Testing Distributed Real-Time Systems

Henrik Thane and Hans Hansson,

Mälardalen Real-Time Research Centre,
Department of Computer Engineering, Mälardalen University,
Västerås, Sweden, hte@mdh.se

Abstract

For testing of sequential software it is usually sufficient to provide the same input (and program state) in order to reproduce the output. For real-time systems, on the other hand, we need also to control, or observe, the timing and order of the inputs. If the system additionally is multitasking, we also need to take timing and the concurrency of the executing tasks into account.

In this paper we present a method for deterministic testing of multitasking real-time systems, which allows explorative investigations of real-time system behavior. The method includes an analysis technique that given a set of tasks and a schedule derives all execution orderings that can occur during run-time. These orderings correspond to the possible interleavings of the executing tasks. The method also includes a testing strategy that using the derived execution orderings can achieve deterministic, and even reproducible, testing of real-time systems. Since, each execution ordering can be regarded as a sequential program, it becomes possible to use techniques for testing of sequential software in testing multi-tasking real-time system software. We also show how this analysis and testing strategy can be extended to encompass distributed computations, communication latencies and the effects of global clock synchronization. The number of execution orderings is an objective measure of the testability of a system since it indicates how many behaviors the system can exhibit during runtime. In the pursuit of finding errors we must thus cover all these execution orderings. The fewer the orderings the better the testability.

Keyword: Determinism, Testing, Distributed real-time systems, reproducibility.

1 INTRODUCTION

In this paper we will present a novel method for integration testing of multitasking real-time systems (RTS) and distributed real-time systems (DRTS). This method achieves deterministic testing of RTS and DRTS by accounting for the effects of scheduling, jitter in RTS, and the inherent parallelism of DRTS applications.

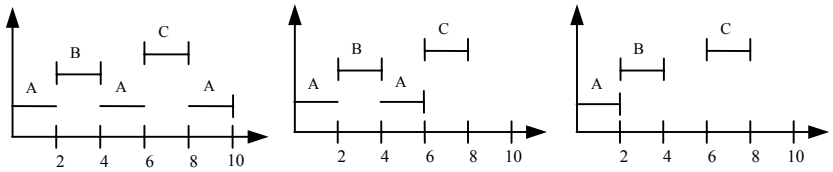
A real-time system is by definition correct if it performs the correct *function* at the correct *time*. Using real-time scheduling theory we can provide guarantees that each task in the system will meet its timing requirements [1][13][27], given that the basic assumptions, e.g., task execution times and periodicity, are not violated at run-time. However, scheduling theory does not give any guarantees for the functional behavior of the system, i.e., that the computed values are correct. To assess the functional correctness other types of analysis are required. One possibility, although still immature, is to use formal methods to verify certain functional and temporal properties of a model of the system. The formally verified properties are then guaranteed to hold in the real system, as long as the model assumptions are not violated. When it comes to validating the underlying assumptions (e.g., execution times, synchronization order and the correspondence between specification and implemented code) we must use dynamic verification techniques which explore and investigate the run-time behavior of the real system. Testing [16] is the most commonly used such technique., and also the state-of-practice in functional verification.

Reproducible and deterministic testing of sequential programs can be achieved by controlling the sequence of inputs and the start conditions [14]. That is, given the same initial state and inputs, the sequential program will deterministically produce the same output on repeated executions, even in the presence of systematic faults [15]. Reproducibility is essential when performing regression testing or cyclic debugging, where the same test cases are run repeatedly with the intent to validate that either an

error correction had the desired effect, or simply to make it possible to find the error when a failure has been observed [12]. However, trying to directly apply test techniques for sequential programs on distributed real-time systems is bound to lead to non-determinism and non-reproducibility, because control is only forced on the inputs, disregarding the significance of order and timing of the executing and communicating tasks. Any intrusive observation of a distributed real-time system will, in addition, incur a temporal probe-effect [6][18] that subsequently will affect the temporal and functional behavior of the system.

The main contribution of this paper is a method for achieving deterministic testing of distributed real-time systems. We will specifically address task sets with recurring release patterns, executing in a distributed system where the scheduling on each node is handled by a priority driven preemptive scheduler. This includes statically scheduled systems that are subject to preemption [27][17], as well as strictly periodic fixed priority systems [1][13]. The method aims at transforming the non-deterministic distributed real-time systems testing problem into a set of deterministic sequential program testing problems. This is achieved by deriving all the possible execution orderings of the distributed real-time system and regarding each of them as a sequential program. A formal definition of what actually constitutes an execution order scenario will be given later in the paper. The following small example presents the underlying intuition:

Consider *Figure 1-1a,* which depicts the execution of the tasks *A*, *B* and *C* during one instant of the repetitive pattern of executions dictated by an off-line generated static schedule, of length equal to the Least Common Multiple (LCM) of the task period times. . The tasks have fixed execution times, i.e. the worst and best-case execution times coincide ($WCET_i=BCET_i$, for $i \in \{A,B,C\}$). A task with later release time is assigned higher priority. These non-varying execution times have the effect of only yielding one possible execution scenario during the LCM, as depicted in *Figure 1-1a*. However, if for example, task *A* would have a minimum execution time of 2 ($BCET_A=2$; $WCET_A=6$) we would get three possible execution scenarios, depicted in *figures 1-1a to 1-1c.* In addition to the execution order scenario in *Figure 1a*, there are now possibilities for *A* to complete before *C* is released (*Figure 1-1b*), and for *A* to complete before *B* is released (*Figure 1-1c*).



(a) $WCET_A=6$ (b) $WCET_A=4$ (c) $WCET_A=2$

Figure 1-1 Three different execution order scenarios.

Given that these different scenarios yield different system behaviors for the same input, due to the order or timing of the produced outputs, or due to unwanted side effects via unspecified interfaces (caused by bugs), we would by using regular testing techniques for sequential programs get non-deterministic results. For example, assume that all tasks use a common resource *X* which they do operations on. Assume further that they receive input immediately when starting, and deliver output at their termination.

We would then for the different scenarios depicted in figures 1-1a to 1-1c get different results:

- The scenario in *Figure 1-1a* would give $A(X)$, $B(X)$ and $C(B(X))$.
- The scenario in *Figure 1-1b* would give $A(X)$, $B(X)$ and $C(A(X))$.
- The scenario in *Figure 1-1c* would give $A(X)$, $B(A(X))$ and $C(B(A(X)))$.

Making use of the information that the real-time system depends on the execution orderings of the involved tasks, we can achieve deterministic testing, since for the same input to the tasks and the same execution ordering, the system will deliver the same output on repeated executions.

In order to address the scenario dependent behavior we suggest a testing strategy consisting of the following:

1. Identify all possible execution order scenarios for each scheduled node in the system during a single instance of the release pattern of tasks with duration *T*; typically equal to the LCM of the period times of the involved tasks
2. Test the system using any regular testing technique of choice, and monitor for each test case which execution order scenario is run during *[0,T],* i.e., which, when and in what order jobs are started, preempted and completed. By jobs we mean single instances of the recurring tasks during *T*.
3. Map test case and output onto the correct execution ordering, based on observation.
4. Repeat 2-3 until the sought coverage is achieved.

The probe effect is avoided by making the probes part of the design and then letting them remain in the target system [24][25]..

Contributions

In this paper we present a method for functional integration testing of multitasking real-time systems and distributed real-time systems: This method is to our knowledge the first testing method to fully encompass scheduling of distributed real-time systems. The main activities of the testing method are to:

- Identify the execution order scenarios for each node in a distributed real-time system
- Compose them into global execution order scenarios
- Make use of them when testing (the test strategy)
- Reproduce the scenarios.

Paper outline

Section 2 presents our system model. Section 3 formalizes the concept of execution orderings and presents the algorithm for identifying all the possible execution orderings in a single node real-time system, and then show how the algorithm can be extended to also cover distributed real-time systems. Section 4 suggests a testing strategy for achieving deterministic and reproducible testing in the context of the presented execution order analysis. Section 5 elaborate on issues like jitter, complexity, and testability. Finally, in Section 6, we conclude and give some hints on future work.

2 THE SYSTEM MODEL

Our system model consists of a distributed system consisting of a set of nodes, which communicate via a temporally predictable broadcast network, i.e. upper and lower bounds on communication latencies are known or can be calculated [8][26]. Each node is a self sufficient computing element with CPU, memory, network access, a local clock and I/O units for sampling and actuation of the external system. We further assume the existence of a global synchronized time base [9][5] with a known precision δ, meaning that no two nodes in the system have local clocks differing by more than δ.

We assume that the software that runs on the distributed system consists of a set of concurrent tasks, communicating by message passing. Functionally related and cooperating tasks, e.g., sample-calculate-actuate loops in control systems, are defined as transactions. The relationship between the cooperating tasks with respect to precedence (execution order), interactions (data-flow), and a period time typically define each transaction. The tasks are distributed over the nodes, typically with transactions that span several nodes, and with more than one task on each node. All synchronization is resolved before run-time and therefore no action is needed to enforce synchronization in the actual program code. Different release times and priorities guarantee mutual exclusion and precedence. The distributed system is globally scheduled, which results in a set of specific schedules for each node. At run-time we need only synchronize the local clocks to fulfill the global schedule [8].

Task model

We assume a fairly general task model that includes both preemptive scheduling of statically generated schedules [27] and fixed priority scheduling of strictly periodic tasks [1][13]:

- The system contains a set of jobs *J,* i.e. invocations of tasks, which are released in a time interval $[t, t+T^{MAX}]$, where T^{MAX} is typically equal to the Least Common Multiple (*LCM*) of the involved tasks period times, and t is an idle point within the time interval $[0, T^{MAX}]$ where no job is executing. The existence of such an idle point, t, simplifies the model since it prevents temporal interference between successive T^{MAX} intervals. To simplify the presentation we will henceforth assume an idle point at 0 for all participating nodes.
- Each job $j \in J$ has a release time r_j, worst case execution time ($WCET_j$), best case execution time ($BCET_j$), a deadline D_j and a priority p_j. J represents one instance of a recurring pattern of job executions with period T^{MAX}, i.e., job j will be released at time r_j, $r_j + T^{MAX}$, $r_j + 2T^{MAX}$, etc.
- The system is preemptive and jobs may have identical release-times.

Related to the task model we assume that the tasks may have functional and temporal side effects due to preemption, message passing and shared memory. Furthermore, we assume that data is sent at the termination of the sending task (not during its execution), and that received data is available when tasks start (and is made private in an atomic first operation of the task) [23][4][10].

Fault hypothesis

Note that, although synchronization is resolved by the off-line selection of release times and priorities, we cannot dismiss unwanted synchronization side effects. The schedule design can be erroneous, or the assumptions about the execution times might not be accurate due to poor execution time estimates, or simply due to design and coding errors.

Inter-task communication is restricted to the beginning and end of task execution, and therefore we can regard the interval of execution for tasks as atomic. With respect to access to shared resources, such as shared memory and I/O interfaces, the atomicity assumption is only valid if synchronization and mutual exclusion can be guaranteed.

Our fault hypothesis is that errors can only occur due to erroneous outputs and inputs to jobs, and/or due to synchronization errors, i.e., jobs can only interfere via specified interactions.

One way to guarantee the fault hypothesis in a shared memory system is to make use of hardware memory protection schemes, or to during design eliminate or minimize shared resources using wait-free and lock-free communication [23][11][3].

3 EXECUTION ORDER ANALYSIS

In this section we present a method for identifying all the possible orders of execution for sets of jobs conforming to the task model introduced in section 2. We will also show how the model and analysis can be extended to accommodate for interrupt interference and multiple nodes in a distributed system by considering clock-synchronization effects, parallel executions, and varying communication latencies.

3.1 Execution Orderings

In identifying the execution orderings of a job set we will only consider the following major events of job executions:

- The start of execution of a job, i.e., when the first instruction of a job is executed. We will use *S(J)* to denote the set of start points for the jobs in a job set J; $S(J) \subseteq J \times [0, T^{MAX}] \times J \cup \{\_\}$, that is *S(J)* is the set of triples (j_1, *time*, j_2), where j_2 is the (lower priority) job that is preempted by the start of j_1 at *time*, or possibly "_" if no such job exists.
- The end of execution of a job, i.e., when the last instruction of a job is executed. We will use *E(J)* to denote the set of end points (termination points) for jobs in a job set J; $E(J) \subseteq J \times [0,$

$T^{MAX}] \times J \cup \{\_\}$, that is *E(J)* is a set of triples (j_1, *time*, j_2), where j_2 is the (lower priority) job that resumes its execution at the termination of j_1, or possibly "_" if no such job exists.

We will now define an execution to be a sequence of job starts and job terminations, using the additional notation that

- *ev* denotes an event, and *Ev* a set of events.
- *ev.t* denotes the time of the event *ev*,
- *Ev\I* denotes the set of events in *Ev* that occur in the time interval *I*,
- *Prec(Ev, t)* is the event in *Ev* that occurred most recently at time *t* (including an event that occurs at *t*).
- *Nxt(Ev, t)* denotes the next event in *Ev* after time *t*.
- *First(Ev)* and *Last(Ev)* denote the first and last event in *Ev*, respectively.

Definition 3-1. An *Execution* of a job set *J* is a set of events $X \subseteq S(J) \cup E(J)$, such that

1) For each $j \in J$, there is exactly one start and termination event in *X*, denoted $s(j,X)$ and $e(j,X)$ respectively, and $s(j,X)$ precedes $e(j,X)$, i.e. $s(j,X).t \leq e(j,X).t$, where $s(j,X) \in S(J)$ and $e(j,X) \in E(J)$.
2) For each $(j_1, t, j_2) \in S(J)$, $p_{j1} > p_{j2}$, i.e., jobs are only preempted by higher priority jobs.
3) For each $j \in J$, $s(j,X).t \geq r_j$, i.e., jobs may only start to execute after being released.
4) After its release, the start of a job may only be delayed by intervals of executions of higher priority jobs, i.e., using the convention that $X\backslash[j.t, j.t) = \emptyset$.

   For each job $j \in J$ each event $ev \in X\backslash[Prec(X,r_j).t, s(j,X).t)$ is either

   - A start of the execution of a higher priority job, i.e. $ev = s(j', X)$ and $p_{j'} > p_j$
   - A job termination, at which a higher priority job resumes its execution, i.e., $ev = (j', t, j'')$, where $p_{j''} > p_j$

5) The sum of execution intervals of a job $j \in J$ is in the range $[BCET(j), WCET(j)]$, i.e.,

$$BCET(j) \leq \sum_{ev \in \{s(j,X)\} \cup \{(j',t,j) \mid (j',t,j) \in E(J)\}} (Nxt(X, ev.t).t - ev.t) \leq WCET(j)$$

   That is, we are summing up the intervals in which *j* starts or resumes its execution.

We will use $EX_t(J)$ to denote the set of timed executions of the job set *J*. Intuitively, $EX_t(J)$ denotes the set of possible executions of the job set *J* within $[0, T^{MAX}]$. Assuming a dense time domain $EX_t(J)$ is only finite if $BCET(j) = WCET(j)$ for all $j \in J$. However, if we disregard the exact timing of events and only consider the ordering of events we obtain a finite set of execution orderings for any finite job set *J*.

Using $ev\{x/t\}$ to denote an event *ev* with the time element *t* replaced by the undefined element "*x*", we can formally define the set of execution orderings *EX(J)* as follows:

Definition 3-2. The set of *Execution orderings EX(J)* of a job set *J* is the set of sequences of events such that $ev_0\{x/t\}, ev_1\{x/t\}, ..., ev_k\{x/t\} \in EX(J)$ **iff** there exists an $X \in EX_t(J)$ such that

- $First(X) = ev_0$
- $Last(X) = ev_k$
- *For any* $j \in [0..(k-1)]$: $Nxt(X, ev_j.t) = ev_{j+1}$

Intuitively, *EX(J)* is constructed by extracting one representative of each set of equivalent execution orderings in $EX_t(J)$, i.e., using a quotient construction $EX(J) = EX_t(J)\backslash \sim$, where $\sim$ is the equivalence induced by considering executions with identical event orderings to be equivalent. This corresponds to our fault hypothesis, with the overhead of keeping track of preemptions and resumptions, although not exactly where in the program code they occur. This overhead means that we can capture more than what our fault hypothesis is supposed to capture. We could thus reduce the number of execution

orderings further if we define $EX(J) = EX_t(J)\setminus \approx$, where $\approx$ is the equivalence induced by considering executions with identical job start and job stop orderings to be equivalent. In the process of deriving all the possible execution orderings we need however to keep track of all preemptions, i.e., $EX_t(J)\setminus \sim$, but after having derived this set we can reduce it to $EX_t(J)\setminus \approx$. Even further reductions could be of interest, for instance to only consider orderings among tasks that are functionally related, e.g., by sharing data.

In the remainder we will use the terms *execution scenario* and *execution ordering* interchangeably.

3.2 Calculating *EX(J)*

This section outlines a method to calculate the set of execution orderings *EX(J)* for a set of jobs *J*, complying with definition 3-2. We will later (in section 3.3) present an algorithm that performs this calculation. In essence, our approach is to make a reachability analysis by simulating the behavior during one $[0,T^{MAX}]$ period for the job set *J*.

The algorithm we are going to present generates, for a given schedule, an Execution Order Graph (EOG), which is a finite tree for which the set of possible paths from the root contains all possible execution scenarios.

But before delving into the algorithm we describe the elements of an EOG. Formally, an EOG is a pair $<N, A>$, where

- N is a set of nodes, each node being labeled with a job and a continuous time interval, i.e., for a job set J: $N \subseteq J \cup \{\text{“}\_\text{”}\} \times I(T^{MAX})$, where {“_”} is used to denote a node where no job is executing and $I(T^{MAX})$ is the set of continuous intervals in $[0, T^{MAX}]$.
- A is the set of edges (directed arcs; transitions) from one node to another node, labeled with a continuous time interval, i.e., for a set of jobs J: $A \subseteq N \times I(T^{MAX}) \times N$.

Intuitively, an *edge*, corresponds to the transition (task-switch) from one job to another. The edge is annotated with a continuous interval of when the transition can take place, as illustrated in *Figures 3-1* and *3-2*.

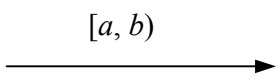


Figure 3-1. A Transition.

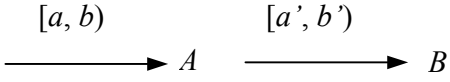


Figure 3-2 Two transitions, one to job A and one from job A to job B.

The interval of possible start times $[a', b']$ for job *B*, in *Figure 3-3*, is defined by:

$$a' = MAX(a, r_A) + BCET_A \qquad (3\text{-}1)$$

$$b' = MAX(b, r_A) + WCET_A$$

The MAX functions are necessary because the calculated start times *a* and *b* can be earlier than the scheduled release of the job *A*.

A *node* represents a job annotated with a continuous interval of its possible execution, as depicted in *Figure 3-3*.

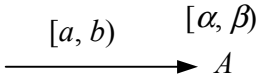


Figure 3-3. A job annotated with its possible execution and start time.

We define the interval of execution, $[\alpha, \beta)$ by:

$$\alpha = MAX(a, r_A) \qquad (3\text{-}2)$$

$$\beta = MAX(b, r_A) + WCET_A$$

That is, the interval, $[\alpha, \beta)$, specifies the interval in which job *A* can be preempted.

From each node in the execution ordering graph there can be one or more transitions, representing one of four different situations:

1) The job is the last job scheduled in this branch of the tree. In this case the transition is labeled with the interval of finishing times for the node, and has the empty job "_" as destination node, as exemplified in *Figure 3-4*.

2) The job has a *WCET* such that it definitely completes before the release of any higher priority job. In this case we have two situations:

   a) *Normal situation*. One single outgoing transition labeled with the interval of finishing times for the job, [*a'*, *b'*). Exemplified by (1) in *Figure 3-4*.

   b) *Special case*. If a higher priority job is immediately succeeding at [*b',b'*] while *b' > a'*, and there are lower priority jobs ready, or made ready during [α,β) then we have two possible transitions: One branch labeled with the interval of finishing times [*a',b*') representing the immediate succession of a lower priority job, and one labeled [*b', b'*) representing the completion immediately before the release of the higher priority job. Exemplified by (2) in *Figure 3-4*.

3) The job has a *BCET* such that it definitely is preempted by another job. In this case there is a single outgoing transition labeled with the preemption time *t*, expressed by the interval [*t*, *t*], as exemplified by (3) in *Figure 3-4*.

4) The job has a *BCET* and *WCET* such that it may either complete or be preempted before any preempting job is released. In this case there can be two or three possible outgoing edges depending on if there are any lower priority jobs ready. One branch representing the preemption, labeled with the preemption time [*t*, *t*], and depending on if there are any lower priority jobs ready for execution we have two more transition situations:

   a) *No jobs ready.* Then there is one branch labeled [*a'*, *t*) representing the possible completion prior to the release of the higher priority job. Exemplified by (4) in *Figure 3-4*.

   b) *Lower priority jobs ready.* If $\beta > \alpha$ then there is one branch labeled [*a'*, *t*) representing the immediate succession of a lower priority job, and one labeled [*t, t*) representing the completion immediately before the release of the preempting job. Exemplified by (5) in *Figure 3-4*.

Example 3-1

Figure 3-4 gives an example of an EOG, using the above notation and the attributes in *Table 3-1*. In *Figure 3-4*, all paths from the root node to the "_" nodes correspond to the possible execution order scenarios during one instance of the recurring release pattern.

Table 3-1 A job set for a schedule with a LCM of 400 ms.

Task	r	p	WCET	BCET
A	0	4	39	9
B	40	3	121	39
C	40	2	59	49
A	100	4	39	9
A	200	4	39	9
A	300	4	39	9
D	350	1	20	9

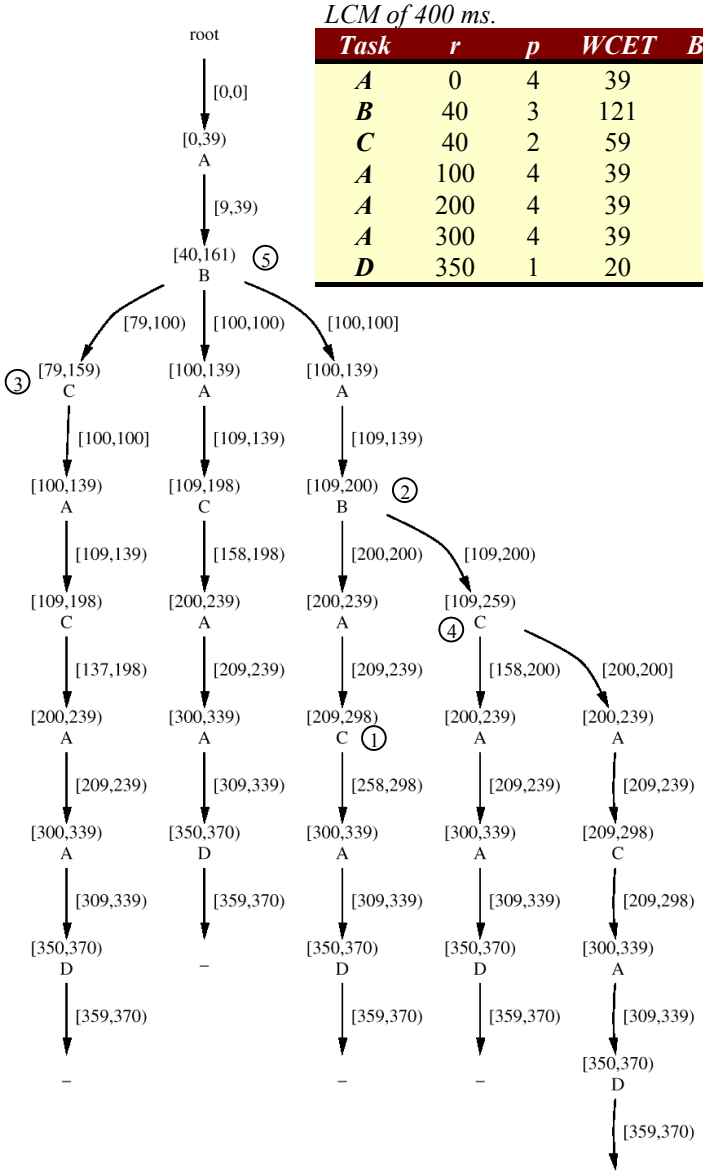


Figure 3-4. The resulting execution order graph for the job set in Table 3-1.

3.3 The EOG algorithm

We will now define an algorithm for generating the EOG. Essentially, the algorithm simulates the behavior of a strictly periodic fixed priority preemptive real-time kernel, complying with the previously defined task model and EOG primitives. In defining the algorithm we use the following auxiliary functions and data structures:

1) *rdy* – the set of jobs ready to execute.
2) Next_release(*I*) – returns the earliest release time of a job $j \in J$ within the interval *I*. If no such job exists then ∞ is returned . Also, we will use *I.l* and *I.r* to denote the extremes of *I*.
3) P(*t*) – Returns the highest priority of the jobs that are released at time *t*. Returns -1 if $t = \infty$.
4) Make_ready(*t, rdy*) – adds all jobs that are released at time *t* to *rdy*. Returns ∅ if $t=\infty$, else the set.
5) X(*rdy*) extracts the job with highest priority in *rdy*.
6) Arc(*n, I, n'*) creates an edge from node *n* to node *n'* and labels it with the time interval *I*.
7) Make_node(*j, XI*) creates a node and labels it with the execution interval *XI* and the id of job *j*.

The execution order graph for a set of jobs J is generated by a call $Eog(ROOT, \{\}, [0, 0], [0, T^{MAX}])$ to the function given in *Figure 3-5*, i.e., with a root node, an empty ready set, the initial release interval [0,0], and the considered interval $[0, T^{MAX}]$ as input parameters.

```
// n- previous node, rdy- set of ready jobs, RI – release interval, SI – the considered interval.
Eog(n, rdy, RI, SI)
{
      //When is the next job(s) released?
      t = Next_release(SI)
      if (rdy = ∅)
          rdy = Make_ready(t, rdy)
          if( rdy ≠ ∅)
              Eog (n, rdy, RI, (t,SI.r] )
          else   Arc(n, RI, _ )
      else
          //Extract the highest priority job in rdy.
          T      = X(rdy)
          [α,β)  = [max(r_T, RI.l), max(r_T, RI.l)+WCET_T)
          a'     = α + BCET_T
          b'     = β
          n'     = Make_node(T, [α,β) ) Arc(n, RI, n')

          // Add all lower priority jobs that are released before
          T's termination, or before a high priority job is preempting T.
          while((t < β) ∧ (P(t) < p_T))
              rdy    =  Make_ready(t, rdy)
              t      =  Next_release((t, SI.r])

          // Does the next scheduled job preempt T?
          if((p_T < P(t)) ∧ (t <  β))
              // Can T complete prior to the release of the next job at t?
              if(t > a')
                  Eog ( n', rdy, [a',t), [t,SI.r] )
                  if(rdy ≠ ∅)
                      Eog(n',Make_ready(t,rdy),[t,t),(t,SI.r])

                  else if(t  = a')
                            Eog(n', Make_ready(t, rdy), [t,t), (t,SI.r])


              //Add all jobs that are released at time t.
              rdy = Make_ready(t, rdy)

              //Best and worst case execution prior to preemption?
              BCET_T = max(BCET_T - (t-(max(r_T, RI.l)),0)
              WCET_T = max(WCET_T - (t-(max(r_T, RI.r)),0)
              Eog( n', rdy + {T}, [t,t], (t,SI.r])

          // No preemption
          else   if(t = ∞ ) //Have we come to the end of the simulation?
                     Eog(n', rdy, [a',b'),[∞,∞]) //Yes, no more jobs to execute

                 else // More jobs to execute

                     //Is there a possibility for a high priority job to succeed immediately,
                     while low priority jobs are ready?
                     if(rdy ≠ ∅ ∧ t = β) //Yes, make one branch for this transition
                         Eog(n', Make_ready(t, rdy),[t,t),(t,SI.r])
                         if(a' ≠ b') //And one branch for the low priority job
                             else Eog(n', rdy, [a',b'),[t, SI.r))
                     // The regular succession of the next job (low or high priority)
                     else Eog(n', rdy, [a',b'),[t, SI.r))
}//End
```

Figure 3-5. The Exection Order Graph algorithm.

3.4 *GEX* – the Global EOG

In a distributed system with multiple processing units (and schedules) we generate one EOG for each processing unit (node). From these, a global EOG describing all globally possible execution orderings can be constructed. In the case of perfectly synchronized clocks this essentially amounts to deriving the set of all possible combinations of the scenarios in the individual EOGs. In other cases, we first need to introduce the timing uncertainties caused by the non-perfectly synchronized clocks in the individual EOGs.

Since the progress of local time on each node keeps on speeding up, and slowing down due to the global clock synchronization, the inter-arrival time of the clock ticks vary. The effect is such that for an external observer with a perfect clock the start times and completion times change, as the global clock synchronization algorithm adjusts the speed of the local clocks to keep them within specified limits. Locally in a node, where all events are related to the local clock tick, the effect will manifest itself as a differential in the actual *BCET* and *WCET* from the estimated values. As the inter-arrival time between ticks increases, the *BCET* will decrease because of the possibility to execute more machine code instructions due to the longer duration between two consecutive ticks (see *Figure 3-6*). Likewise the *WCET* increases due to the decrease in the inter-arrival time of ticks.

When scheduling the distributed real-time system it is essential to accommodate for the clock synchronization effects by time-wise separating the completion times of preceding, or mutually exclusive, tasks from the release time of a succeeding task with a factor δ, corresponding to the precision of the global time base.

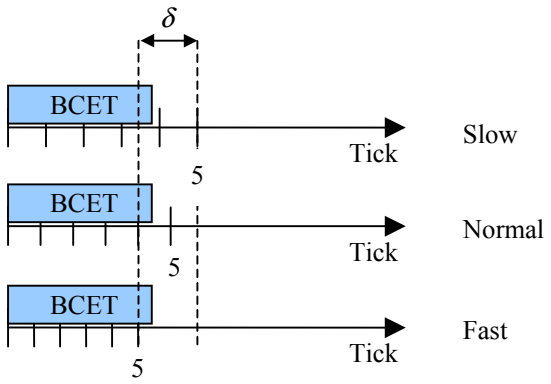


Figure 3-6. The effects of the global clock synchronization on the perceived BCET according to the local tick.

When deriving the execution orderings we need thus change the estimations of the *BCET* and *WCET*:

$$BCET_{new} = \mathrm{MAX}(BCET - \delta/2 * K(BCET), 0) \quad (3\text{-}3)$$

$$WCET_{new} = WCET + \delta/2 * K(WCET) \quad (3\text{-}4)$$

Where the function *K*() is derived from the clock synchronization control loop implementation, and where the argument states how much of the clock synchronization interval is under consideration. The *K()* function substantially increases the precision, since it would be overly pessimistic to add, or subtract, $\delta/2$ when it is possible that the *BCET* and *WCET* will only be fractions of the synchronization interval.

As an illustration, compare the non-adjusted task set in *Table 3-2* and its EOG in *Figure 3-7* with the corresponding adjusted task set and EOG in *Table 3-3*, and *Figure 3-8*.

Table 3-2 The non-adjusted job set for a node in a DRTS, with a clock synchronization precision of $\delta = 4$.

Schedule $\delta = 4$				
Task	*r*	*p*	*BCET*	*WCET*
A	0	1	100	300
B	300	2	100	300

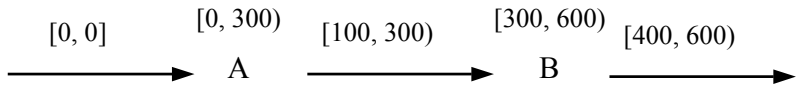


Figure 3-7. The execution ordering for the job set in Table 6-2 without compensating for the precision of the clock synchronization.

Table 3-3 The adjusted job set for a node in a DRTS, with a clock synchronization precision of $\delta = 4$, and K(x) = x.

Schedule $\delta = 4$				
Task	*r*	*p*	*BCET*	*WCET*
A	0	1	98	302
B	300	2	98	302

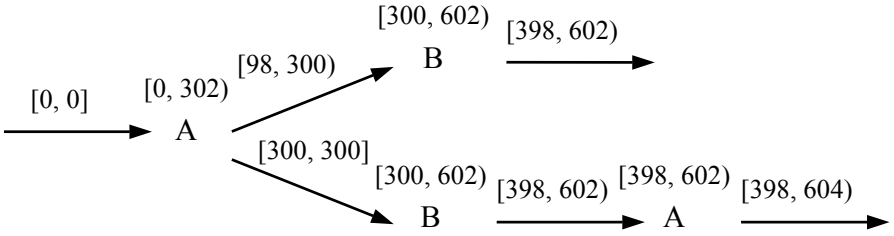


Figure 3-8. The execution ordering for the job set in Table 6-3 where the precision of the clock synchronization has been accounted for.

3.5 Calculating *GEX*

In testing the behavior of a distributed system we must consider the joint execution of a set of local schedules; one on each node. Typically these schedules are of equal length as a result of the global scheduling. However, if they are not, we have to extend each of them by multiples until they equal the global LCM of all task periods in the distributed system.

The next step is to generate EOGs for each node where the effects of global clock synchronization have been accommodated for, as presented above. From these local EOGs we can then calculate the set of global execution order scenarios, *GEX*, which essentially is defined by the product of the individual sets of execution orderings on each node. That is, a set of tuples, $GEX = EX(node_1) \times \ldots \times EX(node_n)$, consisting of all possible combinations of the local execution orderings. This set is however not complete, as illustrated by the following example.

Example 3-2

Assume that we have two nodes, *N1* and *N2*, with corresponding schedules (job sets) *J1* and *J2*, and derived execution orderings of the schedules $EX(J1) = \{q1, q2, q3\}$ and $EX(J2) = \{p1, p2\}$, as illustrated in *Figure 3-9*. Also assume that task *C* is a calculating task in a control application, using sampled values provided by tasks *A*, *B*, and *E*. Assume further that task *G* is another calculating task using values received from tasks *A,* and *E***.** Tasks *C* and *G* receives all inputs immediately at their

respective task starts, and tasks *A*, *B*, *E* all deliver outputs immediately prior to their termination. The resulting set *GEX*, is then equal to {(*q1,p1*), (*q1,p2*), (*q2,p1*), (*q2,p2*), (*q3,p1*), (*q3,p2*)}.

The set *GEX* = {(*q1,p1*), (*q1,p2*), (*q2,p1*), (*q2,p2*), (*q3,p1*), (*q3,p2*)}, is however not complete. Analyzing the data dependencies between task *C* on node *N1* and task *E* on node *N2*, we see that task *C* can receive data from different instances of task *E*. The same goes for task *G*, on node *N2*, with respect to task *A*, on node *N1*.

For example, during the global scenario (*q1, p1*), task *C* has a varying start time of [80,100) and within that interval task instance E_3, has a completion interval of [85,90]. This means, if we assume communication latencies in the interval [3,6), that task *C* can receive data from task instance E_2 or E_3. In order to differentiate between these scenarios we should therefore substitute the global ordering (*q1,p1*) with the set ($q1{:}C(E_2)$, *p1*) and ($q1{:}C(E_3)$, *p1*). Where "($q1{:}C(E_2)$, *p1*)" means that during scenario (*q1,p1*), task *C* receives data from task *E*, and that the originating instance of task *E* is of significance.

For task *G*, assuming the same communication latencies, this means that it can receive data from task instance A_0 or A_1, yielding a new set of scenarios, e.g., scenario (*q1,p1*) is substituted with (*q1, p1*:$G(A_0)$) and (*q1, p1*:$G(A_1)$). These transformations necessitates that we, in addition to recording the execution orderings during runtime, need to tag all data such that we can differentiate between different instances, corresponding to different originators.

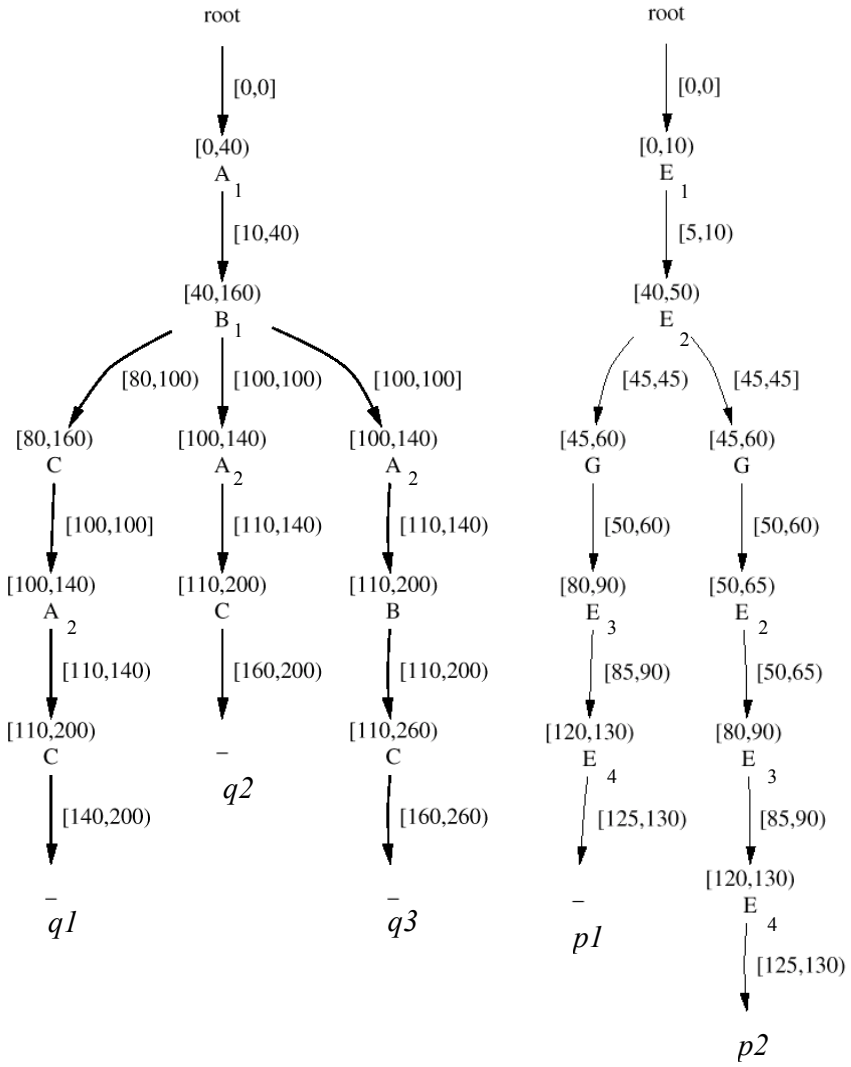


Figure 3-9. The exection orderings for two nodes. The possible global execution orderings are the possible combinations of the individual orderings. for the precision of the clock synchronization.

From the viewpoint of node *N1* the transformed set would look like:

$\{(q1{:}C(E_2), p1), (q1{:}C(E_3), p1), (q1{:}C(E_2), p2), (q1{:}C(E_3), p2),$
$(q2{:}C(E_3), p1), (q2{:}C(E_4), p1), (q2{:}C(E_3), p2), (q2{:}C(E_4), p2),$
$(q3{:}C(E_3), p1), (q3{:}C(E_4), p1), (q3{:}C(E_3), p2), (q3{:}C(E_4), p2)\}$.

From the viewpoint of node *N2* the transformed set would look like:

$\{(q1, p1{:}G(A_0)), (q1, p1{:}G(A_1)), (q1, p2{:}G(A_0)), (q1, p2{:}G(A_1)),$
$(q2, p1{:}G(A_0)), (q2, p1{:}G(A_1)), (q2, p2{:}G(A_0)), (q2, p2{:}G(A_1)),$
$(q3, p1{:}G(A_0)), (q3, p1{:}G(A_1)), (q3, p2{:}G(A_0)), (q3, p2{:}G(A_1))\}$

That is, the new transformed execution orderings are:

- $EX(J1)=\{q1{:}C(E_2), q1{:}C(E_3), q2{:}C(E_3), q2{:}C(E_4), q3{:}C(E_3), q3{:}C(E_4)\}$
- $EX(J2)=\{ p1{:}G(A_0), p1{:}G(A_1), p2{:}G(A_0), p2{:}G(A_1)\}$

The new transformed set $GEX = EX(J1) \times EX(J2)$ becomes:

$\{(q1{:}C(E_2), p1{:}G(A_0)), (q1{:}C(E_3), p1{:}G(A_0)), (q1{:}C(E_2), p2{:}G(A_0)), (q1{:}C(E_3), p2{:}G(A_0)),$
$(q2{:}C(E_3), p1{:}G(A_0)), (q2{:}C(E_4), p1{:}G(A_0)), (q2{:}C(E_3), p2{:}G(A_0)), (q2{:}C(E_4), p2{:}G(A_0)),$
$(q3{:}C(E_3), p1{:}G(A_0)), (q3{:}C(E_4), p1{:}G(A_0)), (q3{:}C(E_3), p2{:}G(A_0)), (q3{:}C(E_4), p2{:}G(A_0)),$
$(q1{:}C(E_2), p1{:}G(A_1)), (q1{:}C(E_3), p1{:}G(A_1)), (q1{:}C(E_2), p2{:}G(A_1)), (q1{:}C(E_3), p2{:}G(A_1)),$
$(q2{:}C(E_3), p1{:}G(A_1)), (q2{:}C(E_4), p1{:}G(A_1)), (q2{:}C(E_3), p2{:}G(A_1)), (q2{:}C(E_4), p2{:}G(A_1)),$
$(q3{:}C(E_3), p1{:}G(A_1)), (q3{:}C(E_4), p1{:}G(A_1)), (q3{:}C(E_3), p2{:}G(A_1)), (q3{:}C(E_4), p2{:}G(A_1))\}$.

3.5.1 *GEX* data dependency transformation

We will now more formally define these transformations. Assume the following auxiliary functions:

- *Start*(*j*), returns the interval of start times for job *j*, (task instance)
- *Finish*(*j*), returns the interval of finishing times for job *j*.
- *ComLatency*(*i*,*j*), returns the communication latency interval for data sent by job *i* to job *j*.
- *Precede*(*j*), returns the immediately preceding instance of the same task as job *j*. Note that this may be a job from the previous LCM.

Data dependency transformation definition

We transform the local execution orderings, $EX(node)$, with respect to the set of global execution orderings, $GEX = EX(node_1) \times \ldots \times EX(node_n)$:

1. For each tuple $l \in GEX$.
    2. For each ordering $x \in EX(n)$, where x is an element in tuple l, and where $K(x)$ is the set of jobs in x consuming data originating from the set of jobs, P, belonging to the other execution ordering elements of l.
        3. For each job $j \in K(x)$.
            4. For each subset $Pt \subseteq P$ representing different instances of the same task producing data for job j.
                5. Let Q be the set of jobs in Pt with finishing times + communication delays, in the start time interval of job j, i.e., $Q=\{u \mid start(j) \cap (Finish(u) + ComLatency(u,j)) \neq \emptyset\}$ The elements in the set can be ordered by earliest finishing time. We use q to denote the earliest job in Q.
                6. If $Q \neq \emptyset$ substitute x. That is,

                $EX(n) = EX(n) \backslash x + x{:}\, j(Precede(q)) + (\forall u \in Q \mid x{:}j(u))$.
        7. Substitute all $x{:}j(y) \in EX(n)$ where $j \in K(x)$ and $y \in P$ such that we for the scenario x, construct a substitute of all possible combinations of $\{x{:}j(y_m), x{:}j'(y'_{m'}),\ldots, x{:}j''(y''_{m''})\} \subseteq EX(n)$ for consuming jobs, $\{j,\ldots,j''\} \subseteq K(x)$ not belonging to the same task instance, i.e., combinations $x{:}j(y_m){:}j'(y'_{m'}){:}\ldots{:}j''(y''_{m''})$ in order of start time for jobs consuming data produced by jobs $\{y_m,\ldots,y''_{m''}\} \subseteq P$.
8. New transformed $GEX = EX(node_1) \times \ldots \times EX(node_n)$

Example 3-3

Assume the same system as in *Example 3-2*, where the initial $GEX = \{ (q1,p1) , (q1,p2), (q2,p1), (q2,p2) , (q3,p1), (q3,p2) \}$.

The first tuple would be $l = (q1,p1)$, the consuming task set in scenario $q1$ would be $K(q1) = \{C\}$, the producing set of jobs for $K(q1)$ would be $Pt=P=\{E_{old},E_1, E_2, E_3, E_4\}$, and the set of jobs Q that has a chance of producing data for the sole job in $K(q1)$, C, is $Q=\{ E_2, E_3\}$. We thus substitute $q1$ with $q1{:}C(E_2)$ and $q1{:}C(E_3)$.

Likewise we can transform all sets:

- $EX(J1)=\{q1{:}C(E_2), q1{:}C(E_3), q2{:}C(E_3), q2{:}C(E_4), q3{:}C(E_3), q3{:}C(E_4)\}$
- $EX(J2)=\{ p1{:}G(A_0), p1{:}G(A_1), p2{:}G(A_0), p2{:}G(A_1)\}$

The transformed global execution ordering set GEX equals the final set described in *Example 3-2*.

4 THE TESTING PROCEDURE

We will now outline a method for deterministic integration testing of distributed real-time systems, based on the identification of execution orderings, as presented above. If we are only interested in addressing failures that tasks can experience in solitude we could make use of regular unit testing [2], and regard each task as a sequential program. However, if we want to take into account the effects that the tasks may have on each other, we need to do integration testing. In any case, we assume that some method for testing of sequential programs is used.

4.1 Assumptions

In order to perform integration testing of distributed real-time systems we require the following:

- A feasible global schedule, including probes that will remain in the target system in order to eliminate the probe effect (as discussed in the introduction).

- Kernel-probes on each node that monitors task-switches. This information is sent to dedicated tasks (probe-tasks) that identify execution orderings from the task-switch information and correlate it to run test cases [25].
- A set of in-line probes that instrument tasks, as well as probe-nodes, which output and collect significant information to determine if a test run was successful or not. This also includes auxiliary outputs of tasks' state if they keep state between invocations [25].
- All probes are left in the target system in order to avoid the probe-effect.
- Control over, or at least a possibility to observe, the input data to the system with regard to contents, order, and timing [25].

4.2 Test Strategy

The test strategy is illustrated in *Figure 4-1* and consists of the following steps:

1) Identify the set of execution orderings by performing execution order analysis for the schedule on each node. This includes compensation for the global clock synchronization jitter, and the global LCM, as introduced in section 3.4.
2) Test the system using any testing technique of choice, and monitor for each test case and node, which execution ordering is run during the interval [0, T^{MAX}]. Where T^{MAX} typically equals the global LCM in the distributed real-time system case.
3) Map the test case and output onto the correct execution ordering, based on observation.
4) Repeat 2-3 until required coverage is achieved.

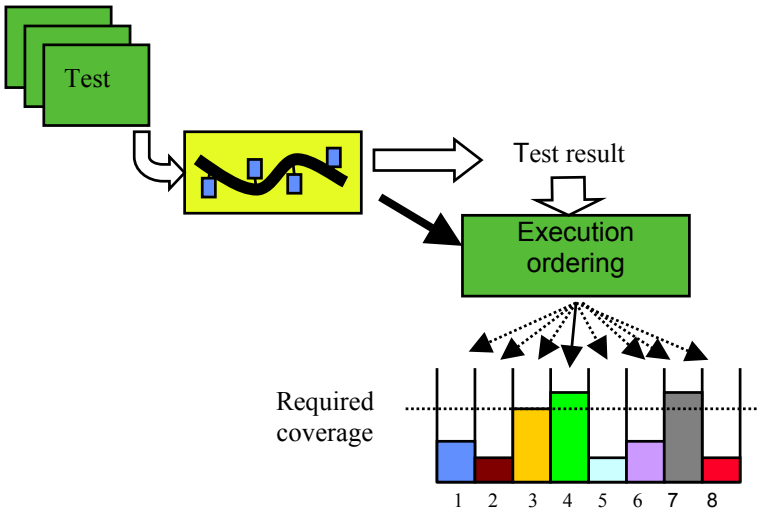


Figure 4-1. The test strategy and testing procedure. The numbers signify the different execution orderings, and the bars the achieved coverage for each ordering.

4.3 Coverage

In order to establish a certain level of confidence in the correctness of a system, we need to define coverage criteria, i.e., criteria on how much testing is required. This is typically defined in terms of the fraction of program paths tested [7][28]; paths, which in the multi-tasking/distributed scenarios we consider, are defined by the execution order graphs. The derived execution orderings also provide means for detecting violations of basic assumptions during testing, e.g., the execution of a scenario not defined by the EOG may be caused by an exceeded worst case execution time.

Complete coverage for a single node is defined by the traversal of all possible execution order scenarios, as illustrated in *Figure 4-2*. The coverage criteria for each scenario is however defined by the sequential testing method applied.

Complete coverage for the distributed system would naively be all combinations of all local execution order scenarios (as introduced in section 3.4). Depending on the design, many of the combinations would however be infeasible due to data dependencies, with the consequence that a certain scenario on one node always gives a specific scenario on another node. Reductions could thus be possible by taking these factors into account. Other reductions could also be facilitated by limiting the scope of the tests, e.g., the entire system, multiple transactions (running over several nodes), single transactions (running over several nodes), multiple transactions (single node), single transactions (single node) or parts of transactions. These issues are however outside the scope of this article, but will definitely be considered in future work. It should also be noted that the correspondence between observations on each node will be limited by the granularity of the global time base, i.e., the globally observable state can at best have a granularity of 2δ, defined by the precision of the clock synchronization [25][24].

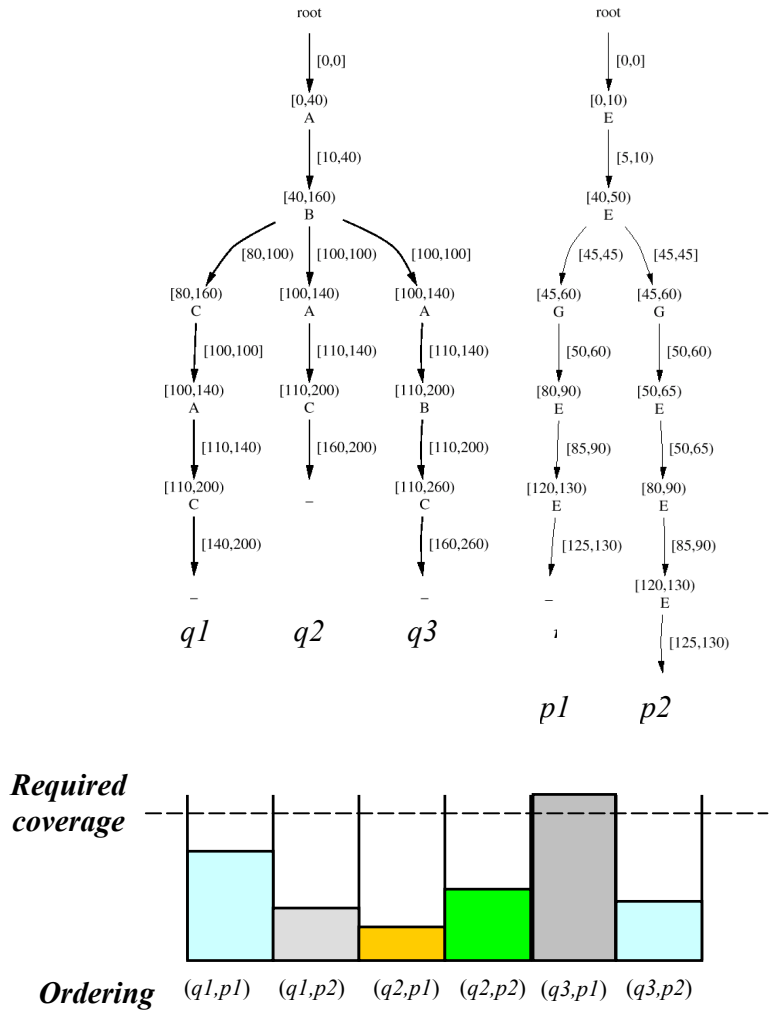


Figure 4-2. The test strategy and testing procedure for a DRTS. The orderings represent the different combinations of the local execution orderings. The bars represent the achieved coverage for each global execution ordering.

4.4 Reproducibility

To facilitate reproducible testing we must identify which execution orderings, or parts of execution orderings that can be enforced without introducing any probe effect. From the perspective of a single transaction, this can be achieved by controlling the execution times of preceding and preempting jobs that belong to other transactions. This of course only works in its entirety under our fault hypothesis, which assumes that the jobs have no unwanted functional side effects via unspecified interfaces, otherwise we could miss such errors. Control over the execution times in other transactions can easily be achieved by incorporating delays in the jobs, or running dummies, as long as they stay within each job's execution time range [*BCET*, *WCET*].

For example, consider *Figure 4-3*, and assume that task *C* and *A* belong to one transaction, and tasks *B*, *D* to another transaction. Assume that task *C* uses the last five samples provided by task *A*. With respect to tasks *A* and *C* we can reproduce the different scenarios by running a dummy in place of task *B*. By varying the execution time of the dummy we can enforce the different scenarios.

$[B_{BCET}, B_{WCET}]$:

1) [39,60)
2) [60,60]
3) [121,121]
4) (60,121)
5) (60,121)

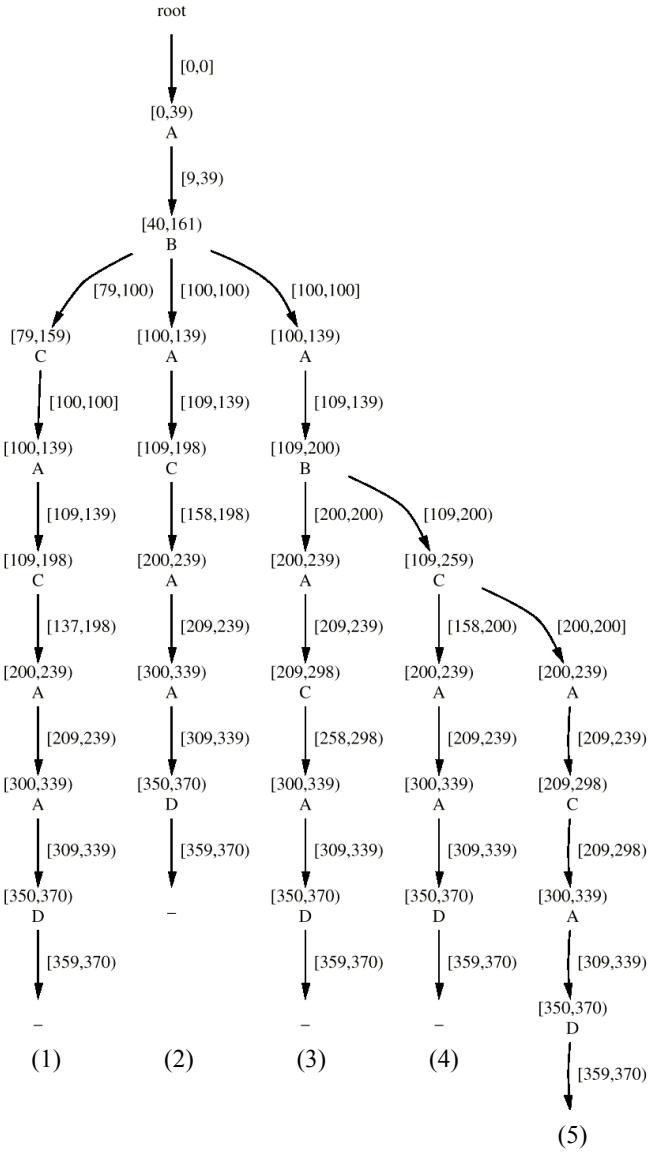


Figure 4-3. The resulting execution order scenarios for the job set in Table 3-1.

5 OTHER ISSUES

We will now outline some specifics of the execution order analysis with respect to jitter, scheduling, testability, and complexity.

5.1 Jitter

In defining the EOG, and in the presented algorithms, we take the effects of several different types of jitter into account:

- *Execution time jitter*, i.e., the difference between *WCET* and *BCET* of a job.
- *Start jitter*, i.e., the inherited and accumulated jitter due to execution time jitter of preceding higher priority jobs.

- *Clock synchronization jitter,* i.e., the local clocks keep on speeding up, and down, trying to comply with the global time base, leading to varying inter-arrival times between clock ticks.
- *Communication latency jitter*, i.e., the varying data transmission times of data passed between nodes in a DRTS, due to arbitration, propagation, etc.

Since any reduction of the jitter reduces the preemption and release intervals, the preemption "hit" windows decrease and consequently the number of execution order scenarios decreases. Suggested actions for reducing jitter is to have fixed release times, or to force each job to always maximize its execution time, e.g. by inserting (padding) "no operation" instructions where needed.

Even if the *BCET* = *WCET* for all tasks there will still be jitter if interrupts are interfering [20][25]. Considering the hypothetical situation that we had no jitter what so ever on the nodes locally, then there could still be plenty of different execution orderings if the communication medium contributed to communication latency jitter.

In general, real-time systems with tighter execution time estimates and WCET ≈ BCET, as well as low communication latency jitter, and better precision of the clock synchronization yield fewer execution orderings than systems with larger jitter.

5.2 Start times and completion times

An interesting property of the EOG is that we can easily find the best and worst case completion-times for any job. We can also identify the best and worst case start times of jobs, i.e., the actual start times of jobs not their release times as dictated by the schedule. For identification of the completion times we only have to search the EOG for the smallest and largest finishing times for all terminated jobs. The response time jitter (the difference between the maximum and minimum response times for a job) can also be quantified both globally for the entire graph and locally for each path, as well as for each job during an LCM cycle. The same goes for start times of jobs. The graph can thus be utilized for schedulability analysis of strictly periodic fixed priority scheduled systems with offsets.

5.3 Testability

The number of execution orderings is an objective measure of system testability, and can thus be used as a metric for comparing different designs, and schedules. It would thus be possible to use this measure, for feedback, or as a new optimization criterion in the heuristic search used for generating static schedules with fewer execution orderings, with the intention of making the system easier to test. We could also use the EOG, as a means for judging which tasks' execution times should be altered in order to minimize the number of execution orderings.

5.4 Complexity

The complexity of the EOG, i.e., the number of different execution orderings, and the computational complexity of the EOG algorithm (section 3.3), is a function of the scheduled set of jobs, *J*, their preemption pattern, and their jitter. From an $O(n)$ number of operations for a system with no jitter which yields only one scenario, to exponential complexity in cases with large jitter and several preemption points. In the latter case the number of scenarios is roughly 3^{X-1} for X consecutively preempting jobs (as illustrated in *Figure 5-1*), given that each preemption gives rise to the worst case situation of three possible transitions (see section 3.2).

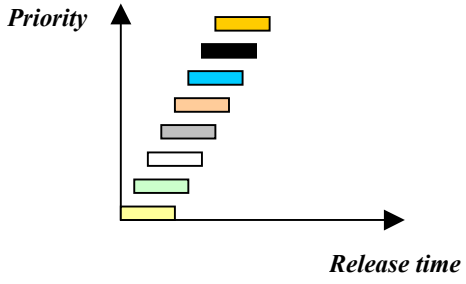


Figure 5-1. Consecutively preempting jobs.

This complexity is not inherent to the EOG but rather a reflection of the system it represents. In generating the EOG we can be very pragmatic, if the number of execution order scenarios exceeds, say a 100, we can terminate the search, and use the derived scenarios as input for revising the design. If the system has no preemption points, but large jitter there will only be one scenario. If the system has plenty of preemption points but no jitter, there will be only one scenario. If the system has plenty of preemption points, and jitter, but the magnitude of the jitter or the distribution of the preemption points are such that they will not give rise to different execution orderings there would be just one scenario. Knowing this, we can be very pragmatic in generating the EOG and try to tune the system in such a way that the jitter is minimized. Alternatively, release-times of tasks can be offset in such a way that they eliminate forks in the EOG (as long as the schedule or requirements allow).

5.5 Event triggered vs. Time triggered

In this paper we do not handle sporadic tasks or interrupts, we have however presented results on this topic in [25][20]. The choice of not trying to model interrupt interference as high priority jobs is the choice of not modeling jobs with arbitrary arrival times. If we were interested in addressing the functional side effects of the interrupts, we would have to model each interrupt as a job. This would force us to consider all possible release times (with a finer granularity than provided by the real-time kernel clock tick) which would result in an (in practice) infinite number of execution orderings. This is exactly why we cannot model interleaving failures, i.e., arbitrary memory corruption failures due to e.g., non-reentrant code deterministically, since it would require us to model all possible preemption points corresponding to each machine code instruction. This could be viewed as a critique against the EOG method but also as a critique against the testability of event triggered systems (systems where the release times of tasks are arbitrary). The complexity and the testability of a system is thus strictly dependent on the infrastructure and the assumed failure semantics. That is, with our fault hypothesis the number of execution orderings would approach infinity if tasks with arbitrary arrival times were allowed. In [25][22] we elaborate on this issue and present a technique using on-line recording and off-line replay of task interleaving that deterministically can reproduce interleaving failures.

6 CONCLUSIONS

We have in this paper introduced a novel method for deterministic testing of multitasking and distributed real-time systems. We have specifically addressed task sets with recurring release patterns, executing in a distributed system with a globally synchronized time base, and where the scheduling on each node is handled by a priority driven preemptive scheduler. The results can be summed up to:

- We have provided a technique for finding all the possible execution scenarios for a DRTS with preemption and jitter.
- We have proposed a testing strategy for deterministic and reproducible testing of DRTS.
- A benefit of the testing strategy is that it allows any testing technique for sequential software to be used to test DRTS.
- Jitter increases the number of execution order scenarios, and jitter reduction techniques should therefore be used whenever possible to increase testability.
- The number of execution orderings is an objective measure of the testability of DRTS, which can be used as a new scheduling optimization criterion for generating schedules that are easier to test, or simply make it possible to compare different designs with respect to testability.

- We can use the execution order analysis for calculating the exact best-case and worst-case response-times for jobs, as well as calculating response-time jitter of the jobs. We could thus make use of the execution order analysis for schedulability analysis of strictly periodic fixed priority scheduled systems with offsets.

With respect to the results presented in this paper we suggest the following extensions and pursuits in the future:

- Experimentally validate the usefulness of the presented results. This pursuit will hopefully be facilitated by the use of the real-time kernel Asterix [23] that has specifically been developed to give the necessary support for monitoring, debugging and testing.
- Extend the execution ordering analysis to also encompass critical regions, by for example assuming immediate inheritance protocols for resource sharing. This analysis would however necessitate assumptions about the duration of the critical regions (intervals), and assumptions about the intervals of delay before entering the critical regions. Knowing these parameters we believe it is straightforward to extend the execution ordering analysis.
- Complete coverage for the distributed system would naively be all combinations of all local execution order scenarios (as introduced in section 3.4). Depending on the design, many of the combinations would however be infeasible due to data dependencies, with the consequence that a certain scenario on one node always gives a specific scenario on another node. Reductions that take these factors into account should be further investigated. One possibility to reduce the required coverage could be to perform parallel composition of the individual EOGs, using standard techniques for composing timed transition systems [19], this would probably reduce the pessimism introduced in the GEX data dependency transformation.

7 REFERENCES

[1] Audsley N. C., Burns A., Davis R. I., Tindell K. W. Fixed Priority Pre-emptive Scheduling: A *Historical Perspective.* Real-Time Systems journal, Vol.8(2/3), March/May, Kluwer A.P., 1995.

[2] Beizer B. *Software testing techniques.* Van Nostrand Reinhold, 1990.

[3] Chen J. and Burns A. *Asynchronous Data Sharing in Muliprocessor Real-Time Systems Using Process Consensus*. 10th Euromicro Workshop on Real-Time Systems, June 1998,

[4] Eriksson C., Mäki-Turja J., Post K., Gustafsson M., Gustafsson J., Sandström K., and Brorsson E. *An Overview of RTT: A design Framework for Real-Time Systems.* Journal of Parallel and Distributed Computing, vol. 36, pp. 66-80, Oct. 1996.

[5] Eriksson C., Thane H. and Gustafsson M. *A Communication Protocol for Hard and Soft Real-Time Systems.* In the proceedings of the 8th Euromicro Real-Time Workshop, L'Aquila Italy, June, 1996.

[6] Gait J. *A Probe Effect in Concurrent Programs*. Software – Practice and Experience, 16(3):225-233, Mars, 1986.

[7] Hwang G.H, Tai K.C and Huang T.L. *Reachability Testing: An Approach to Testing Concurrent Software.* Int. Journal of Software Engineering and Knowledge Engineering, vol. 5(4):493-510, 1995.

[8] Kopetz H. and Grünsteidl H. *TTP - A Protocol for Fault-Tolerant Real-Time Systems*. IEEE Computer, January, 1994.

[9] Kopetz H. and Ochsenreiter W. *Clock Synchronisation in Distributed Real-Time Systems*. IEEE Trans. Computers, 36(8):933-940, Aug. 1987.

[10] Kopetz H., Damm A., Koza Ch., Mulazzani M., Schwabl W., Senft Ch., and Zainlinger R.. *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*. IEEE Micro, (9):25-40, 1989.

[11] Kopetz H. and Reisinger J. *The Non-Blocking Write Protocol NBW: A Solution to a Real- Time Synchronization Problem*. In Proceedings of he 14th Real-Time Systems Symposium, pp. 131-137, 1993.

[12] Laprie J.C. *Dependability: Basic Concepts and Associated Terminology*. Dependable Computing and Fault-Tolerant Systems, vol. 5, Springer Verlag, 1992.

[13] Lui C. L. and Layland J. W.. *Scheduling Algorithms for multiprogramming in a hard real-time environment*. Journal of the ACM 20(1), 1973.

[14] McDowell C.E. and Hembold D.P. *Debugging concurrent programs*. ACM Computing Surveys, 21(4), pp. 593-622, December 1989.

[15] Rushby J. *Formal methods and their Role in the Certification of Critical Systems*. Proc. 12th Annual Center for Software Reliability Workshop, Bruges 12-15 Sept. 1995, pp. 2-42. Springer Verlag. ISBN 3-540-76034-2.

[16] Rushby J., *Formal Specification and Verification for Critical systems: Tools, Achievements, and prospects*. Advances in Ultra-Dependable Distributed Systems. IEEE Computer Society Press. 1995. ISBN 0-8186-6287-5.

[17] Sandström K., Eriksson C., and Fohler G. *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*. In proceedings of the 5th Int. Conference on Real-Time Computing Systems and Applications (RTCSA'98). October 1998, Japan.

[18] Schütz W. *Fundamental Issues in Testing Distributed Real-Time Systems*. Real-Time Systems journal, vol. 7(2): 129-157, Kluwer A.P., 1994.

[19] Sifakis J. and Yovine S. *Compositional specification of timed systems.* In Proc. 13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96, Grenoble, France, Lecture Notes in Computer Science 1046, pp. 347-359. Springer Verlag, February 1996.

[20] Thane H. and Hansson H. *Handling Interrupts in Testing of Distributed Real-Time Systems*. In proc. Real-Time Computing Systems and Applications conference (RTCSA'99), Hong Kong, December, 1999.

[21] Thane H. and Hansson H. *Towards Systematic Testing of Distributed Real-Time Systems*. Proc. 20th IEEE Real-Time Systems Symposium, Phoenix, Arizona, December 1999.

[22] Thane H. and Hansson H. *Using Deterministic Replay for Debugging of Distributed Real-Time Systems*. In proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS'00), Stockholm, June 2000.

[23] Thane H. *Asterix the T-REX among real-time kernels. Timely, reliable, efficient and extraordinary*. Technical report in preparation, Mälardalen Real-Time Research Centre, Dept. Computer Engineering, Mälardalen University, *2000*

[24] Thane H. *Design for Deterministic Monitoring of Distributed Real-Time Systems.* Technical report, Mälardalen Real-Time Research Centre, Dept. Computer Engineering, Mälardalen University, 1999.

[25] Thane H. *Monitoring, Testing and Debugging of Distributed Real-Time Systems.* Ph.D. Thesis. Royal Institute of Technology (KTH), Stockholm, Sweden, May 2000. www.mrtc.mdh.se.

[26] Tindell K. W., Burns A., and Wellings A.J. *Analysis of Hard Real-Time Communications.* Journal of Real-Time Systems, vol. 9(2), pp.147-171, September 1995.

[27] Xu J. and Parnas D. *Scheduling processes with release times, deadlines, precedence, and exclusion, relations*. IEEE Trans. on Software Eng. 16(3):360-369, 1990.

[28] Yang R-D and Chung C-G. *Path analysis testing of concurrent programs*. Information and software technology. vol. 34(1), January 1992.