On Real-Time Software Testing and Debugging

Jeffrey J. P. Tsai, Kang-Ya Fang, and Yao-Dong Bi
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, Illinois 60680

ABSTRACT

Testing and debugging of real-time software systems are difficult because of timing constraints imposed on them and because of their non-deterministic behavior. Conventional static analysis is not adequate to deal with the violation of timing constraints which are inherent in real-time software systems. This paper presents a dynamic analysis method which uses the recorded run-time information to test and debug this kind of timing errors. Based on our approach, a non-interference monitoring system architecture has been developed to collect the process-level program execution data of a target real-time software system without affecting its execution. Different-leveled logical views are then reconstructed from these collected run-time information. A dynamic analysis method is then presented to analyze timing behavior of real-time software systems.

1 INTRODUCTION

Testing and debugging of real-time software are very difficult because of timing constraints imposed on them and because of their non-deterministic execution behavior. In a real-time system, the processes receive inputs from real world processes as a result of asynchronous interrupts. In consequence, it is almost impossible to precisely predicate the exact program execution points at which the inputs will be supplied to the system. Due to the unpredictable sequence in which the inputs arrive at certain points in time, the system may not exhibit the same behavior upon repeated execution of the program. In addition, in a real-time system, the pace of execution of processes is determined not only by internal criteria, e.g., the execution speed of the underlying processor, but also by the timing constraints imposed on real world processes. A proper operation of the whole system—consisting of the real-time target processes and the real world processes—depends on the capability of the target processes to comply with certain timing constraints [1]. Thus one of the difficult problems specially for real-time systems is to test and debug violations of the timing constraints imposed on them.

In the past few years, intensive studies have been actively carried out on the testing and debugging methods of distributed and/or parallel systems. Generally, those methods can be classified into static and dynamic analysis methods. The static analysis methods analyze the program source codes or its abstractions derived from the source codes to detect anomalies of the concurrent programs [2, 3, 4, 5]. Taylor's method is a typical example [2]. He described a static analysis method for detecting anomalies in concurrent programs based on the interaction activities among processes in a program. However, since static analysis methods have their inherent limitations for analyzing software, especially, their incapability of dealing with asynchronous interactions between real-time target processes and real world processes, as well as modeling the timing behavior of hardware implementation, scheduler algorithm, and other environmental activities, they have limitations in analyzing real-time systems. On the other hand, the dynamic analysis methods can use program's run-time information to analyze the program's execution behavior. Most dynamic methods [2, 3, 4, 5] insert codes into a target program or into a process's program or an operating system to collect the run-time information or to control the execution behavior of the target program, in which the timing behavior of the target system is interfered. This interference is prohibited in real-time systems since an error may not manifest due to the change of the timing behavior of the program. These dynamic methods are not suitable for real-time system testing and debugging either.

Using graphs to do the timing analysis has been studied in the VLSI/CAD and in software performance measurement. Timing analysis in VLSI/CAD using graph [10, 11, 12, 13] is to find out the delay time of a circuit from its inputs to the outputs. To do that a circuit is modeled as a process-activity graph, then based on this graph, the critical path analysis method is used to find the longest sensitizable path in the logic network. The difficulty to do that is that the general critical path analysis methods often find an overestimated path which is not sensitizable because of the logic constraints of the circuit. Another issue is how accurate the delay times of the components of the circuit is estimated by the designer. Thus the main concern of timing analysis is to find the longest sensitizable path with the logic constraints and to find an accurate timing model of a circuit. In software performance measurement [14, 15], the execution history of a program is represented as a program activity graph, then based on this graph, the critical path analysis method is used to find a event path which has the longest duration. This path is then used as a guidance for the software developer to find the problems of the program or improve the program efficiency. This method is mainly for parallel and distributed systems, and the main concern of such a technique in distributed systems is to find out an algorithm to find the longest path efficiently. The method to obtain the execution history is not mentioned in [14].

Hsieh [16] provides an approach to timing analysis of cyclic concurrent programs in which GRO path—expressions are used to describe synchronization and concurrency of atomic operations. The behavior of program is represented as a partial order of atomic operations which then is represented as a graph. The timing analysis is then performed on this graph using critical path analysis method. His main concern is to determine whether a program can satisfy a given set of timing constraints. But he did not mention how to compute the execution time of an atomic operation. Another problem is that the implicit assumption that an atomic operation can run whenever it is ready to run is not always true, since the processor may not be available when the operation is ready to run.

Harter [1] proposed another timing analysis approach for level-structured systems. He assumed that the target system employs multiple priorities and is level structured, and that the number of processes and the priorities of the processes in the system are known. The response time of a process is computed by taking the interruptions of the processes at higher priority levels into consideration. In other words, the execution time of the processes at higher priority levels is considered as a net increase in the execution times of the processes at lower priority levels. Since the priorities of the processes and the number of processes in the system must be known in prior, this approach may not suitable to dynamic scheduling system since the priority of a process may vary in time. The author did not mention how to find the execution time of a process in the absence of the higher priority level processes.

This paper presents a dynamic analysis method which uses the run-time information of program. The run-time information is collected in a way such that the execution behavior of the tar-

---

1This research was in part supported by National Science Foundation under grants CCR-8708939 and CCR-8809581, and an award from the IEEE and the Engineering Foundation Society.
The main objective of the non-interference real-time monitoring system is to provide a testing and debugging environment with maximum control over program execution to the software developers. The key is to get program execution history data from the target system for software testing and debugging without interfering with the behavior of the target system. Equipped with a qualification control logic to sample the state of the target processor via its address, data, and control buses during every processor cycle, a developer-selected state or breakpoint is used as a trigger to initiate and stop recording of the execution history.

The highlighted capacities of the non-interference real-time software testing system include: (a) Non-interference software trace and monitoring, (b) User control over the selection of the initial points for the execution of a target program, and (c) Real-time simulation with dynamic control of program execution speed.

Functionally, the non-interference approach in software testing and debugging are divided into two phases: pre-processing and post-processing. The pre-processing is defined as: (a) Providing physical connections without electronic obstruction, (b) Loading conditions for start and end of recording, (c) Initializing interface Module, (d) Recording information, and (e) Transferring recorded information to secondary storage for post-processing.

In the post-processing phase, the data recorded in the pre-processing phase is reconfigured into meaningful information for software testing and debugging. Please refer to [17, 18] for detail on the monitoring system architecture and post-processing mechanisms of our system.
and the time the process was created. Event Chain (EC) describes the events happened in the system. Each node of the EC, corresponding to an event, contains the key values of the events as defined in the above section and two pointers. One pointer is used to link all the events in time sequence (time-link). The other pointer is used to link all the events related to one process (process-link) in time sequence. Each process entry in the PCT is linked to the first node of the Event Chain related to that process.

From IFPEL, we can derive six different process-level logical views, (1) Process Graph, (2) Process Precedence Graph, (3) Process State Transition Sequence, (4) P/V Operation Sequence on the semaphore, (5) Send/Receive Operation Sequence for Interprocess Direct Communication, and (6) Mailbox-driven Operation Sequence for Interprocess Indirect Communication. In this section, we will explain how useful these logical views to software testing and debugging. The algorithms for deriving these logical views from the intermediate data set are also given.

The process graph shows all processes running in the system and their parent-child relations. The process precedence graph gives a picture to show the precedence relations among the processes, such as fork/join. By providing with the time a process created and terminated and the process terminating states (normal, abnormal, killed), these two graphs can be used by the software developers to infer some clues about which process may be responsible for the fault or is suspicious by comparing the two graphs to the expected process graph and precedence relations. The process state transition sequences give another kind of information, state transitions of each process. The software developers can use these sequences to check the staying period for a process at each state and may ask questions like, why a process stayed in that state for such a long time? or why that process changed state at that time? The software developers may find the answers for these questions by checking the rest of the logical views.

One of the timing constraints in real-time systems is in the process synchronization. Processes in the system are synchronized to enter critical sections or to compete for system resources. The semaphore is a common mechanism for synchronizing processes. The correct use of the semaphore is the responsibility of each individual process. The programmer must be completely aware of the information shared among processes. The operating system provides its services only by executing P/V primitive commands whenever they are invoked. Although there exist some models of process synchronization for typical problems, such as producer/consumer and reader/writer problems, the process synchronization is still an error-prone problem in real-time software systems. In real-time software systems, because of the time constraints, the synchronization problems are very hard to be checked by only examining the program code. To discover process synchronization errors, the execution behavior of the synchronized processes is needed. In general, the P/V operation sequence produced in the system reflects the synchronizing behavior of the processes. Combined with the time the operation is taken in the sequence, the software developer can check the P/V operations taken by a process at the time to identify a faulty process. For example, if a process executes a P operation without executing V operation when leaving a critical section, another process might be in the "waiting" state forever.

A common approach in debugging of infrequent and unpredictable problems which do not have any obvious cause is to collect traces of interprocess messages and examine them to see if any abnormality can be detected [18]. The send/receive operation sequences for direct communication and the mailbox-driven operation sequences for indirect communication provide useful information for this kind approach. The software developers can check the messages and time in the sequences to identify a failed process and processes, failed messages and to localize the fault within a process. The Message Trace Analyzer [19] is an example of knowledge-based debugging systems using interprocess messages to localize faults to within a process.

### 3.2.1 Process Graph

The Process Graph (PG) is a directed rooted tree, whose nodes corresponding to processes containing the process identification and the time that the process is created. An edge from node $P_i$ to node $P_j$ means that $P_i$ creates $P_j$, that is, $P_i$ is the parent of $P_j$ (Figure 2). A process graph gives a graphical representation of the hierarchical relationship of all the processes running in the system. This graph can easily be derived from the PCT to show the number of processes in the system during monitoring, the time that each process is created, and the parent-child relationship among processes.

The Process Graph can be created using the following algorithm.

**Algorithm PG:** This algorithm constructs the Process Graph (PG). It is derived from the PCT.

1. **ST1:** Create the root of PG with values of process-id and time of the first entry of PCT; $i := 2$.
2. **Repeat** the following steps until $i > N$ ($N$ is the number of processes):
   1. **ST2:** Search parent-process-id of $i^{th}$ entry of PCT on the PG.
   2. **Link** node with the process-id and time of $i^{th}$ entry as child of the parent-process-id node.
   3. **ST3:** $i := i + 1$ (Link next process to the PG).

**End of Algorithm PG.**

---

### Table 2. Process-Level Event and Key Values

<table>
<thead>
<tr>
<th>Event Code</th>
<th>Event</th>
<th>Key Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Process Creation</td>
<td>Parent Process Identification, Child Process Identification, Time</td>
</tr>
<tr>
<td>P2</td>
<td>Process Termination</td>
<td>Parent Process Identification, Terminating State, Time</td>
</tr>
<tr>
<td>P3</td>
<td>Process Synchronization Operation (PV)</td>
<td>Process Identification, Value of the Semaphore, Time</td>
</tr>
<tr>
<td>P4</td>
<td>Direct Communication</td>
<td>Sending Process Identification, Receiving Process Identification, Operation (Send/Receive), Time</td>
</tr>
<tr>
<td>P5</td>
<td>Indirect Communication</td>
<td>Process Identification, Operation (Indirectly), Message, Time</td>
</tr>
</tbody>
</table>

---

### 3.2.2 Process Precedence Graph

The Process Precedence Graph (PPG) is a directed acyclic graph whose nodes correspond to processes. If a node has two non-null out-edges, the node denotes the creation of a new process, and the out-edges from a node point to the parent process and the newly created process. The node contains the parent process identification and the creation time of the parent process. If a node has two non-null in-edges, it represents a termination of a process and contains the parent process identification and the termination time. The two in-edges represent the parent process and the terminating process. This graph (Figure 3), which can be derived from the PCT and the EC, provides a picture showing the process creation and termination events, such as fork and join, as well as the relative time when these events have occurred.

This graph can be created using the following algorithm.

**Algorithm PPG:** This algorithm is to construct the Process Precedence Graph (PPG) using the PCT and EC.

1. **ST1:** Set the root of PPG with value: p-id and time of PCT(1);
2. **ST2:** Copy the root and link it as the left child of the root;
3. **ST3:** Link a node with value: p-id and time of PCT(2)
as the right child of the root.
ST3: Search EC on time-link for first terminating process node as NODE;
i = i + 1
ST4: Repeat ST5 until meeting the tail of time-link of EC.
ST5: Repeat this step until i = n
IF time of PCT(i)=NODE(time) THEN
Find NODE(pp-id) and NODE(p-id) nodes on the PPG;
Create a node as the son of NODE(pp-id) and NODE(p-id) with value: p-id and time;
Search on EC for next terminating process node as NODE
ELSE
Find the leaf node on PPG with pp-id of PCT(i);
Copy it as its left child;
Link a node with value: p-id and time of PCT(i) as right child;
i = i + 1
End of Algorithm PPG.

3.2.3 Process State Transition Sequence
The Process State Transition Sequence (PSTS) (Figure 4) for a process is a linked list whose nodes correspond to process states in the time sequence order. This sequence, which can be constructed from the EC by searching on the process-link, shows the process state changes together with the reason which causes the state transition and time when the transition occurs.
This sequence can be created using the following algorithm.
Algorithm PSTS: This algorithm is to construct the Process State Transition Sequence (PSTS). PSTS is the PSTS of the process which is the nth entry in the PCT.
ST1: i = 1; Repeat following steps until i = N
(N is the number of processes).
ST2: node := PCT(i) (the first event node in the EC); Repeat until the tail of the process-link of PCT(i).
IF node(code)="E3" THEN
Link this node to the tail of PSTS(i).
node := node(process-link).
ST4: i := i + 1. (To construct PSTS for next process)
End of Algorithm PSTS.

4 FUNCTION-LEVEL MONITORING AND POST-PROCESSING
The collected process-level information can be used to analyze the behavior of the program at process-level. But for repairing the faults, being at process-level is still too high for a software developer since a process may still be too large to handle. Thus monitoring the system at another more detailed abstraction level, that is, function-level abstraction, is needed. After a set of faulty processes is identified, we monitor those processes at function-level to provide information for identifying faulty functions which caused those processes failure.

3.2.4 P/V Operation Sequence on Semaphore
The P/V Operation Sequence (PVOS) (Figure 5) is a linked list in the time sequence order whose nodes correspond to the P's or V's operations on the semaphore. This sequence, which can be easily derived from the EC by searching the time-link, provides a complete picture of process events on the semaphore. This picture gives a detail semaphore operations to show how a process executes what operation and what is the contents of the semaphore. Software developers can use this information to verify process synchronization and process communication.
This sequence can be created using the following algorithm.
Algorithm PVOS: This algorithm constructs the P/V Operation Sequence (PVOS) on a semaphore.

3.2.5 Send/Receive Operation Sequence for Direct Communication
The Send/Receive Operation Sequence (SROS) is a linked list whose nodes correspond to the Send or Receive operations (Figure 6) in time sequence for direct interprocess communication. This sequence provides the message passing information between processes, so that the software developers can verify at what time which process sends what message to or receives what message from which process.
This sequence can be easily derived from the EC using the following algorithm.
Algorithm SROS: This algorithm constructs the Send/Receive Operation Sequence (SROS) for direct communication.
ST1: node := head of the EC.
ST2: Repeat this step until meeting the tail of the time-link.
IF node(code)="E6" THEN
Search on process-link for the process-id in the PCT;
Link this node with process-id to the tail of SROS.
ST3: node := (node(time-link)).
End of Algorithm SROS.

3.2.6 Mailbox-Driven Operation Sequence for Indirect Communication
The Mailbox-driven Operation Sequence (MDOS) (Figure 7) for a mailbox is a linked list whose nodes correspond to send or receive operations on the mailbox in time sequence. This sequence gives a picture showing when a process took what operation (send/receive) on this mailbox for what information. Software developers can use this information to verify the interprocess communication.
This sequence can be derived from the EC using the following algorithm.
Algorithm MDOS: This algorithm constructs the send/receive operation sequence.
ST1: node := head of the EC.
ST2: Repeat this step until meeting the tail of the time-link.
IF node(code)="E6" AND node(mailbox-id)=M-id THEN
Search on the process-link for the process-id in the PCT;
Link this node with process-id to the tail of MDOS.
ST3: node := (node(time-link)).
End of Algorithm MDOS.
key values shown in Table 3 as the information for function-level monitoring and analysis. The key value, time is the time at which the calling or returning event happens.

4.2 Function-Level Monitoring

Before we discuss the monitoring methods for that information, we restrict the programming language in which the target program is written. The programming language is block structured, and a block can be a procedure or a function. A scope of variable is determined statically. A process consists of a set of functions. The main program of a process is also a function. A function can activate other functions by a calling statement. A function terminates by executing a return statement to return the control to the calling function. A function and a called function communicate via passing parameters. Parameters can be passed by value, by result, or by value-result.

To collect the information identified above using our monitoring system, we must know the implementation of block-structured language since it is implementation dependent, we give a brief description of the implementation first.

An activation record for a procedure P might contain:

1. a dynamic link (pointer to the activation record of the caller of P),
2. a static link (pointer to the activation record of the most recent activation of function immediately containing P),
3. the return address (in the caller of P),
4. the parameters passed to P by the caller (copy of actual parameters),
5. the local variables for P,
6. a register save area and area for temporary variables.

Local variables are referenced as offsets from the top of the stack. In a static block-structured language, binding of names to declaration is based on the physical placement of blocks in the program text, not on the order of calls. The dynamic link is used for this purpose.

To call a function, the calling function performs the following steps (here we use "called" to represent the one which is called):

1. allocate part of the called’s activation record including storage for the parameters, the values of the called, the saved area, dynamic and static links,
2. set dynamic link,
3. set static link,
4. set return address,
5. transmit the parameters,
6. push the new activation record into run-time stack,
7. transfer control to the called.

From this procedure, we know that all information identified above for the function calling event is contained in it. If we latch the data on the target system buses during the execution of that procedure, the key values for function calling can be filtered from the latched data. For most computer systems, the step 7 is carried out by a pre-designed instruction, call instruction.

Suppose the above steps can be accomplished with the maximum of N1 instruction cycles of the target system. To latch these data, the system pre-latches data from the buses for N1 cycles. The latching continues in first-in-first-out manner to keep data with length of N1 cycles. When the execution of call instruction is detected by the interface module of the monitoring system, the pre-latched data are saved to the memory of the development module. Thus the information for function calling is collected.

Returning a call to the calling function, the called function performs the following steps:

1. store return values if any in its activation record,
2. restore the run-time stack as before it was called,
3. restore the registers and the status information as before it is called,
4. return control to the calling function.

Same as function calling, the key values for the function returning event are contained in that procedure, and the step 4 is also carried out by a pre-designed instruction, return instruction.

Suppose these steps can be accomplished with the maximum of N2 instruction cycles on the target machine. We can use the same method as for the function calling. The monitoring system pre-latches data for N2 cycles instead of N1 cycles. When the execution of a return instruction is detected, the pre-latched data are saved to the memory of the development module.

From above discussion, we summarize the function-level program execution monitoring as follows:

1. set the pre-latched data length as MAX(N1, N2),
2. save the pre-latched data with length N1, when the call instruction’s execution is detected,
3. save the pre-latched data with length N2, when the return instruction’s execution is detected.

4.3 Function-Level Post-Processing

The collected data from the target system, Function-level Program Execution Log (FPSEL), have the same problems as the PFEL, that is, they are represented at machine-code level, and the interpretation is target processor dependent. Furthermore, the FPSEL contains not only the key values but also other instructions and data. Therefore, it is necessary to filter out non-key value data and re-organize the data.

To achieve this, an intermediate data set, called Integrated Function-level Program Execution Log (IFPSEL), is derived from the FPSEL. The IFPSEL describes the program execution behavior which set limits to the memory of the development module.

The event: IFPSEL describes the program execution behavior which set limits to the memory of the development module. The event: IFPSEL describes the program execution behavior which set limits to the memory of the development module. The event: IFPSEL describes the program execution behavior which set limits to the memory of the development module. The event: IFPSEL describes the program execution behavior which set limits to the memory of the development module.

5 TIMING CONSTRAINT ANALYSIS

One of the correctness of a real-time software system is whether the system can meet the timing constraints imposed by the external process. Timing constraints for tasks in a real-time software system can be very complicated, but the most common timing constraints are either periodic or aperiodic (sporadic). An aperiodic task has a deadline by which it must finish or start, or it may have a constraint on both start and finish time. In case of a periodic task, a period might mean "once per period T" or "exactly T unit apart". Dasarathy [20] categorizes the timing constraints, from a point of view that a timing constraint imposes a temporal restriction on the system or its user, into performance constraints which set limits on the response time of a system and behavioral constraints which make demand on the ratio at which users apply stimulus to the system. He further classifies timing constraints by three of temporal restrictions:
MAXIMUM: no more than \( t \) amount of time may elapse between the occurrence of one event and the occurrence of another; MINIMUM: no less than \( t \) amount of time may elapse between occurrences of two events;

**DURATION**: an event must occur for \( t \) amount of time.

where an event is defined as either a stimulus to the system from its environment, or as an externally observable response that the system makes to its environment. From the above definition, the timing constraints of duration can be eliminated by properly defining the event; separating an event as starting and stopping two events, so the duration can be specified and observed by maximum and minimum constraints.

In the following we present an approach to do the timing analysis for localizing the suspected faulty components in a real-time system. we assume the programming language of the target system has the following features:

1. The execution of a program consists of a set of processes and the program is started by a static process.
2. Processes in the system are synchronized using semaphore.

A semaphore \( S \) is defined as

- **TYPE semaphore**: RECORD;
- **value**: INTEGER;
- **L**: list of processes;
- **END**;

The semaphore operation \( P \) and \( V \) are defined as:

\[
P(S): S.value := S.value - 1;
\]

\[
V(S): S.value := S.value + 1;
\]

if \( S.value < 0 \) THEN BEGIN

- add this process to \( S.L \);
- block;
END;

All semaphores in the system are initialized to 1. That is, only one process can enter the critical section of semaphore \( S \). When a process is in the critical section, all other processes which want to enter the process have to wait in the waiting list of \( S \) until the process in the critical section releases the semaphore.

3. Processes in the system communicate with each other using direct communication mechanism with no buffer. In other words, if a process wants to send or receive a message, it must explicitly identify the partner. In the case of no buffer, the sender must wait until the recipient receives the message. The two processes must be synchronized for a message transfer to take place.

We first introduce Timed Process Interaction Graph (TPIG), which represents the interprocess communications and synchronizations in the program execution. The TPIG is constructed from the logical view of the Process State Transition Sequences (PSTS) in the following way. For each process in the system, draw an Event Flow Graph (EFG) based on its PSTS. Delete all the nodes except those denote one of the following events: 1. the process created or was created by another process, 2. a process sent or received a message, 3. a process woke up and got CPU for running from the release of a semaphore by another process, 4. a process sent a message to or received a message from the external process. Then use directed edges to connect them in time sequence. An edge from node \( n_i \) to \( n_j \) is marked with the process ID, the sum of time the process was in execution state from node \( n_i \) to \( n_j \), and the sum of times the process was in waiting state. After the EFG for each process is constructed, we merge all the EFGs into one single directed graph called Timed Process Interaction Graph (TPIG). The edges of TPIG are the union of all the edges of EFGs. The nodes of TPIG are constructed in the following way. Two nodes, \( N_p \) and \( N_o \), are merged into one node if they meet one of the following conditions: 1. they are associated events that one of the processes created or was created by the other, 2. they are associated events that two processes passed a message, or 3. they are the events that one of them released a semaphore and waked up the other and it got the CPU. Figure 9 shows an example of the Timed Process Interaction Graph. The vertical direction denotes the time and each column is an EFG for a process. The edges connected with a horizontal line denote a merged node of two associated events.

To construct the DTPIG from the TPIG, we define one of the two points as the starting node and the other as ending node of the DTPIG. Then, we cut off all nodes and edges before the starting node in terms of real time. If a process does not have a node at time of starting node, a dummy node is added to it as the first node of that process. For example, in Figure 9, the dotted line shows the real-time point at which \( B \) occurred. Each interaction between the dotted line and a vertical line is a dummy node for that process. We first include the ending node \( E \) in the DTPIG. If an edge is an in-coming edge of a node in the DTPIG, then include this edge and its starting node to the TPIG. Finally, we merge all the nodes with no in-coming edges into one node. Figure 10 shows an example of the DTPIG constructed from Figure 9 for point \( B \) to \( R \).

The Dedicated Timed Process Interaction Graph (DTPIG) constructed in this way can be formally defined as follows. DTPIG is an acyclic directed graph \( G(B, R, N, E) \), where \( B \) is the beginning node without in-going edges, \( R \) is the ending node without out-going edges, \( N \) is a set of nodes denoting the interprocess synchronization events, and \( E \) is a set of edges. Each edge is labeled with the process ID, which is associated with this edge; execution time (ET), the time the process was in
execution; and waiting time (WT), the time the process waited for synchronizing with the other. The DTPIG defined in this way has the following properties. First, each node, except B and R, has at most two in-coming edges. Second, each node, except B and R, has one or two outgoing edges.

Next we define some terms for the convenience of later discussion. MAX-TC and MIN-TC are the maximum and minimum timing constraints from B to R respectively. PET, Execution Time of a path, is the sum of execution time of all edges on the path. CP, Critical Path of a node, is a path from B to the node with no waiting time for all the edges on the path. RT, Response Time, is the maximum execution time of all the paths from B to R. FP, Faulty Path, is a path from B to R whose PET is larger than MAX-TC or less than MIN-TC. The PET of a path is bigger than MAX-TC means that without reducing the execution time on this path the timing constraint can not be satisfied. In other words, there are some faulty components on the path.

The following is an algorithm for timing analysis which will identify all FP's from B to R when the maximum timing constraint is violated. TC here denotes MAX-TC.

Algorithm TIMING-ANALYSIS: This algorithm identifies all the paths whose path execution times (PETS) are bigger than MAX-TC. If a node has only one in-coming edge, then it is treated as left-edge of the node.

ST1: Find the CPs for every node in the DTPIG except node B
ST2: Set PATH to null
ST3: FIND-PATH(EDGE, PATH, PWT).
End of Algorithm TIMING-ANALYSIS

Algorithm FIND-PATH(EDGE, PATH, PWT)

IF (RT-WT(EDGE)-PWT) > TC
PATH := PATH STARTING-Node(EDGE)

PWT := PWT+WT(EDGE)
Print out PATH 
violate timing constraint with time RT-PWT-TC;
S-Node:=Starting-Node(EDGE);
IF Left-Edge(S-Node)< NULL
FIND-PATH(Left-Edge(S-Node), PATH, PWT);
IF Right-Edge(S-Node)< NULL
FIND-PATH(Right-Edge(S-Node), PATH, PWT);
End of Algorithm FIND-PATH.

The first step can employ Prim's minimum spanning tree algorithm (minimum waiting time) with complexity \(O(N^2)\), where \(N\) is the number of nodes in the graph. The second step costs a constant time. The complexity of FIND-PATH is \(O(M)\), where \(M\) is the number of edges in the graph. Since each node in the graph has exactly two in-coming edges, there are at most \(2(N-1)\) edges. So the complexity of FIND-PATH is \(O(N)\). The complexity of TIMING-ANALYSIS is \(O(N^3)\).

For example, in Figure 10 the labels of the DTPIG are shown on the graph. RT=10, MAX-TC=8. The algorithm will identify following faulty paths as shown by bold lines in Figure 10:

1, 6, 7, 10, 11 violates timing constraint with 2,
2, 6, 7, 10, 11 violates timing constraint with 1,
5, 9, 11 violates timing constraint with 1.

The PET of the path [2, 6, 7, 10, 11] is 9. That is, the path [2, 6, 7, 10, 11] is a FP. To satisfy the timing constraint, MAX-TC=8, the PET of this path must be reduced with 1 unit time unit.

6 CONCLUSION AND FUTURE RESEARCH

Testing and debugging real-time software systems are two of the most challenging tasks in software development process. This paper presents a monitoring system to collect run-time information of real-time software systems without interfering with the timing behavior of the target system. Since program analysis is too complex to require all the detailed viewpoints at all time, we provide two levels of program abstract views, process-level and function-level, for program analysis. A dynamic testing and debugging method using collected run-time information is then developed to analyze timing behavior of real-time software systems. The function-level analysis may be still coarse for the software developers to correct errors. For future research, the more detailed level of abstraction, such as, basic program block level, is needed. Analysis methods for various abstraction levels are also needed in order to locate synchronization and computation faults.

References


