Task Dependency Analysis for Regression Test Selection of Embedded Programs

Swarnendu Biswas, *Graduate Student Member, IEEE*, Rajib Mall, *Senior Member, IEEE*, and Manoranjan Satpathy

***Abstract*—Execution dependencies arise among the tasks of an embedded program due to issues such as task priority, task precedence, and intertask communication. We argue that execution dependencies among tasks need to be suitably considered in various embedded software engineering activities such as debugging, regression testing, and computation of complexity metrics. In this letter, we discuss how task execution dependencies among real-time tasks can be identified from static code analysis. Subsequently, we briefly describe an application of our analysis to regression test selection.**

***Index Terms*—Embedded systems, intertask communication, real-time, task execution dependencies.**

I. Introduction

EMBEDDED systems are now extensively being deployed in safety-critical applications such as automotive, avionics, health-care instrumentation, etc. This calls for extremely reliable operation of these applications. Unlike traditional programs, the failures of an embedded program arise from both functional errors as well as timing bugs. Therefore, in addition to functional correctness of an embedded application, it is also necessary to guarantee its temporal correctness. Considerable number of studies on issues such as computation of the worst case execution times (wcet) of tasks and priority inversions arising on account of resource sharing has been reported in the literature. Though research results on such timing analysis of tasks are numerous, no studies on systematic identification of execution dependencies of tasks have been reported [1]. In this context, it is important to note the difference between task timing analysis and execution dependency analysis. While timing analysis deals with prediction of wcet for tasks, the aim of execution dependency analysis is to identify all those tasks in an application which can affect the timing behavior of a given task.

Besides control and data dependencies, additional dependencies arise among tasks due to precedence relations, task priorities, and intertask communications [1], [2]. We call these *task execution* dependencies. Upon a code change, task execution dependencies can cause delays to the completion times of the dependent tasks or cause altered task execution sequences. Thus, it is a major challenge for software developers to ensure that the performance constraints of real-time tasks are still satisfied after minor modifications are made to an embedded program.

In this letter, we argue that systematic identification of execution dependencies among tasks can help in many software engineering activities of embedded programs such as regression test selection (RTS), task prioritization, debugging of timing errors, computation of complexity metrics, etc. For example in RTS, in addition to testing for traditional regression errors induced due to data and control dependencies after a change to an embedded program, it is imperative to test whether any timing errors have been induced. As we show later, this can be achieved by using task execution dependency analysis. Similarly, while *debugging* a timing fault, it becomes necessary to first identify all those tasks that might have contributed to the unexpected timing behavior. The information about the number of tasks that are execution dependent on a particular task can also be used to *prioritize* testing effort.

The rest of this letter is organized as follows: In Section II, we briefly discuss the different types of execution dependencies that may exist among the tasks. We then propose a method to identify task execution dependencies systematically in Section III. Subsequently, we discuss an application of task execution dependency analysis technique in RTS in Section IV. In Section V, we compare our approach with related work, and finally conclude the letter in Section VI.

II. Execution Dependencies Among Tasks

In this section, we first present our assumptions about the underlying task model and then analyze the different types of execution dependence relationships that can arise among tasks in an embedded program.

OSEK/VDX and POSIX RT are two real-time operating system standards that are popularly being used in the development of embedded applications. In this letter, we consider a task model that incorporates features from both these standards. In the following, we list various assumptions that we have made related to the task model:

- the tasks are statically created and are assigned static priority values;
- the tasks are periodic in nature and are scheduled using a priority-driven preemptive task scheduler;
- the tasks communicate using either shared memory or message passing primitives. Furthermore, any access to a shared variable is permitted through the use of some synchronization primitives such as semaphore, lock, etc. We consider only the synchronous message passing scheme, since to achieve predictable results, embedded application developers usually restrict themselves to using only the synchronous message passing model.

Manuscript received June 09, 2011; accepted September 21, 2011. Date of publication October 24, 2011; date of current version December 21, 2011. This manuscript was recommended for publication by P. Chou.

S. Biswas and R. Mall are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, 721302, India (e-mail: swarnendu@cse.iitkgp.ernet.in; rajib@cse.iitkgp.ernet.in).

M. Satpathy is with the GM India Science Lab, Bangalore, India (e-mail: manoranjan.satpathy@gm.com).

Digital Object Identifier 10.1109/LES.2011.2173293

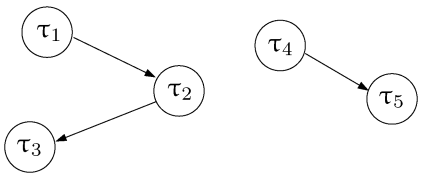

Fig. 1. Representation of precedence relations among tasks.

Our task model reflects the assumptions frequently made in the development of small embedded applications. For example, an adaptive cruise controller (ACC) module in an automotive application is usually implemented with about a dozen periodic real-time tasks with statically assigned priorities. The tasks are scheduled using the rate monotonic scheduling algorithm. Some of the important concurrently executing tasks in a typical ACC implementation include controlling the host vehicle speed (HVSM task), and processing the radar information (RIP task).

Task Execution Dependency Due to Precedences: A precedence relation between two tasks arises when one task is dependent on some actions or results produced by the other task. Precedence relationship defines a partial order among tasks. In Fig. 1, the tasks have been represented using circles and the precedence relations among them are represented using directed edges. A directed edge from a task τ_i to task τ_j indicates that τ_j is dependent on τ_i. The example in Fig. 1 depicts that τ_1 and τ_4 precede τ_2 and τ_5 respectively. However, no precedence ordering can be ascribed between the tasks τ_1 and τ_4 or the tasks τ_1 and τ_5. Let pred(τ_i) be the set of all those tasks that need to be executed before execution of task τ_i, and succ(τ_i) be the set of tasks that can be executed only after the task τ_i has completed execution. According to Fig. 1, succ$(\tau_1) = \{\tau_2, \tau_3\}$, and pred$(\tau_5) = \{\tau_4\}$.

Given a set of time-constrained tasks, the completion time of a task can depend on its precedence ordering with the other tasks. A task τ_i cannot execute unless the set of tasks in pred(τ_i) have already completed their execution. In this case, any delay to the completion time of a task τ_i can affect the completion time of the tasks in the set succ(τ_i). Execution dependencies arising among tasks due to their precedence ordering are transitive in nature.

Task Execution Dependency Due to Priorities: Execution dependencies among tasks can implicitly arise due to task priorities because a delay to the completion time of a higher priority task may delay a lower priority task. Execution dependencies among tasks arising due to priorities are transitive. For a given task τ_i, we denote the set of tasks whose execution times can potentially be affected by τ_i on priority considerations by prior(τ_i).

Task Execution Dependency Due to Message Passing: Synchronous message passing gives rise to execution dependencies among the communicating tasks in an embedded program. When two tasks communicate using a message queue, a delay to one of the tasks can delay the other. Task execution dependencies arising due to message passing are both symmetric and transitive in nature under the synchronous message passing model as both the sender and the receiver tasks can delay each other. For a task τ_i, we denote the set of tasks that can possibly get delayed by it due to messaging passing by ITC$_{\rm mp}(\tau_i)$.

Task Execution Dependency Due to the Access of Shared Resources: Our task model assumes that access to shared resources is guarded using synchronization primitives such as semaphores or locks. In this case, a task locking a synchronization variable for an unusually long duration may delay other tasks sharing the same variable. Task execution dependencies arising due to access to shared resources are transitive and symmetric. For a task τ_i, we denote the set of tasks that are execution dependent on it due to access to shared resources by ITC$_{\rm syn}(\tau_i)$.

Algorithm 1 Pseudocode for identifying execution-dependent tasks.

1: **procedure** IDENTIFYTD $input$
▷ $input$ = Input embedded program
▷ **Output**: set of tasks that are execution dependent on each task τ_i in $input$
2: succ(τ_i) $\leftarrow NULL$
3: prior(τ_i) $\leftarrow NULL$
4: ITC$_{\rm mp}(\tau_i) \leftarrow NULL$
5: ITC$_{\rm syn}(\tau_i) \leftarrow NULL$
6: **for** each task τ_i modified in $input$ **do** ▷ Compute $succ(\tau_i)$
7: **for** every occurrence of a join()/wait() primitive in $input$ **do**
8: Determine whether any task τ_j is dependent on τ_i due to precedence
9: succ(τ_i) = (τ_i) $\cup\, \tau_j$ ▷ Add all τ_j to succ(τ_i)
10: **end for**
▷ Scan $input$ and compare priorities of the other tasks with that of τ_i
11: Determine prior(τ_i)
▷ Scan $input$ to find out matching pairs of message queues and synchronization variables
12: Determine ITC$_{\rm mp}(\tau_i)$ and ITC$_{\rm syn}(\tau_i)$
13: **end for**
14: ***end procedure***

III. Identification of Task Execution Dependencies

Considering all possible types of task execution dependencies, the set of tasks that are execution dependent on task τ_i [denoted by $TD(\tau_i)$] is

$$\text{TD}(\tau_i) = \text{succ}(\tau_i) \cup \text{prior}(\tau_i) \cup ITC_{\rm mp}(\tau_i) \cup ITC_{\rm syn}(\tau_i).$$

Note that this execution dependency result is not applicable to simple embedded operating systems using clock-driven schedulers such as cyclic schedulers. In such an environment, task scheduling takes place solely based on timer interrupts; event-driven task scheduling, task preemption, synchronization, etc., are not supported.

Execution dependencies existing among a set of tasks in an embedded application can automatically be computed by performing static analysis on the source code. The pseudocode for identification of all the tasks which are execution dependent on a

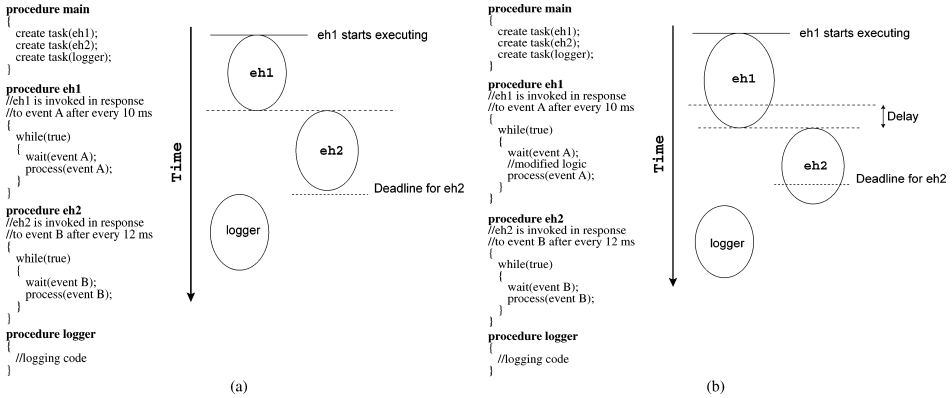

Fig. 2. Example of a regression error introduced due to task execution dependency. (a) Original program P; (b) modified program P'.

task τ_i is shown in Algorithm 1. Algorithm 1 takes an embedded program as input, and computes the set of execution dependent tasks for each task in the program.

Task execution dependency information can be represented using either a task dependency graph [2], or a matrix. The dependency information can also be modeled using petri nets where the execution dependencies are represented using passing of tokens among places which denote the tasks of an embedded program. In the following, we present an example representation using an $n \times n$ *task execution dependency* matrix which we denote by TEDM. An element TEDM$_{ij}$ in the matrix is set to 1 if task τ_j is execution dependent on task τ_i, otherwise TEDM$_{ij}$ is 0. Let the *task execution dependency* matrix for an application having four tasks be as follows:

$$\text{TEDM} = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

This matrix represents that the execution of of tasks τ_3 and τ_4 are dependent on task τ_1. Further, task τ_3 is also execution dependent on the task τ_2.

IV. Application of Task Execution Dependence Analysis to RTS

RTS concerns selection of a subset of valid test cases from an initial test suite that tests the affected but unmodified parts of a program. Use of an effective RTS technique can help to substantially reduce the testing costs in environments in which a program undergoes frequent modifications. Many RTS techniques have been reported in the literature for procedural, object-oriented, and component-based programs [3], [4], but RTS techniques have scarcely been reported in the context of embedded programs.

Procedural RTS techniques usually select regression test cases based on data and control dependency analysis. Therefore, these techniques may not be effective for RTS of embedded programs as they ignore execution dependencies among tasks. This could be the reason why in industry, regression test cases for embedded programs are either selected *ad-hoc* or based on expert judgment or through some form of manual program analysis. In the following, we give an example to show that RTS using existing procedural techniques can be *unsafe* [4] for applications with time-constrained taks.

Example 1: Fig. 2(a) shows the pseudocode for an embedded program P which is composed of three tasks $eh1$, $eh2$ and $logger$. We make the following assumptions: 1) $eh1$ and $eh2$ are invoked in response to events A and B after every 10 ms and 12 ms, respectively; 2) $eh1$ is of higher priority than $eh2$; 3) there is a deadline by which event B needs to be handled by $eh2$ after it is invoked; and 4) there are no data or control dependencies among the program statements in $eh1$ and $eh2$. Note that $logger$ is an auxiliary task and is not dependent on the execution of $eh1$ and $eh2$. Suppose the event handling logic in $eh1$ is changed in P' as shown in Fig. 2(b), and as a result, $eh1$ takes longer to complete after the modification. Since $eh1$ is of higher priority, $eh2$ cannot start until $eh1$ is complete, i.e., task $eh2$ is execution dependent on task $eh1$. Therefore, $eh2$ would get delayed and miss its deadline. Consider a test case t which tests only the event handling logic in task $eh2$. Existing RTS techniques [3], [4] would not select t for regression testing of P' as there exists no data or control dependencies between $eh1$ and $eh2$, and would hence be unsafe.

The drawback of a traditional RTS technique illustrated by the above example can be overcome if it is augmented with task execution dependency information. Whenever the code for one task is changed, it is necessary to test those tasks whose completion times can get affected.

In the following, we outline how regression test cases for an embedded application can be selected based on identification of task execution dependencies. Let us consider an embedded program P consisting of n statically created tasks. Also assume that m out of n tasks in P are modified to meet certain change requirements. After the changes are complete, the modified version of P needs to be regression tested. Let $\Gamma = \{\tau_1 \ldots \tau_m\}$ denote the set of m tasks which have been changed. First, the $TEDM$ matrix for P is computed by computing $TD(\tau_i)$ for each task $\tau_i \in \Gamma$ using a static analysis of the source code. The set of all tasks which have been potentially affected due to task execution dependencies is given by $\bigcup_{\tau_i \in \Gamma} TD(\tau_i)$. The information regarding which test cases execute which tasks in P can be captured during the previous testing cycles. The test cases which execute the tasks in $\bigcup_{\tau_i \in \Gamma} TD(\tau_i)$ need to be selected for regression testing P. Note that regression test cases selected based on task execution dependency considerations serve only

to augment the regression test suite selected using traditional data and control dependency analysis and do not replace it.

We have implemented a prototype tool [5] to study the effectiveness of an RTS technique that has been augmented with our proposed task execution dependency analysis. In our empirical studies, we have considered eight applications from the automotive domain which were developed in C. These include applications such as ACC, Power window controller, and Climate controller. The modified versions were created by systematically adding, modifying, or deleting one or more lines of code from the original versions of the eight programs under test (PUT). We also designed test cases for each PUT to test the functional and temporal correctness of the programs. The functional test cases were designed using blackbox techniques of category partitioning and boundary value analysis, and performance test cases were designed to check whether the timing constraints of tasks are met.

We have implemented Binkley's RTS approach [3] augmented with task execution dependency analysis for experimental evaluation of our approach. We have named our tool ARTS which stands for automated regression test selector. We selected regression test cases using both Binkley's approach and ARTS. We observed that ARTS on the average selected 51.87% of the total test cases from the initial test suite, while only 44.78% test cases were selected on the average using Binkley's approach. Therefore, there was an increase of 15.83% in the number of test cases that were selected by ARTS over Binkley's approach. This increase was expected, since additional test cases get selected based on task execution dependency analysis. In fact, the test cases selected by ARTS is a superset of the test cases selected by Binkley's technique which selects test cases based on only data and control dependencies.

We have defined a metric called *fault-revealing* effectiveness as a measure of the quality of the selected regression test suites. The fault-revealing effectiveness metric can be computed by computing the percentage of test cases selected by an RTS technique from the set of test cases that fail when the valid test cases in the initial test suite are run with the modified program. That is, the fault-revealing effectiveness of the test suite selected by a safe [4] RTS technique is equal to that of the initial test suite. To compute the fault-revealing effectiveness of the regression test suite selected by ARTS, we also ran the complete initial test suite to compute the number of test cases that failed with the modified PUTs. We observed that all the test cases of the initial test suite that failed on the modified versions of the PUTs were included in the test suite selected by our ARTS technique. The fault-revealing effectiveness of the test suite selected using our ARTS approach was 100%, and was 32.08% higher on the average than that of Binkley's approach.

We now give an example to highlight the type of test cases which were omitted by Binkley's approach. In a typical ACC implementation, the HVSM task is execution dependent on the RIP task due to precedence ordering. In the modified version of the ACC program, the RIP task was modified which subsequently delayed the completion of the HVSM task causing the HVSM task to timeout. For such a modification, Binkley's approach failed to select test cases that tested the performance constraints of the HVSM task because there were no data and control dependencies between the RIP and HVSM tasks. The results of our limited experiments confirm our claim that RTS based on control and data dependence analysis augmented with task execution dependency analysis is safe.

V. Comparison With Related Work

Research results on selecting regression test cases for embedded applications have scarcely been reported in the literature. Cartaxo *et al.* [6] have proposed a technique to select functional test cases for embedded applications. Given the initial test suite, their technique aims to *minimize* the test suite while still achieving a desired feature coverage. However, their technique [6] does not consider regression testing of timing-related errors. In [7], Netkow and Brylow have proposed a framework called Xest for automating execution of regression test cases in a test-driven development environment. Their test setup helps to automatically execute regression test cases for kernel development projects on embedded hardware. However, their work does not address the problem of RTS, and is therefore, not directly related to our work.

VI. Conclusion

Execution dependencies arise among tasks of an embedded program on account of task precedence ordering, task priorities, and intertask communication. We have proposed an approach to identify task execution dependencies through static code analysis. We have discussed an application of task execution dependency analysis to RTS. Our empirical studies showed an increase in the number of selected regression test cases by approximately 15.83%. On the other hand, we observed an increase of 32.08% in the number of fault-revealing test cases that were selected for regression testing. We observed that our augmented RTS technique did not miss out on selecting any fault-revealing test case for regression testing. The results of our experiments highlight the necessity of using task execution dependence analysis for RTS of embedded programs. We plan to use our task execution dependency analysis technique to develop an effective debugger for embedded programs.

References

[1] D. Sundmark, A. Pettersson, and H. Thane, "Regression testing of multi-tasking real-time systems: A problem statement," *ACM SIGBED Rev.*, vol. 2, no. 2, pp. 31–34, Apr. 2005.

[2] P. Marwedel, *Embedded System Design*. Berlin, Germany: Springer-Verlag, 2007.

[3] D. Binkley, "Semantics guided regression test cost reduction," *IEEE Trans. Software Eng.*, vol. 23, no. 8, pp. 498–516, Aug. 1997.

[4] G. Rothermel and M. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Software Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[5] S. Biswas, "Model-Based Regression Test Selection and Optimization for Embedded Programs," Master's thesis, Indian Inst. Technol., Kharagpur, India, Jun. 2011.

[6] E. Cartaxo, W. Andrade, F. Neto, and P. Machado, "LTS-BT: A tool to generate and select functional test cases for embedded systems," in *SAC '08: Proc. 2008 ACM Symp. Appl. Comput.*, 2008, pp. 1540–1544.

[7] M. Netkow and D. Brylow, "Xest: An automated framework for regression testing of embedded software," in *Proc. 2010 Workshop Embed. Syst. Edu.*, Oct. 2010, pp. 7:1–7:8, ser. WESE '10. ACM.