Using Property-Based Oracles when Testing Embedded System Applications

Tingting Yu
Computer Science and Engineering
U. of Nebraska - Lincoln
Lincoln, NE, USA
tyu@cse.unl.edu

Ahyoung Sung
Division of Visual Display
Samsung Electronics Co., Ltd.
Suwon, Gyounggi, South Korea
ahyoung.sung@samsung.com

Witawas Srisa-an, Gregg Rothermel
Computer Science and Engineering
U. of Nebraska - Lincoln
Lincoln, NE, USA
{witty, grother}@cse.unl.edu

***Abstract*—Embedded systems are becoming increasingly ubiquitous, controlling a wide variety of popular and safety-critical devices. Effective testing techniques could improve the dependability of these systems. In prior work we presented an approach for testing embedded systems, focusing on embedded system applications and the tasks that comprise them. In this work we focus on a second but equally important aspect of testing embedded systems; namely, the need to provide observability of system behavior sufficient to allow engineers to detect failures. We present several property-based oracles that can be instantiated in embedded systems through program analysis and instrumentation, and can detect failures for which simple output-based oracles are inadequate. An empirical study of our approach shows that it can be effective.**

I. Introduction

Embedded systems are used to control a wide variety of applications, ranging from non-safety-critical systems such as cellular phones and televisions to safety-critical systems such as automobiles, airplanes, and medical devices. Clearly, systems such as these must be sufficiently dependable. There is ample evidence, however, that faults in embedded systems have led to numerous failures [16].

To address dependability problems in embedded systems, researchers have proposed various approaches ranging from formal verification to testing techniques (e.g., [6], [14], [21], [32]). On our analysis of prior work, however (Section II-C), one need that has not been sufficiently addressed is to provide methodologies by which the developers of *embedded system applications* — which run in environments supported by a wide range of underlying software components such as middle-ware, operating system kernels, device drivers, and hardware-related utilities — can *cost-effectively test* those applications *in the context of those environments*.

In previous work [31], we presented an approach to help developers of embedded system applications detect faults that occur as their applications interact with underlying system components. Our approach involves two dataflow-based test adequacy criteria. First, we use dataflow analysis to identify *inter-layer interactions* between application code and lower-level (kernel and hardware-related) components in embedded systems. Second, we use a further dataflow analysis to identify *inter-task interactions* between tasks that are initiated by the application. Application developers then create and execute test cases targeting these interactions.

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. 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. Thus, 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.

In this work we address this observability problem. We present several "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. While creation of the oracles is manual, engineers can create them for specific embedded systems platforms and then reuse them on subsequent systems and releases constructed on these platforms, and with less expense than required by the current typical approach of manually constructing output-based oracles for each test input.

The property-based oracles we study in this work 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.

II. Background and Related Work

A. Embedded Systems

Embedded systems are typically designed to perform specific tasks in particular computational environments consisting of software and hardware components. A typical embedded system is structured in terms of four layers: (1) the application layer, (2) the Operating System (OS) layer, (3) the Hardware Adaptation Layer (HAL), and (4) the underlying hardware substrate. Interfaces between these layers provide access to, and observation points into, the internal workings of each underlying layer.

Embedded system applications operate through interactions between the application layer and lower layers, and interactions between the various *user tasks* that are initiated in the application layer. These include (1) interactions between layers via interfaces, (2) interactions via information created in one layer which then flows to other layers for processing, and (3) interactions between multiple tasks working together concurrently and coordinating their efforts via shared resources. This motivates the use of testing approaches by which application developers can test these interactions.

B. Testing Inter-Layer and Inter-Task Interactions

The foregoing classes of interactions can be modeled in terms of data that flows across software layers, data that flows between components within layers, and data that flows between user tasks. Tracking data that flows across software layers is important because applications use this to obtain services and acquire resources in ways that may 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. Finally, data that flows through resources shared by user tasks can reveal synchronization and other forms of faults.

In prior work [31], we thus investigated the use of dataflow testing techniques which use static analysis to track data usage in programs in terms of the potential flow of data between definitions and uses of variables. We presented two dataflow-based test adequacy criteria. The first criterion tracks data dependencies representing interactions across layers of the system at the intra-task level (i.e., within single user tasks). These *inter-layer* dependencies are calculated through an interprocedural dataflow analysis that identifies *definition-use pairs* that correspond to interactions between system layers and components of system services; i.e., those involving global variables, APIs for kernel and HAL services, function parameters, physical memory addresses, accesses to physical devices, and special registers.

Our second test adequacy criterion identifies *inter-task* interactions in the form of definition-use pairs involving shared variables that can be accessed by multiple user tasks as they operate concurrently. These shared variables are typically represented as global variables; they often represent physical devices accessible to multiple user tasks through memory mapped I/O. Typically, programmers use critical sections to control access to these variables, e.g., by calling kernel APIs to acquire/release semaphores or calling the HAL API to disable/enable interrupt requests.

The two forms of definition-use pairs identified by our adequacy criteria serve as testing requirements (coverage targets) for use by engineers in testing embedded system applications. We empirically studied the use of these criteria in testing a commercial embedded system application. Our results showed that our techniques can be effective [31].

C. Related Work

1) Testing Embedded Systems: Much of the research on testing embedded systems has focused on temporal faults, which arise when systems have hard real-time requirements. Temporal faults are important, but many faults in embedded systems are non-temporal, including algorithmic, logic, and configuration errors, and misuse of resources [4], [18].

There has been some research on techniques for testing for non-temporal faults or faults related to functional behavior in embedded systems. Much of this work has used specifications (e.g., [5], [23]) and applies only when those specifications exist. Among non-specification-based approaches, some test applications without targeting underlying software components (e.g., [30], [32]) while others test software components (e.g., kernels and device drivers) that underlie the applications (e.g., [2], [33]). Certainly applications and underlying components must each be tested, but on our view, interactions among these must also be targeted.

There has been some work using dataflow analysis to analyze properties of embedded systems [36], [37], however this work does not focus on testing. Lai et al. [21] apply inter-context controlflow and dataflow test adequacy criteria to 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. While this work focuses on inter-task interactions, it does not focus on interlayer interactions.

2) Testing Concurrency in Embedded Systems: A common approach used to test concurrent programs is to apply static analysis techniques to discover paths and regions in code that might be susceptible to concurrency faults (e.g., [11], [15], [34]). For example, static analysis techniques based on state modeling and transitions (e.g., [10]) can verify concurrent programs with respect to properties, and have been used to verify correctness of low-level systems including device drivers and kernels [3], [11]. 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.

Dynamic analysis tends to be more accurate than static analysis in detecting concurrency faults (e.g., [12], [19], [22], [28]). Dynamic techniques can also be used to permute thread interleavings at runtime to expose more possible faults. For example, active testing (e.g., [19], [28]) is used to first identify potential concurrency faults, and then control the underlying scheduler by inserting delays at context switch points. The drawback of dynamic analysis, however, is that it can detect faults only on executed paths.

Our goal is to utilize test cases to detect faults, and in particular, faults involving run-time interactions between applications and underlying system components and between user tasks. Thus, we employ dynamic analysis. To overcome the short-comings of dynamic analysis, however, we rely on test adequacy criteria that target definition-use pairs involving inter-layer and inter-task interactions. This allows our test suites to cover richer sets of execution paths.

There are approaches for uncovering concurrency faults by manipulating synchronization sequences (e.g., [6]). However, these approaches rely on program specifications and ours does not.

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. [35] predefine atomicity access patterns associated with locks, and any execution that violates the patterns is reported as a fault. Lu et al. [24] 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.

One major characteristic of our approach is the use of synchronization properties as test oracles. Our synchronization properties are similar to those used in program model checking (e.g., [8], [13], [25]), a formal verification technique that exhaustively tests a model extracted from system code. 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 disciplines (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 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.

There is some prior research on using concurrency properties to test embedded systems. Higashi et al. [14] 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. 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.

III. 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, OS, 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 requires test engineers to create properties relative to particular embedded systems platforms, and thus, manual effort and expertise are required and some properties will be platform dependent. Once created, however, these properties are relevant to all applications built on those platforms so the cost of defining properties for them is amortized. This differs from output-based oracles, which are also typically defined manually, but must be defined for each test case and program individually.

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 [29]. 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 is worth noting that there are existing runtime monitoring tools that can perform instrumentation and violation detection automatically based on specifications [9]. The key

notion in developing such automated tools is having a clear understanding of the correct runtime properties and data structures. In this work, we tackle the observability issue by looking at multiple layers of software. As such, the first step of our work is also to gain the insights about the intricate dependencies between applications and the kernel and among various kernel components. We achieve this through instrumentation. As our work continues to mature, we envision that we will be able to provide an automated tool to instrument various layers of software, monitor relevant events, and detect faults.

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.

As a platform for implementing and studying our approach we chose rapid prototyping systems provided by Altera [1]. 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, and automotive applications. The code base contains about 5000 lines of non-comment C code [20]. For academic research, the source-code of MicroC/OS-II is freely available.

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.

A. 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.

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)∧(T_j,0,R)>

Figure 1. Correct states for binary semaphores. 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.

We encode state information using a triple, S, that contains (i) the identification of a task, T, 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. Each of these operation triples represents a state transition due to an operation performed on the semaphore.

Figure 1 shows the result of applying this process to the states and transitions relevant to MicroC/OS-II.

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)>.

B. 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 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*. 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. If the broadcast option is selected, then all the tasks in the waitlist are blocked; otherwise, only the highest priority task is blocked.

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 2. Correct states for message queue

To observe task states and queue status, we instrument the MicroC/OS-II kernel's functions related to message queue operations and task scheduling (*OSQCreate*, *OSQPend*, *OSQPostOpt*, *OS_EventTaskWait*, and *OS_EventTaskRdy*) to generate the required information.

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.

C. 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 [29]. 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) [27] 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.

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 3.

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 3. Correct properties for critical section

D. 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 based on virtual machines. Presently, we are able to capture control-flow information without instrumenting the source code [7]. Our next step is to extend this framework to capture dataflow information without source code instrumentation. We foresee that the instrumentation framework will enhance the capability and usability of our property-based oracles.

IV. 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.

A. Objects of Analysis

As objects of study we chose two embedded systems applications that are provided with the Altera platform. The first application, MAILBOX, involves the use of mailboxes by several user tasks. The second application, CIRCULAR-BUFFER, involves the use of semaphores by two user tasks, and a circular buffer that operates as a message queue by three tasks. Table I 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.

To evaluate our approaches, we required faulty versions of our object programs. To obtain these we asked a programmer with over ten years of experience with embedded systems, but with no knowledge of our testing approach, to follow a fault injection guide and taxonomy and insert *potential faults* into the applications and into the MicroC/OS-II system (kernel and HAL) code. Potential faults seeded in the code were related to kernel initialization, task creation and deletion, scheduling, resource creation, resource handling interrupt handling, and critical section management.

Given a set of potential faults, we followed [17] in taking two additional steps to ensure that our fault detection results

Table I
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	16	2380	32	39	253	65	26	24
CIRCULARBUFFER	304	17	17	2814	38	49	394	96	40	44

are valid. First we determined, through code inspection and program execution, which potential faults could never be detected by test cases. These included potential faults that were actually semantically equivalent to the original code, or that had been placed in lower layer code that could never be executed in the context of our particular applications, and we eliminated these. Second, after creating test cases for our application (described below), we eliminated any faults that were detected by more than 50% of those test cases. Such faults are likely to be detected during unit testing, and are thus not appropriate for assessing the strength of system-level testing approaches. This process left us with the numbers of faults reported in Table I (the table lists these separately for the application and Kernel/HAL levels).

B. Variables, Measures, and Other 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 III.

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 measured the numbers of faults in our object programs detected by the two oracle approaches.

Additional Factors. To help reduce potential threats to internal validity, we also consider one additional factor. Like many embedded system applications, our object programs run in infinite loops. Although semantically a loop iteration count of two is sufficient to cause the programs to execute one cycle, different iteration counts may have different powers to reveal faulty behavior in the interactions between the user tasks and the kernel. To capture these differences we utilized two iteration counts for all executions, 10 and 100. We selected these iteration counts following preliminary studies of iterations ranging from 5 to 5000, because increases up to 100 were observed to cause fault-detection differences, while further increases above that did not.

C. Study Operation

As test suites we utilize code coverage-based test suites engineered specifically to provide coverage of inter-layer and inter-task definition-use pairs (see Section II). To create these we followed a current typical practice: creating initial black-box test suites and then augmenting these for coverage.

To create initial black-box test suites we used the category-partition method [26], 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 I.

The application of our dataflow analysis techniques to our object programs identified definition-use pairs (see Table I). 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. This process resulted in the creation of additional test cases as reported in Table I.

We executed our test suites on all of the faulty versions of each object program, with one fault activated on each execution. We then applied each output- and property-based oracle to each test execution; this was done independently per oracle and per test case to avoid conflating results.

D. 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 following a fault injection guide and taxonomy, and the test suites are created using practical processes.

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. Our choices of iteration limits may also affect fault detection results but we used reasonable care in exploratory steps to choose meaningful limits.

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

E. Results

Figure 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.

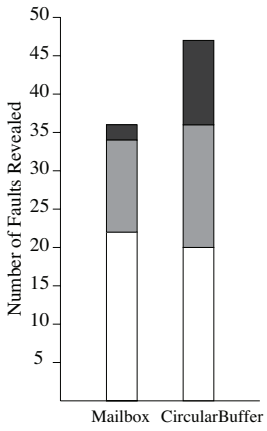

Figure 4. Overview of fault detection data

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 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.

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).

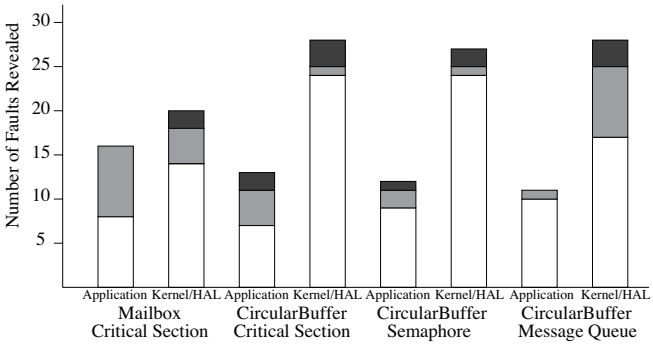

Figure 5. Detailed fault detection data

V. Results Summary and Discussion

While additional studies are needed, if the results we have just reported generalize, then: (1) the property-based oracles we have defined are effective at detecting faults, and can detect faults not detected by output-based oracle; and (2) each property-based oracle demonstrated the ability to detect faults not detected by others.

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, although output-based oracles may detect more faults than property-based oracles, in cases where output-based oracles are not operable property-based oracles can still be useful. Moreover, the creation of output-based oracles is itself time-consuming and manual and must be performed per test case, and these oracles must then be maintained over the system lifetime. It follows that it may be preferable to apply property-based oracles before output-based oracles, because if they do identify failing test cases this obviates the need to spend time using and inspecting output-based oracles for these 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 IV 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 IV, 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 II.) 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 III) 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 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 involves incorrect usage of a binary operator in a kernel's interrupt service routine; this results in multiple tasks concurrently accessing a critical section. Another fault involves improper usage of a unary operator. In this scenario, the system is configured to unblock all tasks when a message arrives; however, only the highest priority task is unblocked in practice. While neither fault causes incorrect system output on the employed test cases, both could emerge during deployment.

VI. Conclusions and Future Work

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.

In addition to the need to perform additional empirical work, there are several other avenues for future work. We have already discussed prospects for using simulation platforms to create non-intrusive instrumentation environments.

We also see 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) and real-time properties. Finally, the insights obtained from this work can lead to the development of tools that can automate the instrumentation process, monitor runtime events, and detect faults.

Acknowledgements

This work was supported in part by the AFOSR through award FA9550-09-1-0129 to the University of Nebraska - Lincoln, and through the Korea Research Foundation under award KRF-2007-357-D00215. Wayne Motycka helped with the setup for the empirical study. Moonzoo Kim provided helpful advice on content and presentation.

References

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

[2] J. Arlat, J. Fabre, M. Rodriguez, and F. Salles. Dependability of COTS microkernel based systems. *TC*, 51(2), Feb. 2002.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EECCS*, pages 73–85, 2006.

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

[5] J. P. Bodeveix, R. Bouaziz, and O. Kone. Test method for embedded real-time systems. In *WSIDES*, 2005.

[6] R. H. Carver and K.-C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *TSE*, 24(6):471–490, 1998.

[7] X. Chen. *SimSight: A Virtual Machine Based Dynamic Call Graph Generator*. PhD thesis, University of Nebraska-Lincoln, Lincoln, NE, USA, 2010.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.

[9] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

[10] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *FSE*, pages 62–75, 1994.

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

[12] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, pages 231–240, 2008.

[13] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2:366–381, 2000.

[14] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue. An effective method to control interrupt handler for data race detection. In *AST*, pages 79–86, 2010.

[15] D. Hovemeyer and W. Pugh. Finding concurrency bugs in Java. In *WCSJP*, 2004.

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

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, May 1994.

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

[19] P. Joshi, M. Naik, C.-S. Park, and K. Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *CAV*, pages 675–681, 2009.

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

[21] Z. Lai, S. Cheung, and C. W.K. Inter-context control-flow and data-flow test adequacy criteria for nesC applications. In *FSE*, Nov. 2008.

[22] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, pages 235–244, 2010.

[23] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using UPPAAL-TRON. In *EMSOFT*, Sept. 2005.

[24] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. *SIGPLAN Not.*, 41(11):37–48, 2006.

[25] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[26] T. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *CACM*, 31(6), June 1988.

[27] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations in C programs. *TSE*, 20(5):385–403, May 1994.

[28] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, pages 135–145, 2008.

[29] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 2008.

[30] A. Sung, B. Choi, and S. Shin. An interface test model for hardware-dependent software and embedded OS API of the embedded system. *CSI*, 29(4), Apr. 2007.

[31] A. Sung, W. Srisa-an, G. Rothermel, and T. Yu. Testing inter-layer and inter-task interactions in RTES application. In *APSEC*, Dec. 2010.

[32] W. Tsai, L. Yu, F. Zhu, and R. Paul. Rapid embedded system testing using verification patterns. *IEEESW*, 22(4), 2005.

[33] M. A. Tsoukarellas, V. C. Gerogiannis, and K. D. Economides. Systematically testing a real-time operating system. *IEEE Micro*, 15(5):50–60, Oct. 1995.

[34] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *ESEC-FSE*, pages 205–214, 2007.

[35] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *TSE*, 32(2):93–110, 2006.

[36] L. Yu, S. R. Schach, K. Cehn, and J. Offutt. Categorization of common coupling and its application to the maintainability of the Linux kernel. *TSE*, 30(10):694–705, Oct. 2004.

[37] L. Zhang and I. G. Harris. A data flow fault coverge metric for validation of behavioral HDL descriptions. In *ICCAD*, Nov. 2000.