of a function.

False positives are mostly caused by the inaccuracy of our static analysis. First, our static analysis tool cannot know if certain statements are not reachable. For example, aComment mistakenly considered that the function call `local_irq_disable()` in the code segment `while (0) {local_irq_disable();}` was executed, therefore, it inferred the wrong interrupt state and reported a false bug. Such false positives can be removed by eliminating obviously not executed code segments. Additionally, function pointers are frequently used in the kernel code, but our static analysis tool cannot adequately discover their aliases, causing false positives. A more advanced pointer aliasing analysis can reduce these false positives.

Detected Bug Examples.

In addition to the bugs shown earlier, we show another bug automatically detected by our aComment tool in Figure 6. The precondition in annotation `@IRQ (1, 1)` is extracted directly from the assertion in `cpa_flush_array` (shown at the bottom of Figure 6), while the postcondition is obtained during propagation. The annotation is propagated to `select_core_and_segment`, which is not satisfiable because the interrupts were disabled right before calling function `select_core_and_segment`. In addition, several other caller functions of `select_core_and_segment` disable interrupts right before calling `select_core_and_segment`. However, we count them as one bug, because they were all solved by one bug fix by the kernel developers: making `select_core_and_segment` no longer call `cpa_flush_array`.

Software	#Sentence	#IRQSent	#HeuSent	#Annot
Linux	1,024,624	23,662	423	226
FreeBSD	420,013	11,117	80	43
NetBSD	680,650	23,942	108	62
OpenSolaris	535,073	8,074	71	24
Total	2,660,360	66,795	682	355

Table 8: Annotations extracted from comments. #Sentence is the total number of comment sentences. #IRQSent denotes the number of comment sentences that contain the keyword 'interrupt' (case insensitive). #HeuSent is the number of comment sentences extracted using our heuristics. #Annot is the number of annotations that are manually verified to be correct.

4.2 Annotation Extraction Results

Comments Versus Code.

In total, 245 seed annotations are inferred from the Linux kernel comments and code assertions, 226 of which are from comments, and 24 of which are from code assertions. A majority of the annotations, 221 from comments and 19 from code assertions, can only be extracted from one of the sources. This result indicates that comments and code complement each other for annotation extraction. Our hybrid approach increases the number of annotations that can be extracted, which helps detect more bugs more accurately.

Annotation Extraction in Other OSs.

Table 8 shows the number of interrupt related annotations extracted from the four popular operating systems' comments. It demonstrates that our heuristics dramatically reduced the number of comments that we need to read manually to verify the annotations: we only need to manually read 682 of the 66,795 comments to find a total of 355 accurate seed annotations from the four operating systems. The annotation extraction precision is the number of verified annotations (355) divided by the total number of extracted annotation containing comments (682), which is 52.1%. We can significantly improve this precision by focusing on function header comments, which are more likely to contain preconditions and postconditions. We calculate the annotation generation accuracy as the number of verified annotations whose preconditions and function names are generated correctly of the total number of verified annotations. The annotation generation accuracy for the four OSs is 90.3-100.0%.

Section 4.1 demonstrates that aComment is effective in leveraging the extracted annotations in the Linux kernel to detect bugs. The annotations in the other three OSs should help us detect more bugs, which remains as our future work.

4.3 Time Overhead

On a server with two 2.4 GHz Intel processors, it took aComment 30 minutes to analyze comments to extract seed annotations. The annotation propagation and bug detection process finished within 235 minutes. Therefore, our aComment tool is practical to be used for real-world large software.

5. DISCUSSIONS AND LIMITATIONS

Alternative Solutions.

To avoid the bug in Figure 5, one solution is to have hardware disable interrupts before entering an interrupt handler (as x86 does). However, this can cause problems as it is hardware architecture dependent. In the Alpha architecture, the hardware does not disable interrupts before entering an interrupt handler, which caused this bug. In addition, it does not solve the general OS concurrency problem. For example, it does not help when interrupts need to be disabled before a function that is not an interrupt handler.

Limitations.

aComment is effective in extracting and inferring annotations for all functions in large software, however, it has limitations. We only considered annotations that require all interrupts to be disabled and enabled. Some functions may assume that a particular interrupt is disabled or enabled on entry or exit, which is out of the scope of aComment. In the future, we can extend our work to extract and analyze these more detailed assumptions.

We assumed that functions have unconditional interrupt related preconditions and postconditions. In other words, if a function must be called with interrupts disabled in some contexts, and must be called with interrupts enabled in other contexts, we simply consider that the precondition is `X` (i.e., don't-care). While it is possible to model conditional preconditions, it is a bad practice for developers to implement functions with conditional preconditions.

As software evolves, comments can become outdated, which may cause some annotations extracted from comments to be obsolete. Our manual verification of the extracted comments did not reveal any wrong annotations caused by this reason. In the future, we would like to send the annotations to the developers to verify the correctness of the annotations. In practice, developers often update comments to keep them in sync with source code [37, 60]; and we believe that important comments are less likely to be outdated because developers may be more motivated to keep them in sync.

6. RELATED WORK

6.1 Annotation Languages

Many annotation languages have been proposed to extend the C type system [9, 15, 38, 39, 62], to specify locking requirements [39, 62], to annotate function interfaces [15, 38, 62], or to mark control flows [15, 38]. Storey et. al [58] studied how programmers use TODO comments for task annotation purpose. None of these annotation languages can express the complex synchronization assumptions that are intertwined with interrupts in operating system code. Moreover, these studies rely on manually written annotations while we advance the state of art by semi-automatically annotating code.

6.2 Concurrency Bug Detection

Many dynamic concurrency bug detection techniques were proposed. Race detectors [5, 8, 45, 47, 54, 67] detect data races; a few studies are conducted to detect, avoid or prevent deadlocks [24, 25, 63]; and other work [17, 20, 29, 35, 36, 65] detects atomicity violations. Several static concurrency bug detection or verification techniques were proposed [22, 41, 42, 50, 53, 56, 57].

The work mentioned above does not handle the complex interaction with interrupts or leverage the preconditions and postconditions embedded in comments for concurrency bug detection, missing the opportunities to detect more OS concurrency bugs. In addition, none of them demonstrated their effectiveness on operating system code, which is extremely challenging due to the amount of drivers, state explosion, bug manifestation, run-time overhead and/or scalability problems. Further, this paper explicitly generates and propagates annotations to prevent the introduction of new bugs.

Some work [12, 28, 49] demonstrated the ability to check OS code. For example, recent work [28] used symbolic execution to test device driver binaries, which detected several interrupt related

race conditions. aComment complements their work in the following ways. aComment infers programming rules (interrupts must be disabled on entry and exit) in the form of annotations in addition to bug detection. The inferred annotations may be used by other detection and verification tools to detect bugs. They can also be used to help developers avoid bugs. Further, aComment detects violations to the inferred programming rules, while they [12, 28] detect data races. As quite often a race is not a bug [36, 43, 44], aComment addresses this limitation by detecting violations to *programmers' intentions*, which directly indicate bugs. Additionally, the previous work did not leverage comments for bug detection or annotation extraction.

A heuristic based dynamic bug detection tool, lockdep, was developed by the kernel developers specifically for the Linux kernel [2]. Lockdep uses several heuristics to detect lock-related bugs, e.g., if two locks are acquired in different orders at different places, and if there are dependencies between a lock that is ever held in interrupt context and a lock that is every held with interrupt enabled. Being a dynamic approach, lockdep requires "*massive* amount of runtime checking" [2], while aComment incurs no runtime overhead for using a static approach. In addition, aComment infers fine-grained (specific to a function) annotations to detect violations to developers' intentions and assumptions.

6.3 iComment

Our previous work, iComment [60], is the first to automatically analyze 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. iComment has demonstrated its effectiveness by automatically analyzing several thousands of lock-related and call-related comments to detect 60 new bugs and bad comments in the Linux kernel, Mozilla, Apache and Wine.

This work is different from iComment [60] in several aspects as this work: (1) proposed and generated a new type of annotations and used these annotations to detect bugs caused by the complex synchronization related to interrupt context; (2) automatically propagated annotations to caller functions to improve annotating and bug detection; and (3) used a hybrid approach to extract annotations from both code and comments, while iComment only extracted rules from comments.

6.4 Rule and Pattern Extraction

Previous work [6, 7, 13, 14, 31, 33, 34, 48, 61, 64] extracted programming rules or models from source code or execution traces for bug detection or other purposes. Although source code and execution traces have been very useful for rule and model extraction, certain important information is documented in comments but are not available in source code or are extremely difficult to extract from source code. Without utilizing information in comments, previous work missed the opportunities to extract more information, detect more bugs or the chance to improve their bug detection accuracy.

Concurrent to or after our prior work iComment [60], a few studies [18, 70] extracted rules or specifications from documents in natural language or the semantics of program identifies. This paper extracts a different type of rules: OS interrupt related annotations, which has its unique challenges and requires different techniques. In addition, we applied a hybrid approach to extract annotations from both code and natural language text (comments).

6.5 Automatic Documentation Generation

Literate programming by Knuth [26] proposes embedding code inside documentation to produce "literature" instead of embedding comments and documentation in the code. GhostDoc [1] generates XML formatted comments from code identifiers that follow a good naming convention. Javadoc [27] let programmers use special *tags* to document functions or data structures. Those tags are later processed by a tool to automatically produce hypertext documentation. While these tools help developers write better documents, they did not address the OS concurrency bug detection problem.

6.6 Transactional Memory

Recently transactional memory [21, 23] is proposed to ease the implementation of concurrent programs. While they can reduce concurrency bugs, they cannot entirely eliminate such bugs. Our approaches could still help find bugs in programs using transactional memory. In addition, it is quite challenging to provide transactional memory support for operating systems.

7. CONCLUSIONS AND FUTURE WORK

To detect operating system concurrency bugs related to interrupts, we design a new type of annotations – interrupt related annotations, and generate 96,821 such annotations for the Linux kernel with little manual effort. These annotations help us automatically detect 9 real bugs in the latest versions of the Linux kernel. Many (245) of the annotations are seed annotations, which are directly inferred from comments and code assertions. We automatically propagate these seed annotations from the callee functions to the caller functions to generate annotations for all of the functions. By extracting seed annotations from both comments and code, we are able to extract more annotations than using a single source, as only a small number (5) of the annotations can be extracted from both sources.

In the future, we plan to extend the analysis to generate annotations for a specific kind of interrupts, and distinguish the different interrupt contexts, e.g., bottom halves, top halves, softirqs, etc. The hybrid approach of extracting specifications from both comments and code can be applied to non-OS code, code written in other programming languages such as Java, and for extracting other types of specifications.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their priceless comments. This research is supported by NSF and NSERC.

9. REFERENCES

[1] Ghostdoc. http://submain.com/products/ghostdoc.aspx.
[2] Runtime locking correctness validator. http://www.mjmwired.net/kernel/Documentation/lockdep-design.txt.
[3] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. In *PLDI*, 2008.
[4] T. Ball, B. Hackett, S. Lahiri, and S. Qadeer. Annotation-based property checking for systems software. Research report MSR-TR-2008-82, Microsoft Research, May 2008.
[5] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *PLDI*, 2010.
[6] L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with Tarantula. In *ISSRE*, 2007.
[7] J. Burnim and K. Sen. DETERMIN: Inferring likely deterministic specifications of multithreaded programs. In *ICSE*, 2010.
[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.
[9] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.
[10] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. Reilly, 2005.

[11] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. In *SOSP*, 1967.

[12] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[13] D. R. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[14] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[15] D. Evans. Static detection of dynamic memory errors. In *PLDI*, 1996.

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

[17] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[18] Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker. Analysing source code: Looking for useful verb-direct object pairs in all the right places. *IET Software Special Issue on Natural Language in Software Development*, 2008.

[19] O. Glickman and I. Dagan. Acquiring lexical paraphrases from a single corpus. In *RANLP*, 2003.

[20] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, 2008.

[21] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 2003.

[22] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, 2004.

[23] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 1993.

[24] P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. In *FASE*, 2001.

[25] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[26] D. E. Knuth. Literate programming. *Computer Journal*, 27(2), 1984.

[27] D. Kramer. API documentation from source code comments: A case study of javadoc. In *SIGDOC*, 1999.

[28] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC*, 2010.

[29] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, 2010.

[30] N. Leveson. The Therac-25 accidents.

[31] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, 2005.

[32] D. Lin and P. Pantel. Discovery of inference rules for question-answering. *Natural Language Engineering*, 2001.

[33] B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *FSE*, 2005.

[34] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, 2008.

[35] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, October 2007.

[36] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[37] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. E. Hassan. Understanding the rationale for updating a function's comment. In *ICSM*, 2008.

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

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

[40] B. Murphy. Automating software failure reporting. *Queue*, 2004.

[41] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[42] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, 2009.

[43] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *PLDI*, 2007.

[44] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[45] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[46] Y. Padioleau, L. Tan, and Y. Zhou. Listening to programmers - Taxonomies and characteristics of comments in operating system code. In *ICSE*, May 2009.

[47] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, 1996.

[48] D. Posnett, C. Bird, and P. T. Devanbu. THEX: Mining metapatterns from java. In *MSR*, 2010.

[49] Z. Rakamaric. STORM: Static unit checking of concurrent programs. In *ICSE Student Research Competition*, 2010.

[50] D. S. Rosenblum. Design and verification of distributed tasking supervisors for concurrent programming languages. 1988.

[51] N. Rungta, E. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic executio. In *SPIN*, 2009.

[52] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *EuroSys*, 2009.

[53] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP*, 2005.

[54] 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.

[55] SecurityFocus. Software bug contributed to blackout.

[56] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based testing of concurrent programs. In *FSE*, 2009.

[57] N. Sterling. WARLOCK - A static data race analysis tool. In *USENIX Winter Technical Conference*, 1993.

[58] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *ICSE '08*, 2008.

[59] L. Tan, C. Liu, Z. Li, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Bug characteristics in modern open source software. In *University of Waterloo Technical Report*, 2011.

[60] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[61] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *ICSE*, 2009.

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

[63] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for mult-threaded programs. In *OSDI*, 2008.

[64] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *FSE*, 2007.

[65] M. Xu, R. Bodk, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[66] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: An extensible, expressive system and language for statically checking security properties. In *CCS*, 2003.

[67] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[68] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, 2010.

[69] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Trans. Softw. Eng.*, 32(4), 2006.

[70] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, 2009.