Integration Testing of Fixed Priority Scheduled Real-Time Systems

Henrik Thane, Anders Pettersson, and Hans Hansson
{hte,apo,han}@mdh.se
Mälardalen Real-Time Research Center, Department of Computer Engineering,
Mälardalen University, Västerås, Sweden

Abstract-- **In order to achieve reproducible and deterministic functional integration testing of real-time systems software it is essential to consider inputs, outputs, and the order in which the tasks communicate and synchronize with each other. In this paper we present a method for deterministic integration testing of strictly periodic fixed priority scheduled real-time systems. Essentially it is a structural white box testing method applied on the system level rather than the individual programs of the tasks. The method includes a reachability technique for deriving all possible orderings of task starts, preemptions and completions for tasks executing in a system where synchronization is resolved using the Priority Ceiling Emulation Protocol (PCEP) or offsets. The method also includes a structural testing strategy for achieving full coverage with respect to the derived execution orderings. The testing strategy also allow test methods for sequential programs to be applied, since each identified ordering can be regarded as a sequential program. In the presented analysis and testing strategy, we consider task sets with recurring release patterns, and take into account the effects of variations in start and execution times of the involved tasks, as well as the variations of the arrival and duration of the critical sections.**

Index Terms—Testing, integration testing, real-time systems, test coverage, priority ceiling emulation protocol, fixed priority scheduling, reachability analysis, determinism

1 INTRODUCTION

In this paper we extend the method for achieving deterministic testing of distributed real-time systems by Thane and Hansson [14][13]. They addressed task sets with recurring release patterns, executing in a distributed system, where the scheduling on each node was handled by a fixed priority driven preemptive scheduler supporting offsets. The method transformed the non-deterministic distributed real-time systems testing problem into a set of deterministic sequential programs testing problems. This was achieved by deriving all the possible execution orderings of the individual nodes in the distributed real-time system, and regarding each ordering as a sequential program. Full test coverage (if needed) was then achieved by testing all obtained scenarios (using monitoring during run-time) until a required level of *secondary* coverage was achieved. For each individual scenario/program, the secondary coverage criteria were defined by the testing technique applied, e.g., white or black box testing.

This method assumed that all synchronization was resolved offline, e.g., by a static scheduler, which assigned offsets and priorities to all tasks in the distributed system. That is, general use of semaphores was not allowed. All tasks in the system were also assumed to receive all input immediately at their start, and to produce all output at their termination. These limitations were quite severe, although the analysis proved that even statically scheduled systems could yield enormous numbers of different scenarios, when subjected to preemption and jitter (execution time-, communication time-, and interrupt induced jitter), especially when the system is of multi-rate character.

In this paper we elaborate on the approach presented in [14][13] and expand the task model to also include critical sections, governed by the Priority Ceiling Emulation Protocol (PCEP) [2], a.k.a. the immediate inheritance protocol and immediate priority ceiling protocol. Since tasks may synchronize/communicate via critical sections, we will also relax Thane's and Hansson's input output assumption. Our extension is however only valid for the individual nodes in the distributed real-time system, unless we assume a global PCEP, which is quite complex to achieve [10]. The subsequent analysis in this paper is hence focused on a single node. The results by Thane and Hansson [13][14] on how to derive *global* execution ordering scenarios can however successfully be applied if global scheduling is relying on offsets between tasks on different nodes, but this is outside the scope of this paper.

1.1 The problem of testing real-time software

Reproducible and deterministic testing of sequential programs can be achieved by controlling the sequence of inputs and the start conditions [9]. 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 [11]. 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 [7]. However, trying to directly apply test techniques for

This work is founded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Stratetgic Research.

sequential programs on 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 real-time system can in addition incur a temporal probe-effect [4] that subsequently will affect the temporal and functional behavior of the system.

In theory it is possible to reproduce the behavior of a real-time system if we can reproduce the exact trajectories of the inputs to the system with an exact timing. The inputs, and state, of the tasks dictates the control flow paths taken, which in turn dictates the execution time of the tasks, which in the end dictates the preemption pattern (for strictly periodic systems). Trying to perform exhaustive black-box testing of individual programs is in the general case infeasible, due to the large number of possible inputs, e.g., two 32 bit inputs yields 2^{64} possible input combinations, not considering state. For a multitasking real-time system the number of possible inputs is similarly bordering on the ludicrous. However, just as individual program's control flow structure can be derived and used for white-box testing of the control-flow paths (which usually are significantly fewer than the number of possible inputs), we can for a set of multi-tasking real-time tasks test the different interleavings of task executions on the system level.

1.2 Contribution

The contribution of this paper is a white-box system level integration testing method that includes:

- A white-box reachability technique for deriving all possible orderings of task starts, preemptions and completions for tasks executing in a system where synchronization is resolved using offsets or the priority ceiling emulation protocol (PCEP).
- A testing strategy for achieving any required level of coverage, with respect to the derived execution orderings. The testing strategy also allows test methods for sequential programs to be applied, since each identified ordering can be regarded as a sequential program.

The results in this paper substantially extends the applicability of the results by Thane and Hansson [14][13], since we now can handle systems with on-line synchronization, for which it is actually more likely that errors have been caused by implementation and synchronization problems. Also, PCEP has been adopted in industry standards like POSIX, ADA95, and OSEK, for its implementation simplicity [12][6].

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. Section 4 presents a testing strategy for deterministic full coverage testing. Section 5 discusses the relation between jitter and testability. Finally, in Section 6, we conclude and give some hints on future work.

2 The system model

We assume that the real-time systems software consists of a set of concurrent tasks, communicating by message passing or shared memory. All synchronization, precedence or mutual exclusion, is resolved either offline by assigning different release times and priorities, or during runtime by the use of semaphores which have Priority Ceiling Emulation Protocol semantics (PCEP) [2].

2.1 Task model

We assume a fairly general task model that includes both preemptive scheduling of statically generated schedules [18] and fixed priority scheduling of strictly periodic tasks [1][8]:

- 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 such that it prevents temporal interference between successive T^{MAX} intervals. To simplify the presentation we will henceforth assume an idle point at 0.
- 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 unique base priority bp_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, $r_j + T^{MAX}$, $r_j + 2\,T^{MAX}$, etc.
- The system is preemptive and jobs may have identical release times.

2.2 Synchronization using PCEP

For PCEP we assume that:

- Each job $j \in J$ has a current priority p_i that may be different from the statically allocated base priority, bp_j, if the job is subject to priority promotion when granted a resource.
- Each resource R, used by a set of jobs S_R, has a statically computed priority ceiling defined by the highest base priority in S_R increased by one, i.e., $p_R = \mathrm{MAX}(bp_i \mid i \in S_R) + 1$.
- Each job, j, that enters a critical section protecting a resource R is immediately promoted to the statically allocated priority ceiling of the resource, if $p_R > p_j$ then $p_j = p_R$.
- Each job, j, that is executing and releases a resource R is demoted immediately to the maximum of the base priority bp_j, and the ceilings of the remaining resources held by the job.
- Each critical section, k, has a worst case execution time ($WCET_k$) and a best case execution time ($BCET_k$).

- Each critical section, *k*, has a release time interval [er_k, lr_k) ranging from the earliest release time to the latest release time.
- All resources are claimed in the same order for all paths through the program in a job.

Note that we here, compared to other ceiling priority models, can take more detailed information of the time when the resources are allocated into account.

2.3 Side effects

Related to the task model we assume that the jobs may have functional and temporal side effects due to preemption, message passing or shared memory.

- Data is sent at the termination of the sending job and received data is available when job start to execute.
- Access to shared memory or I/O is guarded by semaphores, or offsets.

2.4 Fault hypothesis

The 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. This means that interleaving failures, e.g., memory corruptions are not considered. To handle interleaving failures other techniques need to be applied, e.g., deterministic replay debugging [15].

3 Execution Order Analysis

In this section we present a technique for identifying all the possible orders of execution for sets of jobs conforming to the task model of section 2. We first begin with a definition of execution orderings, then continue with a definition of the execution order graph, and finally presents an algorithm that generates this graph.

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 or a critical section, i.e., when the first instruction 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 job that is
    - o Preempted by the start of j_1 at *time*, or possibly the idle job "_" if no j_2 job exists.
    - o Promoted to a higher priority due to the arrival of a critical section, j_1 at *time*, i.e., the same job continues executing, but at higher priority.
    - o Preempted by j_1 when j_2 exits a critical section at *time* and demotes its priority.
- The end of execution of a job or critical section, i.e., when the last instruction 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 job
    - o That is resuming its execution at the termination of higher priority job j_1, or possibly the idle job "_" if no such job exists.
    - o That is demoted to a lower priority when exiting a critical section, j_1, i.e., the same job continues executing, but at lower priority.

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 events 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, or promoted to a higher priority when entering a critical section.
3. For each $j \in J$, $s(j,X).t \geq r_j$, i.e., jobs may only start to execute after being released. 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 priority promotion, due to arrival of a higher priority critical section, i.e. $ev = s(j', X)$ and $p_{j'} > p_j$
    - A priority demotion, due to the exit from a critical section, at which a higher priority job resumes its execution, i.e., $ev = (j', t, j'')$, where $p_{j''} > p_j$
    - A job termination, at which a higher priority job resumes its execution, i.e., $ev = (j', t, j'')$, where $p_{j''} > p_j$
4. The sum of execution intervals of a job $j \in J$ is in the range [*BCET*(*j*), *WCET*(*j*)], i.e.,

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

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 time abstracted 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 $EX(J) = EX_t(J)\backslash\approx$, 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., $EX_t(J)\backslash\sim$, but after having derived this set we can reduce it to $EX_t(J)\backslash\approx$. 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 Deriving the Execution Orderings

This section outlines a technique for deriving the set of execution orderings $EX(J)$ for a set of jobs J, complying with definitions 3-1 and 3-2. We will later (in section 3.3) present an algorithm implementing the technique. In essence, our approach is to make a reachability analysis by simulating the behavior of a real-time kernel conforming to our task model 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, the job's current priority, and a continuous time interval, i.e., for a job set J: $N\subseteq J \cup\{\text{“\_”}\} \times P \times I(T^{MAX})$, where {"_"} is used to denote a node where no job is executing. P is the set of priorities, 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, or when a job enters or leaves a critical section. The edge is annotated with a continuous interval of when the transition can take place, as illustrated in *Figures 3-1* and *3-2*, showing EOGs for simple jobs without critical sections.

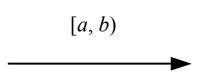

Figure 3-1. A Transition.

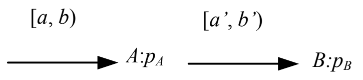

Figure 3-2. Two transitions, one to job A and one from job A to job B.

The interval of possible start times $[a', b')$ for job B, in *Figure 3-2*, is defined by:

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

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 time, $[\alpha,\beta)$, as depicted in *Figure 3-3*.

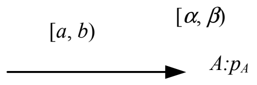

Figure 3-3. A job annotated with its possible execution, start time and current priority.

We define the interval of execution, $[\alpha, \beta)$ as the interval in which job A can be preempted:

$$\alpha = MAX(a, r_A) \quad (3\text{-}2)$$
$$\beta = MAX(b, r_A) + WCET_A$$

3.2.1 Critical sections

Critical sections will be introduced by transforming the job set, such that a job with critical sections is partitioned into a set of jobs corresponding to the different critical sections and executions in between. We assume that each job $i \in J$, which has a set of critical sections CS_i, is split into an ordered list of sub jobs, SJ_i, such that every time there is a change in the job's effective priority a new sub job is added (as illustrated in *Figure 3-4*). Each sub job $s_i \in SJ_i$ of original job i has a release time interval $[er_s, lr_s)$ ranging from its earliest release time to its latest release time. The release time interval for a sub job s_i is given in terms of execution time run by the immediately preceding sub job, q_i, before it enters the critical section represented by sub job s_i, rather than in terms of the system clock tick. This means that all BCETs and WCETs for all sub jobs are calculated such that they represent execution time before entering the immediately succeeding critical section except the last sub job, which runs till termination.

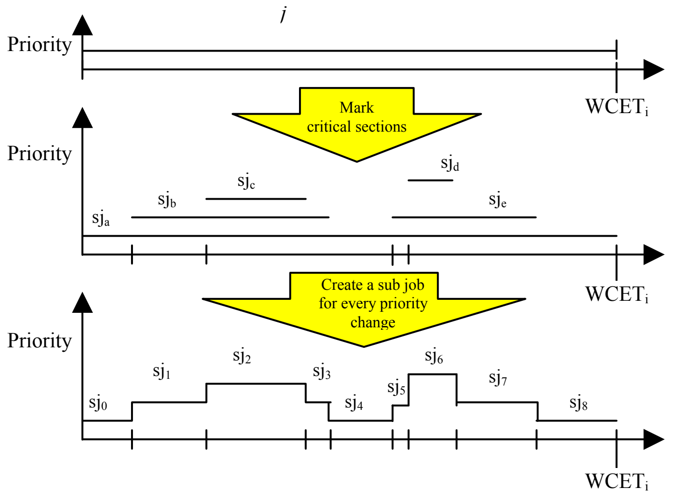

Figure 3-4. A job split into a set of sub jobs, in order of changes in effective priority. The sub jobs sj0, sj4, and sj8 represent the base priority job.

The interval of possible start times [*a'*, *b'*] for the sub job s_i, as illustrated in *Figure 3-5*, is defined relative to its predecessor, q_i, by:

$$a' = MAX(a, r_i) + BCET_q \qquad (3\text{-}3)$$
$$b' = MAX(b, r_i) + WCET_q$$

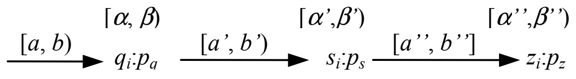

Figure 3-5. Three transitions, one to sub job q_i, one demoting transition from sub job q_i to sub job s_i, and one promoting transition from sub job s_i to sub job z_i.

The MAX function in *Equation 3-3* is needed since the sub job cannot be released earlier than scheduled release of the original job *i*. The transition interval can represent a promoted priority, denoted [*a'*, *b'*], or a demoted priority, denoted [*a'*, *b'*).

A *node* represents a sub job in the same manner as a node represents a job, i.e., the node is annotated with a continuous interval of its possible execution and a priority, in this case the priority ceiling of the critical section.

We define the execution interval, [α', β') for a sub job s_i:

$$\alpha' = MAX(a, r_i) \qquad (3\text{-}4)$$
$$\beta' = MAX(b, r_i) + WCET_s$$

That is, the interval, [α', β'), specifies the interval in which sub job s_i with priority p_s can be preempted by a higher priority job.

3.2.2 *Transitions for a system with no critical sections (base case)*

From each node in the execution order graph there can be one or more transitions, representing one of four different situations (assuming no critical sections) as illustrated by *Figure 3-4*:

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 3-6*.
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 possible modes of transition:
   a. *No high priority job succession*. One single outgoing transition labeled with the interval of finishing times for the job, [*a'*, *b'*). Exemplified by (1) in *Figure 3-6*.
   b. *High priority job succession*. 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 3-6*.
3. The job has a *BCET* such that it definitely is preempted by another job *U*. In this case there is a single outgoing transition labeled with the preemption time r_U, expressed by the interval [r_U, r_U], as exemplified by (3) in *Figure 3-6*.
4. The job has a *BCET* and *WCET* such that it may either complete or be preempted before any preempting job *U* 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 [r_U, r_U], 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'*, r_U) representing the possible completion prior to the release of the higher priority job. Exemplified by (4) in *Figure 3-6*.
   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 [r_U, r_U) representing the completion immediately before the release of the preempting job. Exemplified by (5) in *Figure 3-6*.

3.2.3 *Additional transitions for a system with critical sections*

5. A sub job, *HI*, succeeds a lower priority sub job, *LO*, before the release of any higher priority job, *U*. That is if *b'* < r_U, and $p_{HI} > p_U > p_{LO}$, we have one single outgoing transition labeled with the start interval, [*a'*, *b'*], of the sub job *HI*. Exemplified by (6) in *Figure 3-7*.

6. A sub job, *LO*, succeeded by a higher priority sub job, *HI*, before the release of any higher priority job, *U,* or is preempted by *U*. That is, $a' < r_U < b'$, and $p_{HI} > p_U > p_{LO}$. Then we have two outgoing transitions: one labeled with the possible start interval of the sub job *HI* [a', r_U], and another representing the preemption by *U* at [r_U, r_U]. Exemplified by (7) in *Figure 3-7*.
7. A sub job, *HI,* succeeded by a lower priority sub job, *LO* (if there is one), before the release of any higher priority job, *U*. That is $a' < r_U$. Then, *LO* is entered into the set of ready jobs and then governed by transition rule 4, above.

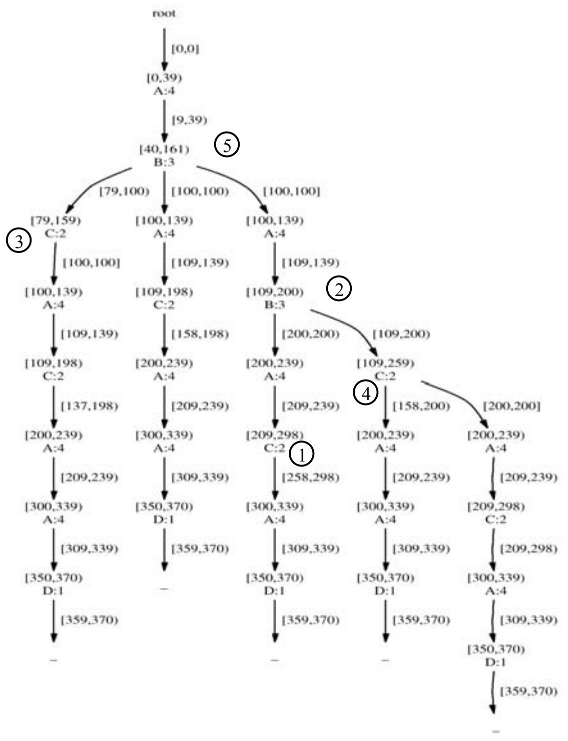

Figure 3-6. The resulting execution order graph for the job set in *Table 3-1.*

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

Example 3-1

Figure 3-6 and Figure *3-7* give examples of EOGs, using the above notation and the attributes in *Tables 3-1* and *3-2* respectively. *Figure 3-7* illustrates the use of critical sections. In *Figure 3-6* and *Figure 3-7*, all paths from the root node to the "_" nodes correspond to the possible execution order scenarios during one instance of the recurring release pattern.

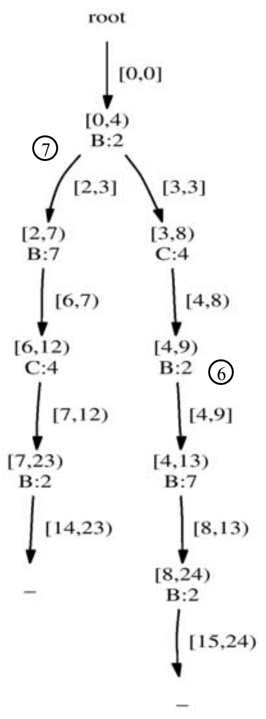

Figure 3-7. The resulting execution order graph for the job set in *Table 3-2.*

Table 3-2. A job set for a schedule where job *B* accesses a shared resource, and when entering the critical section boost its priority to 7. *B* is split into 3 sub jobs.

Task	*r*	*p*	*WCET*	*BCET*
B	0	2	4	2
	-	7	4	4
	-	2	9	7
C	3	4	5	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 $\varnothing$ if $t = \infty$, else the set.
5. X(*rdy*): Extracts the job with highest priority in *rdy*.

```
/* 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(rT, RI.l), max(rT, RI.l)+WCETT)
    a’     = α + BCETT
    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) < pT))
      rdy   =  Make_ready(t, rdy)
      t   = Next_release((t, SI.r])
    /* Does the next scheduled job preempt T? */
    if((pT < P(t)) ∧ (t <  β))
      /* Can T complete prior to the release of the next job at t? */
      if(t > a’)
        /* Enter a critical section? */
        cs_job = Get_nextCS( T )
        if( Is_CS( T ) ∧ Is_lock( T , cs_job  )  )
          Eog(n’, rdy+ {cs_job}, [a’,t], (t,SI.r]) /* Enter */
        else if( Is_CS( T ))
          /* One branch for the next critical section */
          Eog(n’, rdy+ {cs_job}, [a’,t), [t,SI.r]) /* Leave */
    /* One branch for the immediate succession of a higher priority job */
          Eog(n’, Make_ready( t,rdy), [a’,t), [t,SI.r])
        else /* No, T was not a critical section */
          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? */
      BCETT = max(BCETT - (t–(max(rT, RI.l)),0)
      WCETT = max(WCETT - (t–(max(rT, 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 */
          /* Enter a critical section? */
          cs_job = Get_nextCS( T )
          if( Is_CS( T ) ∧ Is_lock( T , cs_job  )  )
            Eog(n’, rdy+ {cs_job}, [a’,t], (t,SI.r]) /* Enter */
          else if(Is_CS( T ))
            Eog(n’, rdy+ {cs_job}, [a’,t), [t,SI.r]) /* Leave */
          else /* No, T was not a critical section */
           /* 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 3-8. The Execution Order Graph algorithm.

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.
8. Get_nextCS(T): Returns the next sub job from an ordered list of sub jobs.
9. Is_CS(T): Determines if job T is a sub job, i.e., a critical section
10. Is_lock(T , cs_job): Determines if the priority of the sub job is promoted.

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 3-8*, 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.

4 THE INTEGRATION TESTING PROCEDURE

The identified execution orderings can be used for determining coverage in integration testing of real-time systems. Our testing method relies on two types of coverage criteria, one defined by the derived execution orderings, and one defined by the actual sequential testing technique applied. In the latter case criteria defined by, e.g., statistical black box testing or structural white box testing [3].

4.1 Test rig assumptions

In order to perform integration testing of an embedded real-time system we require the following: A monitoring mechanism that can extract sufficient information from the target system. This includes, task switches, and inputs-outputs from the jobs. This monitoring mechanism can either be implemented using non-intrusive hardware, intrusive software, or hybrids. If the software approach is chosen then monitors/probes must remain in the target system in order eliminate the probe effect [16][9][4].

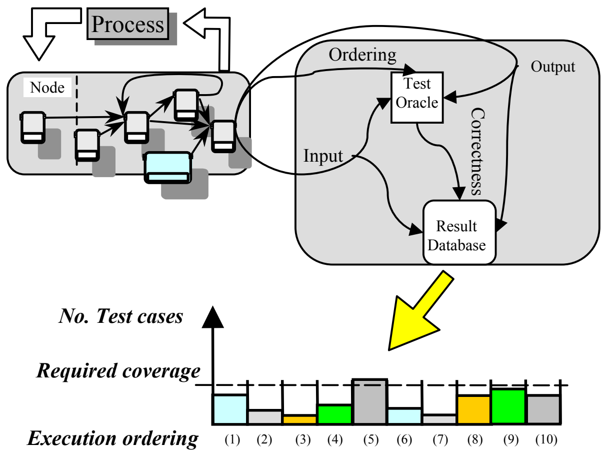

Figure 4-1. The resulting test rig with the coverage of the execution orderings illustrated. Monitoring is implemented in software.

4.2 Test Strategy

The test strategy consists of the following steps:

1. Identify the set of execution orderings by performing execution order analysis of the job set.
2. Test the system using any testing technique of choice, and monitor for each test case, 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.

Complete primary coverage would be to execute all identified execution orderings, complete secondary coverage would be to test all identified orderings with a certain number of unique test-cases defined by the testing method applied. For example, assuming statistical black box testing, with a reliability requirement of 10^{-6} failure/test-case with confidence 0.99, would require (according to probable correctness theory [5]) 4,600,000 test cases per scenario. For the system in *Figure 4-1* with 10 scenarios the total number of test cases would amount to $4{,}600{,}000 \times 10 = 46{,}000{,}000$ test cases.

5 JITTER, TESTABILITY AND COMPLEXITY

We will now outline some specifics of the execution order analysis with respect to jitter, scheduling, testability, and complexity. 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, or a critical section.
- *Start jitter*, i.e., the inherited and accumulated jitter due to execution time jitter of preceding higher priority jobs.
- *Entry jitter,* i.e., the inherited and accumulated jitter due to execution time jitter before entering a critical section.

The complexity/testability of the EOG, i.e., the number of scenarios, is proportional to the number of preemption points and the jitter in the system. This complexity is not inherent to the EOG but rather a reflection of the system it represents. For the original model by Thane and Hansson [13] a rough estimate of the maximum number of execution orderings of a system, is $3^{n \cdot p}$, where n is the number of jobs (excluding sub jobs defined by critical sections) and p the average number of preemptions for each task instance. The base 3 in the expression comes from the possibility for a task to be preempted, the possibility for completion immediately before the start of a higher priority task, or the possibility of completion such that a lower priority task may succeed. These possibilities are strictly dependent on the execution time jitter and the start time jitter of tasks in the system. Consequently if there exists no jitter there will be only one possibility ($1^{n \cdot p}$). The number of preemption points, p, is also strictly dependent on the jitter in the system. There is thus an exponential relation between the complexity and the jitter in the system.

For a system with synchronization using semaphores the above complexity metric is still valid for the original (entire) jobs, not the partitioned sub-jobs. If the metric was applied to the sub-jobs the number of preemption points, p, would have to be reduced since the window of execution where a sub-job could be preempted by a higher priority job decreases.

Since any reduction of the jitter reduce 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 start times, or to force each job to always maximize its execution time, e.g. by inserting (padding) "no operation" instructions where needed. Fixed start times are easier to achieve for offset synchronized systems than mixed systems with offset and on-line synchronization. In order to achieve fixed start times of entry into critical sections we cannot usually make use of regular kernel primitives since the granularity of the system timer tick is not sufficiently fine (due to the prohibitive kernel overhead a too frequent timer tick would cause). We must thus resort to execution time equalization using padding unless some ingenious technique is used.

6 CONCLUSIONS

In this paper we have present a method for deterministic integration testing of strictly periodic fixed priority scheduled real-time systems with offsets, using on-line synchronization, complying with the Priority Ceiling Emulation Protocol (PCEP) [2] (a.k.a., the immediate inheritance protocol). The paper extends the results by Thane and Hansson [14][13] with handling of online synchronization. This substantially increases the applicability of the method, since it is more likely that errors are caused by synchronization and implementation problems, but also that industry standards like POSIX, ADA95, and OSEK have adopted PCEP [12][6].

Essentially the method is a structural white box testing method applied on the system level rather than on the individual tasks. The method includes a reachability technique for deriving all possible orderings of task starts, preemptions and completions for tasks executing in a system, together with a structural testing strategy for achieving full coverage with respect to the derived execution orderings. The testing strategy also allow test methods for sequential programs to be applied, since each identified ordering can be regarded as a sequential program. In the presented analysis and testing strategy, we considered task sets with recurring release patterns, and accounted for the effects of variations in start and execution times of the involved tasks, as well as the variations of the arrival and duration of the critical sections.

The testability/complexity of a system has an exponential relation to the jitter, as identified by Thane and Hansson [14][13].

For those that are interested we have tools for deriving the set of execution orderings as well as a real-time operating system, Asterix [17], which has the necessary infrastructure for monitoring, debugging [15] and testing of real-time systems as described in this paper.

Future research would be to investigate how to resolve execution order analysis with online synchronization protocols, other than PCEP, and to investigate how the testability of those protocols relate to PCEP, and offsets.

Another pursuit would be to extend the analysis with global synchronization using global PCEP [10] in the same manner as Thane and Hansson did with offsets [14][13].

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] Baker T. Stack-based scheduling of real-time processes. Real-Time Systems Journal, 3(1):67-99, March, 1991.

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

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

[5] Hamlet R. G. Probable Correctness Theory. Information processing letters 25, pp. 17-25, 1987.

[6] ISO/IEC. ISO/IEC 8652L 1995 (E), Information Technology – Programming Languages – Ada, Febrary 1995.

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

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

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

[10] Mueller F. Priority inheritance and ceilings for distributed mutual exclusion. Proc. 20th IEEE Real-Time Systems Symposium, pp. 340-349, Phoenix, Arizona, December 1999.

[11] 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.

[12] Technical Committee on Operating Systems and Application Environments of the IEEE. Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API), 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c:Amedment 2: Threads Extension [C Language]

[13] Thane H. and Hansson H. Testing distributed real-time systems. Journal of Microprocessors and Microsystems (24):463-478, Elsevier, 2001

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

[15] 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.

[16] 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.

[17] Thane H., Pettersson A., and Sundmark D. The Asterix real-time kernel. In proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01), Industrial Session, Delft, June 2001.

[18] 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.