Handling Interrupts in Testing of 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

Testing techniques for sequential software are well established. Unfortunately, these testing techniques are not directly applicable to distributed real-time systems. In previous work we have proposed a method for reproducible and deterministic testing of distributed real-time systems. This method identifies all possible orderings of task starts, preemptions and completions. Each ordering is regarded as a sequential program, and can as such be tested with standard sequential testing techniques.

In this paper we extend our method to also handle interrupts. This is essential since critical parts of most real world systems are managed by interrupts.

1 Introduction

We have previously [18] presented a method for achieving deterministic testing of distributed real-time systems. This method transforms the non-deterministic distributed real-time systems testing problem into a set of deterministic sequential programs testing problems. This is achieved by deriving all the possible execution orderings of the real-time system and regarding each of them as a sequential program. In this paper we pursue this line of work and extend the analysis to also cover the interference caused by interrupts. We will specifically address task sets with recurring release patterns, where the scheduling is handled by a priority driven preemptive scheduler. This includes statically scheduled systems that are subject to preemption [22][16], as well as strictly periodic fixed priority systems [2][13].

The analysis presented in this paper handles single nodes, but can easily be extended to distributed real-time systems, by taking clock-synchronization effects into account, and using parallel composition techniques as presented in [18].

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 [17], 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 [11]. However, trying to directly apply test techniques for sequential programs on real-time systems and 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 real-time system will in addition incur a temporal probe-effect [5][12] that subsequently will affect the system’s temporal and functional behavior.

To allow the real-time system to be regarded as a set of sequential programs we derive all execution orderings, that is, the set of possible orderings of task starts, preemptions and completions. A formal definition of what actually constitutes an execution order scenario can be found in [18], but the following small example presents the underlying intuition:

Consider Figure 1a, which depicts a schedule for the duration equal to the Least Common Multiple (LCM) of the period times of the involved tasks $A$, $B$ and $C$, which are generated by a static off-line scheduler and which 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 1a. However, if for

![Figure 1 Three different execution order scenarios.](image-url)

(a) $C_p=6$  (b) $C_p=4$  (c) $C_p=2$
example, task $A$ had a minimum execution time of 2 ($BCET_A=2$; $WCT_A=6$) we would get three possible execution scenarios, depicted in figures 1a - 1c. In addition to the execution order scenario in Figure 1a, there are now possibilities for $A$ to complete before $C$ is released (Figure 1b), and for $A$ to complete before $B$ is released (Figure 1c).

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.

The contributions of this paper are suggestions on:

- how to identify the execution order scenarios when interrupts interfere
- how to use the scenarios when testing (the test strategy), and
- how to reproduce the scenarios.

Concerning monitoring of real-time systems we assume that the probe-effect is eliminated through the allocation of sufficient resources and then letting the probes remain in the target system. This includes allocating resources for the probes’ execution time, memory, communication bus bandwidth and accounting for the probes when designing and scheduling [19].

Paper outline: Section 2 presents our system model. Section 3 describes our method for identifying all the possible execution orderings of a single node real-time system, introduces the extended analysis to cover interrupts, and gives some examples. Section 4 suggests a testing strategy for achieving deterministic and reproducible testing in the context of the execution order analysis. Finally, in Section 5, we conclude and give some hints on future work.

2 The system model

We assume a distributed system consisting of a set of nodes, which communicate via a temporally predictable broadcast network, i.e. upper bounds on communication latencies are known or can be calculated [8][21]. 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] with a known precision $\delta$, meaning that no two nodes in the system have local clocks differing by more than $\delta$.

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 set of jobs $J$, i.e. invocations of tasks, which are released in a time interval $[0, J_{MAX}]$, where $J_{MAX}$ is typically equal to the Least Common Multiple (LCM) of the involved tasks. Each job $j \in J$ has a release time $r_j$, worst case execution time ($WCT_j$), best case execution time ($BCET_j$), a deadline $D_j$ and a unique priority $p_j$. $J$ represents one instance of a recurring pattern of job executions with period $J_{MAX}$, i.e., job $j$ will be released at time $r_j$, $r_j+J_{MAX}$, $r_j+2J_{MAX}$, etc. We further assume that each job $j \in J$ always completes within its deadline $D_j$, i.e., $J$ is schedulable.

We further assume that the system is preemptive (both by jobs and interrupts) and that jobs may have identical release-times. The task model is fairly general since it includes both preemptive scheduling of statically generated schedules [16][22] and fixed priority scheduling of strictly periodic tasks [2][13].

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. We assume however, that interrupts have only temporal side effects and no functional side effects. 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) [4][10].

Fault hypotheses

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.

Depending on pessimism we can therefore identify two fault hypotheses:

1. 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.

2. In addition to (1) jobs can corrupt each others shared memory and I/O interfaces, i.e., they may interfere via unspecified side-effects.

One way to guarantee (1) in a shared memory system is to make use of a hardware memory protection scheme, or during design eliminate shared resources. The analysis of execution orderings in Section 3 essentially corresponds to fault hypothesis (2), but we will also show how the analysis can be abstracted to a less discriminating model corresponding to fault hypothesis (1).

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. A formal definition of what actually constitutes an execution order scenario, and an algorithm that derives the scenarios for a given job set can be found in [18].

The fundament for the analysis is the Execution Order Graph (EOG), that describes which, when and in what order tasks are started, preempted and completed. The EOG represents the behavior of a strictly periodic preemptive fixed priority real-time kernel during one instance of the release pattern of a task set; typically equal to the LCM of the period times of involved tasks. It is sufficient from a determinism and reproducibility point of view to only represent one instance of the release pattern of a task set; typically equal to the LCM of the period times of involved tasks. It is

Figure 2 A Transition

\[ (a, b) \rightarrow A \]

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

\[ (a, b) \rightarrow [a', b') \rightarrow B \]

\[ (a, b) \rightarrow [\alpha, \beta] \rightarrow A \]

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 2 and 3.

The interval of possible start times \([a', b')\) for task \(B\), in Figure 3, is defined by:

\[ a' = \text{MAX}(a, r_j) + BCET_j \]

\[ b' = \text{MAX}(b, r_j) + WCET_j \]

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 4.

\[ (a, b) \rightarrow [\alpha, \beta] \rightarrow A \]

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

\[ \alpha = \text{MAX}(a, r_j) \]

\[ \beta = \text{MAX}(b, r_j) + WCET_j \]

i.e., the interval, \([\alpha, \beta]\), specifies the interval in which \(A\) can be preempted.

From each node 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 5.

2) The job has a WCET such that it definitely completes before the release of any higher priority job. In this case there is a single outgoing transition labeled with the interval of finishing times for the job, \([a', b')\). Exemplified by (1) in Figure 5.

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 (2) in Figure 5.

4) The job has a BCET and WCET such that it may either be preempted or completes 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) **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 (3) in Figure 5.

b) **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 5.

Note that, whenever a preemption of a job is represented in the EOG, we need to take into account the range of possible execution prior to preemption, so that when the job is resumed we can deduct this interval of execution from its remaining execution time.

**Example 1**

*Figure 5* gives an example of an EOG, using the above notation and the attributes in Table 1. In *Figure 5*, 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>
<thead>
<tr>
<th>Table 1 Job set</th>
<th>Job</th>
<th>(r)</th>
<th>(P)</th>
<th>BCET</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>(B)</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>(C)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>(D)</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

### 3.1.1 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 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.

### 3.1.2 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.

#### 3.2 Adding interrupts

We will now incorporate the temporal side effects of interrupts, by regarding them as sporadic jobs with a known minimum and maximum inter-arrival time. 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 however make the EOG practically intractable, since we do not know the release times (or phases) of the interrupts, and must therefore consider all possible release times within their inter-arrival times. We will here make use of standard response time analysis techniques [1][6] for calculating the execution- and completion intervals. This is rather pessimistic when the duration between releases of tasks are non-harmonious with the inter-arrival times of the interrupts, but we use this pessimistic and simple method to simplify the presentation. Thané [20], presents how to exactly calculate the interrupt interference on the execution order graph.

**Figure 5** The EOG for the job set in Table 1.
\( T_{k}^{\text{min}} \), a maximum inter-arrival time \( T_{k}^{\text{max}} \), a best case interrupt service time \( \text{BCET}_{k} \), and a worst case interrupt service time \( \text{WCET}_{k} \).

**Execution interval**

Considering the interrupt interference, the execution interval \([\alpha, \beta]\) (3-2) changes to:

\[
\alpha = \text{MAX}(a, r_{j}) \quad \beta = \text{MAX}(b, r_{j}) + w, \text{ where } w \text{ is the sum of } \text{WCET}_{j} \text{ and the maximum delay due to the preemption by sporadic interrupts, given by:} \\

w = \text{WCET}_{j} + \sum_{\forall k \in \text{int}} \left[ \frac{w}{T_{k}^{\text{min}}} \right] \cdot \text{WCET}_{k} \quad (3-3)
\]

Hence, we calculate the interrupt interference on the preemption intervals in the same way as Response Time Analysis (RTA) [6] is used to calculate the response times for jobs that are subjected to interference by preemption of higher priority jobs.

**Start time interval**

When adding interrupts, the upper bound \( b^{'} \) of the start time interval \([a', b']\) is still equal to \( \beta \), whereas the lower bound \( a' \) changes to:

\[
a' = a + w_{\alpha} \quad (3-4)
\]

Where \( w_{\alpha} \) is defined as:

\[
w = \text{BCET}_{A} + \sum_{\forall k \in \text{int}} \left[ \frac{w}{T_{k}^{\text{max}}} \right] \cdot \text{BCET}_{k} \quad (3-5)
\]

This equation captures that the minimum interference by the interrupts occur when they have their maximum inter-arrival time and execute with their minimum execution time, and when they have their lowest possible number of hits within the interval. The latter is guaranteed by the use of the floor function (\( \lfloor \rfloor \)).

**Execution times**

In EOG we decrease a preempted job \( j \)'s maximum and minimum execution time with how much it has been able to execute in the worst and best cases before the release time \( t \) of the preempting job. Since we are now dealing with interrupts, the effective time that \( j \) can execute prior to the preemption point will decrease due to interrupt interference. The remaining minimum execution time \( \text{BCET}_{j}^{'} \) is given by:

\[
\sum_{\forall k \in \text{int}} \left[ \frac{t - \text{MAX}(a, r_{j})}{T_{k}^{\text{min}}} \right] \cdot \text{BCET}_{k} \quad (3-6)
\]

Note that the sum of interrupt interference is not iterative, but absolute, because we are only interested in calculating how much the job \( j \) can execute in the interval, minus the interrupt interference.

Likewise we can calculate the remaining maximum execution time, \( \text{WCET}_{j}^{'} \):

\[
\text{WCET}_{j}^{'} = \text{WCET}_{j} - (t - \text{MAX}(b, r_{j}) - \sum_{\forall k \in \text{int}} \left[ \frac{t - \text{MAX}(b, r_{j})}{T_{k}^{\text{min}}} \right] \cdot \text{WCET}_{k} \quad (3-7)
\]

**Example 2**

Here we assume that the system is subjected to preemption by interrupts. The attributes are described in Table 2 and 3. The side effects of the interrupts are solely of temporal character. Figure 6 depicts the EOG without interrupt interference, and Figure 7 with interrupts accounted for. We observe that the interrupt may delay the execution of job \( A \) such that it will be preempted by job \( B \), in contrast with the behavior without interrupts, where \( A \) always completes before \( B \).

<table>
<thead>
<tr>
<th>Job</th>
<th>( r )</th>
<th>( p )</th>
<th>\text{BCET}</th>
<th>\text{WCET}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>( B )</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>( T_{\text{min}} )</th>
<th>( T_{\text{max}} )</th>
<th>\text{BCET}</th>
<th>\text{WCET}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I )</td>
<td>( \infty )</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 2 Schedule, \( J \)**

**Table 3 Interrupts**

**Figure 6** Not considering interrupts.

**Figure 7** With interrupts.
In defining the EOG 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.
- **Interrupt induced jitter**, i.e., execution time variations induced by the preemption of interrupts.

In previous work [18] we have also considered **Clock synchronization jitter**, i.e., that the local clocks keep on speeding up and down while synchronizing with the global time base, which leads to varying inter-arrival times between clock ticks.

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 ($WCET$), e.g. by inserting (padding) “no operation” instructions where needed.

Even if the $BCET = WCET$ for all tasks there would still be jitter if interrupts were interfering. In general, real-time systems with tighter execution time estimates and $WCET \approx BCET$, as well as interrupt inter-arrival times of $T_k^{max} = T_k^{min}$, and better precision of the clock synchronization yield better testability than systems with larger jitter.

### 3.3 Complexity

The complexity of the EOG, i.e., the number of different execution orderings, and the computational complexity of the algorithm [18], 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. This is not really inherent to the EOG but rather a reflection of the system it represents.

### 4 Towards systematic Testing

We will now outline a method for deterministic integration testing of distributed real-time systems, based on the identification of execution orderings. Testing of sequential programs (like single tasks) can be performed with regular unit testing [3]. We assume that some method for testing of sequential programs is used. Exactly which is not an issue here, since focus is on the parameters timing and order.

In order to perform integration testing of distributed real-times systems the following is required:

- A feasible global schedule, including probes that will remain in the target system in order to eliminate the probe effect.
- 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.
- 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.
- Control over, or at least a possibility to observe, the input data to the system with regard to contents, order, and timing.

#### Strategy

The test strategy consists of the following steps:

1. Identify the set of execution orderings by performing execution order analysis for the schedule on each node.
2. Test the system using any testing technique of choice, and monitor for each test case and node, which execution ordering is run during $[0, J^{BCET}]$.
3. Map test case and output onto the correct execution ordering, based on observation.
4. Repeat 2-3 until sought coverage is achieved.

#### Coverage

Complete coverage for a single node is defined by the traversal of all possible execution order scenarios. The coverage criteria for each scenario is however defined by the sequential testing method applied.

By limiting the scope of the tests to, e.g., the entire system, multiple transactions, single transactions or parts of transactions, we could also reduce the coverage criteria.

#### Other issues

For a system that keeps state between periods, we must monitor or control, the jobs’ internal variables, and not only the legal inputs defined by the jobs’ interfaces, in order to guarantee determinism and coverage.

#### 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, if we adhere to fault
hypothesis (1), 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 $[BCET, WCET]$.

5 Conclusion

In this paper we have introduced a method for achieving deterministic testing of distributed real-time systems (DRTS), which are subjected to interrupts, where the interrupt interference as been modeled as high priority sporadic tasks. 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 method for finding all the possible execution scenarios for a DRTS with preemption, jitter and interrupt interference.
- 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 when possible to increase testability.

Future pursuits include experimentally validating the usefulness of the presented results, and extending the technique to handle critical regions. It would also be useful to investigate how less pessimistic coverage criteria for DRTS could be developed, as well as devising testability increasing design rules for DRTS. Other pursuits would be to investigate the benefits of using the testability measure as a means for feedback, or a new heuristics in the generation of highly testable static schedules. The ambition is also to use the EOG for deriving tighter response time estimates for fixed priority scheduled systems.

Updated information on this work is available at www.mrtc.mdh.se/projects/tatoo.

6 Acknowledgements

We would like to thank Jukka Mäki-Turja, Christer Norström, Sasikumar Punnekkat, and Kristian Sandström for valuable discussions and for reviewing draft versions of this paper.

This work is funded by the national Swedish Real-Time Systems research initiative ARTES, supported by the Swedish Foundation for Strategic Research.

Mälardalen Real-Time research Centre (MRTC; www.mrtc.mdh.se) is a research centre in Västerås, Sweden, supported by Swedish industry, the Swedish Foundation for Knowledge and Competence Development (KK-stiftelsen) and Mälardalen University.

7 References


