

# Testing Inter-Layer and Inter-Task Interactions in RTES Applications

Ahyoung Sung\*, Witawas Srisa-an, Gregg Rothermel, and Tingting Yu  
Department of Computer Science and Engineering  
University of Nebraska - Lincoln  
Lincoln, NE, USA  
{aysung, witty, grother, tyu}@cse.unl.edu

**Abstract**—Real-time embedded systems (RTESs) 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 this paper we present an approach for testing RTESs, intended specifically to help RTES application developers detect faults related to functional correctness. Our approach consists of two techniques that focus on exercising the interactions between system layers and between the multiple user tasks that enact application behaviors. We present results of an empirical study that shows that our techniques are effective at detecting faults.

## I. INTRODUCTION

Real-time embedded systems (RTESs) 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. RTESs are also being produced at an unprecedented rate, with over four billion units shipped in 2006 [11].

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.<sup>1</sup> Other faults in RTESs have led to the Ariane [28], Therac-25 [27], Mars Pathfinder [19] and other failures [16].

There are thus good reasons for researchers to address dependability issues for RTESs. Section V provides a full discussion of related research to date, but to summarize, much has focused on temporal faults, which arise due to the hard real-time requirements for these systems (e.g., [1], [21], [37], [41]). Temporal faults are important, but in practice, a large percentage of faults in RTESs involve problems with functional behavior including algorithmic, logic, and configuration errors, and misuse of resources in critical sections [5], [20]. Moreover, according to Toyota engineers [42], many RTESs are constructed with interacting software modules, designed and configured by teams of engineers over years. Once integrated, these modules form legacy structures that can be challenging to understand. As

these structures evolve, the effects can be difficult to predict, making further validation more difficult.

There has been some research on techniques for testing for faults related to functional correctness of RTESs. Some efforts focus on testing applications without specifically targeting underlying software components (e.g., [35], [38]) while others focus on testing specific software components (e.g., real-time operating kernels and device drivers) that underlie the applications (e.g., [3], [4], [39]). Certainly applications and underlying components must each be tested, but as Toyota engineers suggest [42], and as we show in Section II, new methodologies designed to capture component interactions, dependency structures, and designers’ intended behaviors are needed to effectively test RTESs. Approaches presented to date do not adequately do this.

We present an approach for testing RTESs that focuses on *helping developers of RTES applications detect faults related to functional correctness 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 RTESs and between individual software components within layers. Technique 2 uses information gathered by Technique 1 to identify *intertask interactions* between the tasks that are initiated by the application layer. Application developers target these interactions when creating test cases. We present results of an empirical study of our techniques applied to a commercial RTES system. Our results show that both techniques are effective at detecting faults.

This work makes the following contributions: (1) we provide techniques for testing RTESs from the context of the application, focusing on faults related to functional correctness and targeting interactions between system layers and user tasks; this is the first work to address RTES dependability in this context; (2) while our testing techniques build on existing dataflow analysis and testing approaches and could be applied to other classes of software, we use them here to address specific problems in the RTES context; and (3) we report results of an empirical study using a commercial RTES kernel and application, that shows that our techniques can be effective. Compared to studies performed in prior work, ours represents a significant improvement in the state of empirical work in this area.

\*New affiliation: Division of Visual Display, Samsung Electronics

<sup>1</sup>CNN news report, February 4, 2010.

## II. REAL-TIME EMBEDDED SYSTEMS

RTOSs are typically designed to perform specific tasks in particular computational environments consisting of software and hardware components. A typical RTES is structured in terms of four layers: (1) the application layer, (2) the Real-Time Operating System (RTOS) 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.

An application layer consists of code for a specific application; applications are written by developers and utilize services from underlying layers.

An RTOS consists of task management, context-switching, interrupt handling, inter-task communication, and memory management facilities [22] that allow developers to create RTES 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* [22]. Thus, many kernels schedule tasks based on priority assignment.

The HAL is a runtime system that manages device drivers and provides interfaces by which higher level software layers 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.

The issues involving classes of faults in real-time systems, the importance of testing functional system attributes and the state of the art in research on testing RTESs sketched in Section I and further detailed in Section V help motivate our focus in this research. Further motivation for both our focus and for the specific approaches we investigate come from two additional central characteristics of RTESs.

*Characteristic 1: Manner of operation.* To illustrate the way in which RTESs 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: (1) interactions between layers (via interfaces) play an essential role in RTES application execution; (2) as part of RTES application execution, information is often created in one

layer and then flows to others for processing; and (3) 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 RTES. This observation is central to our choice of testing methodologies in this work.

*Characteristic 2: Development practices.* A second motivating factor behind this work involves the manner in which RTESs are developed. First, software development practices in the area of portable consumer electronics such as phones and PDAs, which account for about 40% of RTESs [11], often separate developers of application 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, iPod, and iPad applications. Apple provided the *iPhone SDK* — the same tool used by Apple's engineers to develop core applications [9] — to support application development by providing basic interfaces that an application can use to obtain services from underlying software layers.

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 due to failures in the underlying layers that affected operations of the devices [9]. These failures occurred because the developers used the interfaces provided by the iPhone SDK in ways not anticipated by Apple's engineers.

The foregoing discussion provides further support for the need to test RTESs from the perspective of the application, and the need to test interactions between the application and lower layers. Additional support is derived from a somewhat contrasting alternative development process. Specifically, a recent study has shown that the 60% of RTESs (e.g., automotive, avionics, and medical) that are outside the domain of consumer electronics are developed as customized applications [11]. These applications use lower-level software components (runtime services and libraries) that must be developed or heavily customized by in-house software developers. In situations such as this the developers do know what scenarios underlying services will be invoked in, but they still cannot safely treat lower level components as black boxes in testing. Instead, developers and engineers need to specifically observe the execution of each component and test the scenarios in which underlying components are invoked by the application and in which tasks interact.

## III. APPROACH AND TECHNIQUES

The interactions in RTESs that we have just discussed 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 RTEs application programmers detect faults related to functional correctness in their systems. We thus consider dataflow-based testing approaches in the RTEs 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 an RTEs. 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 to determine whether this sharing creates failures.

Our approach involves two techniques designed to take advantage of these aspects of data usage information. The first technique statically analyzes an RTEs at the *intratask* level to calculate data dependencies representing interactions between layers of the system. The second technique calculates data dependencies representing *intertask* interactions. The data dependencies identified by these two techniques become coverage targets for testers.

We next present the basics of dataflow testing, and then we describe our two techniques in turn. (For space limitations we do not present algorithms but these are available in [36].) We follow this with a discussion of approaches for dynamically capturing execution information, which is required for the use of our techniques.

#### A. Dataflow Analysis and Testing

Dataflow analysis (e.g., [29]) 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., [32]) 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*; we utilize the latter, building on algorithms given in [14], [31].

From our perspective on the RTEs testing problem there are several things that render dataflow approaches interest-

ing. 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 RTEs 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 is less expensive than targeting all definition-use pairs in a system.

In the context of testing we also avoid other costs associated with conservative dataflow analysis techniques. Conservative analyses must overestimate dependencies, and this in turn requires relatively expensive analyses of aliases and function pointers (e.g., [24], [34]). Testing, however, 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.

#### B. Technique 1: Intratask Testing

As we have discussed, an application program is a starting point through which underlying layers in RTEs 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 [31], [33] 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 RTEs 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`.

*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., [33]) initialize worklists with definitions and then propagate these around ICFGs. Step 2 of our technique does this by initializing our worklist to every definition of interest in the ICFG for  $T_k$ . These 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 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.

### C. Technique 2: 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 RTEs use various approaches to control access to critical sections, including calling kernel APIs to acquire/release semaphores, calling the HAL API to disable/enable IRQs (interrupt requests), or writing 1 or 0 to a control register using a busy-wait 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.

There are several issues related to the foregoing approach that should be noted. First, we have chosen to focus only on shared variables that appear in critical sections, as these relate directly to problems in the implementation of synchronization mechanisms such as semaphores and monitors in the kernel. It is trivial to expand the approach to consider all the shared variables appearing in a program. Second, we do not attempt to locate cases where entries to or exits from critical sections are erroneously omitted, but it would be possible to do so either statically, or dynamically through analysis of test traces. Third, our intertask analysis is both flow- and context-insensitive; more precise analyses such as inter-context control-flow and dataflow analysis [23] could be used to analyze iterations among user tasks and obtain more precise sets of intertask definition-use pairs. Finally, on some systems, covering intertask pairs may require testers to do more than just vary inputs – they may need to manipulate task sequences or other aspects of system behavior [10].

#### D. 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 sys-

tem states. Currently, the focus of our methodology is to make functional testing more effective, and 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., [8], [26]) can further assist with this process.

That said, 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 [40] 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 techniques.

#### IV. 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.

As a baseline for comparison we utilize two approaches. Our first approach is to consider our coverage criteria within the context of a current typical practice for testing RTEs applications. Engineers currently often use black-box test suites designed based on system parameters and knowledge of functionality to test RTEs 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 RTEs context, our techniques add additional fault detection 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 randomly generated test suites containing the same number of test cases. This lets us assess whether the use of our

testing criteria provides 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 techniques achieve better fault detection than test suites not so augmented, and if so to what extent?

**RQ2:** Do test suites that are coverage-adequate in accordance with our techniques achieve better fault detection than randomly generated test suites containing the same number of test cases, and if so to what extent?

### A. Object 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 [12] 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 RTESSs. A second issue in performing studies involves finding a platform on which to experiment. A platform must provide support for gathering analysis data, implementing techniques, and automating test execution, and 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 [2], a company with over 12,000 customers worldwide and annual revenue of 1.2 billion USD.<sup>2</sup> 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 [22]. For academic research, the source-code of MicroC/OS-II is freely available, so other studies may utilize it.<sup>3</sup>

As an object of study we chose an RTESS application, MAILBOX, that is provided with Altera, and 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. We chose this object because it is a meaningful application, of manageable size but also relatively complex in terms of user tasks and usage of lower layer functionality. The MAILBOX application code itself consists of nine functions and 268 non-comment lines of C code, and it invokes, directly or indirectly, 20 kernel, 10 HAL, and two hardware

layer functions consisting of 2,380 non-comment lines of C. While in absolute terms this codebase might seem small, it is typical of many single-purpose RTESSs. Moreover, the complexity of a program depends on its run-time behavior, inputs, and source code, and even a small program such as ours can have complex and long-running behavior.

To address our research questions we required faulty versions of our object program, and to obtain these we followed a protocol introduced in [17]. We asked a programmer with over ten years of experience including experience with RTESSs, but with no knowledge of our testing approach, to insert potential faults into the MAILBOX application 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), and interrupt handling. From the potential faults seeded in this fashion, we eliminated those that we could determine, by code inspection and execution of the program, could never be revealed by any test cases. This process left us with 45 faults: 14 in the application layer and 31 in lower system layers.

### B. Variables, Measures, and Other Factors

*Independent Variable.* Our independent variable is the testing approach used, and as discussed in relation to our research questions there are two: (1) 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; and (2) 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 techniques.

*Additional Factors.* Like many RTESS applications, MAILBOX runs in an infinite loop. Although semantically a loop iteration count of two is sufficient to cause the program to execute one cycle, different iteration counts may have different powers to reveal faulty behavior in the interactions between the MAILBOX 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. (Performing the setup for and executing test cases under iteration count 100 allowed us to conduct all experimental runs at that iteration count in approximately 950 hours, as the next section will clarify.) For each application of each testing approach, we report results in terms of each iteration count.

<sup>2</sup>[http://www.altera.com/corporate/about\\_us/abt-index.html](http://www.altera.com/corporate/about_us/abt-index.html).

<sup>3</sup>Researchers may also obtain all of the objects, artifacts, and data obtained in our study by contacting the first author.

### C. Study Setup

To create an initial black-box test suite we used the category-partition method [30], 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. To obtain TSL test suites, we asked the fourth author of this paper, who was not yet familiar with our testing approach and had no access to the seeded faults, to employ the category-partition method. This yielded an initial test suite of 26 test cases.

To implement our techniques we used the the ARISTOTLE [15] system 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 prototype to our object program identified 253 intratask definition-use pairs and 77 intertask pairs. We executed the TSL test suite on the program and determined its coverage of intratask and intertask pairs; 76% were covered by this initial suite. We then asked the fourth author to augment their initial TSL test suite twice; the first time by creating new test cases sufficient to ensure coverage of all the executable intratask definition-use pairs in the program, and the second time to ensure coverage of all the executable intertask definition-use pairs. This process resulted in the creation of 24 additional test cases (14 to cover intratask pairs and 10 to cover intertask pairs). Executing these test cases and further analyzing the definition-use pairs revealed that all of the intratask and 65 of the 77 intertask definition-use pairs were executable, and the augmented suites thus raised coverage of the feasible definition-use pairs to 100%.

To create random test suites we randomly generated inputs for MAILBOX, which was feasible since its input consists of a set of integers representing task 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 program, with one fault activated on each execution. In total, for each iteration count, we executed 200 test cases (four suites each of size 50) for the object program and performed 9,000 ( $200 \times 45$ ) executions for the correct and faulty versions of the object program, recording program output 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.

### D. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our object program, 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 investigate is a commercial RTEs, the faults are inserted by an experienced programmer, and the test suites are created using practical processes. Further, with over 11,000 test executions conducted, our data set is substantially larger than those used in prior studies of testing techniques applied to RTEs.

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.

### E. Results

We now present and analyze our results. Section IV-F provides further discussion.

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 algorithms. Table I 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.

As the data shows, 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 22% to 38%. The TSL suites detected at best 10 of the 14 application level faults, while the augmented suites detected all 14. The TSL suites detected at best 14 of the 31 lower layer faults, while the augmented suites detected 18, leaving 13 undetected.

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 II

Table I  
FAULT DETECTION, RQ1 (TSL vs TSL+DU)

Layer	Iter. Count = 10		Iter. Count = 100	
	TSL	TSL+DU	TSL	TSL+DU
Application	8	13	10	14
Kernel/HAL	14	18	14	18

Table II  
FAULT DETECTION, RQ2 (RAND vs TSL+DU)

Layer	Iter. Count = 10		Iter. Count = 100	
	RAND	TSL+DU	RAND	TSL+DU
Application	11	13	11	14
Kernel/HAL	12	18	12	18

compares the two, in the same manner as Table I. 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.

In this case, 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 15% to 33%.

#### F. Discussion.

**Fault detection effectiveness.** 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 the object system, and the TSL+DU suites detected 32.

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 I that at iteration counts 10 and 100, across all software layers, these DU test cases detected nine faults at level 10, and eight 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 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 large advantage.

**Relationship between execution time and faults.** 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.

**Relationship between intratask and intertask techniques.** 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. 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. Moreover, with 253 intratask definition-use pairs and just 65 intertask pairs, 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.

**Types of faults detected.** We also analyzed our data to see which types of faults were revealed. We determined that both TSL and DU test cases detected 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.

**Cost issues.** By restricting our attention to specific types of intratask and intertask definition-use pairs we are able to lower testing costs. The total number of definition-use pairs in our object program using a standard interprocedural analysis on all definitions and uses is 583, and our restricted focus lowers the number to 318. As noted earlier, none of the intratask pairs were infeasible and only 12 of the intertask pairs were infeasible – a relatively low number. Further, these 12 all involved pairs in which the use occurred in the system’s initialization task, and the definition occurred in other user tasks. While our flow-insensitive algorithm for identifying interclass pairs does not rule these particular types of definition-use pairs out, testers can easily do so, and the analysis could be adapted to do so.

## V. RELATED WORK

There has been a great deal of work on using dataflow-based testing approaches to test interactions among system components (e.g., [14], [18], [31]). With the exception of work discussed below, however, none of this work has addressed problems in testing RTEs or studied the use of such algorithms on these systems, and none has directed the analyses at specific classes of definition-use pairs appropriate for exercising intratask and intertask interactions in RTEs.

There has been some work involving testing RTEs. A first class of approaches involves formal specifications. Bodeveix et al. [7] present an approach for using timed automata to generate test cases for real-time embedded systems. En-Nouaary et al. [13] 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 [6] verifies embedded systems that can be modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronization [25]. 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.

Second, informal specifications have been used to test RTEs focusing on the application layer. Tsai et al. [38] present an approach for black-box testing of embedded applications using class diagrams and state machines. Sung et al. [35] test interfaces to the kernel via the kernel API and global variables that are visible in the application layer. These techniques operate on the application; in contrast, our approach is a white-box approach, analyzing code in both the application and kernel layers to find testing requirements.

Third, there has been work on testing kernels using conventional testing techniques. Fault-injection based testing methodologies have been used to test kernels [3], [4] to achieve greater fault-tolerance, and black-box boundary value and white-box basis-path testing techniques have been applied to real-time kernels [39]. All of this work, however, addresses testing of the kernel layer, and none of it specifically considers interactions between layers.

There has been one attempt to apply dataflow analysis to the Linux kernel. Yu et al. [43] 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. This work is not, however, about testing.

There has been work on testing the correctness of hardware behavior using dataflow test adequacy criteria [44]. 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 [1], [37], [41] 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.* [21] creates a light-weight monitoring framework to debug and observe real-time embedded systems.

There has been a great deal of research on testing concurrent programs, of which [8], [26] represent a small but representative subset. Techniques such as these could be complementary to the techniques we have presented.

Work by Lai et al. [23] proposes inter-context controlflow and dataflow 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 approach is similar to the approach used in our second technique, but they consider a much wider 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 second technique, but at additional cost. As noted in Section IV-F, we may be able to improve our technique by adopting aspects of theirs.

Finally, we remark on the state of the art in empirical studies of testing RTEs. Whereas the empirical study of testing techniques in general has made great strides in recent years, studies of RTEs have not. Considering just those papers on testing RTEs cited in this section, many [7], [13], [38] include no empirical evaluation at all. Of those that do include empirical studies, most studies simply demonstrate technique capabilities, without comparing techniques to any baseline techniques [3], [4], [25], [35], [39], [43]. Only two of the papers reports on fault detection capabilities of techniques [23], [35]. Our study goes 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.

## VI. CONCLUSIONS AND FUTURE WORK

We have presented a new approach for use in testing RTEs. Our approach focuses on interactions between layers and user tasks in these systems, to expose faults in functional behavior that developers face as they create new applications. While our testing techniques build on existing dataflow analysis and testing approaches and could be applied to other classes of software, we use them here to address specific problems in the RTE context. We have conducted an empirical study applying our techniques to a commercial RTE, and demonstrated that they can be effective at revealing faults.

There are several avenues for future work. We have already discussed prospects for using simulation platforms to create non-intrusive instrumentation environments. Also, while we have focused here on source code analysis, underlying layers of RTEs are often provided only in object code. Our analyses can be adapted, however, to include such code. In addition, dataflow analysis of C programs is complicated by the presence of aliases and function pointers, and our algorithms have not yet considered these. Incorporating alias and points-to analyses (e.g., [24], [34]) could help us identify additional and potentially important definition-use pairs.

Our next goal in continuing this work, however, involves investigating our techniques on a larger set of object programs. The cost of doing this is quite large – the understanding and preparation of the object used in this study, and the conduct of the study, required between 1200 and 1500 hours of researcher time. Still, it is only through such studies that we can gain sufficient insights into the costs and benefits of our techniques.

## ACKNOWLEDGEMENTS

We thank the anonymous reviewers for comments that improved the paper. This work was supported in part by the Korea Research Foundation under award KRF-2007-357-D00215. Wayne Motycka helped with the setup for the empirical study.

## REFERENCES

- [1] N. Al Moubayed and A. Windisch. Temporal white-box testing using evolutionary algorithms. In *ICST*, 2009.
- [2] Altera Corporation. Altera Nios-II Embedded Processors. <http://www.altera.com/products/devices/nios/nio-index.html>.
- [3] J. Arlat, J. Fabre, M. Rodriguez, and F. Salles. Dependability of COTS microkernel based systems. *TC*, 51(2), Feb. 2002.
- [4] R. Barbosa, N. Silva, J. Dures, and H. Madeira. Verification and validation of (real time) COTS products using fault injection techniques. In *CBSS*, Feb. 2007.
- [5] S. Beatty. Sensible software testing. <http://www.embedded.com/2000/0008/0008feat3.htm>, 2000.
- [6] G. Behrmann, A. David, and K. G. Larson. A tutorial on UPPAAL. <http://www.uppaal.com>, 2004.
- [7] J. P. Bodeveix, R. Bouaziz, and O. Kone. Test method for embedded real-time systems. In *WSIDES*, 2005.
- [8] R. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8(2):66–74, Mar. 1991.
- [9] D. 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.
- [10] J. Chen and S. MacDonald. Testing concurrent programs using value schedules. In *ASE*, 2007.
- [11] Deviceguru.com. Over 4 Billion Embedded Devices Ship Annually. <http://deviceguru.com/over-4-billion-embedded-devices-shipped-last-year/>, Jan. 2008.
- [12] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques. *ESEJ*, 10(4):405–435, 2005.
- [13] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real-time systems. *TSE*, 28(11):1203–1238, 2002.
- [14] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *FSE*, Dec. 1994.
- [15] M. J. Harrold and G. Rothermel. Aristotle: A system for development of program analysis tools. Technical Report OSU-CISRC- 3/97-TR17, Ohio State University, Mar 1997.
- [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] Z. Jin and A. Offutt. Coupling-based criteria for integration testing. *JSTVR*, 8(3):133–154, Sept. 1998.
- [19] M. Jones. What really happened on Mars? [http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html), Dec. 1997.
- [20] 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>.
- [21] J. C. Junior and D. Renaux. Efficient monitoring of embedded real-time systems. In *CIT*, 2008.
- [22] J. J. Labrosse. *MicroC OS II: The Real Time Kernel*. CMP Books, 2002.
- [23] 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.
- [24] B. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39(4):473–489, Mar. 2004.
- [25] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using UPPAAL-TRON. In *EMSOFT*, Sept. 2005.
- [26] Y. Lei and R. Carver. Reachability testing of concurrent programs. *TSE*, 32(6), June 2006.
- [27] N. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7), July 1993.
- [28] J. L. Lyons. ARIANE 5, flight 501 failure. <http://www.ima-umn.edu/arnold/disasters/-ariane5rep.html>, July 1996.
- [29] S. Muchnick and N. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [30] T. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *CACM*, 31(6), June 1988.
- [31] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations in C programs. *TSE*, 20(5):385–403, May 1994.
- [32] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *TSE*, 11(4):367–375, Apr. 1985.
- [33] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *TOSEM*, 10(2):209–254, Apr. 2001.
- [34] B. Steensgaard. Points-to analysis in almost linear time. In *SPPL*, Jan. 1996.
- [35] 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.
- [36] A. Sung, W. Srisa-an, G. Rothermel, and T. Yu. Testing inter-layer and inter-task interactions in RTES applications. Technical Report TR-UNL-CSE-2010-0006, University of Nebraska-Lincoln, June 2010.
- [37] M. Tlili, S. Wappler, and H. Sthamer. Improving evolutionary real-time testing. In *CGEC*, 2006.
- [38] W. Tsai, L. Yu, F. Zhu, and R. Paul. Rapid embedded system testing using verification patterns. *IEEE SW*, 22(4), 2005.
- [39] M. A. Tsoukarellas, V. C. Gerogiannis, and K. D. Economides. Systemically testing a real-time operating system. *IEEE Micro*, 15(5):50–60, Oct. 1995.
- [40] Virtutech. Virtutech Simics. Web-page, 2008. <http://www.virtutech.com>.
- [41] J. Wegener, H. Pohlheim, and H. Sthamer. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. In *RTSS*, 1999.
- [42] H. Yazarel, T. Kaga, and K. Butts. High-confidence power-train control software development. In *SCDAC*, Nov. 2006.
- [43] 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.
- [44] L. Zhang and I. G. Harris. A data flow fault coverage metric for validation of behavioral HDL descriptions. In *ICCAD*, Nov. 2000.