Fall 12-15-2010

TESTING EMBEDDED SYSTEM APPLICATIONS

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

TESTING EMBEDDED SYSTEM APPLICATIONS

by

Tingting Yu

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professors Gregg Rothermel and Witty Srisa-an

Lincoln, Nebraska

December, 2010

TESTING EMBEDDED SYSTEM APPLICATIONS

Tingting Yu, M. S.

University of Nebraska, 2010

Advisors: Gregg Rothermel and Witty Srisa-an

Embedded systems are becoming increasingly ubiquitous, controlling a wide variety of popular and safety-critical devices. Testing is the most commonly used method for validating software systems, and effective testing techniques could be helpful for improving the dependability of these systems. However, there are challenges involved in developing such techniques. Embedded systems consist of layers of software – application layers utilize services provided by underlying system service and hardware support layers. A typical embedded application consists of multiple user tasks. Interactions between application layers and lower layers, and interactions between the various user tasks that are initiated by the application layer, can be a source of system failures in fielded applications. The "oracle problem" is also a challenging problem in many testing domains, but with embedded systems it can be particularly difficult. Embedded systems employing multiple tasks can have non-deterministic output, which complicates the determination of expected outputs for given inputs, and thus, complicates oracle automation.

There are many different classes of real-time embedded systems. For example, hard real-time embedded systems have strict temporal requirements, and include safety critical systems such as those found in avionics and automotive systems. Soft real-time embedded systems, in contrast, occur in a wide range of popular but less safety-critical systems such as consumer electronic devices, and tend not to have such rigorous temporal constraints. In this work, we focus on the latter class of systems.

In this thesis we present an approach for testing soft real-time embedded systems, intended specifically to help developers of soft real-time embedded system applications de-

tect non-temporal faults. Our approach consists, first, of two testing techniques. Our first technique uses dataflow analysis to distinguish points of interaction between specific layers in embedded systems and between individual software components within those layers. Our second technique extends the first to track interactions between user tasks; these are sources of concurrency and other faults common in such systems. We present results of an empirical study that shows that these techniques can be effective. A second aspect of our approach addresses the oracle problem, and the need to provide observability of system behavior sufficient to allow engineers to detect failures. We present a family of property-based oracles 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. We focus on several important *non-temporal* attributes of system behavior; in particular, 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, and critical sections). An empirical study of this oracle approach shows that it can be effective.

ACKNOWLEDGMENTS

My first and biggest thanks goes to my advisor, Dr. Gregg Rothermel. This thesis could not have been completed without him. From him, I learned not only how to do research, but also how to be a good and happy person. He taught me many skills, including how to read research papers, how to talk with people in conferences, how to do time management, and how to brainstorm when I lack new ideas; none of which I had at the beginning of my graduate study. He always believes in me and gives me confidence when I feel I am not smart enough to move forward. I would like to thank him for his patience to guide me from a very basic level, for his encouragement to let me get through every gloomy moment, and for his consideration to help me adapt to a new environment as an international student.

My next big thanks go to my co-advisior Dr. Witty Srisa-an, as this work would not have been possible without him. He spent a lot of time helping me strengthen my background knowledge and inspired and encouraged me when I got stuck. I really appreciate his patience, insights, and enthusiasm during the numerous meetings that we had. I would like to thank him for his constant and valuable advice, for his help in developing ideas for my research topic, and for his kindness and good sense of humor.

I would also like to thank Dr. Matthew Dwyer and Dr. Xue Liu for offering their time to serve as my committee members, and for delivering valuable knowledge and insight.

My special thanks go to Ahyoung Sung. She began this work, and in particular led the efforts described in Chapter 3. I would like to thank her for always sharing her valuable research experience in this area, for her constant encouragement, and for always taking care of me like an older sister. From her, I know how to prepare for meetings with professors, how to be professional when working with other people, how to have a good attitude when facing failures, and how to live a rich life. She helped me grow.

I would like to thank Wayne Motycka for helping with the setup for the empirical study in this work.

I would like to thank my girl friends, Ahyoung, Isis, Katie, Xiao, Zhihong, for sharing gossip and all other girly stuff. It is extremely fun to be with you. I would like to thank all other friends in ESQuaRed lab and UNL. My graduate life could not be that enjoyable without your friendship. I also would like to thank my feline roommate, George, for his companionship during my lonely time.

Last, I would like to thank my parents for absolutely everything. Without their continuous love and encouragement, I could not achieve anything.

This work was supported in part by NSF under grant CNS-0454203, and by the AFOSR through award FA9550-09-1-0129.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems are used to control a wide variety of dynamic and complex applications, ranging from non-safety-critical systems such as cellular phones, media players, and televisions, to safety-critical systems such as automobiles, airplanes, and medical devices. Embedded systems are also being produced at an unprecedented rate, with over four billion units shipped in 2006 [57].

As a prominent example of an embedded system, consider the drive-by-wire systems that are emerging in modern automobiles [32]. In an automobile using a drive-by-wire system, mechanical components such as actuators and cables are replaced by sensors, wires, embedded computers, and software components. Relying on these, the automobile can perform properly regardless of the surrounding environment and operator expertise.

Clearly, systems such as these must be sufficiently dependable; however, there is ample evidence that often they are not. Toyota Corporation has admitted that a "software glitch" is to blame for braking problems in the 2010 Prius, in which cars continue to move even when their brakes are engaged.[1] In addition, a power train control software system used in the 2004/2005 Prius has programming errors that can cause the cars to stall or shut down

[1] CNN news report, February 4, 2010.

while driving at highway speeds.[2] Other faults occurring in embedded systems, such as Ariane [61] crash due to operators environment change; The Therac-25 Tragedies [56] , the Mars Pathfinder [43] failures and many other smaller incidents [40].

There are, however, different classes of embedded systems. Embedded systems can be classified into hard real-time and soft real-time depending on their requirements. Hard real-time embedded systems perform tasks such as digital signal processing and controlling avionics. These systems are often deployed in challenging computing environments, and they have strict temporal requirements. Much of the research on validating hard real-time embedded system has focused on these temporal requirements (e.g., [3, 47, 80, 87]). On the other hand, soft real-time embedded systems are not required to satisfy hard real-time constraints. Examples of these systems include mobile phones, digital cameras, and personal digital assistants. In these systems, timing issues are less important; non-temporal behaviors are also prone to failures, and yet, have been subject to far less research.

There has been a great deal of work on verifying properties of both hard and soft real-time embedded systems using non-testing based approaches. Static analysis is used to enhance the overall quality of software while reducing run-time costs (e.g., [7, 27]). Formal verification techniques are also used (e.g., [10, 12]) by researchers, but these do not address the practical problems and situations that existing in the great majority of industrial settings. In fact, in many industrial settings, testing takes on great importance. For example, our industrial collaborators in the Division of Visual Display at Samsung Electronics Co., Ltd.[3] tell us that testing is the sole methodology used in their division to verify that the televisions, blue-ray disk players, monitors, and other commercial embedded systems applications for display-devices created by Samsung fulfill their functional requirements.

[2]CNN Money report, May 16, 2005

[3]Personal communication, Ahyoung Sung, Senior Engineer, Division of Visual Display, Samsung Electronics Co., Ltd., Suwon, Gyounggi, South Korea, ahyoung.sungsamsung.com.

There has been some research on techniques for testing for soft real-time embedded systems. Some efforts focus on testing applications without specifically targeting underlying software components, and without observing specific underlying behaviors (e.g., [78, 81]) while others focus on testing specific major software components (e.g., real-time operating kernels and device drivers) that underlie the applications (e.g., [5, 8, 48, 82]). Certainly applications must be tested, and underlying software components must be tested, but as we show in Chapter 3, new methodologies designed to capture component interactions, data-access rules, dependency structures, and designers' intended behaviors are needed to effectively test these systems. Approaches presented to date do not adequately do this.

Helping engineers create test cases is important, but test cases are useful only if they can reveal faults. To reveal faults, test cases must produce *observable* failures. Observability requires appropriate *test oracles*. The "oracle problem" is a challenging problem in many testing domains, but with embedded systems it can be particularly difficult. The real-time and concurrency features in multitasking can cause embedded systems to produce non-deterministic outputs.

In this thesis, we address issues that focus on helping developers of embedded system applications use testing techniques to detect faults in their systems, and the oracle problems in testing. We focus on soft real-time embedded systems.

First, we provide techniques for testing embedded systems from the context of applications, focusing on non-temporal faults and targeting interactions between system layers and user tasks; this is the first work to address embedded system dependability in this context.

Second, we report results of an empirical study using two embedded system applications built on a commercial embedded system kernel, that show that our techniques can be effective. In comparison with studies performed in prior work, ours represents a substantial improvement in the state of empirical work in the area of testing embedded systems.

Third, we present a family of "property-based" oracles 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.These can be used during testing to help engineers observe specific system behaviors that reveal the presence of faults. The properties we study here focus on several important *non-temporal* attributes of system behavior; in particular, 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, and critical sections). Furthermore, we track these properties not just across tasks at the application level, but also at lower system levels.

Finally, we present results of an empirical study of our approach applied to two embedded system applications built on a commercial embedded system kernel. Our results show that our property-based oracles can detect faults beyond those detectable by simple output-based oracles.

Chapter 2

Background and Related Work

This chapter provides background information and concepts required to understand this work and its related work. Section 2.1 discusses software testing. Section 2.2 describes background on embedded systems, including their architecture and properties. Section 2.3 discusses related work on verifying embedded systems.

2.1 Software Testing

Software testing is a process, or a series of processes, of *dynamically* executing a program with given inputs, to make sure computer code does what it was designed to do and that it does not do unintended things.

Software testing can be classified as black-box (functional) testing and white-box (structural) testing. The goal of *black-box* testing is to find circumstances in which the program does not behave according to its specifications [20]. In this approach, test data are derived without taking advantage of knowledge of the internal structure of the program [20].

Typical black-box testing techniques include decision table testing [33], all-pairs testing [30], state transition tables [31], equivalence partitioning [70], and boundary value

analysis [70]. As a specific example of a technique referred to in this thesis, the *category-partition method* is a black-box testing technique providing the tester with a systematic method for decomposing a functional specification into test specifications for individual functions. Testers can use a *test specification language* to determine test cases by specifying the interaction of parameters and environment factors relevant to the target program [66].

Generally speaking, using black-box testing is a relatively easy way to create test cases since testers do not have to inspect the internal logic of the program. If a specification is precise and thorough enough, black-box testing can be powerful for revealing faults. However, black-box testing is not easy to fully test a program since it does not look inside the program. This can cause difficulties when a program or a component of system is updated while its corresponding specification is not.

On the other hand, *white-box testing* permits tester to examine the internal structure of the program. That means, a tester is allowed to inspect the code of the program while creating test cases [20]. White-box testing lets testers determine how thoroughly the program needs to be tested, which is often done via *adequacy criterion*. A *test adequacy criterion* is a predicate which is used to determine whether a program has been tested "enough" [30]. Some well known white-box testing criteria include statement testing, branch testing, and path testing, requiring test data to exercise every node, branch, or path in the *control flow graph* of the program, respectively.

Dataflow testing is also a well known white-box testing technique. Dataflow testing selects paths through the program's control flow in order to explore sequences of events related to the status of data objects [11]. There is a family of data flow testing criteria that are applicable for testing programs [30]. Testers are required to generate test cases to cover the selected criteria. More details on data flow testing will be presented in Section 3.2.1.

White-box testing can be more powerful than black-box testing in some aspects. Testers can inspect the implementations of programs, which is helpful for revealing hidden faults.

Testers can also select different testing criteria to satisfy their own needs. However, white-box testing can be more expensive than black-box testing, because it requires testers to understand the internal structure of target program, and typically it requires programs to be instrumented to report coverage. Ultimately, black-box testing and white-box testing are complementary for their individual advantages and limitations.

2.2 Embedded Systems

We begin with an overview of the typical embedded system architecture, and then we discuss some characteristics and properties important to embedded systems that are necessary to understand our approach.

2.2.1 Architecture

Embedded systems are typically designed to perform specific tasks in a particular computational environments consisting of software and hardware components. Figure 2.1 illustrates the typical structure of such a system in terms of four layers – the first three consisting of software and the fourth of hardware.

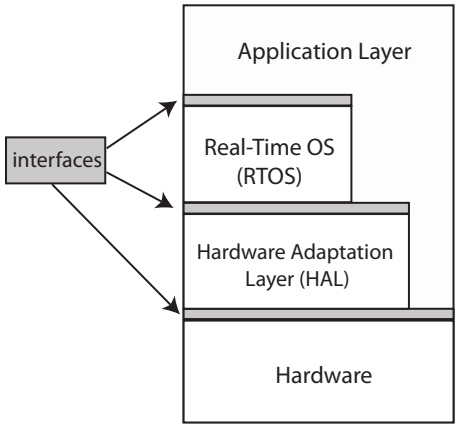

Figure 2.1: Structure of an embedded system

An application layer consists of code for a specific application; applications are written by developers and utilize services from underlying layers including the Real-Time Operating System (RTOS) and Hardware Adaptation Layer (HAL). The gray boxes represent interfaces between these layers. These provide access to, and observation points into, the internal workings of each layer.

An RTOS consists of task management, context-switching, interrupt handling, inter-task communication, and memory management facilities [49] that allow developers to create embedded system applications that meet functional requirements and deadlines using provided libraries and APIs. The kernel is designed to support applications in devices with limited memory; thus, the kernel footprint must be small. Furthermore, to support real-time applications, the execution time of most kernel functions must be *deterministic* [49]. Thus, many kernels schedule tasks based on priority assignment.

The hardware adaptation layer is a runtime system that manages device drivers and provides interfaces by which higher level software systems (i.e., applications and RTOSs) can obtain services. A typical HAL implementation includes standard libraries and vendor specific interfaces. With HAL, applications are more compact and portable and can easily be written to directly execute on hardware without operating system support.

2.2.2 Characteristics of Soft Real-Time Embedded Systems

We summarize five specific characteristics of soft-real time embedded systems. Note that these characteristics can certainly be present in other classes of software systems; however, they are particularly central and defining characteristics of embedded systems

Characteristic 1: Interactions. To illustrate the way in which embedded systems operate, consider an application built to run in a smart phone, whereby a user provides a web address via a mobile web-browser application. The web browser invokes the OS-provided

APIs to transmit the web address to a remote server. The remote server then transmits information back to the phone through the network service task, which instigates an interrupt to notify the web-browser task of pending information. The browser task processes the pending information and sends part of it to the I/O service task, which processes it further and passes it on to the HAL layer to display it on the screen.

This example illustrates three important points:

- Interactions between layers (via interfaces) play an essential role in embedded system application execution.
- As part of embedded system application execution, information is often created in one layer and then flows to others for processing.
- Multiple tasks work together, potentially interacting in terms of shared resources, to accomplish simple requests.

In other words, interactions between the application layer and lower layers, and interactions between the various tasks that are initiated by the application layer (which we refer to as *user tasks*), play an essential role in the operation of an embedded system. This observation is central to our choice of testing methodologies in this work.

Inter-task interaction is also a typical feature in embedded systems. Multiple user tasks execute in parallel, sharing common resources (e.g., CPU, bus, memory, device, global variable, etc.). In addition, strict priority based execution among multi-task applications is a distinguishing feature compared to conventional concurrent software. Faults in such interactions could cause priority inversion, which could result in further execution anomalies. This observation, too, is important in our choice of testing methodologies.

Characteristic 2: Development practices. A second key characteristic involves the manner in which embedded systems are developed. First, software development practices

often separate developers of applications software from the engineers who develop lower system layers. As an example, consider Apple's *App Store*, which was created to allow software developers to create and market iPhone and iPod applications. Apple provided the *iPhone SDK* — the same tool used by Apple's engineers to develop core applications [15] — to support application development by providing basic interfaces that an application can use to obtain services from underlying software layers.

In situations such as this the providers of underlying layers often attempt to test the services they provide, but they cannot anticipate and test all of the scenarios in which these services will be employed by developers creating applications. Thus, on the first day the App Store opened, there were numerous reports that some applications created using the SDK caused iPhones and iPods to exhibit unresponsiveness [15]. The developers of iPhone applications used the interfaces provided by the iPhone SDK in ways not anticipated by Apple's engineers, and this resulted in failures in the underlying layers that affected operations of the devices.

Often, particularly in the realm of portable consumer electronics devices, application developers simply utilize lower system layers that have been created separately by other engineers [21]. In other cases, however, developers must customize lower-level software components to create applications [21]. This is the case at the aforementioned Division of Visual Display at Samsung Electronics, where engineers create various commercial embedded systems applications ranging from televisions to blue-ray disk players, and that are available in a wide range of classes of consumer products. Developers of these systems work initially with a Linux kernel and a wide range of HAL, device drivers, and middleware systems to develop applications. They often must perform low-level programming on these systems, especially HAL and kernel systems, to port their entire applications to various target boards.

In this work, we attempt to provide help for engineers working in both of the foregoing scenarios.

Characteristic 3: Concurrency. Multitasking is one of the most important aspects in embedded system design, and is managed by the RTOS. As in other concurrent programs, multitasking in embedded system applications is usually implemented by classical concurrent approaches, e.g, semaphores, monitors, message queues, barriers, rendezvous, and most of them are implemented as kernel management modules. Since underlying components are more time and resource critical, concurrency is usually implemented in low-level languages, e.g, critical section entry and exit in kernel are implemented by assembly language, for mutual exclusion purposes.

Addressing classes of concurrency aspects in embedded systems is also a goal of this work.

Characteristic 4: Test oracles. Because of thread inter-leaving, embedded systems employing multiple tasks can have non-deterministic output, which complicates the determination of expected outputs for given inputs, and oracle automation. Faults in embedded systems can produce effects on program behavior or state which, in the context of particular test executions, do not propagate to output, but do surface later in the field. Oracles that are strictly "output-based", i.e., focusing on externally visible aspects of program behavior such as writes to *stdout* and *stderr*, file outputs, and observable signals, may fail to detect faults. Thus, designing appropriate *test oracles* can help engineers discover internal program faults when testing embedded systems, and becomes the second primary focus of this work.

Characteristic 5: Timeliness. As described in Chapter 1, timeliness is critical and a major concern in both design and testing phases in hard real-time embedded systems. While in soft-real embedded systems, timing issues may not be that important. A few misses of soft deadlines do no serious harm. Meeting all deadlines is not even the primary consid-

eration. An occasional missed deadline or aborted execution is usually considered tolerable [58]. Our approach is not designed to uncover temporal faults, however, as shown in Section 3.3.5, it is capable of uncovering faults that are sensitive to longer execution time.

2.3 Background: Verifying Embedded Systems

Verification is the process of showing that software meets its specification. If specifications or test oracles are sufficiently precise and informative, verification can be a more objective technique than validation, which is used to validate user requirements. In this thesis, we classify embedded system verification into testing and non-testing-based approaches. We discuss related work for both categories.

2.3.1 Testing

Testing is the process of exercising a product to verify that it satisfies specified requirements or to identify differences between expected and actual results [1]. Input data must be given to the object program, and reactions of the program are observed. In contrast to formal verification, which aims to prove conformance with a predefined specification, the goal of testing is to find cases where the reactions do not meet its expected ones, and we call methods of checking these expected reactions *test oracles*.

Informal specifications have been used to test embedded systems focusing on the application layer. Tsai et al. [81] present an approach for black-box testing of embedded applications using class diagrams and state machines. Sung et al. [78] test interfaces to the kernel via the kernel API and global variables that are visible in the application layer. Cunning et al. [19] generate test cases from finite state machine models built from specifications. All of these techniques operate on the application. The testing approach we

present, in contrast, is a white-box approach, analyzing code in both the application and kernel layers to determine testing requirements.

There has been work on testing kernels using conventional testing techniques. Fault-injection based testing methodologies have been used to test kernels [5, 8, 48] to achieve greater fault-tolerance, and black-box boundary value and white-box basis-path testing techniques have been applied to real-time kernels [82]. All of this work, however, addresses testing of the kernel layer, and none of it specifically considers interactions between layers.

Much of the research on testing embedded systems has focused on temporal faults, which arise when systems have hard real-time requirements (e.g., [3, 80, 87]). Temporal faults are important, but in practice, a large percentage of faults in embedded systems involve problems with functional behavior including algorithmic, logic, and configuration errors, and misuse of resources in critical sections [9, 44].

Our industrial collaborators in the Division of Visual Display at Samsung Electronics Co., Ltd., as discussed above, tell us that in their organization, testing proceeds in several steps. The testing process begins with white-box based unit testing by developers, and black-box based unit testing is done by unit testers. (In this case, the ”unit” are actually functional units defined by the engineers based on requirement and basic architectural analysis.) The most rigorous testing of these systems, however, is applied after system components have been integrated and are being dynamically exercised on simulators and target boards. This rigorous effort is conducted by teams of system testers, in a process that can take as much as four months. At present, system tests are based primarily on expected functional behaviors as inferred from non-formal requirements specifications. Finding approaches by which to better focus this testing on interactions that occur among system components and that can be inferred automatically from the code is a high priority.

There has been work on testing the correctness of hardware behavior using statement and data-flow test adequacy criteria [92]. However, this work does not consider software,

and it focuses on covering definition-use pairs for a hardware description language for circuits, not on interactions between system layers.

There are several methodologies and tools that can generate test cases that can uncover temporal violations or monitor execution to detect violations. In the evolution real-time testing approach [3, 80, 87] evolutionary algorithms are used to search for the worst case execution time or best case execution time based on a given set of test cases. Work by Junior *et al.* [47] creates a light-weight monitoring framework to debug and observe real-time embedded systems. There has also been a great deal of research on testing concurrent programs, of which [13, 55] represent a small but representative subset. Techniques such as these could be complementary to the techniques we.

There has been a great deal of work on using dataflow-based testing approaches to test interactions among system components [35, 42, 67]. With the exception of work discussed below, however, none of this work has addressed problems in testing embedded systems or studied the use of such algorithms on these systems. While we employ analyses similar to those used in these papers, we direct them at specific classes of definition-use pairs appropriate to the task of exercising intratask and intertask interactions within embedded systems.

Work by Chen and MacDonald [16] supports testing for concurrency faults in multi-threaded Java programs by overcoming scheduling non-determinism through value schedules. One major component of their approach involves identifying concurrent definition-use pairs, which then serve as targets for model checking. One algorithm that we present in this thesis (Section 3.2.3) uses a similar notion but with a broader scope. Instead of detecting only faults due to memory sharing in the application layer, our technique detects faults in shared memory and physical devices that can occur across software layers and among tasks.

Work by Lai et al. [50] proposes inter-context control-flow and data-flow test adequacy criteria for wireless sensor network nesC applications. In this work, an inter-context flow graph captures preemption and resumption of executions among tasks at the application layer and computes definition-use pairs involving shared variables. This technique considers a wide range of contexts in which user tasks in the application layer interact, and this is likely to give their approach greater fault detection power than our approach, but at additional cost. However, this work focuses on faults related to concurrency and interrupts at the application layer whereas ours focuses technique focuses on analyzing shared variables in the kernel layer that are accessed by multiple user tasks. Still, as we shall note later (Section 3.2.3), we may be able to improve our technique by adopting aspects of theirs.

There has been a lot of research on testing concurrent programs to reveal faults such as data races [22, 26, 72, 17, 73], deadlock [2, 46], and atomicity violations [60, 86, 34, 51, 68, 29]. Lock based dynamic techniques can predict data races in an execution [72], eg., Eraser records lockset and warnings are reported if there are unprotected variables. Happen-before based dynamic techniques can detect real data races happened in an execution [52]. Some testing techniques can also be used to permute thread interleavings at runtime to expose more possible faults. For example, active testing (e.g., [45, 68]) is used to first identify potential concurrency faults, and then control the underlying scheduler by inserting delays at context switch points.

There has been some work involving testing through SYN-sequences to discover concurrent faults. Non-deterministic testing executes program with the same inputs many times in order to exercise different SYN-sequence, which increases the chance of finding faults [77]. On the other hand, deterministic testing selects a set of SYN-sequences to force programs to deterministically execute with an input; but this requires selecting SYN-sequences derived from static models, such as CFGs [89]. Yu et al. [55] combine both deterministic and non-deterministic approaches to derive SYN-sequences on the fly,

without building any models. These techniques utilize SYN-sequences to guide testing, but they do not provide a way to generate oracles.

One approach to testing concurrent programs is to manipulate synchronization sequences (SYN-sequences) [55, 77, 89]. A notable example of such approaches is the use of sequencing constraints for specification-based testing [14]. A set of constraints is derived from a specification of a program, and nondeterministic testing is used to measure coverage and detect violations according to constraints. Then, additional SYN-sequences are generated to cover the uncovered valid constraints, and deterministic tests are created for these sequences. Unlike these approaches, our approach does not require program specifications; instead, we use basic synchronization properties that can be obtained through component specifications or source code analysis. However, in the cases where program specifications are available this technique can also enhance our capability.

There are also dynamic techniques that can observe concurrency bug patterns without specifications. Such patterns can be treated as one type of oracle. Wang et al. [86] predefine atomicity access patterns associated with locks, and any execution that violates the patterns is reported as a fault. Lu et al. [60] specify violated interleaving access patterns regarding shared variables; an execution that matches the pattern is a fault. These techniques detect only atomicity violations, and they do not take variable accesses and states in underlying components into consideration.

There is some prior research on using concurrency properties to test embedded systems. Higashi et al. [38] detect data races caused by interrupt handlers via a mechanism that causes interrupts to occur at all possible times. They have applied the technique to test the uClinux real time kernel.

Whereas the empirical study of techniques for testing techniques in general has made great strides in recent years, studies of embedded systems have not. Considering just those papers on testing embedded systems that have been cited in this section, many [12, 25, 81]

include no empirical evaluation at all. Of those that do include empirical studies, most studies simply demonstrate technique capabilities, and do not compare the techniques to any baseline techniques [5, 8, 48, 54, 78, 82, 90]. Only two of the papers on embedded systems examine the fault detection capabilities of techniques [50, 78]. Our empirical studies described in Section 3.3 and Section 4.4 go beyond most of this prior work, by examining fault detection capabilities relative to baseline techniques and relative to the use of techniques in common process contexts.

2.3.2 Non-Testing-Based Verification

There have been many non-testing-based verification approaches proposed for use on embedded systems. Although formal verification plays an important role in such approaches, other non-formal techniques will also be discussed in this section.

In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics[88].

Basically, formal verification can be classified into two approaches. One is *theorem proving*, which proves the correctness of a system by theorem derivation. The other one is *model checking*, an approach that does not require complex mathematical proofs, and is more acceptable in software industry.

The essential concept of *model checking* is to check whether a extracted model satisfies a certain specification or property. There are generally three steps involved. First, a model of the target system must be created. Second, system properties must be formalized. Third, a model checker is invoked to make a decision about whether the property is satisfied or not. The advantages of model checking include its ability to perform exhaustive verification, and its capability of proving the absence of faults. Theoretically, this is quite useful for critical

systems requiring absolute reliability, but with the prerequisite that both the system model and model checker themselves are reliable.

There has been a great deal of work involving verifying embedded systems using other forms of formal verification. Bodeveix et al. [12] present an approach for using timed automata to generate test cases for real-time embedded systems. En-Nouaary et al. [25] focus on timed input/output signals for real-time embedded systems and present the timed Wp-method for generating timed test cases from a nondeterministic timed finite state machine. UPPAAL [10] verifies embedded systems that can be modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronization [54]. While these approaches may work in the presence of the required specifications, they are not applicable in the many cases in which appropriate specifications do not exist.

Another common approach used to verify systems with concurrent properties is to apply static analysis techniques to discover paths and regions in code that might be susceptible to concurrency faults [26, 27, 39, 64, 85]. Two notable examples are RacerX [26] and RELAY [85], which are static tools computing the lock sets at each program point, and they are applied to detect races and deadlock in OS kernels. A shortcoming of this approach is that the analysis results can be conservative and imprecise. For example, static analysis techniques based on state modeling and transitions (e.g., [24, 59, 65]) can verify concurrent programs with respect to properties, and have been used to verify correctness of low-level systems including device drivers and kernels [7, 27]. Although static analysis does not involve runtime overhead, one shortcoming of these techniques is state explosion, making them unscalable. Further, due to imprecise local information and infeasible paths, static analysis can report false positives.

Many researchers have created techniques for using dynamic execution information to verify system correctness. Reps et al. [71] use spectra-based techniques that consider execution patterns involving program components. Visser et al. [84] dynamically verify

program properties. These techniques range from heuristics that may signal the presence of anomalies to algorithms that provide guarantees that certain errors do not occur.

There has been one attempt to apply dataflow analysis to the Linux kernel. Yu et al. [90] examine definition and use patterns for global variables between kernel and non-kernel modules, and classify these patterns as safe or unsafe with respect to the maintainability of the kernel.

Chapter 3

Testing Inter-layer and Inter-task Interactions in Embedded Systems [1]

3.1 Introduction

In this chapter, we present an approach for testing embedded systems that focuses on *helping developers of embedded system applications detect non-temporal faults that occur as their applications interact with underlying system components*. Our approach involves two techniques. Technique 1 uses dataflow analysis to identify *intratask interactions* between specific layers in embedded systems and between individual software components within those layers. Technique 2 uses information gathered by Technique 1 to identify *intertask interactions* between the various tasks that are initiated by the application layer. Application developers target both of these classes of interactions when creating test cases.

We present results of an empirical study of our techniques applied to a commercial embedded system. Our results show that both of our techniques are effective at detecting faults. In particular, test suites built initially from specifications and augmented with test

[1] The contents of this chapter have appeared in [79].

cases to achieve coverage of intratask and intertask interactions detect more faults than test suites not thus augmented, and more faults than equivalently sized randomly generated test suites. In focusing on just interactions, however, our techniques are able to restrict testing effort to a relatively small number of coverage requirements.

3.2 Approach and Techniques

The interactions in embedded systems that we discussed in Section 2.2.2 can be directly and naturally modeled in terms of data that flows across software layers and data that flows between components within layers. Tracking data that flows across software layers is important because applications use this to instigate interactions that obtain services and acquire resources in ways that may ultimately exercise faults. Once data exchanged as part of these transactions reaches a lower layer, tracking interactions between components within that layer captures data propagation effects that may lead to exposure of these faults.

For these reasons, we believe that testing approaches that focus on data interactions can be particularly useful for helping embedded system application programmers detect non-temporal faults in their systems. We thus begin by considering dataflow-based testing approaches in the embedded systems context.

There are two aspects of data usage information that can be useful for capturing interactions. First, observing the actual processes of defining and using values of variables or memory locations can tell us whether variable values are manipulated correctly by the software systems. Second, analyzing intertask accesses to variables and memory locations can also provide us with a way to identify resources that are shared by multiple user tasks, and whether this sharing creates failures. Our approach involves two techniques designed to take advantage of these two aspects of data usage information.

The first technique statically analyzes an embedded system at the *intratask* level to calculate data dependencies representing interactions between layers of those systems. The second technique calculates data dependencies representing *intertask* interactions. The data dependencies identified by these two techniques become coverage targets for testers.

3.2.1 Dataflow Analysis and Testing

Dataflow analysis (e.g., [62]) analyzes code to locate statements in which variables (or memory locations) are defined (receive values) and statements where variable or memory location values are used (retrieved from memory). This process yields *definitions* and *uses*. Through static analysis of control flow, the algorithms then calculate data dependencies (*definition-use pairs*): cases in which there exists at least one path from a definition to a use in control flow, such that the definition is not redefined ("killed") along that path and may reach the use and affect the computation.

Dataflow test adequacy criteria (e.g., [69]) relate test adequacy to definition-use pairs, requiring test engineers to create test cases that cover specific pairs. Dataflow testing approaches can be *intraprocedural* or *interprocedural*; in this work we utilize the latter, building on algorithms presented in [35, 67].

Dataflow analysis algorithms and dataflow test adequacy are not themselves novel; however, our interest here is not in creating novel algorithms, but in discovering effective methods for testing embedded systems.

From our perspective on the embedded system testing problem there are several things that render dataflow approaches interesting and somewhat different. In particular, we can focus solely on data dependencies that correspond to interactions between system layers and components of system services. These are dependencies involving global variables, APIs for kernel and HAL services, function parameters, physical memory addresses, and

special registers. Memory addresses and special registers can be tracked through variables because these are represented as variables in the underlying system layers. Accesses to physical devices can also be tracked because these devices are mapped to physical memory addresses. Many of the variables utilized in embedded systems are represented as fields in structures, which can be tracked by analyses that operate at the field level. Attending to the dependencies in variables thus lets us focus on the interactions we wish to validate. Moreover, restricting attention to these dependencies lets us operate at a lower cost than required if we aimed to test all definition-use pairs in a system.

In the context of testing embedded systems, we also avoid many costs associated with conservative dataflow analysis techniques. Conservative analyses such as those needed to support optimization must track all dependencies, and this in turn requires relatively expensive analyses of aliases and function pointers (e.g., [53, 76]). Testing, in contrast, is just an heuristic, and does not require such conservatism. While a more conservative analysis may identify testing requirements that lead to more effective test data, a less expensive but affordable analysis may still identify testing requirements that are sufficient for detecting a wide variety of faults.

3.2.2 Intratask Testing

As we have discussed, an application program is a starting point through which underlying layers in embedded systems are invoked. Beginning with the application program, within given user tasks, interactions then occur between layers and between components of system services. Our first algorithm uses dataflow analysis to locate and direct testers toward these interactions.

Our technique includes three overall steps: (1) procedure *ConstructICFG* constructs an interprocedural control flow graph ICFG, (2) procedure *ConstructWorklist* constructs

a worklist based on the ICFG, and (3) procedure *CollectDUPairs* collects definition-use pairs. These procedures are applied iteratively to each user task T_k in the application program being tested.

Step 1: Construct ICFG. An ICFG [67, 75] is a model of a system's control flow that can support interprocedural dataflow analysis. An ICFG for a program P begins with a set of control flow graphs (CFGs) for each function in P. A CFG for P is a directed graph in which each node represents a statement and each edge represents flow of control between statements. "Entry" and "exit" nodes represent initiation and termination of P. An ICFG augments these CFGs by transforming nodes that represent function calls into "call" and "return" nodes, and connecting these nodes to the entry and exit nodes, respectively, of CFGs for called functions.

In embedded system applications, the ICFG for a given user task T_k has, as its *entry point*, the CFG for a main routine provided in the application layer. The ICFG also includes the CFGs for each function in the application, kernel, and HAL layers that comprise the system, and that could possibly be invoked (as determined through static analysis) in an execution of T_k. Notably, these functions include those responsible for interlayer interactions, such as through (a) APIs for accessing the kernel and HAL, (b) file IO operations for accessing hardware devices, and (c) macros for accessing special registers. However, we do not include CFGs corresponding to functions from standard libraries such as `stdio.h`. Algorithms for constructing CFGs and ICFGs are well known and thus, due to space limitations, we refer the reader to [62, 67] for details.

Step 2: Construct Worklist. Because the set of variables that we seek data dependence information on is sparse we use a worklist algorithm to perform our dataflow analysis. Worklist algorithms (e.g., [75]) initialize worklists with definitions and then propagate these around ICFGs. Step 2 of our technique does this process by initializing our Worklist to every defi-

nition of interest in the ICFG for T_k. Such definitions include variables explicitly defined, created as temporaries to pass data across function calls or returns, or passed through function calls. However, we consider only variables involved in interlayer and interprocedural interactions; these are established through global variables, function calls, and `return` statements. We thus retain the following types of definitions:

1. definitions of global variables (the kernel, in particular, uses these frequently);
2. actual parameters at call sites (including those that access the kernel, HAL, and hardware layer);
3. constants or computed values given as parameters at call sites (passed as temporary variables);
4. variables given as arguments to `return` statements;
5. constants or computed values given as arguments to `return` statements (passed as temporary variables).

The procedure for constructing a worklist considers each node in the ICFG for T_k, and depending on the type of node (entry, function call, `return` statement, or other) takes specific actions. Entry nodes require no action (definitions are created at call nodes). Function calls require parameters, constants, and computed values (definition types 2-3) to be added to the worklist. Explicit `return` statements require consideration of variables, constants, or computed values appearing as arguments (definition types 4-5). Other nodes require consideration of global variables (definition type 1). Note that, when processing function calls, we do consider definitions and uses that appear in calls to standard libraries. These are obtained through the APIs for these libraries – the code for the routines is not present in our ICFGs.

Step 3: Collect DUPairs. Step 3 of our technique collects definition-use pairs by removing each definition from the Worklist and propagating it through the ICFG, noting the uses it reaches.

Our Worklist lists definitions in terms of the ICFG nodes at which they originate. *CollectDUPairs* transforms these into elements on a Nodelist, where each element consists of the definition d and a node n that it has reached in its propagation around the ICFG. Each element has two associated stacks, `callstack` and `bindingstack`, that record calling context and name bindings occurring at call sites as the definition is propagated interprocedurally.

CollectDUPairs iteratively removes a node n from the Nodelist, and calls a function *Propagate* on each control flow successor s of n. The action taken by *Propagate* at a given node s depends on the node type of s. For all nodes that do not represent function calls, `return` statements, or CFG exit nodes, *Propagate* (1) collects definition-use pairs occurring when d reaches s (retaining only pairs that are interprocedural), (2) determines whether d is killed in s, and if not, adds a new entry representing d at s to Nodelist, so that it can subsequently be propagated further.

At nodes representing function calls, *CollectDUPairs* records the identity of the calling function on the `callstack` for n and propagates the definition to the entry node of the called function; if the definition is passed as a parameter *CollectDUPairs* uses the `bindingstack` to record the variable's prior name and records its new name for use in the called function. *CollectDUPairs* also notes whether the definition is passed by value or reference.

At nodes representing `return` statements or CFG exit nodes from function f, *CollectDUPairs* propagates global variables and variables passed into f by reference back to the calling function noted on `callstack`, using the `bindingstack` to restore the variables' former names if necessary. *CollectDUPairs* also propagates definitions that have

originated within f or functions called by f (indicated by an empty `callstack` associated with the element on the Nodelist) back to *all* callers of f.

The end result of this step is the set of intratask definition-use pairs, that represent interactions between layers and components of system services utilized by each user task. These are coverage targets for engineers when creating test cases to test each user task.

3.2.3 Intertask Testing

To effect intertask communication, user tasks access each others' resources using shared variables (SVs), which include physical devices accessible through memory mapped I/O. SVs are shared by two or more user tasks and are represented in code as global variables with accesses placed within critical sections. Programmers of embedded systems use various approaches to control access to critical sections, including calling kernel APIs to acquire/release semaphores, calling the HAL API to disable or enable IRQs (interrupt requests), or writing 1 or 0 to a control register using a busy-waiting macro.

Our aim in intertask analysis is to identify intertask interactions by identifying definition-use pairs involving SVs that may flow across user tasks. Our algorithm begins by iterating through each CFG G that is included in an ICFG corresponding to one or more user tasks. The algorithm locates statements in G corresponding to entry to or exit from critical sections, and associates these "CS-entry-exit" pairs with each user task T_k whose ICFG contains G.

Next, the algorithm iterates through each T_k, and for each CS-entry-exit pair C associated with T_k it collects a set of SVs, consisting of each definition or use of a global variable in C. The algorithm omits definitions (uses) from this set that are not "downward-exposed" ("upward-exposed") in C; that is, definitions (uses) that cannot reach the exit node (cannot be reached from the entry node) of C due to an intervening re-definition. These definitions

and uses cannot reach outside, or be reached from outside, the critical section and thus are not needed; they can be calculated conservatively through local dataflow analysis within C. The resulting set of SVs are associated with T_k as *intertask definitions and uses*.

Finally, the algorithm pairs each intertask definition d in each T_k with each intertask use of d in other user tasks to create intertask definition-use pairs. These pairs are then coverage targets for use in intertask testing of user tasks.

3.2.4 Recording Coverage Information

Both of the techniques that we have described require access to dynamic execution information, in the form of definition-use pairs. This information can be obtained by instrumenting the target program such that, as it reaches definitions or uses of interest, it signals these events by sending messages to a "listener" program which then records definition-use pairs. Tracking definition-use pairs for a program with D pairs and a test suite of T tests requires only a matrix of Boolean entries of size $D * T$.

A drawback of the foregoing scheme involves the overhead introduced by instrumentation, which can perturb system states. There has been work on minimizing instrumentation overhead for applications with temporal constraints (e.g., [28]), and runtime monitoring techniques for real-time embedded systems (e.g., [93]). Currently, the focus of our methodology is to make functional testing more effective; thus, it does not directly address the issue of testing for temporal faults. Furthermore, there are many faults that can be detected even in the presence of instrumentation, and the use of approaches for inducing determinism into or otherwise aiding the testing of non-deterministic systems (e.g., [13, 55]) can further assist with this process.

3.3 Empirical Study

While there are many questions to address regarding any testing technique, the foremost question is whether the technique is effective at detecting faults, and it is this question that we address here. Proper empirical studies of new testing techniques, however, should not simply examine the performance of those techniques; rather, they should compare the techniques to baseline techniques or techniques representing current practice.

In the case of our techniques, it is difficult to find an appropriate baseline technique to compare against, because (as Section 2.3 shows) there simply are none that share our specific focus: non-temporal faults related to interactions between layers and user tasks in embedded systems, that arise in the context of executions of applications. Comparing our approach to standard coverage-based approaches is not appropriate because these target code components at a much finer granularity than our approach. We have found two approaches, however, for addressing this difficulty in relevant ways.

Our first approach is to consider our coverage criteria within the context of a current typical practice for testing embedded system applications. Engineers currently often use black-box test suites designed based on system parameters and knowledge of functionality to test embedded system applications. Ideally, engineers should augment such suites with additional test cases sufficient to obtain coverage. While prior research on testing suggests that coverage-based test cases complement black-box test cases, the question still remains whether, in the embedded system context, our techniques add additional fault detection effectiveness in any substantive way. Investigating this question will help us assess whether our techniques are worth applying in practice. Thus, our first evaluation approach is to begin with a suite of black-box (specification-based) test cases and augment that suite to create coverage-adequate test suites using our techniques.

This approach also avoids a threat to validity. If we were to independently generate black-box and coverage-based test suites and compare their effectiveness, differences in effectiveness might be due not to differences between the techniques, but rather, to differences in the specific inputs chosen for test cases across the two techniques. By beginning with a black-box test suite and extending it such that the resulting coverage-adequate suite is a superset of the black-box suite, we can assert that any additional fault detection is due to the additional power obtained from the additional test cases created to satisfy coverage elements.

The second approach we use to study our techniques is to compare test suites generated by our first approach with equivalently sized randomly generated test suites. This lets us assess whether the use of our testing criteria does in fact provide fault detection power that is not due to random chance.

We thus consider the following research questions:

RQ1: Do black-box test suites augmented to achieve coverage in accordance with our first two techniques achieve better fault detection than test suites not so augmented?

RQ2: Do test suites that are coverage-adequate in accordance with our first two techniques achieve better fault detection than equivalently-sized randomly generated test suites?

3.3.1 Objects of Analysis

Researchers studying software testing techniques face several difficulties – among them the problem of locating suitable experiment objects. Collections of objects such as the Software-artifact Infrastructure Repository [23] have made this process simpler where C and Java objects are concerned, but similar repositories are not yet available to support studies of testing of embedded systems. A second issue in performing such studies in-

volves finding a platform on which to experiment. Such a platform must provide support for gathering analysis data, implementing techniques, and automating test execution, and ideally it should be available to other researchers in order to facilitate further studies and replications.

As a platform for this study we chose rapid prototyping systems provided by Altera [4], a company with over 12,000 customers worldwide and annual revenue of 1.2 billion USD.[2] The Altera platform supports a feature-rich IDE based on open-source Eclipse, so plug-in features are available to help with program analysis, profiling, and testing.

The Altera platform runs under MicroC/OS-II, a commercial grade real-time operating system. It is certified to meet safety-critical standards in medical, military and aerospace, telecommunication, industrial control, and automotive applications. The codebase contains about 5000 lines of non-comment C code [49]. For academic research, the source-code of MicroC/OS-II is freely available, so other studies may utilize it.[3]

For this study we chose two application programs that run on Altera, The first, MAILBOX, involves the use of mailboxes by several user tasks. When one of these tasks receives a message, the task increments and posts the message for the next task to receive and increment. The second application, CIRCULARBUFFER, involves the use of semaphores by two user tasks, and a circular buffer that operates as a message queue by three tasks. As for the semaphore tasks, when one task acquires a semaphore, the number of semaphores acquired by this task is increased by one, and then the semaphore is released. As for the message queue tasks, when one task sends a message, the task increments the number of sent messages by one, and then one of the receiving tasks receives the message and increments the number of received messages by one.

[2] http://www.altera.com/corporate/about_us/abt-index.html.

[3] Researchers may also obtain all of the objects, artifacts, and data obtained in our study by contacting the author.

We chose these two applications because they are of manageable size but also are relatively complex in terms of user tasks and usage of lower layer components. Table 3.1 provides basic statistics on these objects, including the numbers of lines of non-comment code in the application and lower layers, and the numbers of functions in the application or invoked in lower layers.

Table 3.1: Object Program Characteristics

Program	*Application*			*Kernel and HAL*			*Definition-use Pairs*		*Test Cases*	
	LOC	Funcs	Faults	LOC	Funcs	Faults	Intratask	Intertask	Black-box	White-box
MAILBOX	268	9	14	2380	32	31	253	65	26	24
C.-BUFFER	304	17	17	2814	38	49	394	96	40	44

While in absolute terms these code bases might seem small, they are typical of many single-purpose embedded systems. Moreover, the complexity of a program depends on its run-time behavior, inputs, and source code, and even small programs such as ours can have complex and long-running behavior.

Finally, while it would be preferable to study a larger set of object programs, the cost of doing this is quite large. The preparation of the two objects used in this study (e.g., environment configuration), and the conduct of the study, required between 1900 and 2200 hours of researcher time. Further studies of additional artifacts will ultimately be possible as additional artifacts become available, but as we shall show, the artifacts studied here do yield informative results.

To address our research question we required faulty versions of our object program, and to obtain these we followed a protocol introduced in [41] for assessing testing techniques. We asked a programmer with over ten years of experience including experience with embedded systems, but with no knowledge of our testing approach, to insert potential faults into the MAILBOX and CIRCULARBUFFER applications and into the MicroC/OS-II system code. Potential faults seeded in the code of underlying layers were related to kernel initialization, task creation and deletion, scheduling, resource (mailbox) creation, resource handling (posting and pending), interrupt handling, and and critical section management.

Given the initial set of potential faults, we again followed [41] in taking two additional steps to ensure that our fault detection results are valid. First we determined, through code inspection and execution of the program, seeded faults that could never be detected by test cases. These included code changes that were actually semantically equivalent to the original code, and changes that had been made in lower layer code that would not be executed in the context of our particular application. None of these are true faults in the context of our object program and retaining them would distort our results, and thus we eliminated them. Second, after creating test cases for our application (described below), we eliminated any faults that were detected by more than 50% of the test cases executed. Such faults are likely to be detected during unit testing of the system, and are thus not appropriate for characterizing the strength of system-level testing approaches.This process left us with the numbers of faults reported in Table 3.1 (the table lists these separately for the application and Kernel/HAL levels).

3.3.2 Factors

Independent Variable. Our independent variable is the testing approach used, and as discussed in relation to our research questions there are two:

- Begin with a set of black-box test cases, and augment this with additional test cases to ensure that all executable intratask and intertask definition-use pairs identified by our first two techniques are exercised.
- Randomly generate test suites of the same size as the test suite created by the foregoing procedure.

Dependent Variable. As a dependent variable we focus on *effectiveness* measured in terms of the ability of techniques to reveal faults. To achieve this we measure the numbers of faults in our object program detected by the different techniques.

Additional Factors. Like many embedded system applications, MAILBOX and CIRCULARBUFFER each run in an infinite loop. Although semantically a loop iteration count of two is sufficient to cause each program to execute one cycle, different iteration counts may have different powers to reveal faulty behavior in the interactions between user tasks and the kernel. To capture these differences we utilized two iteration counts for all executions, 10 and 100. 10 represents a relatively minimal count, and 100 represents a count that is an order of magnitude larger, yet small enough to allow us to conduct the large number of experimental trials needed in the study within a reasonable time-frame.

For each application of each testing approach, we report results in terms of each iteration count.

3.3.3 Study Operation

To create initial black-box test suites we used the category-partition method [66], which employs a Test Specification Language (TSL) to encode choices of parameters and environmental conditions that affect system operations (e.g., changes in priorities of user tasks) and combine them into test inputs. This yielded initial black-box test suites of sizes reported in Table 3.1.

To implement our techniques we used the ARISTOTLE [36] program analysis infrastructure to obtain control flow graphs and definition and use information for C source code. We constructed ICFGs from this data, and applied our techniques to these to collect definition-use pairs. To monitor execution of definitions, uses, and definition-use pairs we embedded instrumentation code into the source code at all layers.

The application of our dataflow analysis techniques to our object programs identified definition-use pairs (see Table 3.1). We executed the TSL test suites on the programs and determined their coverage of inter-layer and inter-task pairs. We then augmented the initial TSL test suite to ensure coverage of all the executable inter-layer and inter-task definition-use pairs.[4] This process resulted in the creation of additional test cases as reported in Table 3.1.

To create random test suites we randomly generated inputs for MAILBOX and CIRCULARBUFFER, which was feasible since the input consists of a set of integers representing process priorities. To control for variance in random selection we generated three different test suites of size 50.

We next executed all of our test cases on all of the faulty versions of the object programs, with one fault activated on each execution. We used a differencing tool on the outputs of the initial and faulty versions of the program to determine, for each test case t and fault f, whether t revealed f. This allows us to assess the collective power of the test cases in the individual suites by combining the results at the level of individual test cases.

3.3.4 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our object programs, faults, and test suites. Other systems may exhibit different behaviors and cost-benefit tradeoffs, as may other forms of test suites. However, the system we utilized is a commercial kernel, the faults were inserted by an experienced programmer, and the test suites are created using practical processes. And our data set is substantially larger than those used in prior studies of testing techniques applied to embedded systems.

[4]Through this process and further inspection we determined which of the definition-use pairs were infeasible — 12 in MAILBOX and 26 in CIRCULARBUFFER — and eliminated them from consideration; the Table lists the numbers of feasible pairs.

The primary threat to internal validity for this study is possible faults in the implementation of our techniques, and in the tools that we use to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can determine correct results.

Where construct validity is concerned, fault detection effectiveness is just one variable of interest. Other metrics, including the costs of applying the technique and the human costs of creating and validating tests, are also of interest.

3.3.5 Results

We now present and analyze our results, focusing on each research question in turn. Section 3.3.6 provides further discussion.

3.3.5.1 Research Question 1

Our first research question, again, concerns the relative fault-detection abilities of an initial specification-based test suite versus test suites augmented to cover the intratask and intertask definition-use pairs identified by our first two algorithms. Table 3.2 provides fault detection totals for the initial specification-based (TSL) suite, and the suite as augmented to cover definition-use pairs (TSL+DU). The table shows results for each of the iteration counts, and shows results for faults in the application and lower layers separately.

Table 3.2: Fault Detection, RQ1 (TSL vs TSL+DU)

Program	*Layer*	*Iter. Count = 10*		*Iter. Count = 100*	
		TSL	TSL+DU	TSL	TSL+DU
MAILBOX	Application	8	13	10	14
	Kernel/HAL	14	18	14	18
CIRCULARBUFFER	Application	3	11	3	11
	Kernel/HAL	9	24	10	24

As the data shows, for each subject, at both the application and lower levels, and at both iteration counts, the augmented coverage-adequate (TSL+DU) test suites improved fault detection over TSL suites by amounts ranging from 29% to 63% in MAILBOX, and from 140% to 267% in CIRCULARBUFFER. The TSL suites detected at best 10 of the 14 application level faults in MAILBOX, while the augmented suites detected all 14. The TSL suites detected at best 3 of the 17 application level faults in CIRCULARBUFFER, while the augmented suites detected at best 11. The TSL suites detected at best 14 of the 31 lower layer faults in MAILBOX, while the augmented suites detected 18, leaving 13 undetected. The TSL suites detected at best 10 of the 49 lower layer faults in CIRCULARBUFFER, while the augmented suites detected 24, leaving 25 undetected.

3.3.5.2 Research Question 2

Our second research question concerns the relative fault-detection abilities of the augmented coverage-adequate test suites (TSL+DU) compared with equivalently-sized random test suites (RAND). Table 3.3 compares the two, in the same manner as Table 3.2. Note that each of the three random test suites that we executed exhibited the same overall fault detection capabilities (the suites differed in terms of how many test cases detected each fault, but not in terms of which faults were detected). Thus, the entries in the RAND columns represent results obtained for each random suite.

Table 3.3: Fault Detection, RQ2 (RAND vs TSL+DU)

Program	*Layer*	*Iter. Count = 10*		*Iter. Count = 100*	
		RAND	TSL+DU	RAND	TSL+DU
MAILBOX	Application	11	13	11	14
	Kernel/HAL	12	18	12	18
CIRCULARBUFFER	Application	2	11	2	11
	Kernel/HAL	9	24	10	24

In this case, for each subject, at both the application and lower levels and for both iteration counts, the augmented coverage-adequate (TSL+DU) test suites improved fault

detection over equivalently sized random suites by amounts ranging from 18% to 33% in MAILBOX, and from 140% to 450% in CIRCULARBUFFER.

3.3.6 Discussion

The results we have reported thus far all support the claim that our techniques can be advantageous in terms of fault detection, in the context of embedded system applications. We have focused thus far on our research questions; we now turn to additional observations.

To address RQ1 we compared initial black box (TSL) test suites with test suites augmented to achieve coverage, and this necessarily causes our augmented (TSL+DU) suites to detect a superset of the faults detected by the initial black-box suites. Overall, across both iteration counts, the TSL suites detected 24 of the 45 faults present in MAILBOX, and the TSL+DU suites detected 32; the TSL suites detected 13 of the 66 faults present in CIRCULARBUFFER, and the TSL+DU suites detected 35.

We also compared the fault detection data associated with just the TSL test cases with the data associated with just those test cases that were added to the initial suites (DU test cases) to attain coverage. It is already clear from Table 3.2 that at iteration counts 10 and 100, across all software layers, in MAILBOX, DU test cases detected nine faults at level 10, and eight at level 100, that were not detected by TSL test cases; in CIRCULARBUFFER, DU test cases detected twenty-three faults at level 10, and twenty-two at level 100, that were not detected by TSL test cases. TSL test cases, however, did detect one fault (in the kernel layer, at both iteration counts) that was not detected by any of the DU test cases in both MAILBOX and CIRCULARBUFFER. In other words, the two classes of test cases can be complementary in terms of fault detection – although in our study the DU test cases exhibited a great advantage.

Our results also show that some faults are sensitive to execution time; thus our approach, though not designed explicitly to detect temporal faults, can do so. However, this occurred infrequently – at the application level only and for just three faults, each of which was revealed at iteration count 100 but not iteration count 10.

In our study, the faults detected by test cases targeting intertask definition-use pairs were a subset of those detected by test cases targeting intratask pairs. In MAILBOX, the intertask test cases detected 8 of the 14 faults found by the intratask test cases in the application, and 13 of the 18 faults found by the intratask test cases in the lower system levels. In general, however, this subset relationship need not always hold. In CIRCULARBUFFER, the intertask test cases detected 6 of the 9 faults found by the intratask test cases in the application, and 14 of the 23 faults found by the intratask test cases in the lower system levels. However, 2 faults detected by intertask test cases are not detected by intratask test cases in the case of CIRCULARBUFFER. Moreover, with 253 and 394 intratask definition-use pairs and just 65 and 96 intertask pairs in the two subjects, the fact that test cases designed for Technique 2 detected these numbers of faults is encouraging, and suggests that in resource-constrained testing situations, engineers might begin with Technique 2 and still achieve relatively useful results. It is also possible that we can improve Technique 2 by adapting inter-context control-flow and data-flow analysis [50] to analyze iterations among our user tasks at the application layer (see Section 2.3.1).

We also further analyzed our data to see which types of faults were revealed. We were able to determine that both the TSL test cases and the DU test cases can detect faults related to invalid priority values (e.g., priority already in use, priority higher than the allowed maximum), deletion of idle tasks, resource time-outs, and NULL pointer assignments to resources in the kernel layer. The DU test cases, however, detected more faults related to invalid uses of resources (e.g., misused data types of resources, attempts to pend resources in interrupt service routines, misuse of resource posting) and faults related to running out

of available space for allocating data structures needed by the kernel (e.g., task control blocks). Also, the fault types that were detected only by the DU (and not the TSL) test cases were not detected by the random test cases, suggesting that targeting specific definition-use pairs does indeed have specific fault-detection power.

By restricting our attention to specific types of intratask and intertask definition-use pairs we are able to lower testing costs, because the total number of definition-use pairs in our object program using a standard interprocedural analysis on all definitions and uses is more than our restricted focus. Moreover, none of the 318 pairs we targeted were infeasible in MAILBOX, and only 26 pairs were infeasible in CIRCULARBUFFER. While we cannot assert that these results will generalize, it is possible that the restricted sets of definition-use pairs identified by our algorithms may have higher probabilities than others of being feasible. Further study of this issue is needed.

3.4 Summary

We have presented a new approach for use in testing embedded systems. Our approach focuses on interactions between layers and user tasks in soft real-time embedded systems, to expose non-temporal failures that developers face as they create new applications. We have conducted an empirical study applying our techniques to a commercial embedded system, and demonstrated that they can be effective at revealing faults.

Chapter 4

Using Property-based Oracles to Enhance Testing in Embedded System Applications [1]

4.1 Introduction

In Chapter 3, we presented an approach meant to help developers of embedded system applications detect faults that occur as their applications interact with underlying system components. We presented results of an empirical study of our techniques that showed that they can be effective.

As described in Chapter 2, observability requires appropriate oracles, which is a challenging problem in the embedded system context where multiple tasks can have non-deterministic output. In this chapter we address this observability problem. We present a family of "property-based" oracles that use instrumentation to record various aspects of execution behavior and compare observed behavior to certain intended system properties

[1] The contents of this chapter will be appearing in [91].

that can be derived through program analysis. These can be used during testing to help engineers observe specific system behaviors that reveal the presence of faults. The properties we study here focus on several important *non-temporal* attributes of system behavior; in particular, 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, and critical sections). Furthermore, we track these properties not just across tasks at the application level, but also at lower system levels.

We present results of an empirical study of our approach applied to two embedded system applications built on a commercial embedded system kernel. Our results show that our property-based oracles can detect faults beyond those detectable by simple output-based oracles.

4.2 Related Work

One major characteristic of our oracle approach is the use of synchronization properties as test oracles. Our synchronization properties are similar to those used in model checking (e.g., [18, 37, 63]), a formal verification technique that exhaustively tests an extracted model. For example, Corbett et al. [18] automatically extract a finite-state model from Java source code, and use it to verify properties formalized as temporal logic. Artho et al. [6] automatically generate properties expressed as temporal logic, from a given test input; the observer module in their framework checks program behavior from program execution traces against generated sets of properties. Our work also extracts synchronization properties by inferring behaviors from component interfaces and/or program semantics; however, instead of exploring full states, our approach relies on a suite of test inputs that can change runtime behaviors, exploring partial states and avoiding the state explosion problem. These inputs can include those such as priority order, task sleeping time, and task notification dis-

ciplines (e.g., unblock one task, unblock all tasks). We then employ low-level observation points to observe how these test inputs affect the ability of applications to conform to these synchronization properties.

A second major characteristic of our oracle approach is that, unlike most of the approaches discussed in this section, it does not assume that underlying concurrent constructs are correctly implemented. This is important, because many embedded systems are developed as customized applications, using lower-level software components (runtime services and libraries) that are themselves heavily customized by in-house software developers. In such cases, developers cannot treat lower level components as black boxes in testing.

In Section 2.3.1, we also discussed related work on using concurrency properties to test embedded systems [38]. One major difference between this approach and ours is that we do not artificially amplify the frequency of events to evaluate whether these events can cause failures. Our technique identifies observation points in the low-level runtime systems to observe interactions among system layers. The runtime systems are not configured to behave unrealistically except for instrumentation at observation points.

4.3 Property-based Oracles

In this section, we describe our approach for creating test oracles based on properties of various synchronization primitives and common interprocess communications. There are three general steps involved in applying the approach:

1. Create test oracles based on properties – this can be done by analyzing the source programs to extract their interfaces and the program semantics of particular software components.

2. Execute test cases on the system and generate runtime trace information – this requires instrumentation of the source programs, libraries, and runtime systems.

3. Analyze trace information to detect violations – this involves checking generated traces against oracles to verify conformance with properties of interest.

Our approach involves creating properties relative to particular embedded systems platforms, and thus, some properties will be platform dependent. Once created, however, these properties are relevant to all applications built on those platforms.

Also, it is possible to generalize the specification of a property if that property is widely accepted or part of a standard. For example, our approach includes creating an oracle for a binary semaphore, the property for which can be found in any operating system textbook [74].

In addition, while the instrumentation of low-level runtime systems such as kernels can be dependent on those systems, the overall approach can be generalized (e.g., most OSs employ a similar conceptual view of binary semaphores).

It should also be noted that our technique might false-positively report errors in some pathological cases. As an example, one property that our approach can detect is proper pairing of critical section enter and exit statements. It is possible that in some scenarios, a critical section exit might be omitted without causing errors (e.g., a branch in a critical section might lead directly to program termination). In this scenario, our technique would still detect this as an error.

Next, we illustrate the application of our approach to support property-based testing of two synchronization primitives in MicroC/OS-II: *semaphore* and *message queue*. We then apply the approach to *critical sections*. We select these three primitives because, while in principle any primitives could be checked with test cases created by any approach, these three can all arguably be better checked by test suites created in accordance with our test

adequacy criteria. This is because semaphores, message queues, and other shared resources are modeled in embedded systems as variables and buffers, and their usage and interactions can thus be tracked through inter-layer and inter-task definition-use pairs.

4.3.1 Binary Semaphore

A semaphore is an integer variable that is accessed through two atomic operations: *pend* and *post*. The pend operation decrements the semaphore value by one. If the semaphore value is already 0, the task that tries to perform the pend operation is blocked. The post operation increments the semaphore value by one if there are no tasks waiting on the semaphore. If there are tasks waiting, the semaphore value remains at 0, and the waiting task that has the highest priority is admitted directly into the semaphore. A binary semaphore can only contain a value of 0 or 1.

To apply our approach to such semaphores, we first create an oracle based on state transition information. To accomplish this task, we reason about the properties of *pend* and *post* to calculate and encode all possible correct state transitions for a semaphore.

We encode state information using a triple, S, that contains (i) the identification of a task that performed an operation (pend, post, or initialize) on the semaphore, (ii) the semaphore value after the operation, and (iii) the execution state of that task after the operation (*B* for blocked state and *R* for ready state). We record operation information using a tuple A that contains (i) the identification of the task that performs the operation, and (ii) the actual operation performed on a semaphore. For each operation, we encode the associated state transition as triples; each triple contains $<S_i,A_j,S'_j>$, where *i* and *j* denote task identifications and *i and j can but need not differ*. Here, S_i represents the state before the operation, A_j represents the operation (pend or post), and S'_j represents the state after the operation.

Format:
sem_cnt = $\{0,1\}$
state = $\{$B,R$\}$ //B for blocked, R for ready
A = $\{(T$,PEND),$(T$,POST)$\}$
S, S' = $\{(T$,sem_cnt,state)$\}$
= $\{(T$,0,R),$(T$,1,R),$(T$,0,B)$\}$

Oracle:
Case 1: accessed by the same task
$<(T_i$,1,R);$(T_i$,PEND);$(T_i$,0,R)$>$
$<(T_i$,0,R);$(T_i$,POST);$(T_i$,1,R)$>$

Case 2: accessed by different tasks
$<(T_i$,1,R);$(T_j$,PEND);$(T_j$,0,R)$>$
$<(T_i$,0,R);$(T_j$,POST);$(T_j$,1,R)$>$
$<(T_i$,0,R);$(T_j$,PEND);$(T_j$,0,B)$>$
$<(T_i$,0,B);$(T_j$,PEND);$(T_j$,0,B)$>$
$<$ALL$\{(T_i$,0,B)$\}$;$(T_j$,POST);
(highest priority task in ALL$\{T_i\}$,0,R)$\wedge(T_j$,0,R)$>$

Figure 4.1: Correct states for binary semaphores.

Each of these operation triples represents a state transition due to an operation performed on the semaphore.

Figure 4.1 shows the result of applying this process to the states and transitions relevant to MicroC/OS-II.In the last triple, the semaphore value remains 0 after a post because MicroC/OS-II directly admits a waiting task with the highest priority to the semaphore.

Next, we wish to gather dynamic information from program execution, to check against the oracle. To do this we need to observe semaphore states; thus we instrument kernel functions related to semaphore operations and task scheduling to generate trace data including task id, operation, semaphore identification and value, and execution state (ready or blocked). For MicroC/OS-II these functions are *OSSemCreate*, *OSSemPend*, *OSSemPost*, *OS_EventTaskWait*, and *OS_EventTaskRdy*. We then execute the instrumented system with test cases, generating traces. Next, we process the generated traces to obtain information in the form represented in the oracle. Finally, we compare the processed data against the oracle to detect anomalies.

Note that our technique can scale to work with multiple semaphores. To achieve this, we identify transitions based on operations on each semaphore (e.g., a set of transitions for *sem1* and another set for *sem2*). We can then check each set of transitions against the oracle.

To illustrate an application of our technique we provide an example. Suppose we are using a kernel that contains a faulty implementation of binary semaphore. Suppose that an instance of the faulty semaphore (*sem1*) is used in a program with three tasks (T_{main}, T_1, and T_2). The semaphore is created and initialized to 1 by T_{main} so the initial state is (T_{main},1,R). T_1 then performs a pend operation on *sem1*, and the triple representing the state transition of *sem1* ($Trans_1$) is <(T_{main},1,R);(T_1,PEND);(T_1,0,R)>. Later, T_2 performs a pend operation on *sem1*, and the triple representing the state transition of *sem1* ($Trans_2$) is <(T_1,0,R);(T_2,PEND);(T_2,0,R)>. Some time later, the program terminates without any obvious failures.

By comparing the dynamically generated state transitions with the oracle, we can detect that $Trans_2$ does not match any of the transitions in the oracle, and therefore, is incorrect. When T_1 previously performed a pend operation on the semaphore, it left the semaphore value at 0. As such, the pend operation performed by T_2 should have caused T_2 to be blocked. However, this is not the case here as the trace clearly shows that the state of T_2 remains at ready (R) after the pend operation. If the semaphore had operated correctly, the correct transition would have been <(T_1,0,R);(T_2,PEND);(T_2,0,B)>.

4.3.2 Message Queue

MicroC/OS-II provides *message queues* as a way to support indirect communication. In this mechanism, messages are sent to and received from a circular buffer with a user-specified size. In the current implementation, MicroC/OS-II returns an OS_Q_FULL error

code when a task tries to post a message to a fully occupied queue. In addition, applications that use this facility typically prevent a task from posting a message to a full buffer (e.g., by putting the task in a busy-wait loop). Messages are consumed from the queue in FIFO order. When the queue is empty, tasks that try to receive messages are blocked. Application engineers can specify that the highest priority task or all tasks must unblock when a message is deposited into an empty queue.

To create the oracle for the message queue, we reason about the send and receive operations to calculate all correct state transitions. The oracle is the generalization of these transitions (Figure 4.2). This process is similar to the one used to generate state transition information for binary semaphores. We encode the state information using a triple S but replacing the semaphore value with the number of messages in the buffer. We also record send and receive operations using a tuple A. We represent transitions using triples; each triple contains $<S_i,A_j,S'_j>$, where *i* and *j* denote task identifications and *i and j can but need not differ*. Note that the correct state of a message queue in the fourth triple of case 2 depends on whether the system is configured to unblock only the highest priority task waiting for a message or all the waiting tasks.

We next run tests on the program to generate execution traces. We process these traces to generate states of the circular buffer and state transitions based on send and receive operations. We then compare the dynamically generated information with the oracle to detect anomalies. The process is similar to that shown in the example used to illustrate fault discovery with the binary semaphore.

We next run tests on the program to generate execution traces. We process these traces to generate states of the circular buffer and state transitions based on send and receive operations. We then compare the dynamically generated information with the oracle to detect anomalies. The process is similar to that shown in the example used to illustrate fault discovery with the binary semaphore.

Format:
msg_cnt= {empty;num;full}, where (empty $<$ num $<$ full)
state = {B,R}
A = {(T,RECV),(T,SEND)}
S, S' = {(T,msg_cnt,state)}
= {(T,num,R),(T,full,R),(T,empty,B),(T,empty,R)}

Oracle:
Case 1: accessed by the same task
<(T_i,empty,R);(T_i,RECV);(T_i,empty,B)>
<(T_i,empty,R);(T_i,SEND);(T_i,1,R)>
<(T_i,num,R);(T_i,RECV);(T_i,num-1,R)>
<(T_i,num,R);(T_i,SEND);(T_i,num+1,R)>
<(T_i,full,R);(T_i,SEND);(T_i,full,R)>
<(T_i,full,R);(T_i,RECV);(T_i,full-1,R)>

Case 2: accessed by different tasks
<(T_i,empty,R);(T_j,RECV);(T_j,empty,B)>
<(T_i,empty,R);(T_j,SEND);(T_j,1,R)>
<(T_i,empty,B);(T_j,RECV);(T_j,empty,B)>
<ALL{(T_i,empty,B)};(T_j,SEND);
broadcast?(ALL{T_i}:highest priority task in ALL{T_i},empty,R)>
<(T_i,num,R);(T_j,RECV);(T_j,num-1,R)>
<(T_i,num,R);(T_j,SEND);(T_j,num+1,R)>
<(T_i,full,R);(T_j,SEND);(T_j,full,R)>
<(T_i,full,R);(T_j,RECV);(T_j,full-1,R)>

Figure 4.2: Correct states for message queue

4.3.3 Critical Section

Our approach to property-based testing of critical sections differs somewhat from the approaches used for semaphores and message queues. In the foregoing approaches we define oracles, and these are directly used to compare against dynamic information. In the case of critical sections, we require an additional static analysis step to provide the data that dynamic information is compared against.

The basic property of a critical section is that its contents are mutually exclusive in time. That is, when a task is executing in a critical section, no other task is allowed to execute in the same critical section [74]. This also means that given a critical section in user task T_k

that contains a sequence of definitions and uses of shared variables (SVs), these definitions and uses must be executed without intervening accesses by other user tasks to the SVs.

To observe critical section behavior in this regard, we could instrument functions related to critical section entry and exit across all layers, as well as definitions and uses of shared variables within each critical section. By statically calculating the interlayer sequences of definitions and uses of SVs (*SV-sequences*) that are associated with critical sections in a given embedded system application, we can then later compare dynamic sequences obtained in testing against expected sequences and check for anomalies.

We describe the two steps by which this process is accomplished first, and then we provide the oracle.

Step 1: Calculate static SV-sequences. To calculate static SV-sequence information for a given critical section k, we apply an algorithm that utilizes an interprocedural control flow graph (ICFG) [67] for the application under test, and performs a depth-first traversal of the portion of the ICFG created for a given user task T_i beginning at the critical section entry node CS_{enter_k} for k.

During this traversal the algorithm identifies and records the SVs encountered. This process results in the incremental construction of SV-sequence information as the traversal continues. If the algorithm reaches a predicate node in the ICFG during its traversal, it creates independent copies of the SV-sequences collected thus far to carry down each branch. If a back edge is encountered (i.e., there is a loop), the collected SVs in this branch are enclosed by parentheses, followed by a $+$, meaning that such SVs can be repeated.

On reaching the critical section exit (CS_{exit_k}), the algorithm generates CS_k, which contains the set of collected SV-sequences, CS_{enter_k}, and CS_{exit_k} for task T_i and critical section k.[2]

[2]Collection of SV-sequences requires that all critical sections have properly paired entry and exit constructs on all possible paths; however, the algorithm detects cases where this does not hold and informs the user.

Oracle:

CS_k: static SV-sequences in critical section k for task T_i.

1. For each CS_{enter_k} of task T_i there must be a matching CS_{exit_k} retrieved from CS_k of T_i.
2. A critical section must not be simultaneously accessed by multiple tasks.
3. A dynamic sequence between CS_{enter_k} and CS_{exit_k} for task T_i must match with its corresponding static du sequence retrieved from CS_k of T_i.

Figure 4.3: Correct properties for critical section

Step 2: Calculate dynamic SV-sequences. To observe critical section behavior, we instrument functions related to critical section entry and exit, as well as shared variables within the critical section. Instrumentation is done at the boundaries of each critical section across all layers and at each definition and use within it. For MicroC/OS-II, the boundaries are *OSSemPend, OSSemPost, OS_ENTER_CRITICAL, OS_EXIT_CRITICAL, alt_irq_enable*, and *alt_irq_disable*.

Next, the program is run with tests and dynamic definition-use information is collected, for each user task, on SVs occurring in critical sections. This information is then processed with a focus on the SV-sequences associated with each critical section to obtain sets of *dynamic SV-sequences*. Finally, the sets of dynamic SV-sequences (CS_k) are validated against the static SV-sequences in accordance with the properties listed in the oracle shown in Figure 4.3.

4.3.4 Discussion on Effects of Instrumentation

As noted above, we instrument the kernel to obtain observability of low-level runtime systems. Such instrumentation can change system states (e.g., cache, bus, and register usage) and prolong execution time. As a way to eliminate the effects of instrumentation on system

states, we are working to create a non-intrusive instrumentation framework which will be further discussed in Section 5.2.

4.4 Empirical Study

Our property-based oracles are intended to help engineers detect faults in embedded system applications that might not be detectable by output-based oracles alone. We thus designed an empirical study to examine whether this occurs.

We thus consider the following research questions:

RQ1: Are property-based oracles effective at detecting faults and can they detect faults not detected by output-based oracles?

RQ2: Do specific property-based oracles have abilities to detect faults not detected by others, or detected only at specific system layers?

4.4.1 Object of Analysis

As objects of analysis we use the same two programs utilized in our first study, MAILBOX and CIRCULARBUFFER, as described in Section 3.3.1.

4.4.2 Factors

Independent Variable. Our independent variable is the oracle approach used. We consider two: output-based and property-based. As property-based oracles we implemented those described in Section 4.3.

To provide output-based oracles, on MAILBOX, we used a differencing tool on the outputs of the initial and faulty versions of the program to determine, for each test case t

and fault f, whether t revealed f. This allows us to assess the power of the test suites by combining the results measured for individual test cases.

On CIRCULARBUFFER a simple output-differencing approach would not suffice, because like many embedded systems its output can be non-deterministic. However, engineers faced with such systems typically create partial oracles that can check important aspects of system behavior, and we took this approach. On outputs that correspond to exceptional behavior resulting in error messages, output does remain deterministic and was simply differenced. In other cases, we utilized several automatable checks of expected output; namely, (1) "the number of messages received must be less than or equal to the number sent;" (2) "the semaphore must be held by the task whose number of semaphores acquired is not equal to zero"; and (3) "deterministic portions of output strings must match expected contents."

Dependent Variable. As a dependent variable we focus on *effectiveness* in terms of the ability of the oracles to reveal faults. To achieve this we measure the numbers of faults in our object programs detected by the two oracle approaches.

Additional Factors. The one additional factor we consider here involving loop count is the same with factor considered in the first study described in Section 3.3.2

4.4.3 Study Operation

As test suites we utilized the code coverage-based test suites engineered specifically to provide coverage of inter-layer and inter-task definition-use pairs from first study (see Section 3.2).

4.4.4 Threats to Validity

This study shares the same threats to validity as the first (see Section 3.3.4). In addition, we compare our property-based oracles against just two particular output-based oracles.

4.4.5 Results

We now present and analyze our results, focusing on each research question in turn. Section 4.4.6 provides further discussion.

4.4.5.1 Research Question 1

Figure 4.4 provides an overview of our fault detection data. The figure displays segmented bar graphs, one per object program. Each bar graph partitions the faults detected in the program into three disjoint subsets. The white segment of the bar corresponds to faults detected by output oracles only, the black segment corresponds to faults detected by property-based oracles only, and the gray segment corresponds to faults detected by both types of oracles.

As the figure shows, 36 of the 55 faults in MAILBOX were detected, and 47 of the 66 faults in CIRCULARBUFFER were detected. Of the 36 faults detected in MAILBOX, 34 were detected by output-based oracles and 14 were detected by property-based oracles. Of the 47 faults detected in CIRCULARBUFFER, 36 were detected by output-based oracles, and 26 were detected by property-based oracles. Clearly, output-based oracles were more effective than property-based oracles at detecting faults overall, but in the absence of output-based oracles, property-based oracles would still have revealed substantial numbers of faults.

Considering the data further, in MAILBOX, two faults were detected *only* by property-based oracles, and in CIRCULARBUFFER, 11 were detected *only* by property-based oracles. Property-based oracles thus also displayed the potential to detect faults not caught by output-based oracles.

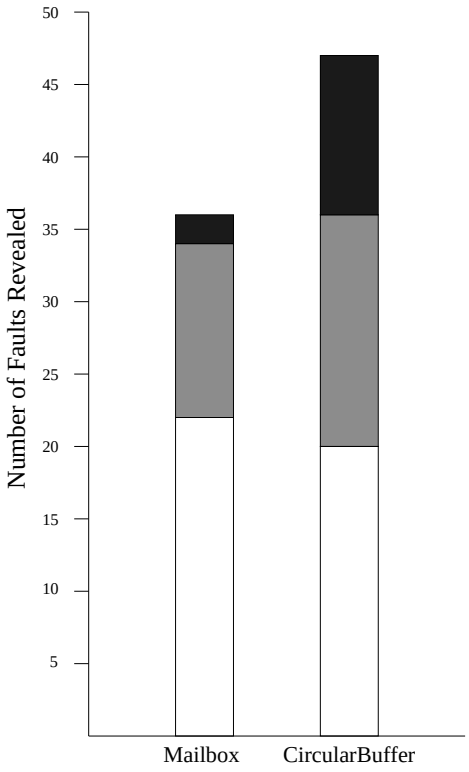

Figure 4.4: Overview of fault detection data

4.4.5.2 Research Question 2

Figure 4.5 uses a similar graphical approach to partition results further, displaying fault detection per program and property-based oracle type, separated by the system layer in which the faults resided (application versus kernel/HAL). For CIRCULARBUFFER, results are shown for each of the three property-based oracle types. For MAILBOX, results are shown only for the critical section property-based oracle because MAILBOX does not utilize semaphores or message queues. Note that in the case of CIRCULARBUFFER, the sets of faults detected per property-based oracle type are not disjoint; that is, some faults are detected by multiple property-based oracles. Layers, however, do form disjoint partitions on faults per program; e.g., of the 36 faults detected in MAILBOX, 16 were in the application layer and the other 20 were in the kernel/HAL layers.

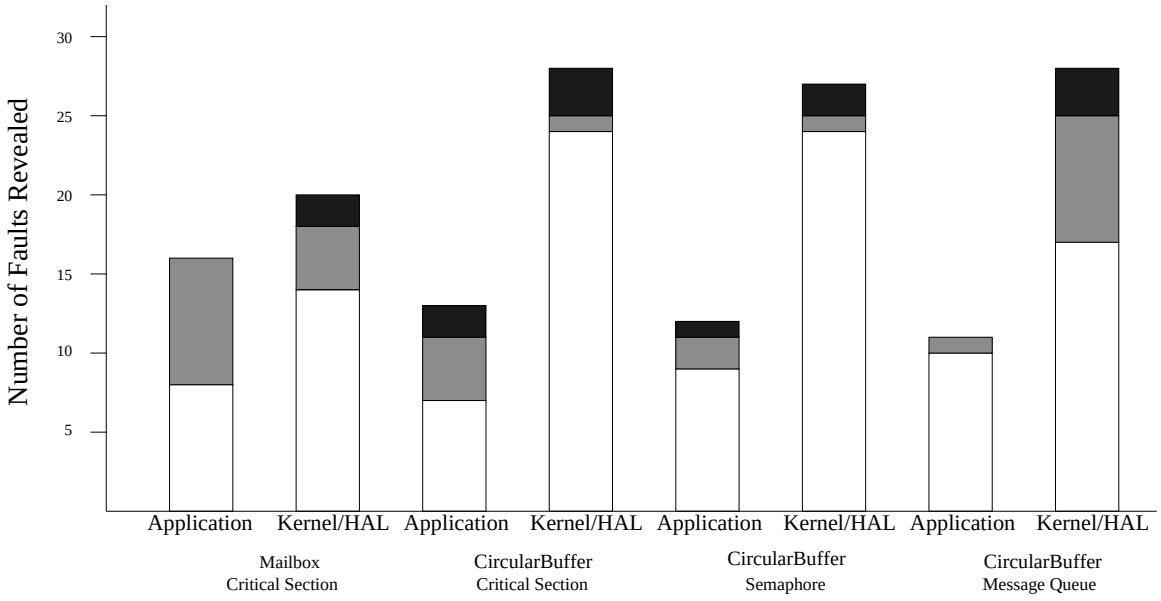

Figure 4.5: Detailed fault detection data

We first consider results across property-based oracles. All three such oracles were successful in revealing faults. The message queue oracle revealed the most faults, and the semaphore oracle the fewest, on CIRCULARBUFFER. More significantly, the property-based oracles each revealed faults that were not revealed by the output-based oracles. The critical section oracle revealed seven such faults, the semaphore oracle revealed three, and the message queue oracle revealed three. Further, each of the 13 faults that were revealed only by a property-based oracle were actually revealed *only* by *one* particular property-based oracle. In other words, each property-based oracle exhibited the potential to reveal faults not revealed by any other property-based oracle.

We next consider differences in fault detection results across system layers. Faults detected by *both* property-based and output-based oracles occurred in equal numbers in the application and kernel/HAL layers (15 each). Faults detected *only* by property-based oracles, however, resided disproportionally in the kernel/HAL layers. More precisely, at the application layer, across all properties, property-based oracles detected only three faults that were not detected by output-based oracles (all on CIRCULARBUFFER), while at the

kernel/HAL layers, property-based oracles detected 10 faults that were not detected by output-based oracles (two on MAILBOX and eight on CIRCULARBUFFER).

4.4.6 Discussion

While additional studies are needed, if the results we have just reported generalize, then:

- The property-based oracles we have defined are effective at detecting faults, and can detect faults not detected by output-based oracles.
- Each property-based oracle demonstrated the ability to detect faults not detected by others.
- Property-based oracles were particularly effective at detecting faults in the kernel/HAL layers.

These results have several implications. They suggest that output-based and property-based oracles are complementary in fault detection effectiveness and ideally both should be used. However, in cases where output-based oracles are not operable, property-based oracles can still be useful. Moreover, in cases where output-based oracles require more substantial human investment than property-based oracles, it may be preferable to apply these oracles before output-based oracles, because if they do identify failing test cases this obviates the need to spend time validating the outputs of those test cases.

We also observed a much larger number of failures revealed only by property-based oracles on CIRCULARBUFFER (11) than on MAILBOX (2). Of course, more of the property-based oracles were applicable to the former program, but the result might also reflect the use of a partial oracle for CIRCULARBUFFER. We conjecture that on programs where weaker output-based oracles are required, property-based oracles could have even greater power to reveal faults that might not otherwise be seen.

Examining the data further yielded additional observations. First, while our property-based oracles could conceivably produce false warnings, they did not do so in any case in our studies. Inspection of our results revealed that in every case in which a property-based oracle signaled a problem in our study, a fault caused the problem.

Second, in Section 4.4 we noted that iteration counts can conceivably cause differences in program behavior relative to fault detection. In our study we observed such differences only in four cases, all involving faults detected by the critical section property-based oracle (three on `Mailbox` and one on `CircularBuffer`). In all four of these cases, faults were detected at iteration level 100 but not at iteration level 10. Choice of iteration level thus can matter, but in our case it did not do so extensively.

Third, as noted in Section 4.4, the test suites that we utilized were composed of black box test cases, augmented with test cases needed to achieve coverage of inter-layer and inter-task definition-use pairs (as defined in Section 3.3.) Of the 40 faults detected by property-based oracles, 32 were detected by the initial black-box test cases, and 39 were detected by the white-box test cases. This result has two implications: (1) when using specification-based (TSL) test cases, property-based oracles can be effective, but their effectiveness increases when test suites are augmented to cover inter-layer and inter-task definition-use pairs; (2) just the test cases created to add additional coverage above those used initially for black-box testing were adequate to achieve most of the property-based fault detection. This result lends credence to our conjecture (Section 4.3) that the dataflow coverage-based test cases created by our approach can be particularly effective at detecting the failures that are detectable using our specific property-based oracles.

Fourth, four of the faults present in MAILBOX involved changes in task sleeping times at the application layer. Such faults are common because sleeping times are embedded as constants by programmers and can easily be chosen incorrectly. Our black-box test cases to detect all four of these faults using output-based oracles at iteration count 10. Critical

section analysis, however, revealed three of these at iteration count 10 using the same test cases. While all three of these faults were revealed by additional coverage-based test cases, and were revealed by black-box test cases at iteration count 100, this result does illustrate the potential power of critical section analysis to detect faults in situations where particular test cases themselves (together with a choice of iteration level) might not.

Finally, we analyzed our data to determine what types of faults were revealed only by our property-based oracles. Of these 13 faults, many have to do with incorrect task initialization, improper semaphore creation, incorrect pairing of critical section enter and exit statements, and incorrect bit operations in the scheduler or wait list, causing tasks to be incorrectly blocked or unblocked. There are two particular faults that are especially interesting since they do not result in incorrect outputs or abnormal program terminations. The first fault has to do with incorrect usage of a binary operator in a kernel's interrupt service routine; this results in simultaneous accesses to a critical section by multiple tasks. Another fault involves incorrectly unblocking tasks. In this scenario, the system is configured to unblock all tasks when a message arrives; however, due to incorrect usage of a unary operator in the post operation, only the highest priority task is unblocked. While neither fault causes incorrect system output on the specific test cases employed, both could emerge during execution in the field.

4.5 Summary

We have presented an approach for using property-based oracles when testing embedded systems. We have conducted an empirical study applying our techniques to two commercial embedded system applications, and demonstrated that they can be effective at revealing faults.

Chapter 5

Conclusion and Future Work

In this chapter we conclude and summarize the main contributions of our work. Finally, we present our future work.

5.1 Conclusion

First, we have proposed a new technique for testing embedded system applications, targeting non-temporal faults that occur due to interactions between system layers and user tasks.

Second, we have conducted a substantial empirical study using two embedded system applications, `Mailbox` and `CircularBuffer`, built on MicroC/OS-II, an RTOS kernel. This empirical study has demonstrated the potential effectiveness of our approach.

Third, we have presented a family of "property-based" oracles using instrumentation to record program behaviors across system layers, and comparing the observed behavior to specified intended properties. This approach can help engineers observe internal system behaviors that can reveal faults. The properties that we have considered involve non-temporal

attributes of system, particularly related to concurrency management among multiple user tasks.

Finally, we have conducted an empirical study of our oracle approach on the `Mailbox` and `CircularBuffer` applications. Our results show that property-based oracles can detect faults that output-based oracles cannot.

5.2 Future Work

The work presented in this thesis is a first milestone in our research on testing embedded system applications. There are several other avenues for future work.

First, we have seen the potential for creating other types of property-based oracles, including those for high-level concurrent programming constructs for multi-task embedded software (e.g., monitors). At the same time, additional empirical work is needed.

Second, recent developments in the area of simulation platforms support testing of embedded systems. One notable development is the use of full system simulation to create virtual platforms [83] that can be used to observe various execution events while yielding minimum perturbation to the system states. As part of our future work, we will explore the use of these virtual platforms to create a non-intrusive instrumentation environment to support our testing techniques. We will utilize such virtual environments to capture complex runtime information, such as shared resource access, memory usage, to identify system resource usage and its execution behaviors.

Third, we will build a testing framework targeting more embedded system specific properties using virtual platforms. We will use both static and dynamic analysis to identify system attributes that can affect temporal behaviors of soft real-time embedded systems. Then we will create test inputs and task interleavings to target such attributes. We will use

checkpoint features of the virtual platform to design testing criteria that are more effective in detecting temporal faults causing by soft real-time variances.

Fourth, we will perform additional substantial empirical studies to evaluate our techniques.

Bibliography

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990.

[2] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 51–60, 2006.

[3] Noura Al Moubayed and Andreas Windisch. Temporal white-box testing using evolutionary algorithms. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 150–151, 2009.

[4] Altera Corporation. Altera Nios-II Embedded Processors. http://www.altera.com/products/devices/nios/nio-index.html.

[5] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of cots microkernel-based systems. *IEEE Trans. Comput.*, 51:138–163, February 2002.

[6] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Rosu, and Willem Visser. Experiments with test case gen-

eration and runtime analysis. In *Proceedings of the abstract state machines 10th international conference on Advances in theory and practice*, pages 87–108, 2003.

[7] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 73–85, 2006.

[8] R. Barbosa, N. Silva, J. Duraes, and H. Madeira. Verification and validation of (real time) cots products using fault injection techniques. In *Proceedings of the Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, pages 233–242, 2007.

[9] Sean Beatty. Sensible software testing. http://www.embedded.com/2000/0008/0008feat3.htm, 2000.

[10] G. Behrmann, A. David, and K. G. Larson. A tutorial on UPPAAL. http://www.uppaal.com, 2004.

[11] Boris Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[12] J. P. Bodeveix, R. Bouaziz, and O. Kone. Test method for embedded real-time systems. In *ERCIM Workshop on Dependable Software Intensive Embedded systems*, 2005.

[13] Richard H. Carver and Kuo-Chung Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8:66–74, March 1991.

[14] Richard H. Carver and Kuo-Chung Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24(6):471–490, 1998.

[15] David Chartier. Phone, app store problems causing more than just headaches. http://arstechnica.com/apple/news/2008/- 07/iphone-app-store-problems-causing-more-than-just-headaches.ars, 2008.

[16] Jun Chen and Steve MacDonald. Testing concurrent programs using value schedules. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 313–322, 2007.

[17] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, 2002.

[18] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *Proceedings of the 22nd international conference on Software engineering*, pages 439–448, 2000.

[19] S. J. Cunning and J. W. Rozenblit. Automatic test case generation from requirements specifications for real-time embedded systems. In *IEEE Conference and Workshop on Engineering of Computer-Based Systems*, 1999.

[20] J.L. Dalley. The art of software testing. In *Aerospace and Electronics Conference, 1991. NAECON 1991., Proceedings of the IEEE 1991 National*, pages 757 –760 vol.2, May 1991.

[21] Deviceguru.com. Over 4 Billion Embedded Devices Ship Annually. http://deviceguru.com/over-4-billion-embedded-devices-shipped-last-year/, January 2008.

[22] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, 1991.

[23] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10:405–435, October 2005.

[24] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 62–75, 1994.

[25] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real-time systems. *IEEE Transactions on Software Engineering*, 28(11):1203–1238, 2002.

[26] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, 2003.

[27] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 1–1, 2000.

[28] S. Fischmeister and P. Lam. On time-aware instrumentation of programs. In *Real-Time and Embedded Technology and Applications Symposium. 15th IEEE*, pages 305 –314, 2009.

[29] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, 2004.

[30] P. Frankl and E. Weyuker. An applicable family of data flow criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[31] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *Software Engineering, IEEE Transactions on*, 17(6):591 –603, June 1991.

[32] Gizmo Highway. Drive by Wire Cars. Web page, 2003. http://www.gizmohighway.com/autos/drive_by_wire.htm.

[33] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *SIGPLAN Not.*, 10:493–510, April 1975.

[34] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *Proceedings of the 30th international conference on Software engineering*, pages 231–240, 2008.

[35] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 14–25, December 1994.

[36] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC- 3/97-TR17, Ohio State University, March 1997.

[37] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2:366–381, 2000.

[38] Makoto Higashi, Tetsuo Yamamoto, Yasuhiro Hayase, Takashi Ishio, and Katsuro Inoue. An effective method to control interrupt handler for data race detection. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 79–86, 2010.

[39] David Hovemeyer and William Pugh. Finding concurrency bugs in java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[40] Huckle, T. Collection of software bugs. http://www5.in.tum.de/%7Ehuckle/bugse.html, 2007.

[41] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, May 1994.

[42] Zhenyi Jin and A. Jefferson Offutt. Coupling-based criteria for integration testing. *The Journal of Software Testing, Verification, and Reliability*, 8:133–154, 1998.

[43] M. Jones. What really happened on Mars? http://research.- microsoft.com/ mbj/Mars_Pathfinder/Mars_Pathfinder.html, December 1997.

[44] Nigel Jones. A taxonomy of bug types in embedded systems, October 2009. http://embeddedgurus.com/stack-overflow/- 2009/10/a-taxonomy-of-bug-types-in-embedded-systems.

[45] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 675–681, 2009.

[46] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *SIGPLAN Not.*, 44:110–120, June 2009.

[47] João Cadamuro Junior and Douglas Renaux. Efficient monitoring of embedded real-time systems. In *Proceedings of the Fifth International Conference on Information Technology: New Generations*, pages 651–656, 2008.

[48] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, pages 72–, 1997.

[49] Jean J. Labrosse. *MicroC OS II: The Real Time Kernel*. CMP Books, 2002.

[50] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 94–104, 2008.

[51] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings*

of the 32nd ACM/IEEE International Conference on Software Engineering, pages 235–244, 2010.

[52] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[53] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not.*, 27:235–248, July 1992.

[54] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306, 2005.

[55] Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32:382–403, June 2006.

[56] N. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7), July 1993.

[57] LinuxDevices.com. Four billion embedded systems shipped in 06. http://www.linuxdevices.com/news/- NS3686249103.html, January 2008.

[58] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[59] Douglas Long and Lori A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 21–35, 1991.

[60] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. *SIGPLAN Not.*, 41(11):37–48, 2006.

[61] J. L. Lyons. ARIANE 5, flight 501 failure. http://www.ima.-umn.edu/ arnold/disasters/- ariane5rep.html, July 1996.

[62] S.S. Muchnick and N.D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[63] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 446–455, 2007.

[64] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, 2006.

[65] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data flow analysis for checking properties of concurrent java programs. In *Proceedings of the 21st international conference on Software engineering*, pages 399–410, 1999.

[66] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31:676–686, June 1988.

[67] Hemant D. Pande and William Landi. Interprocedural def-use associations in c programs. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 139–153, 1991.

[68] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145, 2008.

[69] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11:367–375, April 1985.

[70] Stuart C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. *Software Metrics, IEEE International Symposium on*, 0:64, 1997.

[71] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Softw. Eng. Notes*, 22:432–449, November 1997.

[72] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[73] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 11–21, 2008.

[74] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2001.

[75] S. Sinha, M. J. Harrold, and G. Rothermel. Computation of interprocedural control dependencies. *ACM Transactions on Software Engineering and Methodology*, 10(2):209–254, April 2001.

[76] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[77] Scott D. Stoller. Testing concurrent java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4):142 – 157, 2002.

[78] Ahyoung Sung, Byoungju Choi, and Seokkyoo Shin. An interface test model for hardware-dependent software and embedded os api of the embedded system. *Comput. Stand. Interfaces*, 29:430–443, May 2007.

[79] Ahyoung Sung, Witty Srisa-an, Gregg Rothermel, and Tingting Yu. Testing inter-layer and inter-task interactions in RTES application. In *Software Engineering Conference, 2010. Proceedings. 17th Asia-Pacific*.

[80] Marouane Tlili, Stefan Wappler, and Harmen Sthamer. Improving evolutionary real-time testing. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1917–1924, 2006.

[81] Wei-Tek Tsai, Lian Yu, Feng Zhu, and Ray Paul. Rapid embedded system testing using verification patterns. *IEEE Software*, 22:68–75, 2005.

[82] Manthos A. Tsoukarellas, Vasilis C. Gerogiannis, and Kostis D. Economides. Systematically testing a real-time operating system. *IEEE Micro*, 15:50–60, October 1995.

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

[84] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.

[85] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, 2007.

[86] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, 2006.

[87] Joachim Wegener, Hartmut Pohlheim, and Harmen Sthamer. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. *Real-Time Systems Symposium, IEEE International*, 0:270, 1999.

[88] Wikipedia. Formal verification. Web page. http://en.wikipedia.org/wiki/Formal_verification.

[89] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 153–162, 1998.

[90] Liguo Yu, Stephen R. Schach, Kai Chen, and Jeff Offutt. Categorization of common coupling and its application to the maintainability of the linux kernel. *IEEE Transactions on Software Engineering*, 30:694–706, October 2004.

[91] Tingting Yu, Ahyoung Sung, Witty Srisa-an, and Gregg Rothermel. Using property-based oracles when testing embedded system applications. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation*, 2011.

[92] Qiushuang Zhang and Ian G. Harris. A data flow fault coverage metric for validation of behavioral hdl descriptions. *Computer-Aided Design, International Conference on*, 0:369, 2000.

[93] Haitao Zhu, M.B. Dwyer, and S. Goddard. Predictable runtime monitoring. In *Real-Time Systems, 2009. 21st Euromicro Conference on*, pages 173 –183, 2009.