

# Testing Inter-Layer and Inter-Task Interactions in Real-Time Embedded Systems

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

## Abstract

Real-time embedded systems are becoming increasingly ubiquitous, controlling a wide variety of popular and safety-critical devices. Testing is the most commonly used method for validating software systems, and effective testing methodologies could be helpful for improving the dependability of these systems. In this paper we present a methodology for testing real-time embedded systems, directed specifically at exercising the interactions between system layers, and between the multiple user tasks that enact application behaviors, from the application layer. We augment this with a dynamic analysis on testing data that can detect failures related to incorrect usage of resources within critical sections. We show that our methodology can effectively detect faults in these systems.

## 1 Introduction

Real-time embedded systems are being increasingly relied on to control a wide range of dynamic and complex applications. These applications range from non-safety-critical systems such as cellular phones, media players, and televisions, to safety-critical systems such as automobiles, airplanes, and medical devices. In fact, in the recent past, these systems have been produced at an unprecedented rate, with over four billion units shipped in 2006 by leading manufacturers and independent vendors [25].

Clearly, systems such as these must be sufficiently dependable. However, there is ample evidence that often they are not, with consequences that are expensive or even tragic, as the Ariane explosion [26], the Therac-25 [24], Mars Pathfinder [18], and Chernobyl reactor [2] failures, and numerous reports of lesser accidents (e.g., [16]) illustrate.

Testing is the most commonly used method for validating software systems, and effective testing methodologies could be helpful for improving the dependability of real-time embedded systems. However, there are challenges involved in creating such methodologies. Real-time embedded systems consist of layers of software – application layers utilize services provided by underlying system service and hardware support layers. While this characteristic is also common to other types of software such as desktop and server applications, the pressure to create new products that have more features than the current generation of products and utilize new hardware with new specifications leads the software layers in these systems to evolve at a much higher pace than those of other systems [32]. At the same time, the rapid pace of change leaves less time

to thoroughly test applications as they are developed. This is reflected in recent increases in the number of software faults occurring in newly released versions of embedded systems [32].

As a second challenge, software development practices in portable consumer electronics such as phones and PDAs often separate developers of applications software from the engineers who develop lower system layers. An example of such practices involves Apple's *App Store*, which was created to allow software developers to create and market iPhone and iPod applications. Apple provided the *iPhone SDK*, which is the same tool used by Apple's engineers to develop core applications [11], as a tool 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 of these applications caused these devices to exhibit unresponsiveness in basic applications and system functionalities (e.g., Internet connectivity) when the new applications were launched [8].

In situations such as this, the providers of underlying layers often attempt to test the services that they provide, but they cannot anticipate and test for all of the scenarios in which these services will be employed by developers creating applications. The developers of iPhone applications may have used the interfaces provided by the iPhone SDK in ways not anticipated by Apple's engineers. Improper usage of these interfaces results in failures in the underlying layers that affect operations of these devices. In practice, interactions between application layers and lower layers, and interactions between the various tasks that are initiated by the application layer (referred to here as *user tasks*), are a significant source of such failures in fielded applications.

As a third challenge, a recent study finds that systems running on more than half of small embedded devices are implemented as fully customized applications with few or no standardized software components [25]. These customized applications often use lower-level software components (custom operating systems, device drivers, and libraries) developed by in-house software developers. In this scenario, the common testing practice (used to test desktop and server systems) of assuming that low-level components are correctly implemented and treating them as black-boxes is unsound since these components are developed together to support applications. Because these low-level software components are accessible by these software developers, there is an opportunity to explore new testing methodologies that can increase testing effectiveness by focusing on interactions across these software layers.

In this paper we present a testing methodology addressing the foregoing challenges, that can be used by the developers of real-time embedded applications to increase the dependability of their systems, by focusing on interactions created by their applications as they use underlying systems facilities. Our methodology involves three techniques. Our first technique uses dataflow analysis to distinguish points of interaction between specific layers in real-time embedded systems and between individual software components within those layers. Tracking data that flows across software layers is important because applications instigate these

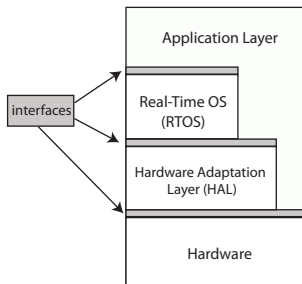


Figure 1: Structure of an embedded system

interactions to obtain services and acquire resources in ways that may ultimately lead to failures. 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 failures. Our second technique extends the first to track interactions between user tasks; these are sources of concurrency and other faults common in such systems. Finally, we augment these techniques with a third that uses additional dynamic analysis on data gathered during testing to detect failures related to incorrect usage of resources within critical sections. We present results of an empirical study of our methodology applied to a commercial real-time embedded system application, that shows that the methodology can be effective in detecting faults.

## 2 Background and Related Work

### 2.1 Overview of Embedded Systems

Embedded systems are designed to perform specific tasks, such as digital signal processing, controlling avionics, or communications. These systems are often deployed in challenging computing environments, most of which have strict real-time requirements. Therefore, correctness of such systems is a matter of both functional and temporal attributes. *In the remainder of this paper, when referring to “embedded systems” we imply systems that must operate under real-time requirements.*

Figure 1 illustrates the structure of a typical embedded system. The application layer consists of applications that interact with the users; these are written by developers and utilize services from underlying software systems such as Real-Time Operating Systems (RTOSs) or Hardware Adaptation Layers (HALs). The gray boxes represent interfaces between these layers. They are central to this work as they can be used to partially observe the internal workings of each layer.

An RTOS is designed to support real-time applications in devices with limited memory; thus, the execution time of most kernel functions must be *deterministic* [20]. As a result, many of these kernels schedule tasks based on priority assignment.

The hardware adaptation layer is a runtime system that manages device drivers and provides interfaces by which higher level software systems (i.e., applications and RTOSs) can obtain services. A typical HAL

implementation includes standard libraries and vendor specific interfaces. With HAL, applications are more compact and portable and can easily be written to directly execute on hardware without operating system support.

To illustrate the way in which these systems interact, consider a case where a user provides a web address to a mobile web-browser application running in a smart phone. 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 then sends part of the processed information to the I/O service task, which processes the information and passes it on to the HAL layer to display it on the screen.

This example illustrates three important points. First, interfaces play an essential role in embedded system application execution. Second, as part of execution, information is often created by one layer and then flows to the other layers for further processing. Third, multiple tasks work together to accomplish this simple request.

## 2.2 Related Work

Differences between desktop and real-time embedded systems have prompted research on testing methodologies explicitly targeting the latter. For example, fault-injection based testing methodologies have been used to test kernels [3, 4, 19] to achieve greater fault-tolerance. Tsoukarellas et al. [38] provide an approach for applying black-box boundary value and white-box basis-path testing techniques to real-time kernels. All of this work, however, addresses testing of the kernel layer, and none of it specifically considers interactions between layers.

Formal specifications have also been used to test real-time embedded systems. Bodeveix et al. [6] present an approach for using timed automata to generate test cases for real-time embedded systems. En-Nouaary et al. [10] 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 [5] verifies embedded systems that can be modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronization. While these approaches may be successful in the presence of the required formal specifications, they are not applicable in the vast majority of practical cases in which such formal models do not exist.

Informal specifications have also been used to test real-time embedded systems from the application level. Tsai et al. [37] present an approach for black-box testing of embedded applications using class diagrams and state machines. Sung et al. [36] test interfaces to the kernel via the kernel API and global variables that are visible in the application layer. The methodology we present, in contrast, is a white-box approach, analyzing

code in both the application and kernel layers to determine testing requirements. Moreover, even in cases where specification information exists, it is well known that code-based and specification-based approaches are typically complementary in the classes of errors that they detect.

There has been some work that applies dataflow analysis to real-time embedded systems. Yu et al. [40] examine definition and use patterns for global variables between kernel and non-kernel modules, and classify these definition and use patterns as being safe or unsafe with respect to the maintainability of the Linux kernel. This work is not, however, directed at testing.

There has been a great deal of research on testing concurrent programs, of which [7, 23] represent only a small subset. In our search of the literature, however, we have not uncovered efforts to apply that research within the context of real-time embedded systems. As discussed in Section 3.5, however, there are many aspects of that work that may be applicable in the context of this work.

Finally, work by Chen and MacDonald [9] supports testing for concurrency faults in multithreaded Java programs by overcoming scheduling non-determinism through value schedules. One major component of their approach involves identifying concurrent definition-use pairs. These are then used as targets for model checking. The third algorithm proposed in this paper also uses a similar notion, but with a broader scope. Instead of detecting only faults due to memory sharing in the application layer, our technique detects faults in shared memory and physical devices that can occur across software layers and among tasks.

### 3 Methodology and Techniques

Our goal is to create a methodology for testing real-time embedded systems that focuses on interactions between layers and user tasks that occur in the context of executing an application. These interactions can be directly and naturally modeled in terms of data that flows from one layer to the next as well as data that flows through resources shared by multiple user tasks. For this reason, in this work we adapt dataflow-based testing approaches to the real-time embedded systems context.

There are three aspects of data usage information that can be useful for capturing interactions. First, observing the actual processes of defining and using values of variables or memory locations can tell us whether variable values are manipulated correctly by the software systems. Second, analyzing intertask accesses to variables and memory locations can also provide us with a way to identify resources that are shared by multiple user tasks, and whether this sharing creates failures. Third, observing the sequences in which data is accessed within critical sections can help us determine whether concurrency faults may have occurred.

Our methodology involves three techniques designed to take advantage of these three aspects, and in so doing, to expose interlayer and intertask faults. The first technique statically analyzes a real-time embedded system at the *intratask* level to calculate data dependencies representing interactions between layers of those

systems. The second technique calculates data dependencies representing *intertask* interactions. The data dependencies identified by these two techniques become coverage targets for testers. The third technique uses information on data usage within critical sections together with data collected during test execution to search for additional anomalies in execution of critical section code that might signal the presence of synchronization errors such as data races.

We next present the basics of dataflow testing, and then we describe our three techniques in turn. (Algorithms are available in the Appendix.) We follow this with a discussion of approaches for dynamically capturing execution information, required for the use of our techniques.

### 3.1 Dataflow Testing

Dataflow analysis algorithms (e.g., [1, 27]) analyze 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 thereby influence the computation.

Dataflow test adequacy criteria, (e.g., [31]), relate test adequacy to definition-use pairs, requiring testers to create test cases that cover certain specified pairs. Dataflow testing approaches can be *intraprocedural* or *interprocedural*; in this work we utilize the latter, building on interprocedural dataflow analysis algorithms presented in [13, 30].

Dataflow test adequacy criteria and dataflow analysis algorithms are not themselves novel. The contribution of this work, however, is the adaptation of these to the real-time embedded systems context, and to the problem of testing interactions between layers and user tasks in the context of an applications program. We 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. Many of these variables are represented as fields in structures, so our technique analyzes structures at the field level. Attending to these dependencies lets us focus on the interactions we wish to validate, and restricting attention to these dependencies lets us do so at a cost much less than would be required in an approach aimed at testing all definition-use pairs in a system.

### 3.2 Technique 1: Intratask Testing

As shown in Section 2, an application program is a starting point through which underlying layers in real-time embedded systems are invoked. Beginning with the application program, within given user tasks,

interactions then occur between layers and between components of system services. Our first algorithm uses dataflow analysis to locate and direct testers toward these interactions.

Our technique includes three overall steps: (1) procedure *ConstructICFG* constructs an interprocedural control flow graph (ICFG), (2) procedure *ConstructWorklist* constructs a worklist based on the ICFG, and (3) procedure *CollectDUPairs* collects definition-use pairs. These procedures are applied iteratively to each user task  $T_k$  in the application program being tested.

*Step 1: Construct ICFG.* An ICFG [30, 22] is a model of a system’s control flow suitable for use in performing 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 real-time embedded systems 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, and also 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 of the code) 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 (e.g., `irq_disable()`), (b) file IO operations for accessing hardware devices (e.g., `fopen()`), and (c) macros for accessing special registers (e.g., `_builtin_writel()`). However, we do not include CFGs corresponding to functions from standard libraries such as `stdio.h`. Algorithms for constructing CFGs and ICFGs are well known and thus we do not discuss them here; see [1, 27, 30, 22] for details.

*Step 2: Construct Worklist.* Because the sets of variables that we seek data dependence information on are sparse we use a worklist algorithm to perform our dataflow analysis. Worklist algorithms (e.g., [34]) initialize worklists with definitions and then propagate these around ICFGs. Step 2 of our technique does this process by initializing our Worklist to every definition of interest in the ICFG for  $T_k$ . Such definitions include variables explicitly defined, created as temporaries to pass data across function calls or returns, or passed through function calls. However, we consider only variables involved in interlayer and interprocedural interactions; these are established through global variables, function calls, and `return` statements. We thus retain the following types of definitions:

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

4. variables given as arguments to **return** statements
5. constants or computed values given as arguments to **return** statements (passed as temporary variables)

The procedure to construct 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 consideration of 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 also has two associated stacks, **callstack** and **bindingstack**, that keep track of 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  $n$  to determine effects that happen at that successor. 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 **binding** stack 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 definition-use pairs that are intratask and interprocedural, and represent interactions between layers and components of system services utilized by each user task. These are then coverage targets for use in testing each user task.

### 3.3 Technique 2: Intertask Analysis

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

Our aim in intertask analysis is to identify intertask interactions by identifying definition-use pairs involving SVs that may flow across user tasks.

The algorithm begins by iterating through the CFGs that are included in ICFGs corresponding to one or more user tasks. For each CFG the algorithm locates statements 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 that CFG.

Next, the algorithm iterates through each  $T_k$ , and for each CS-entry-exit pair  $C$  associated with  $T_k$  it collects a set of SVs, consisting of each definition or use of a global variable in  $C$ . The algorithm omits definitions (uses) from this set that are not “downward-exposed” (“upward-exposed”) in  $C$ , that is, definitions (uses) that cannot reach the exit node (cannot be reached from the entry node) of  $C$  due to an intervening re-definition. These definitions and uses cannot reach outside, or be reached from outside, the critical section and thus are not needed; they can be calculated conservatively through local dataflow analysis within  $C$ . The resulting set of SVs are associated with  $T_k$  as *intertask definitions and uses*.

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

### 3.4 Technique 3: Critical Section Analysis

As discussed in Section 3.3, shared resources are represented in code as shared variables (SVs). Testing intertask interactions between these SVs can reveal intertask errors in embedded systems applications programs. Further analysis of the SVs occurring in critical sections, paired with data retrieved from execution of test cases, can help detect various classes of anomalies related to intertask synchronization, including data

races, synchronization, deadlock, and priority inversion. Our third technique provides *an analysis technique designed to detect data races* (cases in which protected variables are accessed by multiple user tasks within a critical section) that can occur due to intertask interactions. In systems that utilize disabling interrupts to provide execution isolation, a service request requiring execution isolation from within a critical section can cause the critical section to be prematurely unprotected.

Our technique relies on the fact that, given a critical section in user task  $T_k$  that contains a sequence of definitions and uses of SVs in all possible layers, these definitions and uses must be executed without intervening accesses by other user tasks to the SVs. By statically calculating the interlayer sequences of definitions and uses of SVs (*SV-sequences*) associated with critical sections, we can later compare sequences obtained in testing against expected sequences, and check for anomalies. Notably, this approach is inspired by the way a typical transactional memory system detects memory access conflicts from multiple memory transactions [12, 15].

Calculating static SV-sequences is accomplished for each user task  $T_k$  by postprocessing the ICFG created for  $T_k$  by Step 1 of Technique 1, and considering each critical section  $C$  gathered for  $T_k$  during Technique 2.

Note that a critical section may contain multiple paths, and thus, for a given critical section there may be a set of SV-sequences. In principle, keeping such sets could be expensive if critical sections contain loops or deeply nested decision constructs, because these could lead to large numbers of SV-sequences. In practice, however, critical sections are deliberately kept short and loop-free by programmers of real-time embedded systems because large critical sections result in significant execution delay due to contention.

The SV-sequence calculation algorithm performs a depth-first traversal of the portion of the ICFG beginning at the critical section entry. 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. On reaching the critical section exit the algorithm outputs the set of SV-sequences collected, removing any duplicates that might occur (i.e., if identical sequences of SVs exist along multiple paths).

Having calculated the SV-sequences for each critical section in each user task  $T_k$ , the process of using these to detect anomalies is as follows. First, the program is run with tests and dynamic definition-use information is collected relative to SVs occurring in critical sections (Section 3.5 describes this process). Second, the collected information is analyzed with a focus on the SV-sequences associated with each critical section, to obtain sets of *dynamic SV-sequences*. Third, the sets of dynamic SV-sequences are compared to the static SV-sequences to identify cases in which protected shared variables occur within the dynamic sequences that are not contained in the static sequences. Such cases indicate data races.

### 3.5 Recording Coverage Information

All three of the techniques that we have described require access to dynamic execution information, the first two in the form of definition-use pairs and the third in the form of SV-sequences. 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 and SV-sequences. 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  $D * T$ . Tracking SV-sequences could be costly since these could involve space exponential in program size; however, a listener can monitor actively for violations using the approach described in Section 3.4 rather than recording entire sequences, reducing this cost.

A drawback of the foregoing scheme involves the overhead introduced by instrumentation, which can alter time-dependent aspects of program behavior. Currently, the focus of our methodology is to make functional testing more effective; thus, it does not directly address the issue of testing for temporal faults. Still, 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., [7, 23]) 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 [28, 39]. Currently, most of these platforms do not support testing based on time-sensitive execution; however, we believe that they can be extended to support minimally intrusive instrumentation for the purpose of data collection. These systems provide a global clock that can be used for coarse-grained synchronization of events. The global clock can be adjusted to make the amount of time needed to simulate a task reflective of the actual time to execute the same task on real systems. It can also be stopped during simulation, making it possible to factor out instrumentation time from virtual execution time. 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.

## 4 Empirical Study

To empirically assess our approach we conducted an empirical study. The research question we address is whether the use of our algorithms can create test suites that are more effective at detecting faults than those which might be created by a common current practice.

### 4.1 Object of Analysis

As an object of study we chose an embedded system application, MAILBOX, that involves the use of a real-time embedded system’s mailboxes by three different user tasks. Once a task receives a message, the task

increments and posts the message for the next task to receive and increment. The application consists of nine functions and 268 non-comment lines of C code; it invokes, directly or indirectly, 20 kernel, 10 HAL, and two hardware layer functions consisting of 2,380 non-comment lines of C.

To address our research question we required faulty versions of our object program, and to obtain these we followed a protocol introduced in [17] for assessing testing techniques. We asked a programmer with over ten years of experience including experience with real-time embedded systems, but with no knowledge of our testing approach, to insert potential faults into the application and system code relevant to MAILBOX. This process resulted in the introduction of 19 potential faults into the application layer code; 196 potential faults into the kernel code related to task management, task initialization, task scheduling, and resource (mailbox) management; and 38 potential faults into HAL code related to IRQ management.

## 4.2 Variables, Measures, and Additional Factors

### 4.2.1 Independent Variable

Our independent variable is the testing approach used, and we use the three algorithms described in Section 3. As a baseline for comparison we use an existing black-box testing methodology, the category-partition method [29], which uses a Test Specification Language, TSL, to encode choices of parameter and environmental conditions that affect system operations 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 program faults, to employ the category-partition method. This yielded a test suite of 26 test cases.

### 4.2.2 Dependent Variable

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

## 4.3 Additional Factors

Like many real-time embedded 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 three MAILBOX user tasks and the kernel. To capture these differences we utilized two loop counts for all executions, 10 and 100, representing a relatively minimal count and a count that is an order of magnitude larger. For each application of each testing approach, we report results in terms of each loop count.

A second factor of interest involves the locations of faults, and differences between techniques as related to those locations. To consider this factor, in our analysis we examine fault detection for the three layers — application, kernel, and HAL — independently.

Table 1: Test Cases Created and Percentage Coverage Achieved

<i>Test Type</i>	<i>Intratask</i>		<i>Intertask</i>	
	Number	Coverage	Number	Coverage
TSL	26	80.6 %	26	75.4 %
DU	14	100 %	10	100 %

#### 4.4 Study Setup

We implemented our algorithm to operate on rapid prototyping systems from Altera. 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 platform runs under MicroC/OS-II, a commercial grade real-time operating system, which hosts the MAILBOX object that we selected.

Our prototype relies on the ARISTOTLE [14] program analysis infrastructure to gather control flow graphs and basic definition and use information for C source code. We construct ICFGs from this data, and use the definition and use information together with additional analysis of source code to apply the algorithms and collect definition-use pairs. To monitor execution of definitions, uses, and definition-use pairs we use the GNU debugger and scripts to record accesses to variables at definition and use points.

We used our prototype to calculate intratask and intertask definition-use pairs in the program using the algorithms described earlier. This process identified 253 intratask pairs and 65 intertask pairs. We executed the TSL test suite on the program and determined its coverage of intratask and intertask pairs. 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 second set is created independently of the first; that is, it does *not* include the test cases created to cover intratask pairs.) Table 1 reports the numbers of test cases created and the percentage of definition-use pairs they covered, for the intratask and intertask cases. All of the intratask and intertask pairs were executable (feasible).

Note that by choosing to create coverage-adequate test suites by *augmenting* the TSL suite, we ensure that these coverage-adequate suites are supersets of the TSL suite. This is important, because it allows us to assert that any differences seen in the results of these suites are not due simply to differences in the specific inputs chosen for the test cases, but rather, due to the additional power obtained from expanding the initial TSL suite to achieve coverage of intratask or intertask definition-use pairs.

Finally, we 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 loop count, we executed 50 test cases for the object program (26 + 14 + 10; the TSL test cases are the same for the intratask and intertask cases) and performed 950 (50 × 19) executions for the faulty applications, 9,800 (50 × 196) executions for the faulty kernels, and 1,900 (50 × 38) executions for the faulty HALs. For each loop count, we also executed these test cases to record DU sequences.

Table 2: Fault Detection per Layer, Test Type, and Loop Count for Intratask and Intertask

Layer	(A) Intratask								(B) Intertask							
	Loop Count = 10				Loop Count = 100				Loop Count = 10				Loop Count = 100			
	# of Faults Detected				# of Faults Detected				# of Faults Detected				# of Faults Detected			
	Total	TSL	DU	Both	Total	TSL	DU	Both	Total	TSL	DU	Both	Total	TSL	DU	Both
Application	14	0	4	10	15	0	4	11	12	3	2	7	13	3	2	8
Kernel	24	3	4	17	28	4	7	17	23	5	3	15	26	6	5	15
HAL	13	0	0	13	13	0	0	13	13	0	0	13	13	0	0	13

We then 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. Similarly, we used a differencing tool on the DU sequences for the initial and faulty versions of the program to determine whether these exhibited differences.

## 4.5 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 program we investigate is a commercial real-time embedded systems program, the faults are inserted by an experienced programmer, and the test suites are created using practical processes.

The primary threat to the internal validity of this study is possible faults in the implementation of the algorithms, 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 manually determine correct results.

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

## 4.6 Results

### 4.6.1 Technique 1: Intratask Testing

Table 2(A) shows fault detection results for loop counts of 10 and 100, listing the total number detected, the numbers detected by just TSL and DU test suites, respectively, and the numbers detected by both suites. At loop count 10, in the application layer, TSL and DU suites are able to detect 10 faults in common and the DU suite detects four additional faults. In the kernel layer, the TSL and DU suites detect 17 faults in common, the TSL suite detects three more faults, and the DU suite detects four additional faults. In the HAL layer, TSL and DU suites detect the same 13 faults.

At loop count 100, in the application layer, TSL and DU test suites detect 11 faults in common, and DU suites detect four additional faults. In the kernel layer, TSL and DU suites detect 17 faults in common, the

Table 3: Fault Detection for Critical Section Analysis

<i>Layer</i>	<i>Loop Count = 10</i>				<i>Loop Count = 100</i>			
	# of Faults Detected				# of Faults Detected			
	Total	TSL	DU	Both	Total	TSL	DU	Both
App.	12	2	1	9	14	4	1	9
Ker.	23	2	7	14	8	3	2	3

TSL suite detects four additional faults, and the DU suite detects seven different additional faults. In the HAL layer, TSL and DU suites detect the same 13 faults.

#### 4.6.2 Technique 2: Intertask Testing

Table 2(B) shows the fault detection results for TSL and DU test suites (at the intertask level) for both loop counts. At loop count 10, in the application layer, TSL and DU suites detect seven faults in common, the TSL suite detects three additional faults, and the DU suite detects two different additional faults. In the kernel layer, the TSL and DU suites detect 15 faults in common, the TSL suite detects five additional faults, and the DU suite detects three different additional faults. In the HAL layer, TSL and DU suites detect the same 13 faults.

At loop count 100, in the application layer, TSL and DU test suites detect eight faults in common, TSL suites detect three additional faults, and DU suites detect two different additional faults. In the kernel layer, TSL and DU suites detect 15 faults in common, the TSL suite detects six additional faults, and the DU suite detects five different additional faults. In the HAL layer, TSL and DU suites detect the same 13 faults.

#### 4.6.3 Technique 3: Critical Section Analysis

Table 3 shows the fault detection results obtained by our critical section analysis approach, for the TSL and DU test suites at loop counts of 10 and 100, for the application and kernel layers. (We omit results for the HAL layer because the TSL and DU test suites did not reveal different faults at the HAL level.) At loop count 10, in the application layer, TSL and DU suites detect nine faults in common, the TSL suite detects two additional faults, and the DU suite detects one different additional fault. In the kernel layer, the TSL and DU suites detect 14 faults in common, the TSL suite detects two additional faults, and the DU suite detects seven different additional faults.

At loop count 100, in the application layer, TSL and DU suites detect nine faults in common, the TSL suite detects four additional faults, and the DU suite detects one different additional fault. In the kernel layer,<sup>1</sup> the TSL and DU suites detect three faults in common, the TSL suite detects three additional faults, and the DU suite detects two different additional faults.

<sup>1</sup>Due to time constraints we were able to execute only 20% each of the TSL and DU test cases at this layer.

## 4.7 Discussion.

### 4.7.1 Intra and intertask testing

As our results show, the use of our intra and intertask testing techniques help us detect more faults than can be detected using the TSL test suite alone. For intratask testing, in the application layer the TSL suites detect a subset of those detected by the DU suites, whereas in the kernel layer, the two types of suites are complementary in terms of faults they detect. For intertask testing, in the application and kernel layers DU and TSL test suites are complementary.

Our results also show that some faults are sensitive to execution time. Table 2 does not show the precise number of such faults, but further analysis reveals that at the intratask level, loop count 100 reveals five faults not revealed at loop count 10, and at the intertask level, loop count 100 reveals four faults not revealed at loop count 10.

We also analyzed the data to determine how the two approaches, intratask and intertask, compared to each other. In the application layer at both loop counts and in the kernel at loop count 10, the intratask DU test suite is a superset of the intertask DU test suite in terms of the faults that they can detect. In the kernel at loop count 100, the intratask and intertask DU test suites are complementary.

By further analyzing the types of faults revealed we were able to determine that both the TSL and the DU test suites can detect faults related to invalid priority values (e.g., priority already in use, priority higher than the allowed maximum), deletion of idle tasks, and NULL pointer assignments to resources in the kernel layer. The DU test suites, however, detected faults related to resource time-outs and deletion of tasks that do not exist in the kernel layer that were not revealed by the TSL test suites.

At the intratask and intertask levels, both the TSL and DU test suites detected the same faults in HAL. Further examination shows that in the case of the MAILBOX application, the execution paths invoked by TSL test cases do, by themselves, cover all interlayer definition-use pairs involving the HAL layer related to the faults that were present in that layer.

### 4.7.2 Critical section analysis

Critical section analysis also revealed faults in the application. Both DU and TSL test suites were able to do this at the application and kernel layers, and here too they were complementary in terms of the faults that they can detect. Further analysis of the data also shows that loop count 100 reveals two distinct application layer faults not revealed at loop count 10.

We also analyzed the data to determine the relationship between critical section analysis and the two testing techniques. In the application layer for both loop counts, executing TSL test suites and analyzing critical sections are complementary in terms of the faults that they can detect. For the DU test suites, however, the faults detected by critical section analysis were a subset of those detected by testing.

Interestingly, there was one particular fault related to manipulating sleeping times of user tasks that went undetected by the test suites at loop count 10, but was detected at loop count 100. By analyzing sequences in the critical sections, we discovered that this fault was, in fact, detectable with the TSL test suite at loop count 10 when the sequences had already begun to differ but not to the point that resulted in detectable failures. One potential implication of this is that analyzing violated sequences may enable detection of time sensitive faults independent of execution times.

## 5 Conclusions and Future Work

We have presented a new methodology for use in testing real-time embedded systems. Our methodology focuses on interactions between layers and user tasks in these systems, to expose classes of failures that developers face as they create new applications. We have conducted an empirical study applying our techniques to a commercial real-time embedded system, and demonstrated that the approach has the potential to reveal faults not revealed by a common black-box application testing approach.

There are several avenues for future work. We have already discussed (Section 3.5) prospects for using simulation platforms to create non-intrusive instrumentation environments. Also, while we have focused here on source code analysis, underlying layers of real-time embedded systems are often provided only in object code. Our analyses can be adapted, however, to include such code (see, e.g., [33]). 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. Note that, unlike analyses that depend on conservative approximations of dataflow such as for optimizations, testing techniques do not require such conservativeness; underestimates of dependencies do identify useful interactions to exercise. Still, incorporating alias and points-to analyses (e.g., [21, 35]) could help us identify additional and potentially important definition-use pairs.

## Acknowledgements

This work was supported in part by NSF under Award CNS-0720757 to the University of Nebraska - Lincoln. This work was supported in part by a Korea Research Foundation Grant funded by the Korean Government (MOEHRD) under award KRF-2007-357-D00215. Wayne Motycka helped with the setup for the empirical study.

## References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 1986.

- [2] L. R. Anspaugh, R. J. Catlin, and M. Goldman. The global impact of the Chernobyl reactor accident. *Science*, 242(4885):1513–1519, 1988.
- [3] J. Arlat, J. Fabre, M. Rodriguez, and F. Salles. Dependability of COTS microkernel based systems. *IEEE Trans. Comp.*, 51(2):138–163, 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 *Proc. Conf. COTS-Based Softw. Sys.*, pages 233–242, Feb. 2007.
- [5] G. Behrmann, A. David, and K. G. Larson. A tutorial on UPPAAL. <http://www.uppaal.com>, 2004.
- [6] J. P. Bodeveix, R. Bouaziz, and O. Kone. Test method for embedded real-time systems. In *Proc. W. Soft.-Intensive Dep. Emb. Sys.*, pages 35–38, 2005.
- [7] R. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8(2):66–74, Mar. 1991.
- [8] 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.
- [9] J. Chen and S. MacDonald. Testing concurrent programs using value schedules. In *Proc. Int’l. Conf. Auto. Softw. Eng.*, pages 313–322, 2007.
- [10] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.*, 28(11):1203–1238, 2002.
- [11] A. Gonsalves. Apple iphone sdk downloads reach 100,000. <http://www.informationweek.com/news/software/showArticle.jhtml?articleID=206903229>, 2008.
- [12] T. Harris, A. Cristal, O. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, May 2007.
- [13] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Symp. Found. Softw. Eng.*, Dec. 1994.
- [14] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC- 3/97-TR17, Ohio State University, Mar 1997.
- [15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. Int’l. Symp. Comp. Arch.*, pages 289–300. May 1993.

- [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 *Proc. Int'l. Conf. Softw. Eng.*, pages 191–200, May 1994.
- [18] 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), December 1997.
- [19] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Proc. Int'l. Symp. Rel. Distr. Sys.*, pages 72–79, Oct. 1997.
- [20] J. J. Labrosse. *MicroC OS II: The Real Time Kernel*. CMP Books, 2002.
- [21] B. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39(4):473–489, Mar. 2004.
- [22] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Symp. Princ. Prog. Lang.*, pages 93–103, Jan. 1991.
- [23] Y. Lei and R. Carver. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 32(6), June 2006.
- [24] N. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.
- [25] LinuxDevices.com. Four billion embedded systems shipped in 06. <http://www.linuxdevices.com/news/-NS3686249103.html>, Jan. 2008.
- [26] J. L. Lyons. ARIANE 5, flight 501 failure, report by the inquiry board. <http://www.ima.umn.edu/arnold/disasters/-ariane5rep.html>, July 1996.
- [27] S. Muchnick and N. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [28] Open Virtual Platforms. Open Virtual Platforms. Web-page, 2008. <http://www.ovpworld2.org>.
- [29] T. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6), June 1988.
- [30] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations in C programs. *IEEE Trans. Softw. Eng.*, 20(5):385–403, May 1994.

- [31] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, Apr. 1985.
- [32] B. Robinson. Software faults in devices on the increase. <http://www.intomobile.com/2009/01/11/software-faults-on-devices-on-the-increase.html>, 2009.
- [33] J. Seo, A. Sung, B. Choi, and S. Kang. Automating embedded software testing on emulated target board. In *W. Auto. Soft. Test*, pages 1–7, May 2007.
- [34] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *Trans. Softw. Eng. Meth.*, 10(2):209–254, Apr. 2001.
- [35] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. Symp. Princ. Prog. Langs.*, pages 32–41, Jan. 1996.
- [36] A. Sung, B. Choi, and S. Shin. An Interface test model for hardware-dependent software and Embedded OS API of the embedded system. *ELSEVIER Comp. Stds. Int.*, 29(4):330–343, Apr. 2007.
- [37] W. Tsai, L. Yu, F. Zhu, and R. Paul. Rapid embedded system testing using verification patterns. *IEEE Softw.*, 22(4):68–75, 2005.
- [38] 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.
- [39] Virtutech. Virtutech Simics. Web-page, 2008. <http://www.virtutech.com>.
- [40] L. Yu, S. R. Schach, K. Cehn, and J. Offutt. Categorization of common coupling and its application to the maintainability of the Linux kernel. *IEEE Trans. Softw. Eng.*, 30(10):694–705, Oct. 2004.

## Appendix

Our intratask analysis consists of three overall steps, procedure *ConstructICFG*, procedure *ConstructWorklist*, and procedure *CollectDUPairs*, which are applied iteratively to each user task  $T_k$ .

---

**Algorithm.** *IntraTask Analysis*

**Require:**

$T_k$ : all CFGs associated with user task  $k$

**Return:**

Set *DUPairs* of definition-use pairs to cover

**begin**

1.  $ICFG = \text{ConstructICFG}(T_k)$
2.  $Worklist = \text{ConstructWorklist}(ICFG)$
3.  $DuPairs = \text{CollectDuPairs}(Worklist)$

**end**

---

---

**Procedure** *ConstructICFG*( $T_k$ )

**Require:**

$T_k$ : all CFGs associated with user task  $k$

**Return:**

*ICFG* for  $T_k$

**begin**

1. **for** each *CFG*  $G$  in  $T_k$
2.   **for** each node  $n$  in  $G$
3.     **if**  $n$  is a call to function  $f$  **then**
4.       **if**  $f$  is not a library routine and *CFG* of  $f$  is not marked **then**
5.          create edge from  $n$  to entry node for *CFG* of  $f$
6.          create return node  $r$  in  $G$  associated with  $n$
7.          create edge from exit node in *CFG* of  $f$  to  $r$
8.       **endif**
9.     mark  $G$  visited
10. **endfor**

**end**

---

---

**Procedure** *ConstructWorklist(ICFG)*

**Require:**

*ICFG* from Step 1

**Return:**

*Worklist* of definitions

**begin**

1. **for** each node  $n$  in *ICFG*
  2.     **switch** ( $n.type$ )
  3.         *entry node:*
  4.             **continue** /\* no action required \*/
  5.         *function call:*
  6.             **for** each parameter  $p$
  7.                 **if**  $p$  is a variable  $v$  **then** add  $(n, v)$  to *Worklist*
  8.                 **if**  $p$  is a constant or computed value **then** create a
  9.                     definition  $d$  representing it and
  10.                     add  $(n, d)$  to *Worklist*
  11.             **endfor**
  12.         *return statement node:*
  13.             **if**  $n$  returns variable  $v$  **then** add  $(n, v)$  to *Worklist*
  14.             **if**  $n$  returns a constant or computed value **then**
  15.                 create a definition  $d$  representing it
  16.                 add  $(n, d)$  to *Worklist*
  17.         *other:*
  18.             **for** each definition of global variable  $v$  in  $n$
  19.                 add  $(n, v)$  to *Worklist*
  20.             **endfor**
  21.     **endswitch**
  22. **endfor**
- end**
-

---

**Procedure** *CollectDuPairs(Worklist)***Require:**

*Worklist* from Step2 and *ICFG* from Step1

**Return:**

Set *DuPairs* of form  $(d, u)$

**begin**

1. **for** each definition  $d$  in *Worklist*
2.     *Add\_to\_Nodelist(d.node,  $\phi$ , d.id)* /\* add  $d$  to *Nodelist*\*/
3.     **while** (*Nodelist*  $\neq \phi$ )
4.         extract the first item  $N$  from *Nodelist*
5.         mark  $N.node$  visited in *ICFG*
6.         **switch** ( $N.type$ )
7.             call to function  $f$ :
8.                  $N.callstack\_old = N.callstack$
9.                 push ( $N.callstack$ , caller of  $f$ )
10.                 /\* if there is a use of global variable  $d$  \*/
11.                 **if**  $d$  is a global variable and  $N$  contains use  $u$  of  $d$  **then**
12.                     add  $(d, u)$  to *DuPairs*
- 13.
14.                 /\* if  $d$  is a global variable and  $f$  has no parameters \*/
15.                 **if**  $d$  is a global variable and parameter  $p$  of  $f$  is  $\phi$  **then**
16.                     *Add\_to\_Nodelist(entrynode of  $f$ ,  $N.callstack$ ,  $d.id$ )*
- 17.
18.                 **for** each parameter  $p$  in call to  $f$
19.                     /\* if  $d$  is a global variable and not used as a parameter \*/
20.                     **if**  $d$  is a global variable and  $d \neq p$  **then**
21.                         *Add\_to\_Nodelist(entrynode of  $f$ ,  $N.callstack$ ,  $d.id$ )*
22.                     /\* if  $d$  is a global variable and  $p$  contains use  $u$  of  $d$  \*/
23.                     **if**  $d$  is a global variable and  $p$  contains use  $u$  of  $d$  **then**
24.                         add  $(d, u)$  to *DuPairs*
25.                     /\* if  $d$  appears as  $p$  or  $N$  has a definition,
26.                         e.g., as for a local variable  $x$ ,  $f(x)$  or  $y = f(x+1)$  \*/
27.                     **if**  $p$  is  $d$  or  $N.node$  is  $d.node$  **then**

```

28.         binding_info = get_formal_parameter(f, d.id)
29.         push (N.bindingstack, binding_info)
30.         if f is call-by-value then
31.             Add_to_Nodelist(entrynode of f, N.callstack, N.bindingstack)
32.             Propagate(N.node, N.callstack_old, d.id)
33.         else if f is call-by-reference then
34.             Add_to_Nodelist(entrynode of f, N.callstack, N.bindingstack)
35.         endfor
36.
37.     return statement node:
38.         if return statement node contains use u of d then
39.             add (d, u) to DuPairs
40.         /* if d appears as a return variable rv or
41.         N has a definition at the return statement e.g., return 1 */
42.         if rv is d or N.node is d.node then
43.             /* not popping but reading the top of callstack */
44.             t = read_top_of(N.callstack)
45.             if t is  $\phi$  then /* find possible callers by examining
46.                 the exit node of current CFG */
47.                 find_callers(ICFG, CFG of d)
48.             for each caller
49.                 get the return node r in the caller
50.                 if r has a definition then
51.                     get use u of d associated with r
52.                     add (d, u) to DuPairs
53.             endfor
54.         else /* t  $\neq$   $\phi$  */
55.             get the return node r in t
56.             if r has a definition then
57.                 get use u of d associated with r
58.                 add (d, u) to DuPairs
59.         /* successor of each return statement node is exit node */
60.         Add_to_Nodelist(exitnode of CFG of d, N.callstack, N.bindingstack)
61.

```

```

62.      exit node:
63.          /* go back to return node r in the caller */
64.          pop (N.callstack)
65.          if r is call-by-reference or d is a global variable then
66.              Propagate (r, N.callstack, pop(N.bindingstack))
67.
68.      other:
69.          if N.node contains use u of d
70.              add (d, u) to DuPairs
71.          if d is killed at N.node
72.              continue
73.          Propagate(N.node, N.callstack, d.id)
74.      endswitch
75.  endwhile
76. endfor
end

```

**Procedure** *Add\_to\_Nodelist*(*node*, *callstack*, *bindingstack*)

**begin**

1. create new node *N* on *Nodelist*
2. *N.node* = *node*
3. *N.callstack* = *callstack*
4. *N.bindingstack* = *bindingstack*

**end**

**Procedure** *Propagate*(*node*, *callstack*, *bindingstack*)

**begin**

1. **for** each successor *s* of *node* in the *ICFG*
2. **if** *s* is not visited **then**
3. *Add\_to\_Nodelist*(*s*, *callstack*, *bindingstack*)
4. **endfor**

**end**

---