Ultra Scientist Vol. 24(1)B, 41-48 (2012).

Testing faults in embedded system application

NEELESH JAIN[1] and ASHISH VERMA[2]

[1]Research Scholar, Department of Computer Science and Application
[2]Assistant Professor, Department of Physics and Electronics
Dr. Hari Singh Gour Central Vishwavidyalaya Saugor M.P. (INDIA)
#neelsgr@rediffmail.com
*vermaashish31@rediffmail.com

(Acceptance Date 26th December, 2011)

Abstract

In this paper we have presented an approach of testing faults in embedded system application software. Faults occur when application software communicates with the underlying components of the system. We have introduced two methods which involves dataflow analysis to identify intra-task communication and inter-task communication between specific layers in embedded systems and between software components within those layers.

Key words: Embedded System Testing, Faults.

I. Introduction

Software testing[1-10] is a process or a sequential processes, of dynamically executing a program with given inputs, to make sure that code designed for embedded system do the things what it is designed for. In this paper we have presented an approach of testing faults in embedded system application built to run in a mobile. Non-temporal faults occur when application software communicates with the underlying components of the mobile. Our approach involves two methods. First method involves dataflow analysis to identify intratask communication between different layers of embedded system and between software components with those layers. Second Method uses this information to identify inter-task communication between the various task invoked by the application layer. Test cases have been defined using both of these classes.

II. *Structure of Embedded System :*

Embedded system is designed to perform specific tasks in an environment consist of software and hardware components. Figure 1 below shows the structure of embedded system Embedded system is divided into four layers in which software covers three layers and hardware covers only single layer. An application layer consist of a program written in specific language

like C / C++ / Java etc, which utilizes Real Time Operating System (RTOS) and Hardware Adaptation Layer (HAL) to do things for which it is designed for. The dark layer shows the interfaces between these layers[11-20].

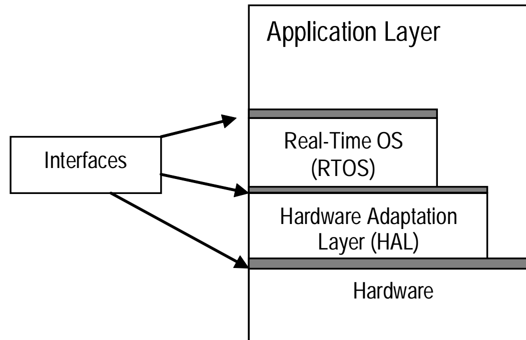

Figure 1. Structure of embedded system

III. *Design of testing method :*

One of the characteristics of soft real time embedded system is communications with the system components. In a mobile when a user uses the internet and provides a web address to the mobile web-browser, corresponding no of sequential processes starts. The web-browser invokes the operating system - provided API to transfer the web address to a remote server. The remote server then transmits information back to the mobile phone through the network service task, which set off an interrupt to notify the web-browser task of pending information. The browser processes the task and 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. So we can conclude[21-30] the following:

- Communication between layers (via interfaces) plays an important role in execution of an embedded system application.
- Information is created in one layer and then it flows to another layer for processing.
- To accomplish a simple request, multiple tasks work together, communicating every layer

So we can conclude that communication between different layers and various tasks plays an important role in operation of an embedded system.

IV. *Test method :*

The communication in embedded system can be defined as the data that flows across software layers and the data that flow between components within layers. The sample data used to track the faults across software layers are important because application use this to initiate communication. As the data is exchanged with the lower layer, tracking communication data between components within that layer capture s data propagation effects that may result to the exposure of faults[31-40].

So we can result in, that testing approach involving data communication is very useful and helpful for detecting faults in embedded system. We thus consider dataflow based testing approach in the embedded system context. There are two methods of data usage information that can be used for recording communication. 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 inter-task accesses to variables and memory locations can also provide us with a way to identify resources

that are shared by multiple user tasks, and whether this sharing creates failures.

Our approach involves two methods designed to take advantage of these two aspects of data usage information. First method calculate the data dependencies of communication between layers and the system, *i.e.*, intra-task and the Second method calculates the data dependencies representing inter-task communications. The data identified is the target under test (TUT).

Before we explain our testing methods, let us review the basics of dataflow testing. This is reviewed because data acquisition method will be required in our methods[5-19].

[1] A. Dataflow Testing :

Dataflow analysis method is used to 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 affect the computation.

Dataflow test adequacy criteria[6,9] relate test adequacy to definition-use pairs, requiring test engineers to create test cases that cover specific pairs. Dataflow testing approaches can be *intraprocedural* or *interprocedural;* in this work we utilize the latter, building on algorithms presented[6,7,35].

In our TUT, we will focus on data dependencies that correspond to communication between system layers and component 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 variable because these are represented as variables in the underlying system layers. Accesses to physical devices can also be tracked because these devices are mapped to physical memory addresses. Many of the variables utilized in embedded system are represented as fields in structures, which can be tracked by analyses that operate at the field level. Attending to the dependencies in variable thus lets us focus on the communications between different layers and system, which is to be validated. Moreover, restricting attention to these dependencies is less expensive than targeting all definition-use pairs in a system[21-40].

[2] Method 1: Intra-task Testing :

An application program invokes the underlying layers of embedded systems. Beginning with the application program, within given user tasks, communication then occurs between layers and between components of system services. Our first algorithm uses dataflow analysis to locate and direct testers toward these communications.

Our method includes three steps in our method: (1) procedure *ConstructICFG* constructs an interprocedural control flow graph (ICFG), (2) procedure *Construct Worklist*

constructs a worklist based on the ICFG, and procedure[3] *CollectDUPairs* collects definition-use pairs. These procedures are applied iteratively to each user task T_k in the application program being tested[1-11].

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

In embedded system applications[11-36], the ICFG for a given user task Tk 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 communication, 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: ConstructWorklist. 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 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 Tk. 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 communication; these are established through global variables, function calls, and return statements. We thus retain the following types of definitions:

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

The procedure for constructing a worklist considers each node in the ICFG for T_k, and depending on the type of node (entry, function call, return statement, or other) takes specific actions. Entry nodes require no action (definitions are created at call nodes). Function calls require parameters, constants, and computed values (definition types 2-3) to be added to the

worklist. Explicit return statements require consideration of variables, constants, or computed values appearing as arguments (definition types [4-5]). Other nodes require consideration of global variables (definition type 1). Note that, when processing function calls, we do consider definitions and uses that appear in calls to standard libraries. These are obtained through the APIs for these libraries - the code for the routines is not present in our ICFGs.

Step 3: Collect DUPairs. Step 3 of our method 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[2-39].

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[1-15] 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[16-40]. *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 intra-task definition-use pairs, which represent communication between layers and components of system services utilized by each user task. These are coverage targets for engineers when creating test cases to test each user task.

[3] Method 2 : Inter-task Testing :

To effect inter-task communication, user tasks access each others' resources using shared variables (SVs), which include physical devices accessible through memory mapped I/O. SVs are shared by two or more user tasks and are represented in code as global variables with accesses placed within critical sections.

Programmers of embedded systems use various approaches to control access to critical sections, including calling kernel APIs to acquire/release semaphores, calling the HAL API to disable or enable IRQs (interrupt requests), or writing 1 or 0 to a control register using a busy-wait macro.

Our aim in inter-task analysis is to identify inter-task communication 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 is associated with Tk as inter-task definitions and uses. Finally, the algorithm pairs each inter-task definition d in each Tk with each inter-task use of d in other user tasks to create inter-task definition-use pairs. These pairs are then coverage targets for use in inter-task testing of user tasks.

V. Conclusion and future work

In this paper we have presented a new approach for testing embedded software. Our approach was focused on the communication between layers and user task in the systems. We have also exposed some non-temporal failures faced by the new software developers.

There are several opportunities for future work. The empirical study applying our techniques to the embedded system can be done. Also the study of these techniques on large set of objects programs is to be continued.

VI. References

1. Neelesh Jain, Ashish Verma, Shalini Jain and Ramashanker, Testing Embedded Software in Microcontroller Based Blood Transfusion System In WECON (2011).
2. N. Al Moubayed and A. Windisch. Temporal white-box testing using evolutionary algorithms. In ICST, (2009).
3. Altera Corporation. Altera Nios-II Embedded Processors. http://www.altera.com/products/devices/nios/nio-index.html.
4. J. Arlat, J. Fabre, M. Rodriguez, and F. Salles. Dependability of COTS microkernel based systems. TC, 51(2), Feb. (2002).
5. 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).
6. S. Beatty. Sensible software testing. http://www.embedded.com/2000/0008/0008feat3.htm, (2000).
7. G. Behrmann, A. David, and K. G. Larson. A tutorial on UPPAAL. http://www.uppaal.com, 2004.

8. J. P. Bodeveix, R. Bouaziz, and O. Kone. Test method for embedded real-time systems. In WSIDES, (2005).
9. R. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Softw., 8(2),* 66-74, Mar. (1991).
10. 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-justheadaches.ars, (2008).
11. J. Chen and S. MacDonald. Testing concurrent programs using value schedules. In ASE, (2007).
12. Deviceguru.com. Over 4 Billion Embedded Devices Ship Annually. http://deviceguru.com/over-4-billion-embeddeddevices-shipped-last-year/, Jan. (2008).
13. H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques:. *ESEJ, 10(4),* 405-435, (2005).
14. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wpmethod: Testing real-time systems. TSE, *28(11),* 1203-1238 (2002).
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. Offut. 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, 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. 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. IEEESW, *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.