Timing Errors in Real-Time Systems and Their Detection

Jeffrey J. P. Tsai and Yaodong Bi
Department of Electrical Engineering and Computer Science
University of Illinois, Chicago, ILL 60680
E-mail: tsai@uicbert.eecs.uic.edu

Abstract

Most real-time applications require a high degree of reliability. Past research efforts have not fully addressed issues peculiar to real-time systems. Recently much attention has been focused on formal methods to verify timing properties of real-time systems. Despite the contribution of these formal methods, the dynamic method for analyzing timing properties is still necessary because of the inherent limitations of formal methods and imperfect program execution environment. This paper presents a dynamic timing analysis method for identifying the causes of timing errors in real-time systems.

Index terms: Real-time software system, fault diagnosis, timing analysis.

1 Introduction

Computers are increasingly being used as passive (monitoring) and active (controlling) components of real-time systems, e.g., air traffic control, aerospace, aircraft, industrial plants, and hospital patient monitoring systems. The problems of safety becomes important when these applications include systems where the consequences of failure are serious and may involve danger to human life and property. Therefore, most real-time applications require a high degree of reliability. As it's well known, the software systems for these applications are difficult to design and analyze because of their time-critical nature, and their reliability has been of much concern to software designers and users. Two major areas of research addressing software reliability have been: formal verification methods which attempt to prove the correctness of programs with respect to system specifications and software engineering which promotes adherence to programming principles to restrict the complexity of large software systems.

Although prevention and elimination of errors are essential in any trustworthy software systems, past research efforts have not fully addressed issues peculiar to real-time systems. Specifically, they do not take into account two important characteristics of the real-time environment, namely, the need for the computation to continually satisfy stringent timing constraints, and the need to guard against an imperfect execution environment which may violate design assumptions.

In recent years, much attention has been focused on formal methods to verify timing properties of real-time systems. The formal methods include extensions of temporal logic to allow quantitative reasoning about time [2, 5], reachability analysis for timed Petri nets [7], real-time logic [3], and timing analysis using graphs [3, 5]. Despite the contributions from formal methods and software engineering, the dynamic timing analysis is still needed for the following reasons. First, the formal methods have their inherent limitations in dealing with asynchronous interactions between real-time target processes and real world processes and in modeling the timing behavior of hardware implementations, scheduling algorithms. Secondly, the execution environment of real-time systems is not perfect, and the design assumptions made about the system are sometimes violated at run-time. In designing real-time systems, for example, it may be necessary to assume an upper bound on communication delays between processes or to require a hard deadline on the execution of a task. However, violation of such assumptions must be detected dynamically. In some cases, it may be infeasible to verify some real-time properties at the design time, especially, when the satisfaction of a timing constraint with dynamically created processes is involved.

This paper presents an analysis method based on system execution traces for real-time systems [10]. The main concern of this paper is to find the causes of violation of the timing constraints based on a given system execution trace. In this method, the timing behavior of the target system is presented with the colored process interaction graph, which depicts the process interactions and the times at which the interactions are performed in the target system. The dedicated colored process interaction graph is derived from the colored process interaction graph for a given timing constraint to reduce the complexity of the timing analysis. Based on these timing behavior presentations, the causes of violation of a timing constraint are identified. In our method, we assume that the events and their corresponding key-values shown in Table 1 are collected from the target system using the non-interference monitoring system introduced in [10]. These events are of the
interactions among the software processes and the interactions between the target computing system and its environment. These events are chosen in such a way that the timing behavior of the target system at process-level can be represented. Based on these events and their key-values, the timing analysis is carried out. The reader can refer [10] for the detailed information about the data collection using that non-interference monitoring system.

The computation refers to the execution time. If the relevant processes of a timing constraint spend more time on computation (staying in the “running” state) than expected, the program needs to be re-designed to reduce the execution time. If the relevant processes cannot obtain enough CPU time to execute (staying in the “ready” state), the timing constraint cannot be satisfied, and therefore, the processes need to be re-scheduled or re-assigned with higher priorities. If the relevant processes spend too much time on waiting for inputs from the environment (staying in the “waiting” state), the timing constraint cannot be satisfied either.

We refer immediate causes of a timing error to the closest reason for the timing errors. The immediate causes can be a combination of the basic causes. For instance, if a deadlock caused by a value assignment to a variable results in a timing error, we say that the deadlock is an immediate reason for the timing error although the deadlock was caused by the assignment statement. There are seven possible combinations for the three basic causes, and we conclude that four of them are identified as the immediate causes of a timing error: synchronization, scheduling, computation, and the combination of computation and scheduling.

One exception necessary to be mentioned is about the I/O operations. We here consider the I/O operations as direct interprocess communications between the processes in the target program and the external process from the environment. If a process is waiting for an input or completion of an I/O operation, we assume that there was a dummy process, \( P_{\text{external}} \), which was “running” externally during the time that the process is waiting. The dummy process \( P_{\text{external}} \) is also considered as a relevant process to the timing constraint.

### 2. Timing Errors in Real-Time Systems

In this paper, a timing constraint is defined as a time restriction between two events [1]. We call these two events the starting event and the ending event and their times the starting time \( TC_{ts} \) and the ending time \( TC_{te} \) respectively. We also call the processes which perform the starting and ending events the starting process \( TC_{ts} \) and the ending process \( TC_{te} \) respectively. We define the relevant processes \( TC_{rp} \) of a timing constraint as a set of processes which interact with the \( TC_{rp} \) or the \( TC_{te} \) of the timing constraint directly or indirectly through message passing, semaphore synchronization, or I/O operations during the time interval from \( TC_{ts} \) to \( TC_{te} \).

To find a debugging method for timing errors of real-time systems, we first identify what may cause the violation of a timing constraint in the system. A process in a real-time system may be in one of the three states: “running”, “ready”, and “waiting” after its birth and before its death. We identify three basic causes for timing errors, corresponding to this process state space (some timing errors may be caused by the combination of these three causes):

- **computation** — running,
- **scheduling** — ready, and
- **synchronization** — waiting.

### 3. Timing Error Debugging for the Synchronization Causes

There are four possible cases for a set of processes that are in the “waiting” state for an unlimited time. The first is that they are involved in a deadlock situation. The second is that they are involved in a synchronization in which the partners have terminated. That is so called the distributed termination. The third is that they are involved in a synchronization in which the partners fail to synchronize. A typical example is that, a process in a semaphore synchronization is occupying the semaphore and never releases it, or a partner in an interprocess communication never executes the requested send/receive operation. We call this as missed operations in the sense that the requested operations are never executed. The last case is that they are in a starvation situation. This only happens when semaphores are used in process synchronization, in which some relevant processes requested semaphores and never had a chance to obtain them since other processes with a
higher priority had frequently accessed the semaphores. Since the synchronization error leaves the relevant processes in the “waiting” state for an unlimited time, the ending event of the timing constraint never occurs.

The deadlocks can be detected by determining whether the processes, which are in “waiting” state, are also in a cyclically waiting state. The distributed termination can also be detected by determining whether the partner process has terminated while the process is still in “waiting” state. From the collected data, it is virtually impossible to determine whether there are missed operations or not, while the partner processes are still running. This is because the infinite event sequence of program execution cannot be obtained by monitoring the program execution, and therefore one never knows whether the partner process will execute the requested operation in the future. For the same reason, it is also virtually impossible to tell whether there is a starvation from the collected data, since one cannot determine whether the processes with a higher priority would occupy the semaphore forever or not. Here we suppose that the monitoring has been performed long enough so that if the starved processes could obtain the semaphore, it would happen and be collected in the execution history.

Thus we summarize our timing analysis procedure for the synchronization as follows. If we did not observe the ending point of the timing constraint, we perform a synchronization analysis to detect the synchronization problems. For the synchronization analysis, we first detect deadlocks. The deadlock detection procedure detects the cyclically waiting processes and excludes these processes from the “waiting” process set. If there are still any processes left in the “waiting” set after the deadlock detection, the distributed termination detection procedure is started. After excluding the processes in the deadlocks and in the distributed terminations, we detect the starvation and the missed operations, if there are still any process left in the “waiting” process set. In this section, we assume that there was a dummy process, $P_{\text{external}}$, which is “running” externally, while a process in the target program is waiting for an input or the completion of an I/O operation.

A. Deadlock Detection

A deadlock can be detected through the collected program execution data [10]. To simplify the description, all semaphores and messages in the system are considered as shared resources. To detect deadlocks, we search the process waiting chains among the waiting processes. If a process waiting chain becomes a cycle, there is a deadlock and the processes involved in the deadlock are reported. If a waiting chain ends with a process which is not in “waiting” state, then these processes in the waiting chain are not involved in deadlocks. If a waiting chain ends with a process which has been in a deadlock, then the processes in the waiting chain are blocked by the deadlock.

B. Distributed Termination Detection

If no process is left in waiting state after the deadlock detection, all the processes in “waiting” state are involved in the deadlocks and there is no distributed terminations, and no missed operations and starvation either. The analysis process can be stopped here. If there are still any processes left, we then continue the synchronization analysis. The distributed termination detection is to find out whether the partners of the processes in “waiting” state have previously terminated. To do this, the distributed termination detection procedure search the process waiting chains. If a waiting chain ends on a process which has terminated at an earlier time, we report the involved processes and the terminated process. If a waiting chain ends on an active process, all the processes on the waiting chain are kept for the missed operation and starvation detection.

C. Starvation and Missed Operation Detection

After the detection for deadlocks and distributed terminations, if there are still any processes left in the “waiting” state, we then check whether they were involved in starvations or missed operations. Since dynamic methods cannot determine whether a set of events will happen infinitely, it is not possible to determine whether an event is going to happen again in the future. Because of that, we never can tell whether there was a starvation or not. Due to the same reason, it is not possible to determine whether an event will happen in the future. Therefore, we cannot exactly tell whether there is a missed operation in the interprocess communication and the semaphore synchronization. Here we also assume that the monitoring time is long enough, so that if the corresponding operation occurred, it would be collected in the execution history.

Next, we consider starvation and missed operations in two different cases: the interprocess communication and the semaphore synchronization. In the interprocess communication, starvation cannot exist since a process can not execute send/receive operations more times than its partner does. This is true under our target system specification in which a process communicates with others only using direct communication without buffer. Since we consider the I/O operations as a kind of direct interprocess communication between the processes in the target system and external processes, we also consider there is no starvation in the I/O operations. Therefore, if a process is waiting for interprocess communication or an I/O operation, we conclude that there is a missed operation in its partner process.

To detect the missed operations in the interprocess communication, a procedure similar to the deadlock detection procedure can be used. The difference from the deadlock detection is that if a waiting chain ends with a process which is waiting for the interprocess communication and its partner is not in “waiting” state, then we report that these processes in the waiting chain and the partners are in a missed operation situation. If a waiting chain ends with an I/O operation, then we report that the processes in the waiting chain and the dummy process, $P_{\text{external}}$, are in a missed operation situation.

In a semaphore synchronization, both deadlock and
missed operation errors may occur. If a process did not execute a V operation before exiting the critical section, and other processes are waiting for the semaphore forever, then this is the case of the missed operation. A process can not get the semaphore because other processes with a higher priority infinitely access the semaphore. This is the case of starvation. Because we cannot obtain any infinite event sequence, we cannot exactly differentiate them.

Suppose a process waiting chain ends with process Pi which is waiting for semaphore S at time T. Instead of reporting the starvation or the missed operation, we provide the user with information of the P and V operations which manipulate the semaphore S after the time T. We report the processes, which executed P and/or V operations on the semaphore, and the frequency of the P and V operations of each of these processes. The determination of whether there is a starvation or a missed operation is left to the user. For example, if the last process which occupied the semaphore had executed one or several pairs of P and V operations on the semaphore between the time T and the time at which the monitoring ended, it is possible there is a starvation. If the last process did not execute any V operation (the number of the V operation is zero), it is very possible that there is a missed operation (V operation) in the last process.

Since the interprocess communication only has missed operation errors, we exclude the processes involved in the missed operations in the interprocess communication from the processes in the "waiting" state. Thus the processes left now are only those involved in semaphore synchronizations. If a waiting chain ends with a process Pk which was waiting for semaphore S, then we provide the user with the information of the P and V operations on the semaphore S after process Pk executed the P operation, and the number of times P and V operations executed by each process.

4 A Graph Representation for Timing Analysis

To analyze a timing error caused by the computation and the scheduling, we construct a graph which can represent the timing behavior of the program execution. Since the processes which are not relevant to the timing constraint have no direct effect on the timing error, we only include the relevant processes in the graph to reduce the computation complexity. We also define relevant segments in a timing constraint as a set of segments of the process execution history, which have affected the satisfaction of the timing constraint, according to the collected data. It is obvious that the relevant segments belong to the relevant processes. To construct this graph, we first introduce the Colored Process Interaction Graph (CPKG), which will present the program execution behavior in terms of time with all the processes in the system. The colors depict the different states of a process at different time. Then we construct a graph, which contains only the relevant segments, from the CPIG, and call it the Dedicated Colored Process Interaction Graph (DCPIG), dedicated to an observed timing error whose ending point was observed. We emphasize that the ending point is observed because the DCPIG can not be constructed without the ending point.

The analysis will identify a set of processes with their segments which are responsible for the violation of the timing constraint. As mentioned earlier, the real cause of a fault may not occur in this DCPIG, it may be in a part of program executed in an earlier time. Therefore, we only say that the set of processes identified by the analysis are immediately responsible for this timing constraint violation, and their faults may be caused by other processes at other time. To detect the real source of the fault, some other analysis methods may be needed.

The Colored Process Interaction Graph (CPKG), which represents the system scheduling, the interprocess communication, and the synchronization of the program's execution, is constructed from the program's execution data collected from the target systems [lo]. For each process Pi in the system, we draw an Event Flow Graph (EFG) by connecting all the events performed by Pi with directed edges in the order of time sequence. Each edge is associated with the time duration of the edge and a color: running, ready, or waiting, to depict the state of the process during the time the edge implies. The three colors correspond to the three possible states in the process state space.

After the EFG for each process is constructed, we merge all the EFGs into one single graph, the Colored Process Interaction Graph (CPKG), by connecting two associated event nodes with an additional horizontal edge to represent the process interactions. The edge between two nodes, Np, of process Pi and Nq, of process Qj, is drawn if the two nodes meet one of the following conditions:

1. one of the nodes is an initial state of a process and the other is the event of creating the process;
2. one of the event is a termination of a process and the other is a state transition event that the parent process of the terminating process which has been waiting for its termination changes the state into "ready";
3. one of the them is a sending/receiving event and the other is the state transition event that its partner changes the state to "ready", and the state transition happened at the same time as the send/receive event; or
4. one of them is a V operation and the other is the state transition event that a process in the waiting queue of the semaphore changes into "ready"
After the Colored Process Interaction Graph is constructed, the Dedicated Colored Process Interaction Graph, which contains only the relevant segments of the relevant processes, is derived from it. The DCPIG consists of all the nodes and edges which are connected to the starting or ending nodes between the starting and ending time points. The starting point and the ending node of the DCPIG correspond to the starting event and the ending event of the timing constraint. The DCPIG is constructed in the following way. We cut off all nodes and edges before the starting node and after the ending node in terms of real time. If a process does not have a node at the time of starting node, a dummy node is added to it as the first node of that process. We first include the ending node in the DCPIG. If a direct edge is connected to a node which is in the DCPIG, then include this edge and its other end node into the DCPIG. If a horizontal line is connected with a node which is in the DCPIG, then include this line and the node at the other end of the line into the DCPIG. This is repeated until all the nodes and edges, connected to the starting and ending nodes, are included in the DCPIG. The final step for constructing the DCPIG is to augment the DCPIG with a coordinate with process and time as its axes. The process axis corresponds to the processes in the system, and time to the real-time of the system. Now, an event in the graph can be denoted by its coordinate (process, time).

As an example, we assume that the program shown in Fig. 1 is monitored. Each column represents the codes of the process named at the top of the column. For brevity, we only show the statements to be monitored in the program. In the execution time, the process P1 is created by the system process P0, the process P2 by P1, P3 by P2, and P4 by P3. Two timing constraints are imposed on the system:

1. P2 should finish its job within TC1 time unit, i.e. P2 should terminate within TC1 time unit after it is created;
2. P1 should perform the output operation within TC2 time unit after P2 creates P3;

After postprocessing the recorded execution data of the program, we represent the program execution behavior using the CPIG as shown in Fig. 2. For the timing constraint 1, the starting point of the timing constraint is the create event occurring at t3, (P2, t3), and the ending point is the terminate event at t10, (P2, t10). For the timing constraint 2, the starting point is the create event at t4, (P2, t4), and the ending point is the output event at t24, (P1, t24). The DCPIG's for both timing constraints are shown in Fig. 3 and Fig. 4 respectively.

5 Timing Analysis for Computation and Scheduling Causes

In this section, we first define a basic time interval in a given DCPIG as a time interval during which there is no other event nodes involved in the DCPIG. For example in the DCPIG shown in Fig. 4, the time interval t6—t7 is a basic time interval, but t6—t8 is not, because two events, (P1, t7) and (P3, t7) occur at time t7 during the interval. But the time interval t17—t19 is also a basic time interval in this DCPIG, since the I/O event is not involved in this DCPIG, although it occurs at time t18 during the interval in the CPIG. Now we re-define the relevant processes of a timing constraint based on the basic time intervals. We say a process is a relevant process during a given basic time interval for a timing constraint, if one of its segments is involved in the DCPIG during the given time interval. For example, in Fig. 4, the process P1, P2 and P3 are relevant processes during the basic time interval 48—49, and P1 and P3 are the only relevant processes during the time interval 9—10.

We say that a set of processes are in “running” state during a basic interval if one of them is in “running” state during the basic interval; a set of processes are in “ready” state during a basic interval if at least one of the processes is in “ready” state and none of them is in “running” state during the basic interval; a set of processes are in “waiting” state during a basic interval if all of them are in “waiting” state. For example, all of the relevant processes for every basic interval, in Fig. 3, are in “running” state. In Fig. 4, the relevant processes P1 and P3 are in “ready” state during the basic interval t17—119; all the relevant processes P1, P3, and P4 are in “waiting” state during the interval t15—t16.

There are three possible timing error causes: computation, scheduling, and the combination of computation and scheduling. For the computation and the scheduling, we have the following theorems.

Theorem 1: If the total time of the basic time intervals during which all relevant processes are in “running” state, i.e. the total execution time between the starting and the ending points of the timing constraint, is larger than the timing constraint, the computation must be an immediate cause of the timing error.

Theorem 2: If the total time of the basic time intervals during which all relevant processes were in “ready” state is larger than the timing constraint, the scheduling must be an immediate cause of the timing error.

The converse of the two theorems is not always true. Neither does the computation cause imply that the total “running” time is larger than the timing constraint, nor does the scheduling cause imply that the total “ready” time is larger than the timing constraint. If the execution time of the relevant processes for a timing constraint is larger than the timing constraint, the computation time must be reduced in order to satisfy the
timing constraint. If the total waiting time for the CPU of the relevant processes is larger than the timing constraint, the relevant processes' priorities must be adjusted so that the scheduler of the operating system can assign more CPU time to the relevant processes.

If the total waiting time for the CPU of the relevant processes is larger than the timing constraint, the relevant processes' priorities must be adjusted so that the scheduler of the operating system can assign more CPU time to the relevant processes.

If neither the total "running" time nor the total "ready" time is larger than the timing constraint, we cannot determine which one is the cause of the timing error or may be both. As we did for the starvation and missed operations, we provide the user with timing information of the execution of the relevant processes between the starting and ending points. The relevant processes, the total execution time, and the total "ready" time are reported to the user. The determination of what are the causes of the timing error is left to the user.

First we provide the user with the total execution time of the basic intervals during which all relevant processes are in "running" state. Since the target system is a single processor system, only one process is executing at any time point. Thus, we compute the execution time by projecting the color edges of running to the time axis and summing all of them up. This sum is the total execution time and is reported to the user. If the total time is larger than the timing constraint, the computation error is reported. For example, suppose that the timing constraint 1 is violated, i.e. \((t_{10} - t_3) > TC1\). The total execution time from the starting point to the ending point, in Fig. 3, is \((t_{10} - t_3)\), of the consecutive basic intervals from \(t_3\) to \(t_{10}\). The computation is the only immediate cause of the timing error. The total execution time from the starting point to the ending point in Fig. 4 is \((t_9 - t_4) + (t_{15} - t_{10}) + (t_{17} - t_{16}) + (t_{20} - t_{19}) + (t_{24} - t_{21})\), of the consecutive basic intervals.
from t4 to t9, the consecutive basic intervals from t10 to t15, the basic interval t16-t17 and t19-t20, and the consecutive basic intervals from t21 to t24.

To compute the total "ready" time, of the relevant processes, we first discard the "running" basic intervals, and then project all of the color edges of ready to the time axis and sum them up. If the total "ready" time is larger than the timing constraint, we report that the scheduling is a cause of the timing error. If the total "ready" time is zero, we report that the scheduling is not the cause of the timing error. If the total "ready" time is less than the timing constraint, we report the total time to the user. The determination whether the scheduling is a cause of the timing error is left to the user. For example, the total "ready" time for the timing constraint 2 of the example, as shown in Fig. 4, is \((t10 - t9) + (t10 - t17) + (t21 - t20)\). If this value is larger than TC2, we report that the scheduling is a cause of the timing error. If this value is less than TC2, we report it to the user. In the DCPIG, Fig. 3, of the timing constraint 1 of the example, the total "ready" time is zero, the operating system has givend the whole CPU time to its relevant processes. Therefore, the scheduling is not the cause of the timing error. The non-relevant processes and their segments which are occupying the CPU during these "ready" basic intervals are also reported, along with the total "ready" time. This information can be produced by searching the processes and their segments on the CPIG. For example, in Fig. 4, the process P2 is occupying the CPU during the time interval t9-t10; P4 during t17-t19; and P3 during t20-t21.

As can be seen in the above discussion, the external process for I/O operations is not considered in these DCPIG's. If some of the relevant processes are waiting for an I/O operation and the rest of the processes are in "running" or "ready" state, the external process would not be considered since it has no effect on the computation and the scheduling. If all of the relevant processes during a basic interval are waiting for the I/O, the waiting time will affect the satisfaction of the timing constraint. If the total waiting time is larger than the timing constraint, the external process is faulty since it has failed to send a message to the process in the chain. If P1 is not waiting for an I/O, i.e. for communication or a semaphore, its partner P2 must be in the DCPIG and P2 is added to WAITING-CHAIN as the last process P1. Suppose there are N relevant processes during the interval I. After this repeat N times and none of the P1 is waiting for I/O, P1 must have been in WAITING-CHAIN already. If so, there is a deadlock because of the cyclically waiting. This is contradictory to the construction of the DCPIG. Therefore, the waiting chain must end with an I/O.

For example, during the basic interval t15-t16 in Fig. 4, the relevant process P1 is waiting for its partner P3, which is occupying the semaphore S. The process P3 is waiting for P4 to receive the message. The waiting chain ends with P4 which is waiting for the input from the external process. From the discussion above, we can compute the total waiting time for I/O's by accumulating the time intervals which are not covered by the projection of the "running" and "ready" edges. For example, the total waiting time for the I/O operation in Fig. 4 is \((t16 - t15)\), which is not covered by the projection of the color edges of running and ready.

Theorem 4: If the total time that all relevant processes are in "waiting" state is larger than the timing constraint, then the external process is responsible for the timing error.

If the total waiting time is larger than the timing constraint, the external process must have failed to give the stimulus to the system within time. Therefore, the external process is faulty. For example, If the total waiting time \((t16 - t15)\), in Fig. 4, is larger than TC2, we report the external process as a faulty process. If the total waiting time is less than the timing constraint, we report the time to the user, and the determination of whether the external process is faulty is left to the user.

6 Conclusion and Future Research

This paper presents an analysis method to detect timing errors in real-time systems. The main contribution of this paper is to identify three basic and immediate timing error causes based on the three process states and their possible combinations, and to provide corresponding timing analysis methods for those immediate timing error causes.

For future research, the three basic and immediate timing error causes may be extended to multiple processor distributed systems. For distributed systems, the real-time monitoring system architecture presented in [10] can be employed for data collection. New methods for computing the execution time and the ready time should be designed. The Dedicated Colored Process Interaction Graph needs to be modified to accommodate different nodes and communication delays.
References


