2013

An Observable and Controllable Testing Framework for Modern Systems

Tingting Yu
University of Nebraska-Lincoln, tyu@cse.unl.edu

Software Engineering (ICSE), 2013 35th International Conference on
Year: 2013
Pages: 1377 - 1380, DOI: 10.1109/ICSE.2013.6606721

An Observable and Controllable Testing Framework for Modern Systems

Tingting Yu
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska, 68588, USA
tyu@cse.unl.edu
http://cse.unl.edu/∼tyu

***Abstract*—Modern computer systems are prone to various classes of runtime faults due to their reliance on features such as concurrency and peripheral devices such as sensors. Testing remains a common method for uncovering faults in these systems. However, commonly used testing techniques that execute the program with test inputs and inspect program outputs to detect failures are often ineffective. To test for concurrency and temporal faults, test engineers need to be able to *observe faults as they occur* instead of relying on observable incorrect outputs. Furthermore, they need to be able to *control thread or process interleavings* so that they are deterministic. This research will provide a framework that allows engineers to effectively test for subtle and intermittent faults in modern systems by providing them with greater *observability* and *controllability*.**

I. Introduction

Modern computer systems are highly concurrent, memory intensive, and sensor intensive. For example, most computer systems currently employ multi-core processors, making concurrent programming a natural way for developers to achieve higher performance. Furthermore, today's embedded systems ranging from consumer electronics to safety-critical devices are equipped with various sensors and peripherals to enable advanced features. These characteristics make these systems very complex and can result in both functional and temporal[1] faults that can be difficult to identify, isolate, and correct. Despite advances in the area of software verification such as model checking, such complexities make these verification techniques infeasible; therefore, testing is still commonly used to assess and find faults in these systems.

To efficiently and effectively test software, developers must be able to *observe* and *control* execution. Where *observability* is concerned, *test oracles* are needed to inspect system behavior for correctness. Unfortunately, testing for faults in modern systems is difficult simply because the classes of faults (e.g., data races) that occur in these systems are often "intermittent" making the traditional testing approaches of using *output-based* oracles ineffective.

Over the past decade, researchers have developed approaches that achieve observability through runtime monitors (e.g., [1]). These approaches tend to focus only on application execution and do not monitor events occurring in lower-level software components such as device drivers and OS modules. As such, they are not capable of revealing subtle faults that can appear in hardware, device drivers, and kernels. In addition, these approaches can obscure lower-level information as they rely on instrumentation which can perturb lower-level system states (e.g., cache, bus, and register usage).

Recently, researchers have investigated test oracles that can detect the presence of faults at internal points in program execution instead of by observing program outputs (e.g., [2]). These *internal oracles* detect whether data or system state manipulations lead the system into some potentially incorrect state. Monitoring internal states rather than outputs should be more effective because it can increase the probability of fault detection. However, internal oracles (as well as output-based oracles) can also fail to detect the presence of faults (producing *false negatives*), and signal the presence of anomalies that are *not* faults (producing *false positives*). There has been little work investigating the relative effectiveness and tradeoffs of internal oracles in practice.

Where *controllability* is concerned, testing techniques must be developed to increase the chance of exposing faults. This means that sources of unpredictability in program execution due to scheduling and interrupt events must be controllable during testing. As an example, to reveal an interrupt related fault, engineers should be able to force interrupts to occur at a particular location. Unfortunately, existing approaches for randomly forcing interrupts (e.g. [3]) are not powerful enough to support such a precise requirement. In the ideal case in which randomly invoking interrupts does expose faults, it may miss faults that can occur due to other interleavings.

Several existing approaches have tried to abstract away scheduling non-determinism in concurrent programs to achieve greater execution control (e.g., [4]). These approaches often control thread scheduling within a single application process. However, they have rarely been adapted to detect concurrency or temporal faults that occur on shared hardware resources or across different applications in modern computer systems.

The goal of our research is to provide a testing framework for modern systems with the aid of *observability* and *controllability* support at lower system layers. To achieve observability,

[1]Faults that occur due to violation of real-time constraints.

we will create a family of internal oracles targeting "hard-to-observe" faults. We will investigate the tradeoffs between internal and output-based oracles, as well as between different internal oracles designed to detect specific fault types. To achieve controllability, we will create techniques for forcing systems to reveal interrupt related concurrency and temporal faults, and process level concurrency faults. Finally, we will empirically evaluate the techniques on real complex software systems.

II. Background and Related Work

A. Background

The classification of modern systems is broad, and we summarize five characteristics of the software running on top of these systems: 1) it can frequently interact with hardware to control system behaviors and thus it is hardware dependent, 2) it can employ different concurrency mechanisms to coordinate threads and processes, 3) it can be programmed with interrupts to interact with the external environment, 4) it can have real-time constraints, and 5) it can produce internal faults that cannot be detected using traditional output-based oracles.

Note that these characteristics may occur to different extents in different systems, and they can also be present in other classes of systems, however they are centrally important to the modern computing systems we are considering in this work, and our solutions must accommodate them.

B. Related Work

There have been many *non-testing-based verification* approaches proposed for use on modern systems (e.g., model checking and static analysis). For example, Brylow et al. [5] apply a model checker to check properties such as interrupt latency. Engler et al. [6] use static analysis based on program state modeling and transitions to verify device drivers and kernels. Tan et al. [7] statically analyze comments and code to detect OS concurrency faults related to interrupts. The drawbacks associated with these approaches involve state explosion, imprecise local information and infeasible paths, and difficulties annotating hardware bit operations.

In this work, we focus on *testing* approaches. In this context, with respect to *observability*, there has been a great deal of research on basic techniques for testing concurrency properties using internal test oracles (e.g., [1]). These techniques enhance observability by monitoring events such as memory access and synchronization operations. However, they do not consider concurrency properties inside lower-level software components (e.g., the kernel), and they may report imprecise results due to unrecognized synchronization primitives.

There has been some research on how internal test oracles affect testing effectiveness. Voas et al. [8] use testability information to identify program locations where assertions may improve testing effectiveness. Staats et al. [2] provide a theoretical foundation describing the importance of oracles in software testing. None of this work considers factors such as false positives and negatives, or the effectiveness of different oracles for certain types of faults.

For *controllability*, there have been some testing techniques that permute thread interleavings to increase the possibility of exposing faults. For example, active testing (e.g., [4]) has been used to test for concurrency faults. These techniques, however, focus on thread level concurrency while ignoring concurrency faults caused by interrupts and different processes. Higashi et al. [9] improve random testing via a mechanism that causes interrupts to occur at all instruction points to detect interrupt related data races. However, this approach can be both ineffective and inefficient as it cannot determine the location at which to issue an interrupt. Laadan et al. [10] note the importance of process races and design a technique to detect and replay them. However, their technique can produce false negatives, and cause replay divergence failures in which the actual system environment does not match the replayed execution.

There has been work on using search based algorithms to test for temporal faults. However, most techniques focus on sequential programs [11]. Iqbal et al. [12] evaluate two search-based algorithms for real-time embedded systems considering interrupts. Briand et al. [13] encode chromosomes by seeding task arrival times, and search for schedules that cause the greatest delays. However, both of these techniques involve black-box testing based on environment or software modeling, and do not consider real system runtime states.

III. Goals and Approaches

The overall goal of this research is to provide a testing framework for modern systems with fine-grained observability and controllability.

A. Activities

To achieve this goal, we propose the following four activities that are mapped to the boxes in Figure 1. We first enumerate these activities, and then elaborate on how we expect to complete each activity.

1. Develop a family of internal oracles at lower software layers to enhance observability for fault detection.
2. Thoroughly investigate the effectiveness of, and trade-offs involved in, the use of internal oracles.
3. Develop techniques to enhance controllability that amplify the chance of revealing faults.
4. Empirically study techniques on real programs.

The first activity aims to address observability problems. This activity will target faults in modern systems that can be hard to detect using existing runtime monitor approaches. Our focus will be on (i) concurrency faults that occur in kernels, (ii) concurrency faults that occur in device drivers and interrupt handlers, (iii) temporal faults caused by interrupts, and (iv) concurrency faults caused by inter-process communication. We will create a family of internal oracles to detect these faults by monitoring program internal states. This corresponds to the box labeled "Internal oracles" in Figure 1. We will use code instrumentation in our initial investigation to detect faults related to concurrent constructs in the kernel (the first box in "Internal oracles" in Figure 1). However, source

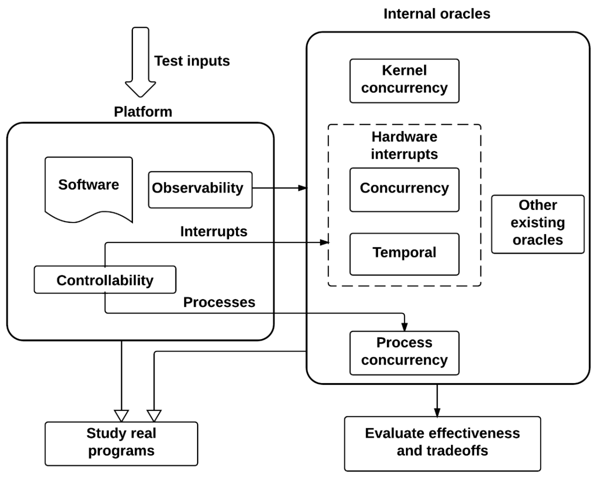

Fig. 1. Overview of Research

code instrumentation is not desirable for modern systems due to their complex and hardware dependent characteristics. Thus, we will utilize virtual platforms (VMs) [14], [15] that can further support testing efforts to help developers non-intrusively uncover faults at lower layers. This corresponds to the "Concurrency", "Temporal", and "Process concurrency" boxes in the "Internal oracles" box in Figure 1.

After developing internal oracles and monitoring techniques, we will study their fault detection effectiveness compared to output-based oracles. Also, we will study tradeoffs involved in using internal oracles. In addition to the internal oracles we have developed, we will include other internal oracles associated with faults that have been well addressed such as memory leaks ("Other existing oracles" box in the "Internal oracles" box in Figure 1), by using existing widely used tools for testing and debugging (e.g.,Valgrind [16]). Since a fault could infect different program points during program execution, different internal oracles may apply to capture the fault. For each fault class of interest, we will define one or more internal test oracles that can be applied to systems to detect the presence of these faults at infection points. We will study the effectiveness (e.g, fault detection) of, and the trade-offs (e.g., false positives and false negatives) involved in, the use of internal oracles. This activity corresponds to the "Evaluate effectiveness and tradeoffs" box in Figure 1.

While conducting these first two activities, we may find that internal oracles provide better fault detection effectiveness than output-based oracles. To amplify effectiveness, we will create techniques to control program execution that force systems to reach error states that are revealable by internal oracles – our third research activity. We will consider three classes of subtle faults including concurrency faults caused by interrupt handling, concurrency faults caused by unsynchronized application processes, and temporal faults caused by intensive interrupts. These are specific to modern systems, and their exposures are highly non-deterministic. Specifically, we will precisely control the occurrences of hardware interrupts so that they test only the locations that have potential races and deadlocks; this corresponds to the "Concurrency" box in the "Internal oracles" box in Figure 1. We will actively control process scheduling to reveal process-level concurrency faults involving improper accessing of shared resources; this corresponds to the "Process concurrency" box in the "Internal oracles" box in Figure 1. To reveal temporal faults, we will use search-based algorithms to generate inputs and interrupt schedules to force real-time constraints to be violated. (We use search-based algorithms because the number of execution paths through the system can grow exponentially with varieties of interrupt souces that can occur at any time.) This corresponds to the "Temporal" box in the "Internal oracles" box in Figure 1.

As for the last activity, we will apply our testing framework with both observability and controllability on a selection of non-trivial software systems. This corresponds to the "Study real programs" box in Figure 1.

B. Scope

The proposed research will be scoped to make its completion feasible in a reasonable amount of time.

We will limit the number of oracles and fault classes that will be explored in the first two activities. We will choose representative fault classes for modern systems, and develop effective oracles to detect these faults; at this time, we expect to identify at least four fault classes in Activity 1, and at least six fault classes in Activity 2. We also expect to create at least three controllability techniques targeting three types of faults. We will study our techniques on at least three non-trivial software systems in Activity 4.

IV. Preliminary Work

We have completed preliminary work towards the first three activities. We briefly discuss the work in this section.

A. Observability

We have developed a family of internal oracles for embedded systems that use instrumentation to record various aspects of execution behavior and compare observed behavior to certain intended system properties that can be derived through program analysis [17]. We focus on several attributes related to proper uses of synchronization primitives and other mechanisms important to managing cooperation and concurrency among tasks (e.g., semaphores, message passing protocols, critical sections, and monitors). Since embedded systems consist of both application and low-level components that should be tested as a whole system [18], our internal oracles operate across system layers.

We have introduced *SimTester*, a testing tool that utilizes VMs to tackle the challenges of testing for concurrency errors involving interactions between software and hardware [19]. We have defined internal oracles involving data races and deadlocks caused by untimely occurrences of interrupts at improperly synchronized code locations. An *execution observer* in SimTester monitors memory accesses and hardware states as a program executes and reports faults related to oracles; the observed information is also used to guide controllability at certain program points. We have evaluated our approach on a release of Linux kernel.

B. Comparison of Internal Oracles

We have conducted an empirical study on several embedded system applications. We have also investigated the tradeoffs between output-based oracles and different internal oracles designed to detect the same fault type but utilizing different execution information [20]. We utilized a family of internal oracles that target faults involving lock management, resource management, interrupt management, critical section protection, and buffer management: faults in modern systems that can lead to well-known and common failures involving data races, deadlock, livelock, critical section violations, and buffer overflow. In our empirical study we used the oracles under consideration in the testing of five concurrent programs and one device driver. We compared the tradeoffs between oracles in terms of effectiveness, false positive rates, and false negative rates. Our results show that internal oracles are substantially more effective at detecting faults than output-based oracles, and that the incidence of false-positive reports associated with them is relatively low, especially when compared to the high incidence of false-negative reports associated with output-based oracles.

C. Controllability

We have developed an *e*xecution controller in SimTester [19] that provides *fine-grained controllability* to actively test for concurrency faults due to interactions between software and hardware. We achieve the level of controllability needed to test such systems by utilizing the VM's abilities to interrupt execution without affecting the states of the virtualized system. Engineers are able to manipulate memory and buses directly to force events such as interrupts and traps. As such, SimTester is able to stop execution at a point of interest monitored by an observability module and force a traditionally non-deterministic event to occur. We have applied SimTester to test for interrupt related data races and deadlocks. It can precisely control the occurrences of interrupts so that it can test every variable that can be accessed by both the application and interrupt handler for vulnerabilities to concurrency faults. SimTester yields precise detection of concurrency faults; that is, it produces no false positives. It is also effective; that is, if races or deadlocks are possible on a shared variable under test, they can be found more easily than with testing approaches that do not incorporate our controllability.

V. Remaining Work

In the future, we are going to enhance the ability of our framework to detect other fault types by iterating over each of the activities listed in Section III-A. First, we will extend our testing framework to detect process level concurrency faults. Second, we will use search-based algorithms to test for temporal faults involving intensive use of interrupts, with the aid of controllability. Third, we will proceed to more comprehensive studies on real software including both effectiveness and cost.

VI. Contributions and Merit

Through the activities described in Section III, this research is expected to develop a VM based testing framework combined with *observability* and *controllability* to effectively detect varieties of faults that are specific to modern systems. Our testing framework aims to bring the notion of observability and controllability into testing for modern computer systems, to provide insights for practitioners and researchers on the use of test oracles, and to provide a prototype tool for future use in academia.

References

[1] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: Proportional detection of data races," in *PLDI*, 2010.

[2] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *ICSE*, 2011.

[3] J. Regehr, "Random testing of interrupt-driven software," in *EMSOFT*, 2005.

[4] K. Sen, "Race directed random testing of concurrent programs," in *PLDI*, 2008.

[5] D. Brylow, N. Damgaard, and J. Palsberg, "Static checking of interrupt-driven software," in *ICSE*, 2001.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *OSDI*, 2000.

[7] L. Tan, Y. Zhou, and Y. Padioleau, "aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *ICSE*, 2011.

[8] J. Voas and K. Miller, "Putting assertions in their place," in *ISSRE*, 1994.

[9] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue, "An effective method to control interrupt Handler for Data Race Detection," in *AST*, 2010.

[10] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, "Pervasive detection of process races in deployed systems," in *SOSP*, 2011.

[11] H. Pohlheim and J. Wegener, "Testing the temporal behavior of real-time software modules using extended evolutionary algorithms," in *GECCO*, 1999.

[12] M. Z. Iqbal, A. Arcuri, and L. Briand, "Empirical investigation of search algorithms for environment model-based testing of real-time embedded software," in *ISSTA*, 2012.

[13] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *GECCO*, 2005.

[14] Virtutech, "Virtutech Simics," Web-page, 2008, http://www.virtutech.com.

[15] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *IPSN*, 2005.

[16] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[17] T. Yu, A. Sung, W. Srisa-an, and G. Rothermel, "Using property-based oracles when testing embedded system applications," in *ICST*, 2011.

[18] A. Sung, W. Srisa-an, G. Rothermel, and T. Yu, "Testing inter-layer and inter-task interactions in RTES application," in *APSE*, 2010.

[19] T. Yu, W. Srisa-an, and G. Rothermel, "Simtester: A controllable and observable testing framework for embedded systems," in *VEE*, 2012.

[20] T. Yu, W. Srisa-an, and G. Rothermel, "An empirical comparison of the cault-detection capabilities of internal oracles," in *ESEM (submitted)*, 2013.