Reproducible and Deterministic Testing of Distributed Real-Time Systems

Henrik Thane

Department of Computer Engineering
Mälardalen University
P.O. Box 883, S-721 23, Västerås, Sweden
Tel. +46 21 103157, Fax. +46 21 103110
henrik.thane@mdh.se

ABSTRACT

Reproducible and deterministic testing of sequential programs can in most cases be achieved by controlling the sequence of inputs to the program. The behavior of a distributed real-time system, on the other hand, is not only dependent on the inputs but also on the *order*, and *timing* of the tasks (concurrent processes) that execute and communicate with each other and the environment. Trying to apply sequential test techniques on distributed real-time systems is therefore sentenced to lead to non-determinism, because these techniques disregard the significance of order and timing.

In this paper we provide methods for identifying all the possible execution order scenarios for tasks in distributed real-time systems. Each identified scenario can be regarded as a sequential program, and consequently, test methods for sequential programs can be applied. We can achieve *deterministic testing* by observing the actual order of execution during runtime and map the test-cases and outputs to the corresponding scenario. Even more powerful is the possibility to force the system into specific scenarios and thus achieve *reproducible testing*.

More specifically this paper investigates statically scheduled distributed real-time systems, which are subjected to preemption, interrupts and jitter.

As a consequence of the proposed methods we propose design criteria for increasing the testability of distributed real-time systems, as well as new optimization criteria for scheduling.

Keywords: Testing, distributed real-time systems, determinism, reproducibility, testability.

1 INTRODUCTION

In this paper we will consider the reproducibility of distributed real-time systems. Testing of distributed systems is generally considered an intractable task, because the execution order and timing of the concurrent processes in the system seem non-deterministic. Our thesis is that this is a misconception, and as an argument for this we present a method for finding all the possible execution orderings of statically scheduled distributed preemptive real-time systems, which are subjected to interrupts.

We define execution order as the succession (line) of starting and completing tasks during an LCM period (the least common multiple of the tasks' period times) for a generated static schedule.

Each identified execution order constitutes a scenario, which can be regarded as a sequential program and can thus be tested by traditional methods, given that the test method can handle (or be modified to handle) the intricacies of observing concurrent process [Schütz1994].

If we run the system for a period equal to the entire static schedule (i.e. the LCM) for a specific test-case, and observe the actual execution scenario, we can always differentiate between different outputs using the observed execution scenario. So, by dedicating an equivalence class to each scenario, we can achieve *deterministic testing* by observing the actual order of execution during runtime, and map the test-cases and their resulting outputs to the corresponding equivalence class. The derived scenarios can therefore be used as coverage criteria for testing of the system's behavior during one LCM period. This is under the assumption that we can consistently observe the global state in the distributed real-time system, and in order to guarantee this we assume that the system is globally scheduled and has a global synchronized time base.

Even more powerful is the possibility to enforce certain execution scenarios with the benefit of achieving *reproducible testing*, which increases the effectiveness of testing by eliminating redundant test cases, and making it easier to achieve the desired level of coverage.

The number of scenarios is an objective measure of system testability, and thus it can be used as a new optimization criterion in the heuristic search used to generate static schedules [Xu1991]. The method also lays a foundation for exact calculations of the worst, and best, case response-times of tasks [Audsley1995].

The paper outline: Section 2 gives a background description of the problem when testing distributed real-time systems, and some related work. Section 3 gives a definition of the system model. Section 4 elaborates on what makes a system deterministic, and how determinism and reproducibility relate. Section 5 presents the algorithm that finds all the possible execution order scenarios for statically scheduled preemptive distributed real-time systems. We also give some examples, and extend the analysis to consider the effects of interrupts. Section 6 suggests a testing strategy for achieving deterministic and reproducible testing in the context of execution order scenario analysis. Finally we conclude in section 7 and give some hints on future work.

2 BACKGROUND

Over the years plenty of effort, time, and money have been put into research and development of testing techniques for sequential programs. A basic assumption has always been that that program behavior is reproducible during testing [McDowell1989]. This property of reproducibility is essential when performing regression testing or cyclic debugging [Schütz1994]. The same test cases are run over and over with the intent to validate that either an error correction had effect, or simply, just to make it possible to find the error when a failure has been observed.

The basic assumption of reproducibility is unfortunately not valid in parallel (concurrent) systems, such as distributed real-time systems, due to the systems (seemingly) non-deterministic behavior, which makes the application of test techniques for sequential programs void. The cause for this non-determinism is twofold:

(1) *The inherent non-determinism in the system due to its parallel character* – which leads to race situations. In a concurrent system with shared resources, different inputs may lead to different execution paths. These paths will in turn lead to different execution times for the processes, which depending on the design of processes may lead to different *orders* of access to the shared resources. As a consequence there may be different system behaviors if the operations on the shared resources are not idempotent (i.e., serially equivalent). A mathematical example of non-idempotency could be $F(G(x)) \neq G(F(x))$.

(2) *The side effects caused by observing the system*. In a system where race conditions exist, the act of (intrusively) observing the system will always change the odds for the racing processes to win, because the probes will add to the execution times of the racing processes. The outcome of a race could therefore be dependent on the presence (or absence) of a probe. If we later remove the probes after observation we do not only decrease the observability, but also change the execution times again – which affects the races, and this time we do not have the probes in place to observe how the system behaves. This act of intrusively observing a system is called the *probe-effect* [McDowell1989, Gait1985] or the *Heisenberg uncertainty* in software [LeDoux1985].

2.1 Related work

The work on testing distributed real-time systems is meager compared to the work on testing sequential software. Related work can be summed up to:

- A nice problem formulation with respect to *observability* by Schütz [1994] describing to some extent what happens to the reproducibility of a distributed real-time system when it is observed (the probe-effect). A preceding and similar problem formulation for parallel programs without real-time requirements has been presented by [McDowell1989] and extended in [McDowell1996]. [Schütz1994] has added the aspects of real-time to McDowell's work, and made a comparison of how time-triggered and event-triggered systems compare with respect to testability. According to Schütz the testability of a distributed real-time system depends on three factors: observability, reproducibility, and controllability. With respect to these factors Schütz describes the laudability of the time-triggered approach over the event-triggered approach, and briefly describes testing of the Mars system [Schütz1994]. He does however, not pursue the issue on how to test distributed real-timer systems further.
- A somewhat more extensive research effort has been put into the field of *monitoring* (supervision) of timing, e.g., deadline violation supervisions, execution time measurements, context switch logging, etc. A few references are: non-intrusive monitoring using special hardware without introducing any probe-effect [Tsai1996], monitoring using software by either accounting for the probe-effect, or simply ignoring it [Tai1991], and monitoring using both hardware and software (hybrid) with the intent to minimize the effects of observation [Dodd1992].

To the authors knowledge there exists no work that constructively tries to do something about the quintessential obstacle when testing distributed real-time systems – namely reproducibility.

3 THE SYSTEM MODEL

We assume that we have a distributed system consisting of a set of nodes, which communicate via a network. 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.

The software that runs on the distributed system consists of a set of concurrent processes, which communicate by sending messages to each other. Although the processes are geographically distributed on the nodes, more than one process may execute on one node.

We assume an execution strategy based on static scheduling, i.e., pre run-time schedule design [Xu1991]. The schedule is during run-time managed by a dispatcher. The objective of the dispatcher is to run the tasks according to the schedule, so that the tasks are periodically activated. The entire schedule is repeated with a period time equal to the least common multiple (LCM) of all the tasks' period times. The schedule fulfills the temporal requirements by construction that is, the schedule is constructed in such a way that the requirements are checked for each task during the construction of the schedule. It is possible to construct several different kinds of schedules: allowing preemption, schedules where each task has fixed start times [Xu1991], schedules where several tasks are assigned to the same start time, schedules where interrupts are accounted for [Sandström1998], etc.

We assume that each task has the following attributes and relations to other tasks when scheduled (task model):

- Period time, Release time (start time relative period start), Deadline (latest completion time relative period start), Worst case execution time (maximum) and Best case execution time (minimum).
- Relations between tasks can be specified by Precedence relationships, Mutual exclusion relationships (shared resources) and Communication.
- All interrupts are specified by minimum, and maximum inter-arrival time, as well as worst and best case execution time of the interrupt routines.

We assume that the system is preemptive (both by tasks and interrupts) and that tasks can have the same release-time (start time).

All synchronization and communication is resolved before run-time. As a consequence there are no actions to enforce synchronization, like locking of semaphores, present in the actual program code. In the generated schedule mutual exclusion and precedence is guaranteed by different release-times relative LCM period start (all release-times in the generated schedule are relative LCM period start.) Furthermore, we assume that received data is available when tasks start, and that sent data is delivered first when tasks terminate, not during execution.

4 DETERMINISM AND REPRODUCIBILITY

The reproducibility problem with distributed real-time systems is actually due to a misunderstanding. When a sequential program is tested it is necessary (and sufficient) to control the sequence of inputs to the program (and perhaps the start conditions). The behavior of a concurrent program, on the other hand, is not only dependent on the sequence of inputs but also on the *order* in which the concurrent programs execute and communicate. In real-time systems the behavior of the system is in addition also dependent on *when* the inputs arrive and *when* the programs execute and communicate with each other, or with the environment. Trying to apply test techniques for testing of sequential programs, on distributed real-time systems, is therefore sentenced to lead to non-determinism because control is only forced on the inputs, disregarding the significance of order and timing. To facilitate testing of distributed real-time systems we must control the timing and order of the sequence of inputs, as well as the timing and order of program execution.

There are thus two problems to solve: (1) reproducing the inputs with fidelity to contents, order, and timing, (2) reproducing the order and timing of the execution of the parallel programs as well as their communication with each other and the environment (output).

In this paper we are only going to address (2) – the reproducibility of program execution and communication.

Using scheduling theory for real-time systems we can make predictions of the systems temporal behavior. For example, using either a static scheduling or a fixed priority scheduling approach we can make predictions about whether task sets can meet their deadlines or not. That is, we can guarantee that a system designed with purely periodic timing requirements will always meet its deadlines. For ideal circumstances we could stop and just deploy the system. In reality however, we must validate our model, both temporally and functionally, because the basic assumptions of e.g., the execution times might not be accurate, the model might be erroneous, and quite likely the implementation of the model into program code is faulty. We must thus verify the system, e.g., by testing. When we want to test a distributed real-time system we are not solely interested in observing its ability to meet deadlines, but are also interested in observing its functional behavior. A real-time system is per definition correct if it performs the correct *function* at the correct *time*.

When we refer to a predictable real-time system we usually mean that we can make predictions about its ability to meet its deadlines, or not. If such a system is free from errors and has been predicted to always meet its deadlines, we can run the system over and over and always observe that it meets its deadlines. The property of always meeting the deadlines will thus be reproducible, and as such the system is deterministic.

Definition. *Determinism*. Given (Knowing) all necessary and sufficient conditions that must occur for a system to have a certain property, and the occurrence of these conditions *uniquely* determines this property, then the system is defined as deterministic. Using this knowledge we can make predictions whether the system will have this property, or not, for an arbitrary set of conditions.

The implication of this definition is that if we have a function *f(a, b, c, d)* and the property *P* of this function is deterministically determined by the necessary and sufficient conditions (or parameters) of *a* and *b*, then we can execute the function *f(a, b, c, d)* an infinite number of times and deterministically observe if this property is valid or not, by observing *f* and by observing *a* and *b*. The values of *c* and *d* are of no significance because they are not necessary for *P's* determinism. If we also can control, not only observe, the values of *a* and *b* we can also reproduce the property *P*.

Definition. *Reproducibility*. A system is reproducible if it is deterministic with respect to a property *P*, and it is possible to control all necessary and sufficient conditions for *P*.

When testing the *correct function* of a real-time system, the predictability (or determinism) of deadlines is necessary but not sufficient, because we need also to make predictions about other properties than just meeting deadlines. As discussed before, the function of a distributed real-time system depends on the order in which system processes execute and when they start, and stop. We must therefore be able to predict release-times of tasks and predict task execution order.

In the next section we describe a method for finding all the possible execution orders for statically scheduled preemptive distributed real-time systems. Using the method we can derive *when* the different execution orders occur and in what manner they manifest themselves. We can thus find all possible execution order scenarios for the system.

5 EXECUTION ORDER ANALYSIS

We are now going to present a method for finding all the possible orders of execution for statically scheduled task sets when preemption is allowed. Later in the paper we also account for the interference caused by interrupts. The method builds a graph with all the possible execution order scenarios.

But before we go into the details of the algorithm we will try to present the underlying intuition.

Consider *figure 1*, which depicts an LCM schedule generated by a static off-line scheduler and where the execution times for the tasks *A*, *B*, and *C* are fixed. This has the effect of only yielding one possible execution scenario during one LCM period. However, if for example, the task *A* had a minimum execution time of 5 we could get another scenario when *A* completes before *B* is released, or another when *A* completes before *C* is released. Anyway, we would end up with three execution order scenarios instead of one. Another example, could be task *B* having a minimum execution time such that it is possible for *A* to complete prior to the release of *C*, giving a total of two execution order scenarios.

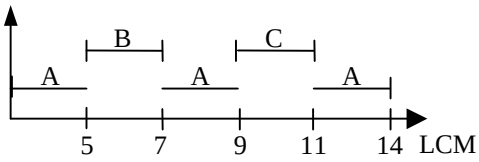


Figure 1 A schedule with the tasks B and C preempting A where the execution times for all tasks are fixed.

Our new algorithm uses information about varying execution times, release times and the precedence relations for the tasks in the schedule, in order to mathematically derive all possible execution orders.

5.1 Definitions and notation

We assume that we have run a static scheduler and found a feasible schedule. The schedule is in the form of a stack, *S*, where the elements, the tasks, *T*, are in order of increasing release-times, and ordered according to precedence for tasks with the same release-time. The stack contains the entire schedule for one LCM, the least common multiple of the tasks' period-times. This means that the schedule will contain several instances of the same task when it has a period time less than the LCM. The task instances will consequently be given different release times, but depending on the generated schedule the instances might not always be separated exactly by their period time.

The following functions operate on the stack *S*:

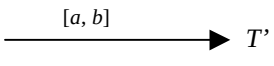
- Push(*T*,*S*) – Push the task *T* on the top of the schedule stack *S*.
- Pop(*S*) – Remove the top element of *S* and return it.
- Remove(*P*, *S*) – Remove the first element in the stack *S* identical to *P* and return it.
- Search(*S*, *a*, *b*) – Search the stack *S* for the first element with a release-time *t* such that $a < t \leq b$, and return a copy of it.

Any arbitrary task *T*, in the stack has attributes defined by the following notation:

- *T.rel* – The earliest release-time for the task.
- *T.min* – The best-case (minimum) estimated execution time.
- *T.max* –The worst-case (maximum) estimated execution time.
- *T.maxp* – The maximum time of execution that can occur from *earliest release* of *T* to the instant when a preempting task P is released. This is a variable, which value changes depending on where in the schedule the task execute. *Set to zero initially.*
- *T.minp* – The minimum time of execution that can occur from *latest release* of *T* to the instant when a preempting task P is released. This is a variable, which value changes depending on where in the schedule the task execute. *Set to zero initially.*

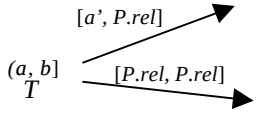
The elements in the Execution Order Graph (EOG):

- A transition from some task to task, *T'*, which can occur in the interval [*a, b*]. This is the interval of possible release-times for *T'*.



- A task *T* with an interval specifying the interval [*a, b*] in which the task *T* can be preempted.

- Two transitions from a task *T* which is preempted by a task *P*. The top transition depicts the scenario when *T* completes prior to *P*. This transition is impossible if *P.rel* < *a'*. The bottom scenario depicts *P* preempting *T*.



5.2 The Graph algorithm

The algorithm mathematically derives all possible execution orders for a given schedule during one LCM period, and represents the derived scenarios in an acyclic graph.

The algorithm is recursive and takes four arguments *B*, *S*, *a*, and *b*, where *S* is a schedule stack, *a*, *b* the earliest, and latest possible release-times for the next coming task on top of the schedule stack. The argument *B* is the transition from the previous node in the graph. The algorithm generates the graph in depth-first order.

The algorithm is initially called, with the branch from the root node, the original schedule stack, and the release-times *a* and *b* set to zero: Graph(ROOT, S_0, 0, 0)

Graph(*B, S, a, b*)

{

1. *T* = Pop(*S*). Pop the first element of *S* and assign its value to *T*.
2. If (T ≠ *e*) //Stack is not empty, else branch complete – return.
3. {
4. *a = MAX(T.rel, a)* //Adjust the release times for the task
5. *b = MAX(T.rel, b)*
6. *(a, b)* = (*T.rel, b + T.max – T.minp)* //Calculate the preemption interval
7. *Annotate T* with the interval **(***a, b***).**
8. P = Search(*S, a, b*). // Find first task in *S* that has a release time within the interval
9. If (*P* = *e*) //NO PREEMPTION
10. {
11. Put the task *T* at the end of *B*.
12. Create a new branch, B_{new}, from *T* in the graph.
13. *[a', b'] = [MAX(a, a + T.min – T.maxp), b + T.max – T.minp]//* New release interval

14. Annotate B_{new} with [*a', b'*].
15. Call Graph(B_{new}, *S, a', b'*)
16. }
17. Else //PREEMPTION
18. {
19. If (*P.rel < a) //T*he preempting task *P* should actually precede *T*.
20. {
21. Push(*T, S*). Put *T* back on the stack again
22. Remove(*P*).
23. Push(*P, S*). Move *P* up to the top position in the stack.
24. Graph(*B, S, a, b*). Redo everything, but this time let *P* be first out.
25. }
26. Else // *P.rel ≥ a*
27. {
28. Create a transition for the non-preempted part of the interval.
29. *[a', b'] = [MAX(a, a + T.min – T.maxp), P.rel]*
30. *If (a' ≤ b') //* The transition is possible.
31. {
32. Put the task *T* at the end of *B*.
33. Create a new branch, B_{new}, from *T* in the graph.
34. Annotate B_{new} with [*a', b'*].
35. Call Graph(B_{new}, *S, a', b'*)
36. }
37. //A transition for the preempted part of the interval.
38. Put the task *T* at the end of *B*.
39. Calculate *T.maxp = T.maxp* + (*P.rel – a*).
40. Calculate *T.minp = T.minp + MAX*(*P.rel – b, 0)*.
41. Remove(*P*) //from the stack.
42. [*a', b'*] = [*P.rel, P.rel*]. //Calculate the release interval for the preempting task.
43. Create a new branch, B_{new}, from *T* in the graph.
44. Annotate B_{new} with [*a', b'*].
45. Push(*T, S*). //Put T on the top of the stack again.
46. Push(*P, S*). //Put *P* on the stack again also
47. Call Graph(B_{new}, *S, a', b'*)
48. }//END Else *P.rel > a*
49. }//END PREEMPTION
50. }
51. Return(). //This invocation is complete

}//END

Additional comments to the algorithm (row wise):

4-5: Adjust the release times for the task. This is necessary because the inherited upper and lower release times from the execution of preceding tasks might be less than the actual release-time for the task T.

6: The interval states that only tasks with later release time than a and less than b can preempt *T*. Tasks with the same release time as *T* are stored in the schedule stack in order of precedence and can therefore not preempt *T*, by definition. Worth noting is that a is pessimistic because tasks with the same release-time as *T*, and preceding *T*, will in actuality increase a with their minimum execution time. This is however of no practical consequence because tasks that have preempted the preceding tasks, with the same release-times as *T*, will have been removed from the schedule stack and can thus not

preempt *T*. However, there might still be tasks $P_1...P_n$ on the schedule stack with greater release-times than *T* but where $P_?.rel < a$, and *a* is the actual earliest release of *T*. In order to accommodate for this we set $a = T.rel$. There is a special case row 19-25 handling this.

The upper end of the interval, *b*, is limited by the latest release time, *b* and *T´*s maximum execution time *T.max*, i.e., $b + T.max$. However due to the possibility that the task *T* has been preempted and is now running the remaining part of its program we must accommodate for the case where *T* had the opportunity to run for minimum *T.minp* time units before the preempting task was released, i.e., $b + T.max - T.minp$. This is safe because if $T.minp > T.max$ then the task would have completed and consequently not end up here. The value of *T.minp* is defined in row 40.

8: Search for the first possible task in *S* that can preempt *T*, i.e., the first task that has a release time within the interval, and return it if found

13: That is, *a´* is earliest release point for the next coming task. The earliest this can occur would seemingly be task *T*'s earliest release-time, *a*, added with *T*'s minimum execution time: $a + T.min$. However, due to the possibility that the task *T* on a previous earliest-release invocation has been preempted, and is now running the remaining part of its program. We must accommodate for the case where *T* had the opportunity to run for maximum *T.maxp* time units before the preempting task was released, i.e., $a + T.min - T.maxp$. However, again, there is a possibility that this sum could end up less than *a* when *T* on the prior invocation had run for more than *T.min*, but less than *T.max*. The resulting sum is infeasible because the next task can never be released earlier than *a*, thus MAX(*a*, $a + T.min - T.maxp$). The value of *T.maxp* is defined in row 39.

19: The task *P* has a release time prior to *T*'s earliest release specified by *a*. *P* must therefore precede *T*.

29: The arguments for the definition of a' is equal to the ones in row 13. The argument for the validity of *b'* is based on the fact that if *T* had a finish time later than *P.rel* it would have been preempted, and consequently not end up in this branch.

30: If $a' > b'$ had been valid then this transition would have been infeasible, because the task *T* has no chance to ever complete its execution before P's release-time.

39: $T.maxp = T.maxp + (P.rel - a)$. The maximum number of time units that the task *T* can have executed from its earliest release, *a*, to the release of *P*, *P.rel*. The value, *T.maxp*, accumulates depending on how many times *T* has previously been preempted.

40: $T.minp = T.minp + MAX(P.rel - b, 0)$. The minimum number of time units that the task *T* can have executed from its latest release, *b*, to the release of *P*, *P.rel*. There is an accumulative effect however, depending on how many preemptions it has experienced previously. However this is only valid for $b < P.rel$, because *T* can not start execution at *b* and then be preempted by *P* since *P* would then already have completed and thus could not preempt.

41: Remove *P* from the stack. Find the first element that has the same value as *P* in *S* and remove it. This removes the task *P* from further preemption possibilities in this part of the graph.

45: Put *T* on the top of the stack again. So it can continue after the completion of *P*.

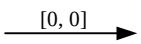
46: Put *P* on the stack also, so it on the next invocation of Graph can go through steps 1-51.

5.2.1 Example 1 – *Preemption*

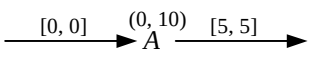
Here we give an example of the resulting the execution order graph (EOG) generated by the algorithm:

Assume that we have a feasible schedule (se figure 1) and a schedule stack S =
{*A.rel=0, A.min=7, A.max=10, A.maxp=0, A.minp=0;*
B.rel=5, B.min=1, B.max=2, B.maxp=0, B.minp=0;
C.rel=9, C.min=1, C.max=2, C.maxp=0, C.minp=0}

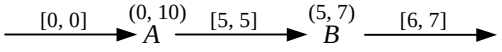
Graph(*ROOT, S, 0,0*) :



A=POP(*S*). Calculate the preemption interval (*0, 0+A.max*] = (*0,10*]. Preemption? Yes by *B*. According to Graph():30 the non preempted part is infeasable. Put *A* at the end of the release-transition. *T.maxp=5, T.minp*=5. Set new release-interval: [*B.rel, B.rel*] =[*5,5*]. Push(*A,S*), Push(*B,S*).

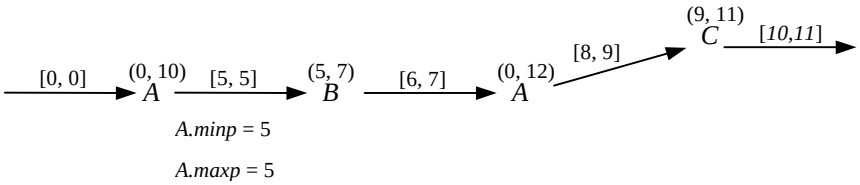


B=POP(*S*). Calculate the preemption interval (*5, 5+B.max*] = (*5,7*]. Preemption? No. Put *B* at the end of the release-transition. Calculate a new release-interval: [5+*B.min, 5+B.max*] =[*6,7*].

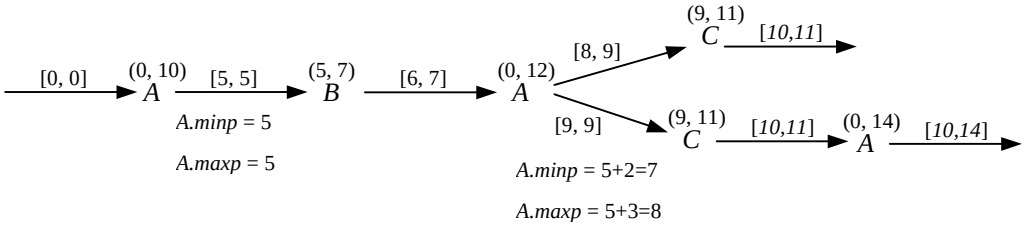


A=POP(*S*). Calculate the preemption interval (*A.rel, 7+A.max-A.minp*] = (*0,12*]. Preemption? Yes by *C*. Create two new transitions according to Graph():39-40.

Here is a complete path for the non-preempted part with respect to *C*:



And, the complete graph.



An interesting observation is that the release intervals from the last incarnation of a task in a certain path represent the response-times for the task. The worst and best-case response time can therefore be found by simply searching the graph for the smallest and the greatest release time. The response time jitter (the difference between the maximum and minimum response times for a task) can also be quantified both globally for the entire graph and locally for each path, as well as for each instance of a task that runs several times during an LCM cycle.

5.3 Adding interrupts

We will now incorporate the effects of interrupts on the execution order graph. We assume that the interrupts have just temporal side effects and no functional side effects on the scheduled tasks. We could however address the functional side effects by modeling the interrupts as aperiodic tasks, and put them into the EOG, but this is out of the scope of this paper.

The attributes of any interrupt *I* is as follows:

- *I.max* – The maximum calculated execution time.
- *I.min* – The minimum calculated execution time.
- *I.maxi* – The maximum inter arrival time between occurrences of *I*.
- *I.mini* – The minimum inter arrival time between occurrences of *I*.

The preemption interval $(a, b) = (T.rel, b + T.max - T.minp]$ changes to:

$\boldsymbol{a} = T.rel$, i.e., keeps the same value as before.

If $T.rel \geq a$ then *T's* release time is the scheduled *T.rel* and thus not afflicted by the interrupt interference on preceding tasks' completion times.

$\boldsymbol{b} = b + w$, i.e., the sum of the latest release time *b*, and the sum, *w*. Where *w* is the sum of (*T.max – T.minp*) plus the maximum delay due to the preemption by aperiodic interrupts with a minimum and maximum inter arrival time of *I.mini* and *I.maxi* respectively for a specific interrupt *I*, belonging to the set of preempting interrupts.

The formula for calculating *w* is similar to Response Time Analysis (RTA) calculations [Audsley1995], which are based on fix-point iteration. As, RTA calculates the response times for tasks that are subjected to interference by preemption of higher priority tasks, we here calculate the interrupt interference on the preemption intervals. The difference is that the response time analysis is more precise here, because it is always applied locally in the graph – defined by the interval of interest, and therefore more constrained than the more global and thus more pessimistic RTA.

The maximum value of *w* is defined by (1) and is based on the assumption that all the interrupts are ready for preemption exactly at the beginning of the interval – this maximizes the number of times the interrupts can preempt the interval.

$$w = (T.max - T.minp) + \sum_{\forall interrupts\ I} \left\lceil \frac{w}{I.mini} \right\rceil \cdot I.max \qquad (1)$$

Because *w* is on both sides in the equation, and due to the intractability of algebraically substituting the ceiling function ($\lceil\ \rceil$) we must use iteration:

$$w^{n+1} = (T.max - T.minp) + \sum_{\forall interrupts\ I} \left\lceil \frac{w^n}{I.mini} \right\rceil \cdot I.max \qquad (2)$$

The iteration proceeds until $w^{n+1} = w^n$. This can be considered as having the window, *w*, widening until it allows *T* to execute *T.max – T.minp*. The iteration is guaranteed to converge in a finite number of steps if the processor utilization is less than a 100% [Audsley1995].

Set $w^0 = T.max - T.minp$, because this is the minimum upper bound of the preemption interval where preemption by interrupts can occur.

However if $T.rel < a$ then *T's* release time is afflicted by the interrupt interference on preceding tasks' completion times and must therefore be accommodated for. We otherwise run the risk of too pessimistic estimations of the interrupt interference when preceding tasks' maximum completion times have been based on interrupt hits which periodicity is out of phase with a first interrupt hit at time *a*.

The execution time that must be accommodated for then is of course (*T.max – T.minp)* but in addition also $q = (b - \sum_{\forall interrupts\ I} \left\lceil \frac{b}{I.mini} \right\rceil \cdot I.max)$, that is $(T.max - T.minp) + q$.

The release intervals for the non-preempted task

$$[\boldsymbol{a}', \boldsymbol{b}'] = [MAX(a, a + T.min - T.maxp), b + T.max - T.minp]$$

and for the preempted task but completing transition

$$[\boldsymbol{a}', b'] = [MAX(a, a + T.min - T.maxp), P.rel]$$ **change to**:

For the lower bound, $\boldsymbol{a}'$, of the release interval we are interested in finding the minimum possible interference by the interrupts. This minimum can be found by assuming that the interrupts occur with their maximum inter arrival time and execute with their minimum execution time, and by finding the lowest possible number of hits within the interval. The latter is guaranteed by the use of the floor function ($\lfloor\ \rfloor$).

The lower bound $\boldsymbol{a'}$, is defined by:

$$\boldsymbol{a'} = a + w_{a,} \tag{3}$$

Where w_a, is a response time analysis function defined as:

$$w_a = MAX(0, T.min - T.maxp) + \sum_{\forall interrupts\ I} \left\lfloor \frac{w_a}{I.maxi} \right\rfloor \cdot I.min \tag{4}$$

Set $w_a^0 = MAX(0, T.min - T.maxp)$, because this is the minimum of the release interval.

The upper bound of the release interval $\boldsymbol{b'}$, is identical to the preemption intervals $\boldsymbol{b}$.

The correction factors, *T.maxp*, *T.minp* **change to**:

$$T.maxp = T.maxp + MAX(0, P.rel - a) - \sum_{\forall interrupts\ I} \left\lfloor \frac{MAX(0, P.rel - a)}{I.maxi} \right\rfloor \cdot I.min \tag{7}$$

Note that the sum of interrupt interference is not iterative, but absolute, because we are only interested in calculating how much the task *T* can execute in the interval, minus the interrupt interference.

Likewise we calculate the minimum number of time units the task *T* can execute prior to preemption

$$T.minp = T.minp + MAX(0, P.rel - b) - \sum_{\forall interrupts\ I} \left\lceil \frac{MAX(0, P.rel - b)}{I.mini} \right\rceil \cdot I.max \tag{8}$$

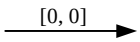
5.3.1 Example 2 – Preemption and Interrupts:

Here we assume that the system is subjected to preemption by interrupts. The side effects of the interrupts are solely of temporal character.

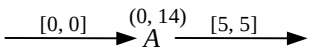
Making matters simple we assume that there is only one interrupt *I* interfering, which has a minimum inter arrival time of *I.mini* = 4 time units, and a maximum inter arrival time of *I.maxi* = ∞ time units. The maximum and minimum execution time, *I.max* = *I.min* = 1 time units.

The feasible schedule in the schedule stack *S* looks like this =
{*A.rel=0, A.min=7, A.max=10, A.maxp=0, A.minp=0;*
B.rel=5, B.min=1, B.max=2, B.maxp=0, B.minp=0; }

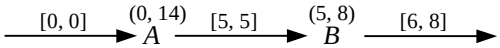
Graph(*ROOT, S, 0,0*) :



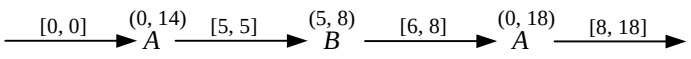
A=POP(*S*). The preemption interval? (*a, b)* = (*0, 0+w)* = (*0,14)*. Preemption? Yes by *B*. According to Graph():30 the non preempted part is infeasable. Put *A* at the end of the release-transition. *T.maxp=5, T.minp=3*. Set new release-interval: [*B.rel, B.rel*] =[*5,5*]. Push(*A,S*), Push(*B,S*).



B=POP(*S*). Calculate the preemption interval (*5, 5+w)* = (*5,8)*. Preemption? No. Put *B* at the end of the release-transition. Calculate a new release-interval: [$5+w_a$, $5+w_b$] =[*6,8*].



A=POP(*S*). Calculate the preemption interval (*A.rel, 8+w)* = (*0,18)*. Preemption? No. Calculate a new release-interval:
[$6+w_a$, $8+w_b$] =[8,*18*].



The complete graph.

Note that the release-times of the tasks are affected by the interrupts, and therefore also affecting the response times.

5.4 Jitter

In the presented algorithms we take into account the effects of jitter, namely:

- *Schedule jitter*, i.e., different instances of the same task in an LCM cycle may have different relative release times for each period depending on precedence between the tasks for the same release-time. For example, A.B.C…B.A.C…A.B.C…
- *Release jitter*, i.e., the inherited and accumulated jitter due to varying execution times for preceding tasks with the same release-time.
- *Completion jitter*, i.e., the response-time jitter due to schedule jitter, release jitter and the variation of the execution time for the task (*Execution time jitter*).

One type of jitter that we have not explicitly addressed, is the jitter inherent to global clock synchronization. The local clocks keep on speeding up, and down, depending on the global time base. Possibly could this jitter be accommodated for, by adjusting the preemption and release intervals in the EOG.

An interesting conclusion that can be drawn from these types of jitter, and their effect on the execution order graph, is that:

1. Minimizing the execution time jitter minimizes the preemption and release intervals – with the positive effect of reducing the number of execution order scenarios.
2. By reducing the release jitter we reduce the execution order graph. This can be achieved by doing (1) or having fixed release-times.

The conclusion is thus that less jitter gives better testability.

6 TEST STRATEGY

How should one go about testing a distributed real-time system? Given the presented method for deriving all execution order scenarios we suggest the following strategy for deterministic testing of a statically scheduled distributed real-time system during one LCM period:

1. Do the execution order analysis for a given static schedule
2. Create an equivalence class for each derived scenario.
3. Test the system using any testing technique of choice, and monitor which execution scenario is run during an LCM cycle for each test case.
4. Map test case and output into correct equivalence class, based on observation.
5. Do 3-4 until sought coverage is achieved.

To facilitate reproducible testing we must identify which scenarios, or parts of scenarios that can be enforced without introducing any probe effect. This can be done by forcing later release-times on tasks that have terminated earlier than desirable, all in order to achieve a specific execution scenario. This can however only be done, in general, when tasks with the same release-time and precedence relation have no communication. Otherwise we run the risk of introducing unwanted probe-effects.

The choice of testing method must be based on the coverage needed and depending on if the system keeps state between LCM periods. In order to guarantee determinism for state-keeping systems we must monitor or control the tasks' internal variables and not only the legal inputs defined by the tasks' interfaces.

7 CONCLUSION

In this paper we have given a description of what constitutes a deterministic and reproducible behavior of a distributed real-time system. Specifically we have addressed testing of statically scheduled distributed preemptive real-time systems (DRTS). The results can be summed up to:

- We have provided a method for finding all the possible execution order scenarios for DRTS with preemption and interrupts.
- We have proposed a test strategy for deterministic and reproducible testing of DRTS.
- A benefit of the testing strategy is it allows any testing technique for sequential software to be used for test of DRTS.
- With the number of execution order scenarios we have got an objective measure of the testability of DRTS, and therefore a new scheduling optimization criteria.
- Any type of jitter reduction reduces the number of execution order scenarios, and should therefore be targeted when possible.
- A positive side effect of the execution order analysis is that we get exact response-times for the tasks, even when interrupts afflict the system.

Future pursuits would be to in practice validate the claims in this paper, and make a similar analysis for periodic fixed-priority scheduled preemptive real-time systems, and devise testing strategies, as well as testability increasing measures for such systems.

8 REFERENCES

[Audsley1995] N. C. Audsley. Fixed Priority Pre-emptive Sceduling: An Historical Perspective. Real-Time Systems, The Int. Journal of Time-Critical Computing Systems, Vol. 8, no. 2/3, March/May, 1995.

[Dodd1992] P. S. Dodd, C. V. Ravishankar. *Monitoring and debugging distributed real-time programs*. Software-practice and experience. Vol. 22(10), pp. 863-877, October 1992.

[Gait1986] J. Gait. A Probe Effect in Concurrent Programs. Software – Practice and Experience, 16(3):225-233, Mar. 1986.

[LeDoux1985] C.H. LeDoux, D.S. Parker. Saving Traces for Ada Debugging. In the proceedings of Ada int. conf. ACM, Cambridge University press, pp. 97-108, 1985.

[McDowell1989] C.E. McDowell and D.P.Hembold. Debugging concurrent programs. ACM Computing Serveys, 21(4):593-622, December 1989.

[McDowell1996] C. E. McDowell, D. P. Helmbold: A taxanomy of race conditions. Journal of Parallel and Distributed Computing, pp. 159-164,Vol 33, no. 2, Mar. 15, 1996.

[Sandström1998] K. Sandström, et al. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. 5th Int. Conference on Real-Time Computing Systems and Applications. October 1998, Japan.

[Schütz1994] W. Schütz. Fundamental Issues in Testing Distributed Real-Time Systems. Real-Time Systems, 7, 129-157, 1994.

[Tai1991] K.C Tai, et. al. *Debugging concurrent Ada programs by deterministic execution*. IEEE transactions on software engineering. Vol. 17(1), pp. 45-63, January 1991.

[Tsai1996] J.P. Tsai, et. al. *A system for visualizing and debugging distributed real-time systems with monitoring support*. Journal of Software engineering and Knowledge engineering. Vol. 6(3), pp. 355-400, 1996.

[Xu1991] J. Xu and D. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. Proceedings of the ACM Sigsoft 91 Conference on Software for Critical Systems, Software Engineering Notes 16(5), pp. 132-145, 1991.