Testing and Debugging Concurrency Bugs in Event-Driven Programs

Guy Martin Tchamgoue, Kyong-Hoon Kim and Yong-Kee Jun

Department of Informatics, Gyeongsang National University,
Jinju 660-701, Republic of Korea
guymt@ymail.com, khkim@gnu.ac.kr, jun@gnu.ac.kr

Abstract

Event-driven programs are prone to concurrency bugs due their inherent nature of handling asynchronous events. Asynchronous events introduce logical concurrency into these programs making them hard to be thoroughly tested and debugged. However, understanding the root causes and characteristics of concurrency bugs can ease the debugging process and help developers to avoid introducing them. Unfortunately, previous taxonomies on concurrency bugs do not provide enough knowledge on event-driven concurrency bugs. This paper classifies the event-driven program models into low and high level according to the type of events they handle, categorizes concurrency bug patterns and surveys existing techniques to detect them in event-driven programs.

Keywords: *Events, event-driven programs, concurrency bugs, taxonomy, concurrency bug detection.*

1. Introduction

Caused by non-deterministic interleaving between shared memory accesses [1], concurrency bugs are a well-documented topic in shared-memory programs including event-driven programs which handle asynchronous events. Asynchronous events introduce fine-grained concurrency into these programs making them hard to be thoroughly tested and debugged. However, concurrency bugs such as unintended data races, atomicity and order violations, and deadlocks are not only common, but also notoriously hard to uncover. Despite the numberless tools and techniques devised to detect concurrency bugs, they often remain undetectable until the exploitation phase leading the application into unpredictable executions sometimes with severe consequences. A typical example is the well-known accident of the Therac-25 [2] where, as the result of a data race caused by the keyboard event handler, many people received fatal radiation doses.

Event-driven programs are becoming pervasive with applications ranging from web servers to operating system kernels and safety-critical embedded software. Understanding the causes and characteristics of concurrency bugs may ease the debugging process and can help creating new heuristics for some debugging tools as shown for shared-memory parallel programs [3]. Unfortunately, previous taxonomies or studies are not applicable to the debugging of event-driven programs or do not provide enough knowledge on event-driven concurrency bugs. A taxonomy of concurrency bugs in shared-memory parallel programs is presented by Farchi et al. [3] and used to create new heuristics for an existing detection tool. However, this taxonomy cannot only fit

for event-driven software due to the differences in the concurrency models. A short taxonomy of bugs in device drivers is given by Ryzhyk et al [4]. Although the classification is not centered on concurrency bugs, their work showed however that concurrency bugs accounts for 19% of the total number of bugs in device drivers. Many other existing works [5, 6, 7] focus on understanding the characteristics of concurrency bugs in real world programs and their impact on new testing and debugging tools. However, all these studies consider only shared-memory parallel programs giving less or no attention to event-driven concurrency bugs.

This paper classifies the event-driven program models into low and high level based on the type of events they handle and carefully examines and categorizes concurrency bug patterns in such programs. As software bugs represent the major cause for system failures, it is therefore important to deeply understand the causes and characteristics of bugs in order to effectively design tools and support for detecting and recovering from software failures [6]. Detecting concurrency bugs in event-driven programs is particularly difficult since they usually contain a very large number of executable paths [8] due to asynchronous events. Nevertheless, many tools and techniques [8, 9, 10, 11, 12, 13, 14] have been proposed for uncovering such bugs for various application domains. These detection techniques can be roughly classified into three major groups: testing methods, static analysis and dynamic analysis. This work also provides a survey of existing tools and techniques for detecting concurrency bugs in event-driven programs.

In the remainder of this paper, Section 2 describes the event-driven programs and gives our motivation. Section 3 provides details on the proposed taxonomy. Section 4 presents a survey of existing detection techniques and gives useful recommendations to avoid concurrency bugs in event-driven programs. Finally, our conclusion comes in Section 5.

2. Background

In order to understand the causes and to categorize concurrency bugs in event-driven programs, it is crucial to have a look on their inner characteristics. Thus, this section describes the event-driven programs, presents the general properties of events and gives the motivation that sustains this work.

2.1. Events and Programs

Globally, events can be classified into two categories: the *low-level* events or interrupts and the *high level* events or signals (UNIX-like signals). Thus, in this paper, when not specifically mentioned, the term event refers to both interrupts and signals and not to the actions that generate them. A signal is a message sent by the kernel or another process (using the kill system call) to a process. Often referred to as *software interrupts*, signals are used as basic inter-process communication mechanisms. In the other hand, interrupts are defined as *hardware signals* as they are generated by the hardware in response to an external operation or environment change. These two classes of events share almost the same properties.

2.1.1. Event Handling: To service each event, an asynchronous callback subroutine called *event handler* is required. In the case of interrupts, the interrupt handler generally resides in the operating system kernel. However, for some embedded systems with a thin kernel layer like TinyOS [15], applications have to provide and manage their own interrupt handlers.

Before its invocation, any signal handler must be carefully registered with the kernel using the `signal()` or `sigaction()` system calls. Each signal has a default handler or action and depending on the signal, the default action may be to terminate the receiving process, to suspend the process, or just to ignore the signal. When an event is received, it remains pending until it is delivered or handled.

2.1.2. Blocking, Preemption, Nesting, and Reentrancy: Contrarily to threads, event handlers cannot block: they run to completion except when preempted by another event handler [8, 11]. Events have an asymmetric preemption relation with the non-event code: event handlers can preempt non-event code but not the contrary. Events are nested when they preempt each other. Nesting events are used to allow time-sensitive events to be handled with low latency [8]. An event is said to be reentrant when it directly or indirectly preempts itself.

2.1.3. Split-Phase Operations: In order to minimize the impact of event handlers on non-event code, all long-latency operations must run in split-phase. With this mechanism, event handlers immediately return after servicing critical operations and post heavy computations for later execution as a new task or process. This technique is known under different names according to the environment: deferred procedure call in Windows [16], bottom-half in Linux [17] or split-phase in TinyOS [11].

2.1.4. Synchronous or Asynchronous: As described by Regehr [8], asynchronous interrupts are signaled by external devices such as network interfaces and can fire at any time that the corresponding device is enabled. Synchronous interrupts, on the other hand, are those that arrive in response to a specific action taken by the processor, such as setting a timer. Similarly, signals may also be generated synchronously or asynchronously. A synchronous signal pertains to a specific action in the program and is generally generated by some errors in the program like the division by zero. Asynchronous signals are generated by actions outside the control of the process (e.g. resizing an application's window) that receives them and may arrive at unpredictable times during execution.

2.1.5. Disabling/Enabling: Event-driven programs present a very simple concurrency model which however allows high and complex concurrency at runtime with an exponentially increasing number of execution paths. Since the only way to share data between a program and its event handlers is through global variables, concurrency bugs like data races may show up. To protect sensitive parts of the program, events must be disabled before and enabled only after critical sections. However, it is not recommended to call non-reentrant functions or system calls within event handlers. In the case of signals, system calls like `getuid()` or `rmdir()` recommended to be invoked in signal handlers are referred to as *async-signal-safe*. Other system calls like `printf()` are classified non async-signal-safe.

2.1.6. Event-Driven Programs: A program is event-driven when a considerable amount of its computation is initiated and influenced by external events like interrupts and/or signals. As described above, depending on the environment, a program can be interrupt-driven or signal-based. For programs that implement only high level events, the operating system is responsible to handle the low-level event and to generate a corresponding high level event for them. In accordance with the type of events they handle, Figure 1 shows a classification of event-driven programs and their interaction with the operating system kernel.

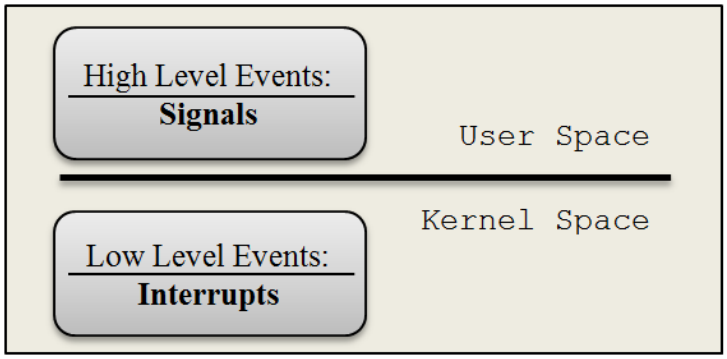


Figure 1. Event-Driven Programs

2.2. Motivation

Event-driven programs are becoming pervasive with the vulgarization of embedded systems. For applications like those in embedded sensor networks which need to interact with the external world, events provide means to quickly respond to changes from the outside environment. Since parallel programs are difficult to implement and to debug, events have been viewed by many researchers as alternative to threads [18, 19, 20].

As event-driven software become more and more complex, there is a real need for more effective and robust tools for testing and debugging in order to reduce the number of bugs that escape in production runs [6]. Event-driven programs present a complex concurrency structure with an exponentially increasing number of execution paths at runtime. Since the only way to share data between an event handler and the non-event code is through global variables, event-driven programs are also prone to concurrency bugs. Designing effective tools for improving the quality of software requires a good understanding of software bug characteristics [6]. However, understanding bug characteristics alone is not enough for designing effective tools for testing and debugging concurrency bugs. Additionally to bug characteristics, a deep understanding of the concurrency model and the bug root causes is also required. Having such knowledge on concurrency bugs certainly helps developers to avoid introducing them.

A taxonomy of concurrency bugs in shared-memory parallel programs is presented by Farchi et al. [3] and used to create new heuristics for a concurrency bug detection tool. Unfortunately, this taxonomy cannot only fit for event-driven software due to the differences in the concurrency models. For example, the concurrency bugs due to the activation of dangerous event handlers when executing sensitive operations (cf. Section 3.1) or to the use of unsafe functions in event handlers (cf. Section 3.2) cannot be captured by the taxonomy presented by Farchi et al [3]. These bugs are directly related to the concurrency model of event-driven programs. A short taxonomy of bugs in device drivers is given by Ryzhyk et al [4]. Although the classification is not centered on concurrency bugs, this work showed that concurrency bugs accounts for 19% of the total number of bugs in device drivers. Many other existing works [5, 6, 7] focused on understanding the characteristics of concurrency bugs in real world programs and their impact on new testing and debugging tools. However, all these taxonomies or studies cannot directly be applicable to event-driven programs or do not provide enough knowledge on event-driven concurrency bugs or other event-related bugs.

This paper categorizes concurrency bug patterns and surveys existing tools for detecting them in event-driven programs. Since software bugs may have similar patterns, a taxonomy of concurrency bugs can also be useful to identify and reveal hidden bugs. Also, studies to

understand the characteristics of software bugs provide useful insights and guidelines for software engineering tool designers and reliable system builders [6].

3. Concurrency Bug Classification

In this section, we present our taxonomy of concurrency bugs in event-driven programs as summarized in Figure 2. This classification tries as much as possible to highlight the potential causes and risks for each class of bugs. Roughly, concurrency bugs can be classified into three main categories: data races, atomicity and order violations, and deadlocks.

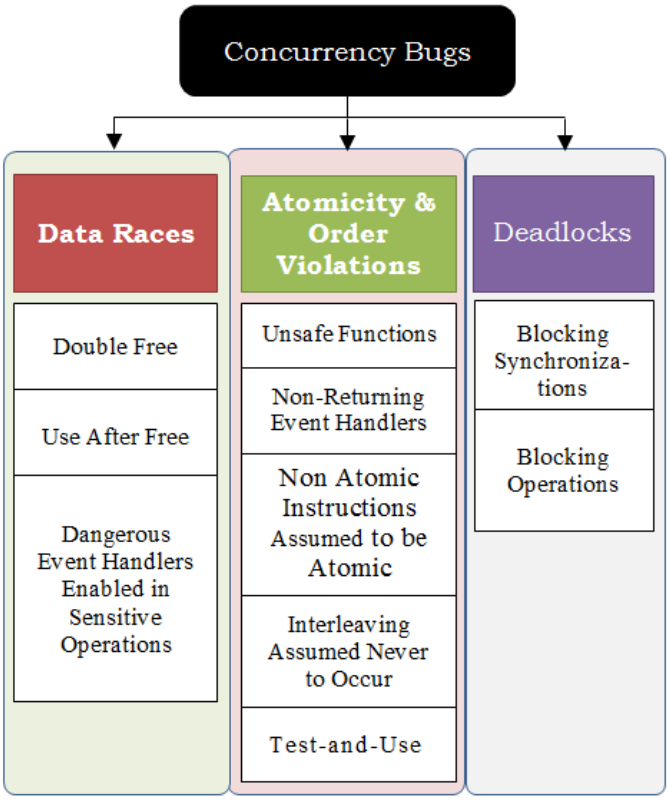


Figure 2. A Taxonomy of Concurrency Bugs

3.1. Data Races

As with shared-memory parallel programs, unintended data races [21] represent one of the most notorious concurrency bugs in event-driven programs. Data races happen when a shared memory is uncoordinatedly accessed with at least one write by both an event handler and a non-event code or by multiple event handlers. Event handlers introduce logical concurrency into event-driven programs and exponentially increase their execution paths at runtime. Ryzhyk et al [4]. showed that concurrency bugs in device drivers are mostly introduced in situations where a sporadic event, such as a hot-unplug notification or a configuration request, occurs while the driver is handling a stream of data requests. Data races can lead to different other concurrency bugs, resulting on unpredictable output, loss or inconsistency of data, data corruption or even on a crash of a program. Following are some examples of bugs belonging to this category:

3.1.1. Double Free: It is common in event-driven programs to share data through global variables. This situation can sometimes lead to pointers or shared memory areas been freed by both the event handler and the non-event code, resulting to the well-known double free bug. This bug may happen because the programmer is confused over which part of the program is responsible for freeing the memory area. Worst, this can also happen if the developer misunderstands the event handling mechanism or has wrong assumptions about the behavior of events. A typical example of such bug is shown in Figure 3 where the shared variable *ptr* can be subject to a double free. If the program receives a signal right before the instruction in line 17 is executed, *ptr* will be doubly freed.

```
1:   #include <signal.h>
2:   #include <stdio.h>
3:   #include <stdlib.h>
4:
5:   void *ptr;
6:
7:   void sh(int sig){
8:     if(ptr!=NULL) free(ptr);
9:   }
10:
11:  int main(int argc, char* argv[])
12:  {
13:    ptr=strdup(argv[2]);
14:    signal(SIGTERM,sh);
15:
16:    if(ptr!=NULL)
17:      free(ptr); //ptr=strdup(argv[2]);
18:  }
```

Figure 3. An Example Program with a Double Free Bug

As we can see with the program of Figure 3, this bug may remain hidden as the probability to reproduce it is very low. Therefore, once it shows up at runtime, there are possibilities for the program to crash or to face data corruption. The double free in this example is a consequence of a race condition between the event handler and the non-event code over the variable *ptr*.

3.1.2. Use after Free: This bug pattern is somehow similar to the double free bug. Only, in this case, we face a situation where a previously deallocated memory location is referenced for computations. However, this bug will not be caused only by the fact that the programmer is confused over which part of the program is responsible for freeing the memory area, but also because of his misunderstanding of the event handling mechanisms. The use after free bug can have number of consequences ranging from data corruption to the execution of arbitrary code and crash of the program.

An example of this bug can be observed in Figure 3 where the instruction in line 17 is replaced by the one in the comment (i.e. `ptr=strdup(argv[0])`).

3.1.3. Dangerous Event Handlers Enabled During Sensitive Operations: Asynchronous events can fire at any time preempting non-event code or other event handlers. A global variable shared by the program and its event handlers can then be exposed to concurrency

bugs. Thus, it is recommended to disable dangerous events before sensitive operations like accessing shared states and to enable them only after. Forgetting to do so might lead the program to unintended data races or other concurrency bugs like atomicity or order violations with severe consequences.

As an example, we can consider again the program in Figure 3 which exposes a double free bug. However, this bug happens because the signal handler is still enabled when the variable *ptr* is accessed in lines 16 and 17. For more precaution, the signal handler must be disabled in line 15 and enabled again only in line 18.

3.2. Atomicity and Order Violations

In this section, we consider concurrency bugs related to *atomicity violations* or to *order violations*. A method or a sequence of instructions is atomic if its execution is not affected by and does not interfere with concurrently executing threads [22]. In the other hand, the order violation bugs occur when a program fails to enforce the programmer's execution order intention [5]. Generally, developers tend to make wrong assumptions about the behavior of event handlers. These wrong assumptions come in different flavors and result in different concurrency bug patterns. The following bugs belong to this class of concurrency bugs:

3.2.1. Use of Unsafe Functions: Due to the fact that global variables are exposed to concurrency bugs, non-reentrant functions or system calls are not recommended in event handlers. A function is said reentrant if multiple instances of the same function can run in the same address space concurrently without creating the potential for inconsistent states [23]. For signals, only a reduced set of reentrant system calls referred to as *async-signal-safe* are recommended to be invoked within signal handlers.

An unsafe function in an event handler cannot guarantee the consistency of data structures when preempted. This may lead to data corruption, loss or inconsistency of data or to unpredictable program output. An example of such vulnerability in signal handler that would have been exploited to perform remote authentication on a server was reported and fixed in OpenSSH [23, Chapter 13]. Another example is the bug detected in Bash 3.0 [24] where the global variable `errno` is overwritten by the non async-signal-safe `ioctl`() called in a signal handler.

3.2.2. Non-Returning Event Handlers: In this scenario, a process is preempted by an event handler, but the execution control never returns back to the interrupted function. There are two ways this can happen [23]: the event handler can either explicitly terminate the process by calling `exit()`, or return to another part of the application in the specific case of signal handler using `longjmp()`. Jumping back to another part of a program might be risky and unsafe if the reachable code is not async-signal-safe. Examples of data races due to non-returning signal handlers were reported in the well-known Sendmail SMTP and WU-FTPD [23].

3.2.3. Non Atomic Instructions Assumed To Be Atomic: This bug pattern [3] is usually related to some non-atomic operations or instructions in high level programming languages like C/C++ that are viewed as atomic by developers. This class of bugs is also referred to as the *load-store bug pattern* [30]. A classic example is the well-known increment operator ++ that is often considered atomic as it seems to consist of a single machine operation. However, during compilation, an $x++$ operation corresponds to three machine instructions: (1) load the current value of x into memory, (2) increment the memory copy of x, and (3) store the

memory value of x. An event can then preempt the execution of this operation after each of these machine instructions resulting to unpredictable results. An example of such bug was detected and reported in Bash 3.0 [24].

3.2.4. Interleaving Assumed Never To Occur: For some reasons, the programmer assumes that a certain program interleaving never occurs [3]. This assumption might be, for example, the relative execution length of a given part of the program, considerations about the underlying hardware, the reentrancy or the repeatability of an event handler, or simply the arrival time of an event. This bug might also happen because a segment of code is wrongly assumed to always run before or after an event handler. In any case, this bug might result in severe data corruption.

An example of such was found in a TinyOS application and described by Regehr [8]. The reported bug effectively involved the analog-to-digital converter interrupt handler which was written under the assumption that every posted task runs before the interrupt handler next fires.

3.2.5. Test and Use: The test-and-use bug pattern is a conditional statement where the condition is checked at the beginning of the statement and then the result is used inside the statement without making sure that the condition still holds [30]. A similar bug pattern referred to as *repeated test-and-use* arises in loops [30]. An example of such bug described in [30] is shown in Figure 4. In this example, the value of the shared variable *ptr* can be modified by an event handler at any time right after the condition if no precaution is taken by the programmer to protect the code, making the assignment instruction to eventually fail.

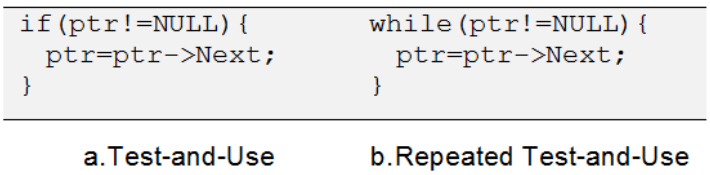


Figure 4. Test-and-Use Bug Patterns

3.3. Deadlocks

In section, we classify all kind of concurrency bugs that manifest themselves as a hang of the system. This commonly happens when an instruction or interleaving contains a blocking operation that blocks indefinitely the execution of a program. In this category, we can enumerate the following bug patterns:

3.3.1. Blocking Synchronizations: In a multithreaded environment, it is recommended to protect accesses to shared variables using synchronization primitives like locks. However, misusing the locking mechanism can lead a program to deadlock. An event handler is supposed to run as fast as possible and returns the control to the main program. Accessing a lock in an event handler might not only block, but cause a deadlock that will surely hand the entire system since the handler might never return. As explained by Dowd et al. [23] for example, the use of a mutex data type in a signal handler can cause a Pthreads program to deadlock.

3.3.2. Blocking Operations: Event handlers always have higher priority than the non-event code with which they have an asymmetric preemption relation. Event handlers must therefore be written efficiently not to monopolize the processor and run as fast as possible since they

can also preempt each other. Thus, in the context of event handlers, it is not allowed to perform any potentially blocking operations [4].

Directly executing blocking operations like some system calls (e.g. `sleep()`) or other file system blocking operations (e.g. manipulating bulk data from an external storage) within an event handler can penalize and even paralyze the entire program execution. An important number of such bugs due to calls to blocking functions in an atomic context were reported in several commonly used Linux device drivers [4].

4. Debugging Event-Driven Programs

Detecting concurrency bugs is particularly difficult and generally requires sophisticated techniques and tools. In this section, we present a survey of existing techniques and tools to detect and avoid concurrency bugs in event-driven programs.

4.1. Detection Techniques

Concurrency bugs are an important issue in event-driven programs which are getting pervasive with the popularization of embedded systems like networked embedded systems. Ever since, there is an urgent need to explore more effective software testing and debugging tools and software engineering methods to minimize the number of bugs that escape into production runs [6]. This section provides a survey of tools and techniques devised for this purpose as summarized in Figure 5. Note that these techniques can also be classified into low level tools [8,9,10,11,12,13,25,26,27] and high level tools [14,24,28] according to the type of events or programs they target.

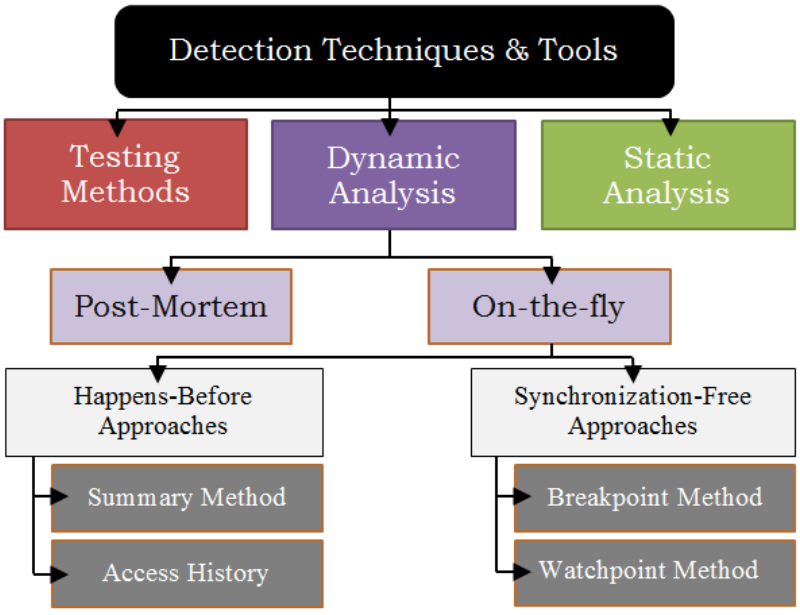


Figure 5. Existing Detection Techniques

4.1.1. Testing Methods: These methods mostly focus on generating test cases that can reveal common bugs as well as concurrency bugs. However, testing methods can only reveal the presence of bugs in a program, but generally require important manual inspection for

detecting their root causes. Regehr [8] proposed a method to random test event-driven applications by randomly fire interrupts at random time intervals. The main challenge in this work is to obtain adequate interrupt schedules by solving the tradeoff between generating a dense or a sparse interrupt schedule. With a dense interrupt schedule, there are always many pending interrupts, making the processor busy in interrupt mode and starving non interrupt code. On the other hand, a sparse schedule provokes less preemption among interrupts, then no or only few bugs might be detected.

To provide a more accurate testing tool, Higashi et al. [23] propose to generate one interrupt after every instruction that accesses a shared variable and to substitute a corrupted memory with a value provided by the developer. The work done by Lai et al. [25] models with an inter-context flow graphs, the interleaving of operations on different contexts in an event-driven application. Two test adequacy criteria, one on inter-context data-flows and another on inter-context control-flows, are then proposed for failures detection in TinyOS applications.

4.1.2. Static Analysis: Static analysis techniques must deal with exponential number of states during the analysis process. They are then less scalable than other techniques and may produce both false positives and false negatives. Some programming languages for event-driven systems like nesC [11] contain a native concurrency bug detection tool. Every nesC program has to enforce a race-free invariant stating that any update to a shared variable must occur either in a synchronous code reachable only from tasks, or within an atomic section.

Regehr and Cooprider [26] propose to transform an interrupt handler code into a corresponding multithreaded code and to use existing static race checkers to detect data races in the newly generated program.

4.1.3. Dynamic Analysis: Dynamic analysis techniques focus only on the specific execution path followed at runtime given an input. These techniques statically or dynamically instrument a program to collect runtime data needed for bug detection. False negatives are inherent to dynamic analysis techniques since bugs in unexecuted paths cannot be detected. Dynamic analysis techniques are classified into two main methods.

Post-Mortem Methods: These techniques instrument the program to record runtime information and data for a post-execution analysis. The runtime overhead may be small, but the size of the log file can be very huge. The main challenge of these techniques is to reduce the size of the log and to accurately record asynchronous events at runtime.

Audenaert and Levrouw [9] proposed a method based on software instruction counter to record and replay multithreaded programs in the presence of interrupts. Gracioli and Fischmeister [12] devised two functions to reduce the size of the log file when recording an event-driven program: the selector function to select parts of the code to be logged and the hashing function to normalize the size of collected data in the log file. A lightweight control flow tracing and encoding scheme to generate a highly compressed log file is presented for networked embedded software debugging by Sundaram et al [27].

On-the-fly Methods: These techniques will instrument a program for runtime detection. The detection overhead may be high, but on-the-fly techniques require still less storage space than post-mortem methods since unneeded information can be discarded as the detection progresses. Existing on-the-fly techniques can further be categorized into several methods we divide into two main approaches:

a. Happens-Before Based Approaches: these techniques implement the Lamport happens-before to detect apparent races [21] in a given program by tracking all synchronization primitives. In this category, we classify the following detection techniques:

- **Summary Methods:** these methods report concurrency bugs with incomplete information about the references that caused them. Ronsse et al. [28] adapted an existing on-the-fly race detector for multithreaded programs to handle data races in sequential programs with concurrent signal handlers. This tool employs a two-phase technique for race detection. During the first pass, memory accesses are collected, concurrent threads detected and primary information about data races found, stored in a log file. The second phase is therefore necessary to refine the information collected during the first execution.
- **Access History Methods:** these techniques maintain an access history to precisely determine each of a pair of concurrent accesses to every shared variable. Tchamgoue et al. [14] proposed a labeling scheme for race detection in sequential programs that use concurrent signals. This scheme generates concurrency information, called label, with constant size for the sequential program or every instance of the concurrent signal handlers.

b. Synchronization-Free Approaches: The works we classify in this group do not implement the happens-before at all and therefore, do not track any synchronization primitive. Moreover, these techniques focus only on detecting actual data races [21]. We group them into two methods according to the detection policy they implement:

- **Watchpoint Methods:** Tahara et al. [24] presented an approach based on the `/proc` system file (for Solaris 10) or the debug registers (for IA32 Linux), that uses the watchpoint facilities provided by modern systems to monitor accesses to shared variables for data race detection in sequential programs that use concurrent signal handlers.
- **Breakpoint Methods:** Erickson et al. [10] presented a technique that randomly samples parts of the program to be used as candidates for the data race detection, and uses the code-breakpoint and data-breakpoint facilities provided by many recent hardware architectures to detect data races. Their technique is indeed designed for low-level operating system kernel code.

4.2. Correction Techniques

Detecting concurrency bugs is difficult, but once detected; correcting them is somehow an easy job. In general, there are simple basic rules to meet in order to avoid concurrency bugs in event-driven programs. In common cases, the developer has to enforce the following:

- Disable events before accessing shared data and enable them only after;
- Make sure to use only reentrant functions in event handlers or within user-defined functions called from event handlers;
- Avoid blocking operations or functions in event handlers;
- Define an atomic data type *volatile sig_atomic_t* for shared variables accessed from signal handlers.

Some other rules may be added according to some specific platforms. In nesC applications [11] for example, any update to a shared variable must occur either in a synchronous code reachable only from tasks, or within an atomic section.

5. Conclusion

Event-driven programs are prone to concurrency bugs due to asynchronous event handling. Concurrency bugs are difficult to reproduce making event-driven programs hard to be thoroughly tested and debugged. Concurrency bugs are not only the consequence of a complex concurrency model due to event handlers, but are also due to programmers' mistakes. A programmer may, for example, misuse existing facilities like disabling/enabling events or may make incorrect assumptions during programming; exposing the program to concurrency bugs.

In this paper, we classified the event-driven program models into low and high level based on the type of event they handle, categorized concurrency bug patterns and surveyed existing detection techniques for concurrency bugs in event-driven programs. We believe that such taxonomy can help the developer to understand the causes of concurrency bugs and to avoid introducing them. It can also ease the debugging process, and help developing heuristics for more precise detection tools.

In the future, it might be interesting to investigate and provide useful statistics on bug characteristics including security issues in event-driven programs by conducting an empirical study on real world event-driven software.

Acknowledgement

This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency), NIPA-2011-C1090-1131-0007.

References

[1] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, and G. Jin, “ConSeq: Detecting Concurrency Bugs through Sequential Errors,” ACM SIGPLAN Notices, vol. 46(3), **(2011)**, pp. 251-264.

[2] N. G. Leveson, and C. S. Turner, “An Investigation of the Therac-25 Accidents,” IEEE Computer, vol. 26(7), **(1993)**, pp. 18-41.

[3] E. Farchi, Y. Nir, and S. Ur, “Concurrent Bug Patterns and How to Test Them,” Proceedings of the IEEE Parallel and Distributed Processing Symposium, **(2003)**, pp.22-26.

[4] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, “Dingo: Taming Device Drivers,” Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09), **(2009)**, pp.275-288.

[5] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from Mistakes? A Comprehensive Study on Real World Concurrency Bug Characteristics,” Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, **(2008)**, pp.329-339.

[6] Z. Li, T. Lin, X. Wang, S. Lu, Y. Zhou, and C. Zhai, “Have Things Changed Now? - An Empirical Study of Bug Characteristics in Modern Open Source Software,” Proceedings of the 9th ACM Asian Symposium on Information Display (ASID'06), **(2006)**, pp.25-33.

[7] S. K. Sahoo, J. Criswell, and V. Adve, “An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis,” Proceedings of the ACM International Conference on Software Engineering (ICSE'10), **(2010)**, pp.485-494.

[8] J. Regehr, “Random Testing of Interrupt-Driven Software,” Proceedings of the ACM International Conference on Embedded Software (EMSOFT'05), **(2005)**, pp.290-298.

[9] K. M. R. Audenaert and L. J. Levrouw, “Interrupt Replay: A Debugging Method for Parallel Programs with Interrupts,” Microprocessors and Microsystems, Elsevier, vol. 18(10), **(1994)**, pp. 601-612.

[10] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective Data-Race Detection for the Kernel," Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10), **(2010)**.

[11] D. Gay, P. Levis, R.V. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A holistic Approach to Networked Embedded Systems," Proceedings of the ACM Programming Language Design and Implementation (PLDI'03), **(2003)**, pp.1-11.

[12] G. Gracioli, and S. Fischmeister, "Tracing Interrupts in Embedded Software," Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'09), **(2009)**, pp.137-146.

[13] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue, "An Effective Method to Control Interrupt Handler for Data Race Detection," Proceedings of the 5th International Workshop on Automation of Software Test (AST'10), **(2010)**, pp.79-86.

[14] G. M. Tchamgoue, O.-K. Ha, K.-H. Kim, and Y.-K. Jun, "Lightweight Labeling Scheme for On-the-fly Race Detection of Signal Handlers," Proceedings of the International Conference on Ubiquitous Computing and Multimedia Applications (UCMA'11), **(2011)**, pp.201-208.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System Architecture Directions for Networked Sensors," Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems, **(2000)**, pp.93-104.

[16] A. Baker and J. Lozano, "The Windows 2000 Device Driver Book: A Guide for Programmers," Prentice Hall, Second Edition, **(2009)**, pages 480.

[17] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux Device Drivers," O'Reilly Media, Third Edition, **(2009)**, pages 640.

[18] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative Task Management without Manual tack Management," Proceedings of the 2002 Usenix Annual Technical Conference, **(2002)**.

[19] E. A. Lee, "The Problem with Threads," IEEE Computer, vol. 36(5), **(2006)**, pp. 33-42.

[20] J. K. Ousterhout, "Why Threads are a Bad Idea (for most purposes)," Invited talk at the 1996 USENIX Technical Conference, **(1996)**.

[21] R. H. B. Netzer, and B. P. Miller, "What Are Race Conditions? Some Issues and Formalizations," ACM Letters on Programming Languages and Systems, vol. 1(1), **(1992)**, pp. 74–88.

[22] C. Flanagan and S. N. Freund, "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs," ACM SIGPLAN Notices, vol. 39(1), **(2004)**, pp. 256-267.

[23] M. Dowd, J. McDonald, and J. Schuh, "The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities," Addison-Wesley, 1st Edition, **(2006)**.

[24] T. Tahara, K. Gondow, and S. Ohsuga, "Dracula: Detector of Data Races in Signals Handlers," Proceedings of the 15th IEEE Asia-Pacific Software Engineering Conference (APSEC'08), **(2008)**, pp.17-24.

[25] Z. Lai, S. C. Cheung, and W.K. Chan, "Inter-Context Control-Flow and Data-Flow Test Adequacy Criteria for nesC Applications," Proceedings of the 16th ACM International Symposium on Foundations of software engineering (SIGSOFT'08/FSE-16), **(2008)**, pp.94-104.

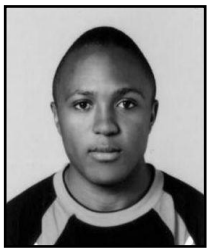
[26] J. Regehr, and N. Cooprider, "Interrupt Verification via Thread Verification," Electronic Notes in Theoretical Computer Science, vol.174, **(2005)**, pp.139-150.

[27] V. Sundaram, P. Eugster, and X. Zhang, "Lightweight Tracing for Wireless Sensor Networks Debugging," Proceedings of the ACM Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens'09), **(2009)**, pp.13-18.

[28] M. Ronsse, J. Maebe, and K. De Bosschere, "Detecting Data Races in Sequential Programs with DIOTA," Proceedings of the Euro-Par 2004, LNCS, vol. 3149, **(2004)**, pp.82-89.

[29] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, Healing Data Races On-The-Fly, Proceedings of PADTAD, pp.54-64, ACM, **(2007)**.

Authors



Guy Martin Tchamgoue is currently enrolled as a PhD candidate with the Department of Informatics in Gyeongsang National University, South Korea. He received his Bachelor of Science, Master of Science and Master of Advanced Studies in Computer Science from the University of Yaoundé I in Cameroon in 2004, 2005 and 2006 respectively. He worked as IT manager for several years in a Cameroonian enterprise. His research interests include event-driven programming and debugging, parallel and distributed computing, operating systems and real-time systems.

Kyong-Hoon Kim received his B.S., M.S., and Ph.D. degrees in Computer Science and Engineering from POSTECH, Korea, in 1998, 2000, 2005, respectively. Since 2007, he has been an assistant professor at the Department of Informatics, Gyeongsang National University, Jinju, Korea. From 2005 to 2007, he was a post-doctoral research fellow at CLOUDS lab in the Department of Computer Science and Software Engineering, the University of Melbourne, Australia. His research interests include real-time systems, Grid and Cloud computing, and security.

Yong-Kee Jun received his BE degree in Computer Engineering from Kyungpook National University in 1980. He obtained his Master and PhD degrees in Computer Science from the Seoul National University in 1982 and 1993 respectively. He is now a Full Professor in the Department of Informatics, Gyeongsang National University (GNU), where he had served as the first director of the GNU Research Institute of Computer and Information Communication (RICIC), and as the first operating director of the GNU Virtual College. He is also the head the Embedded Software Center for Avionics (GESCA), a national IT research Center (ITRC) in South Korea. As a scholar, he has produced both domestic and international publications developed by some professional interests including parallel/distributed computing, embedded systems, and systems software. Professor Jun is a member of the Association for Computing Machinery (ACM) and the IEEE Computer Society.