TOWARDS SYSTEMATIC TESTING OF DISTRIBUTED REAL-TIME SYSTEMS

Henrik Thane, Hans Hansson

Mälardalen Real-Time Research Centre
www.mrtc.mdh.se
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, not only depends on the inputs but also on the *order* and *timing* of the concurrent tasks that execute and communicate with each other and the environment. Hence, sequential test techniques are not directly applicable, since they disregard the significance of order and timing.

In this paper we present a method for identifying all the possible orderings of task starts, preemptions and completions for tasks executing in distributed real-time systems. This allows test methods for sequential programs to be used, since we can regard each identified ordering as a sequential program.

The number of identified execution orderings can be used as an objective measure of the testability of the distributed real-time system. Such a measure is an important quality attribute, which can be utilized as a new scheduling optimization criterion for generating static schedules of high testability.

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

1 INTRODUCTION

A real-time system is per definition correct if it performs the correct *function* at the correct *time*. Using real-time scheduling theory we can provide guarantees that each task in the system will meet its timing requirements [Liu1973,Audsley1995,Xu1990], providing that the basic assumptions hold during run-time, e.g., task execution times and periodicity. However, scheduling theory does not give any guarantees for the system's functional behavior, i.e., that the computed values are correct. To assess the functional correctness other types of analysis are required. One possibility is to use formal methods to verify certain functional and temporal properties of a model of the system. The formally verified properties are then guaranteed to hold in the real system, as long as the model assumptions are not violated. When it comes to validating the underlying assumptions (e.g., execution times, synchronization order and the correspondence between specification and implemented code) we must use dynamic verification techniques which explore and investigate the run-time behavior of the real system, i.e., testing [Rushby1995a]. Testing can also be used as a complement to, or to replace formal methods, in the functional verification.

When a sequential program is tested, it is necessary to control the sequence of inputs, and the start conditions, in order to guarantee reproducibility [McDowell1989]. That is, given the same initial state and input, the sequential program will deterministically produce the same output on repeated executions, even in the presence of systematic faults [Rushby1995b]. Reproducibility is essential when performing regression testing or cyclic debugging [Schütz1994], where the same test cases are run

repeatedly with the intent to validate that either an error correction had the desired effect, or to simply make it possible to find the error when a failure has been observed.

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 also dependent on *when* the inputs arrive and *when* the programs execute and communicate with each other, and with the environment. Trying to apply test techniques of sequential programs to distributed real-time systems is therefore bound to lead to non-determinism (and non-reproducibility), because control is only forced on the inputs, disregarding the significance of order and timing. For instance, in a real-time system with shared resources different inputs may lead to different execution paths. These paths will in turn lead to different execution times for the tasks, which depending on the design may lead to different *orders* of access to the shared resources. As a consequence there may be different system behaviors if the outcome of the operations on the shared resources depend on the ordering of the accesses. Hence, in order to facilitate systematic testing of distributed real-time systems we must, in addition to observing inputs and outputs, observe (or control) the timing and order of the sequence of inputs, as well as the timing and order of program executions.

In a system with race conditions, which naturally occur in many real-time systems, the act of intrusively observing the system will always change the odds for the outcome of the race, because the probes will add to the execution times of the racing tasks. 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].

There are thus three main problems that need to be solved to make systematic testing of distributed real-time systems possible: (1) reproducing the inputs with respect 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, and (3) eliminating the probe-effect.

We are briefly going to discuss (1) and (3), but the focus of this paper is on (2) by presenting a method for deriving all the possible execution orderings for preemptive periodic real-time systems with fixed priorities, that are subjected to interrupts and jitter. Each identified execution order constitutes a scenario, which can be regarded as a single sequence of subroutine calls, where each continuous execution of a task corresponds to a subroutine, and thus a scenario could be viewed as a sequential program. We can thus test each scenario with traditional sequential testing methods.

We consider task sets with recurring release patterns, executing in a distributed system, where the scheduling on each node is handled by a priority driven preemptive scheduler. This includes statically scheduled systems that are subject to preemption [Xu1990], as well as strictly periodic fixed priority systems [Liu1973, Audsley1991]

If we run the system for a duration equal to the entire schedule (i.e., a single instance of the release pattern; typically equal to the Least Common Multiple (LCM) of the period times of the involved tasks) for a specific test case, and observe the actual execution scenario, we can identify the input and the produced outputs, including their timing, to be a test in the observed execution scenario, i.e., for the sequential program it represents. Hence, by classifying tests to belong to different scenarios we can, given that each scenario yields a deterministic execution (which is our hypothesis) achieve *deterministic testing*. The number of tested scenarios can be used as coverage criteria for testing of the system's behavior during an LCM period. This is all under the assumption that we can consistently observe the global state in the distributed real-time system. In order to guarantee this consistency we assume that the system is globally scheduled, i.e., the release and execution times can be related to "the global time", and has a global synchronized time base with a known precision.

If we also have the possibility to control the parameters: input, output, time and synchronization, not only observe them, we can enforce specific execution scenarios thereby achieving *reproducible testing.* Reproducibility increases the effectiveness of testing by eliminating redundant test cases, and by making it easier to achieve the desired level of coverage.

Paper outline: *Section 2* provides a discussion on the main problems in testing of distributed real-time systems, and how to deal with them. *Section 3* presents our system model. *Section 4* formalizes the concept of execution order and presents the algorithm for identifying all the possible execution

orderings in distributed real-time systems. We also give some examples, and extend the analysis to consider the effects of interrupts. *Section 5* suggests a testing strategy for achieving deterministic and reproducible testing in the context of execution order analysis. Finally, in *Section 6*, we conclude and give some hints on future work.

2 TESTING DISTRIBUTED REAL-TIME SYSTEMS

In this section we further discuss the handling of inputs, probe effects and reproducibility in testing of distributed real time systems.

2.1 Inputs

The problem with reproducing the inputs with respect to contents, order and timing has been addressed [Glass1980, DeMillo1987, Somerville 1992] specifically through the use of environment simulators. This is a problem inherent in all software testing. For example, the levels of reliability that can be assessed using experimental statistical methods for sequential programs is limited to about 10^{-4} failures/hour because of the difficulty to represent very rare events accurately [Rushby1995b]. The outlook of statistical assessment of real-time software does therefore look quite grim due to even more complex input data profiles. This is however an issue which we will not consider further in this paper.

2.2 Probe-effects

When it comes to dealing with the probe-effect in distributed real-time systems there are basically two approaches:

- Use special hardware (dual-port memories, etc.) that allows for transparent monitoring of the system [Plattner1984, Haban1990, Tsai1990, Tsai1996]. This has the severe drawback of being expensive and only enabling observations of certain aspects of the systems behavior, such as those related to the external interfaces of the micro-controller, shared resources (such as dual-port memories) and broadcast communication busses. The ever-increasing integration of functionality in general-purpose micro-controllers makes it even harder to observe the internal behavior of the system. The viability of this approach is therefore limited. However, the current trend of making application specific hardware using FPGAs and VHDL [Calvez1998] gives an opportunity to conveniently integrate non-intrusive monitoring mechanisms in the hardware.
- Use software for instrumentation [Dodd1992, Tokuda1988]. By including instrumentation code in the software (application and operating system), we can observe more than possible with the hardware approach. The main problem here is to eliminate the probe-effect. For a distributed real-time system we must allocate resources for the probes, including execution time, memory, communication bus bandwidth and account for the probes when scheduling [Thane1999]. Probes can be placed at several levels, including [Thane1999]:
  - *Kernel-probes* – that collect information from within the kernel, task-switches, execution times, etc.
  - *Inline-probes* – that are inserted into application tasks
  - *Probe-tasks* – that are tasks dedicated to collecting data from kernel-probes, inline-probes and other probe-tasks, and
  - *Probe-nodes* – which are dedicated nodes that collect data from probe-tasks, and which can monitor communication busses.

  By allocating resources for observation and then leaving them in place when the system is delivered, we eliminate the probe effect – any other approach will lead to probe-effects, or very limited observability. Some execution strategies, e.g., statically scheduled real-time systems, allow us to remove probes without temporal side-effects if they are situated within *temporal firewalls* [Schütz1994]. That is, as long as we do not change the start-times of tasks, and their times of output (communication or access to shared resources), we can remove the probes.

In this paper we assume that the probe-effect has been eliminated by allocating sufficient resources for probes (kernel-, in-line-, task- and node-level ones), and by scheduling the system with the probes as part of the design.

2.3 Reproducibility

For reproducing the execution order and timing of the tasks in a distributed real-time system the two main approaches are:

- *Deterministic replay*. Here the tasks' run-time behavior is recorded in a log over a period of time. The execution of the system can then be deterministically replayed off-line. The system cannot be suspended during run-time, but the off-line replay can be suspended and examined. If any modifications have been made to the tasks or to the inputs the logging must be repeated.

  A deterministic replay method for concurrent Ada programs is presented in [Tai1991]. They log the synchronization sequence (rendezvous) for a concurrent program *P*, with input *X*. The source code is then modified to facilitate replay; forcing certain rendezvous so that *P* follows the same synchronization sequence for *X*. This approach can reproduce the synchronization orders for concurrent Ada programs, but not the duration between significant events, because the enforcement (changing the code) of specific synchronization sequences introduces gross temporal probe-effects. The replay scheme is thus not suited for real-time systems, neither does it consider the effects of interrupts, and it is unclear how the method can be extended to handle interrupts.

  [Tsai1990] presents a hardware monitoring and replay mechanism. Their approach can replay significant events with respect to order, access to time, and asynchronous interrupts. Monitoring is apt for real-time systems because it minimizes the probe-effect. Although replay can be performed accurately it may be slower than real execution. The main disadvantages are that the approach needs special hardware and is intended for single-processor systems only. Adapting the approach to distributed real-time systems requires extensive hardware support and rework.

  Another software-based approach is *HMON* [Dodd1992], which is designed for the HARTS distributed (real-time) system multiprocessor architecture [Shin1991]. A general-purpose processor is dedicated to monitoring in each multiprocessor. The probe-effect due to monitoring is eliminated by modifying system service calls to incorporate monitoring mechanisms, and by letting these be present also in the final system. The events monitored are system calls, context switches, interrupts, shared variables references, and application specific events (chosen by the programmer). The recorded events can then order wise be deterministically replayed, in logical-time [Lamport1978], but not in real-time. The system can thus not actually monitor a consistent global state and then reproduce its real-time behavior.

  There are a few disadvantages with all the above approaches [Schütz1994]:

  - One can only replay what has previously been observed, and no guarantees that every significant system behavior will be observed accurately can be provided. Also, if a program has been modified (e.g., corrected) completely new traces have to be generated.
  - Dedicated (special) hardware has to be used in order to eliminate (or minimize) the probe-effect.
  - Since replay takes place at machine level the amount of information required is usually large. All inputs and intermediate events (e.g. messages) must be kept.

- *Deterministic testing*. A distributed real-time system can usually be described by a set of *use-cases*, which are sets of cooperating tasks that jointly perform specific functions, e.g., a sample-calculate-actuate loop in a control system. A precedence relation (execution order), interactions (data-flow), and a period time [Eriksson1996a] typically define each use-case. To test a use-case, we need to control the inputs and observe (or control) the execution order.

  Because a distributed real-time system may contain several use-cases that run on the same processor, there might be many different execution orderings, due to variations in preemption points, varying execution times, and interrupts.

  The method presented in this paper, aims at transforming the non-deterministic distributed real-time systems problem to a set of deterministic sequential program testing problems. This is

achieved by deriving all the possible execution orderings of the distributed system, and regarding each of them as a sequential program.

Related work can be found in [Yang1992], where all possible synchronization sequences (rendezvous) for Ada programs are identified, similar to [Tai1991], but [Yang1992] do not attempt deterministic replay, instead they test the system and consider the actual synchronization sequence as being part of the output. The number of synchronization sequences, and execution paths, exercised are used to define coverage. Similar work can be found in [Hwang1995] where they also attempt deterministic replay, though with the same side effects as [Tai1991]

3 THE SYSTEM MODEL

We assume a distributed system consisting of a set of nodes, which communicate via a broadcast network, that is asumed to be temporally predictable, i.e., upper bounds on communication latencies are known or can be calculated [Kopetz1994,Tindell1995, Eriksson1996b]. 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 [Kopetz1987, Eriksson1996b] with a known precision $\boldsymbol{d}$, meaning that no two nodes in the system have local clocks differing by more than $\boldsymbol{d}$.

The software that runs on the distributed system consists of a set of concurrent tasks, communicating by message passing. The tasks are geographically distributed over the nodes, typically with more than one task on each node. All synchronization is resolved before run-time. As a consequence no action is needed to enforce synchronization in the actual program code. Mutual exclusion and precedence is guaranteed by the different release-times. 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 [Kopetz1994].

Task model

We assume a set of jobs (i.e. invocations of tasks) J that are released in a time interval [0, J_{max}]. 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 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 the schedule is feasible, i.e., that each job j is always completed within its deadline D_j.

We additionally assume a set of interrupts I, where each interrupt $k \in I$ has the following attributes

- Minimum, and maximum inter-arrival time (T_k^{max} and T_k^{min}, respectively), priority p_k (interrupts can preempt each other), as well as worst and best case execution time of the interrupt routines ($WCET_k$ and $BCET_k$, respectively).

We finally assume that the system is preemptive (both by jobs and interrupts) and that jobs may have identical release-times.

Relation to other task models

The above task model is fairly general since it includes both preemptive scheduling of statically generated schedules [Xu1990] and fixed priority scheduling of strictly periodic tasks [Liu1973, Audsley1991].

To see how static periodic scheduling can be mapped to our task model consider a static schedule for a set of periodic tasks T, where each task $T_i \in T$ has the following attributes and relations to other tasks:

- Period (T_i), release time (r_i; start time relative period start), deadline (D_i; latest completion time relative r_i), worst case execution time ($WCET_i$), best case execution time ($BCET_i$), and priority (p_i; each priority is unique).
- Relations between tasks can be specified by precedence relations ($T_i \rightarrow T_j$; the execution of T_i preceeds that of T_j), mutual exclusion relations ($T_i \# T_j$; the execution of T_i does not overlap with that of T_j) and communications ($T_I Y T_j$; at the end of its execution T_i sends a message to T_j).

The length of the generated schedule will be the least common multiple of the period times of the involved tasks, $LCM(T)$; corresponding to J_{max} in our task model. For tasks with periods less than

LCM(T), multiple releases will be made in the interval [0, J_{max}], e.g., for a task *i* with $2T_i = LCM(T)$ there will be two corresponding jobs in our task model, released at r_i and r_i+T_i.

The static schedule fulfills the temporal requirements by construction. Schedules with different properties and for different task models can be constructed, e.g., allowing preemption [Xu1990], giving each task a fixed start time [Xu1990], assigning the same start time to several tasks [Xu1990], and taking interrupts into account [Sandström1998].

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) [Kopetz1989, Rubus1995, Eriksson1996a].

Fault hypotheses

Note that, although synchronization is resolved by the off-line selection of release times, we cannot dismiss unwanted synchronization side-effects, because the schedule design can be erroneous, or the assumptions about the execution times might not be accurate due to poor execution time estimations, 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 transactions interior to 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 levels of fault hypotheses:

1. Errors can only occur due to erroneous outputs and inputs (messages) 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.

The only possibility to guarantee (1) in a shared memory system is to use a hardware memory protection scheme, or to by design eliminate shared resources.

The analysis of execution orderings in Section 4 corresponds to fault hypothesis (2), but we also show how the analysis can be abstracted to a less discriminating model corresponding to fault hypothesis (1).

4 EXECUTION ORDER ANALYSIS

In this section we present a method for identifying all the possible orders of execution for job sets conforming to the task model introduced in Section 3. We will also show how the model and analysis can be extended to account for the interference caused by interrupts.

We will first present analysis for the single node case and then in Section 4.7 indicate how this analysis can be extended to the multi-node distributed system case.

4.1 Execution Orderings

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

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

- Job preemptions, i.e., the points in time when jobs are preempted by other jobs. We will use $Pp(J)$ to denote the set of job preemptions when executing J; $Pp(J) \subseteq [0, J^{MAX}] \times J \times J$, where each triple $(time, j_1, j_2) \in Pp(J)$ denotes a point in time when job j_1 is preempting job j_2.

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

- $ev.time$ denotes the time of the event ev,
- $E\backslash j$ denotes the set of events in the execution E related to job j,
- $E\backslash I$ denotes the set of events in E that occur in the time interval I,
- $First(E)$ and $Last(E)$ denote the first and last events in E, respectively,
- $nxt(E,ev,t)$ denotes the next occurence of event ev in E after time t; $nxt(E,\_,time)$ denotes the next occurence of any event in E after time t, and
- $prec(E, t)$ denotes the event in E that occurred most recently in the past at time t (including events that occur at t).

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

1. For each $j \in J$, there is exactly one start and termination event in E, denoted $s(j,E)$ and $e(j,E)$ respectively.
2. For each $(t, j_1, j_2) \in E \cap Pp(J)$, $p_{j1} > p_{j2}$, i.e., jobs are only preempted by higher priority jobs.
3. For each $j \in J$, $s(j,E) \geq r(j)$, i.e., jobs may only start to execute after being released.
4. After its release, the start of a job may only be delayed by intervals of executions of higher priority jobs, i.e., using the convention that $E\backslash[j,j)=\emptyset$, for each job $j \in J$: each event $ev \in E\backslash[prec(r(j)), s(j,E))$ is either
   - A start of the execution of a higher priority job, i.e. $ev = s(j',E)$ and $p_{j'} > p_j$
   - A preemption, where a higher priority job preempts a high priority job, i.e. $ev = (t, j', j'')$, $p_{j'} > p_j$ and $p_{j''} > p_j$
   - A job termination, at which a higher priority job resumes its execution, i.e., $ev = (j', t, j'')$, where $p_{j''} > p_j$
5. The sum of execution intervals of a job $j \in J$ is in the range $[BCET(j), WCET(j)]$, i.e.,

$$BCET(j) \leq \sum_{ev \in E\backslash j - \{e(j,E)\}} nxt(E,ev,ev.time) - ev.time \leq WCET(j)$$

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

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

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

- $First(E) = ev_0$
- $Last(E) = ev_k$
- For any $j \in [0..(k-1)]$: $nxt(E,\_,evj.time) = ev_{j+1}$

Intuitively, $EX_o(J)$ is constructed by extracting one representative of each set of equivalent execution orderings in $EX_t(J)$, i.e., using a quotient construction $EX_o(J) = EX_t(J)\backslash \sim$, where $\sim$ is the equivalence induced by considering executions with identical event orderings to be equivalent. This corresponds to our fault hypthesis (2), with the relaxation that we only keep track of the timing of preemptions, not exactly where in the program code they occur. If we assume fault hypotheses (1), the set of execution

orderings can be reduced further, since the preemptions are of no significance in this case, i.e., we can define $EX_o(J) = EX_t(J)\backslash \approx$, where $\approx$ is the equivalence induced by considering executions with identical start and stop event orderings to be equivalent. 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 interchangebly.

4.2 Calculating $EX_o(J)$

This section outlines a method to calculate the set of execution orderings $EX_o(J)$ for a set of jobs *J*. We will later (in section 4.4) present an algorithm that perform this calculation. In essence, our approach is to make a reachability analysis by simulating the behavior of the kernel during one $[0,J^{MAX}]$ period for the job set *J*. We use a small example to present the underlying intuition.

Consider *figure 1a*, which depicts an $[0,J^{MAX}]$ schedule generated by a static off-line scheduler and where the execution times for the jobs *A*, *B*, and *C* are fixed, i.e. $WCET_{\{A,B,C\}}=BCET_{\{A,B,C\}}$. Later release time gives higher priority. This has the effect of only yielding one possible execution scenario during $[0,J^{MAX}]$. However, if for example, job *A* had a minimum execution time of 2 ($BCET_A=2$; $WCET_A=6$ as before) we would get three possible execution scenarios, depicted in *figure 1*. The cause is that, in addition to the previous execution scenario, there are now possibilities for *A* to complete before *B* is released (*figure 1b*), and for *A* to complete before *C* is released (*figure 1c*).

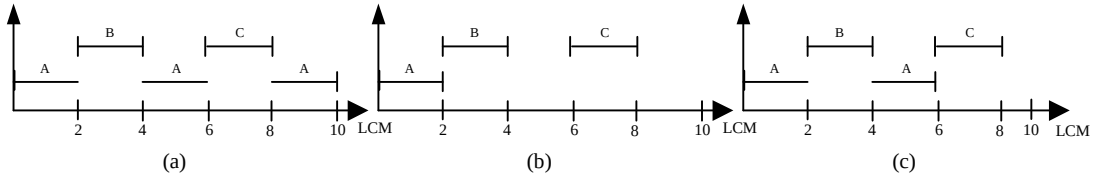


Figure 1. Three different execution order scenarios when job *A* has $BCET_A=2$ and $WCET_A=6$.

4.3 The Execution Order Graph (EOG)

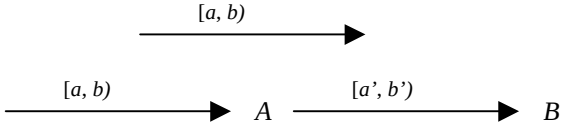
The algorithm we are going to present generates, for a given schedule, an Execution Order Graph (EOG), which is a finite tree for which the set of possible paths from the root contains all possible execution scenarios.

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

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

Intuitively,

- An **edge**, corresponds to the transition (task-switch) from one job to another. The edge is annotated with a continuous interval of when the transition can take place.



Where the interval $[a', b')$ is defined by: (4-1)

$$a' = MAX(a, r_A) + BCET_A$$

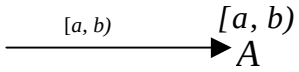
$$b' = MAX(b, r_A) + WCET_A$$

That is, $a´$ is the earliest start time of job *B*. Which is the immediately preceding job's (in this case *A's)* earliest start time, a or *A*'s earliest scheduled release time, r_A, added with *A*'s minimum execution time, $BCET_A$.

The $b´$ represents the latest possible release-time for job *B*, obtained by adding the latest possible release time for job *A* with $WCET_A$, the maximum execution time for *A*.

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

- A **node**, is a job annotated with a continuous interval of its possible execution, as illustrated by the following edge leading to a node:



Using the convention that intervals $[q, q) = [q, q]$, we define the interval of execution, $[a, b)$ by:

$$a = MAX(r_A, a) \qquad (4\text{-}2)$$

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

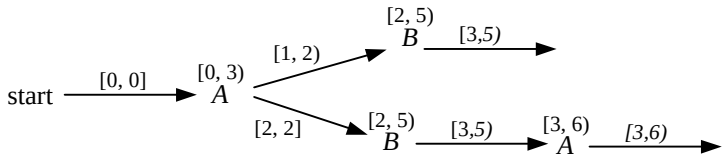
The interval, $[a, b)$, specifies thus the interval when *A* can be preempted

From each node there can be one or more transitions, representing one of four different situations:

1. That 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 no destination node.
2. That the job has a *WCET* such that it definitely completes before the release of any preempting job. In this case there is a single outgoing transition labeled with the interval of finishing times for the job.
3. That 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 interval when the preemption may occur, [*t(preem),t(preem)*], where *t(preem)* is the preemption time.
4. That the job has a *BCET* and *WCET* such that it may either be preempted or completes before any preempting job is relaeased. In this case there will be two outgoing edges, one representing the completion (labeled with the time interval when the completion may occur), the other representing the preemption (labeled with the preemption time).

Example 1

Job	Release time	Min execution time	Max execution time
A	0	1	3
B	2	1	3



The graph gives an example of how an EOG could look like, using the above notation, and the attributes in the table.

4.4 The EOG algorithm

In defining the execution order analysis 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*. That is, using *I.l* and *I.r* to denote the end points of *I*, **min**($\{r_j \mid j \in J \wedge I.l \leq r_j < I.r\}$). If no such job exists then ∞ is returned
3. Make_ready(*t, rdy*) – adds all jobs that are released at time *t* to *rdy*. Returns FALSE if *t*=∞, else TRUE.
4. X(*rdy*) denotes the job with highest priority in *rdy*
5. Arc(*n, I, n'*) creates an edge from node *n* to node *n'* and labels it with the time interval *I*.
6. Make_node(*j, XI*) creates a node and labels it with the execution interval *XI* and the id of job *j*.

The execution order graph for a set of jobs *J* is generated by a call Eog(*ROOT*, {}, 0, 0, [0, J^{MAX}]), i.e., with a root node, an empty ready set, and the release-times *a* and *b* set to zero, plus the considered interval *SI*.

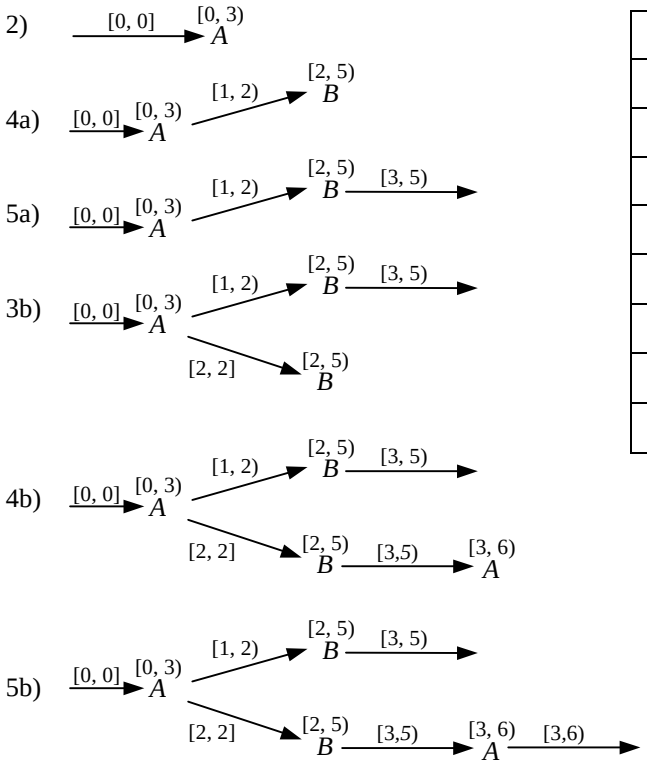
```
// n- previous node, rdy- set of ready jobs, a to b – release interval, SI – the considered interval.
Eog(n, rdy, a, b, SI)
{
    t = Next_release(SI)                                  //What is the time of the next release ?
    if (rdy = ∅)                                          // Then there are no jobs ready for execution
        if( Make_ready(t, rdy) = TRUE)                    //Make more jobs ready to execute?
                Eog (n, rdy, a, b, (t, SI.r] )            // Yes there were some jobs
        else    Arc(n, [a,b), _ )                         //No more jobs to execute – simulation done

    else                                                  // We have a set of ready jobs for execution
        T = X(rdy)                                        // copy the properties of the highest priority job in rdy.
        [a, b ) = [ max(r_T, a), max(r_T, b) + WCET_T)    //Calculate the execution interval for T.   (*A)
        a' = a + BCET_T                                   //Earliest completion time.                 (*B)
        b' = b                                            //Latest completion time.                   (*C)
        n'=Make_node(T, [a, b) )          // Create a node and label it with the execution interval [a, b ).
        Arc(n, [a, b), n')                // Create an arc from the previous node to the node representing T.
        if(t < b)                  // Does the next scheduled job have a release time less than T's completion time?
                if(t > a' )                               // Can T complete prior to the release of the next job at t?
                                                          //Yes T can complete.
                        Eog ( n', rdy \{T}, a', t, [t, SI.r] )
                                                                  // More jobs to execute
                Make_ready(t, rdy)                        //Add all jobs that are released at time t.
                if(T = X(rdy))                            //Do the new jobs have lower priority than T?
                        Eog( n', rdy \{T}, a', b',[t, SI.r] )     // Yes
                else                                              // No, they preempt
                        rdy = rdy \{T}
                        BCET_T = max(BCET_T - (t – (max(r_T , a)), 0) //Best case execution prior to preemption?  (*D)
                        WCET_T = max(WCET_T - (t – (max(r_T , b)), 0) // Worst case…                              (*E)
                        Eog( n', rdy + {T}, t, t, (t, SI.r] )
        else                                  //The next higher priority job starts later than T's latest completion time
                if (t ≠ ∞ )
                        Eog( n', rdy \{T}, a', b',[t, SI.r] )     //Next higher priority job
                else                                              // t = ∞ no more jobs to execute
                        Eog( n', rdy \{T}, a', b',[ ∞,∞] )
}//End
```

Example 2

Here we give an example of the resulting execution order graph (EOG) generated by the EOG algorithm for a set of jobs *J*

Schedule *J*				
Job	*r*	*p*	*BCET*	*WCET*
A	0	1	1	3
B	2	2	1	3



No.	Recursive call sequence
1	Eog(*ROOT*, {}, 0, 0, [0, J^{MAX}]
2	Eog(*ROOT*, {*A*}, 0, 0, (0, J^{MAX}]
3a	Eog(*node(A)*, {}, 1, 2, [2, J^{MAX}]
4a	Eog(*node(A)*, {*B*}, 1, 2, (2, J^{MAX}]
5a	Eog(*node(B)*, {}, 3, 5, [∞,∞]
3b	Eog(*node(A)*, {*A, B*}, 2, 2, (2, J^{MAX}]
4b	Eog(*node(B)*, {*A*}, 3, 5, [∞,∞]
5b	Eog(*node(A')*, {}, 3, 6, [∞,∞]

Table 1 A trace of the recursive calls to the EOG algorithm.

Table 1, gives a trace of the recursive calls to the algorithm Eog() until it terminates. Corresponding to each call that adds to the EOG graph, a graphical trace of how the EOG construction progresses, is shown on the left side of the table.

Completion times

An interesting property of the EOG is that we can easily find the worst and best-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 instance of a job that runs several times during an LCM cycle.

Complexity

The complexity of the algorithm (the number of recursion calls and the number of different execution orderings) 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 (just one scenario) to exponential complexity in cases with large jitter.

4.5 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 are interested in addressing the functional side effects of the interrupts, we will have to model each interrupt as a job. This would however make 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 periodicity. We would thus end up with an infinite number of paths in practice.

Execution interval

The execution interval **[***a, b)* (location **A* in the algorithm) changes to:

a *= MAX(r_A, a),* i.e., keeps the same value as in (4-2).

b *= MAX(r_A, b) + w,* i.e., where *w* is the sum of $WCET_A$ and the maximum delay due to the preemption by sporadic interrupts.

The formula for calculating *w* is similar to Response Time Analysis (RTA) calculations [Joseph1986]. As, RTA calculates the response times for jobs that are subjected to interference by preemption of higher priority jobs, we here calculate the interrupt interference on the preemption intervals.

The maximum value of *w* is defined by (4-3) 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.

$$w = WCET_A + \sum_{k \in interrupts} \left\lceil \frac{w}{T_k^{min}} \right\rceil \cdot WCET_k \qquad (4\text{-}3)$$

This equation can be solved using the standard RTA iteration technique [Joseph1986,Audsley1995].

Release time interval

Adding interrupts, the release time interval *[a´, b']* (location **B* in the algorithm) changes to:

For the lower bound, ***a´***, we are interested in finding the minimum interference by the interrupts, which 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 ***a'***, is defined by:

$$a' = a + w_a \qquad (4\text{-}4)$$

Where w_a, is defined as:

$$w = BCET_A + \sum_{k \in interrupts} \left\lfloor \frac{w}{T_k^{max}} \right\rfloor \cdot BCET_k \qquad (4\text{-}5)$$

The upper bound of the release interval ***b'***, is identical to the new ***b,*** (location **C* in the algorithm.)

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 best cases before the release time *t* of the preempting job.

Affected parts in the algorithm are *(*D)* and *(*E).*

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 $BCET_T$' is given by:

$$BCET_T' = BCET_T - (t - MAX(a, r_T) - \sum_{\forall interrupts\ k} \left\lfloor \frac{t - MAX(a, r_T)}{T_k^{max}} \right\rfloor \cdot BCET_k) \qquad (4\text{-}6)$$

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, $WCET_T'$

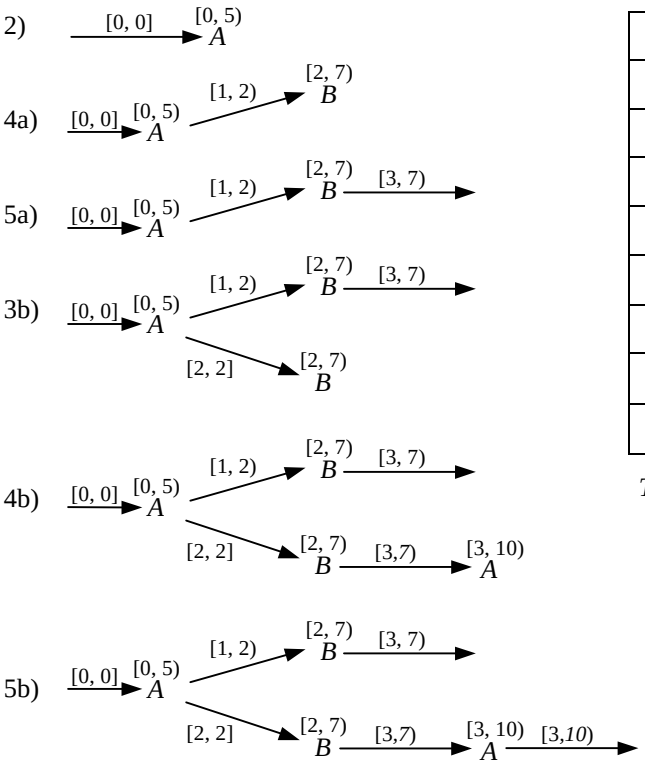
$$WCET_T' = WCET_T - (t - MAX(b, r_T) - \sum_{k \in interrupts} \left\lceil \frac{t - MAX(b, r_T)}{T_k^{min}} \right\rceil \cdot WCET_k) \qquad (4\text{-}7)$$

Example 3

Here we assume that the system is subjected to preemption by interrupts. The side effects of the interrupts are solely of temporal character. We make use of the same parameters as in example 2, and we assume that there is only one interrupt, *I* interfering.

Schedule, *J*				
Job	*r*	*p*	*BCET*	*WCET*
A	0	1	1	3
B	2	2	1	3

Interrupts				
Interrupt	T^{max}	T^{min}	*BCET*	*WCET*
I	∞	3	1	1



No.	**Recursive call sequence**
1	Eog(*ROOT*, {}, 0, 0, [0, J^{MAX}]
2	Eog(*ROOT*, {*A*}, 0, 0, (0, J^{MAX}]
3a	Eog(*node(A)*, {}, 1, 2, [2, J^{MAX}]
4a	Eog(*node(A)*, {*B*}, 1, 2, (2, J^{MAX}]
5a	Eog(*node(B)*, {}, 3, 7, [∞,∞]
3b	Eog(*node(A)*, {*A, B*}, 2, 2, (2, J^{MAX}]
4b	Eog(*node(B)*, {*A*}, 3, 7, [∞,∞]
5b	Eog(*node(A')*, {}, 3, 10, [∞,∞]

Table 2 A trace of the recursive calls to the EOG algorithm.

Table 2, gives a trace of the recursive calls to the algorithm Eog() when it incorporates the effects of interrupts. For each call that adds to the EOG graph, there is on the left side of the table a graphical trace of how the EOG construction progresses. Note that the release-times of the jobs are affected by the interrupts, and therefore also affecting the response times.

4.6 Jitter

In the presented algorithms we take the effects of several different types of jitter into account:

- *Schedule jitter*, i.e., different instances of the same job in an LCM may have different relative release times for each period.
- *Release jitter*, i.e., the inherited and accumulated jitter due to varying execution times for preceding higher priority jobs.

- *Completion jitter*, i.e., the response-time jitter due to schedule jitter, release jitter and the variation of the execution time for the job (*Execution time jitter*).

One type of jitter that we have not explicitly addressed, is the jitter inherent in global clock synchronization. The local clocks keep on speeding up, and down, depending on the global time base. This jitter could possibly 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 preemption "hit" window, and thus 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 that less jitter gives better testability. Jitter reduction could be achieved by padding; the execution time for each execution path through a task is equalized by "no operation" instructions. Other possibilities, are to make use of delays in the code, or having a kernel that does not release a lower priority task until a preceding higher priority task has used up its scheduled WCET.

4.7 The multi-node case

The presented algorithm generates an EOG for a schedule executed on a single node. In a distributed system with multiple nodes (and schedules) we can use the algorithm to generate one EOG for each node. From these, a global EOG describing all globally possible execution orderings can be constructed.

In the case of perfectly synchronized clocks this essentially amounts to perform a parallel composition of the individual EOGs, using standard techniques for composing timed transition systems [Sifakis1996]. In other cases, we first need to introduce the timing uncertainties caused by the non-perfectly synchronized clocks in the individual EOGs. The details of this are presented in [Thane1999].

5 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 [Sommerville1992, Beizer1990], using a black-box or white-box approach [Beizer1990].

We assume that some method for testing of sequential programs is used. Exactly which is not an issue here, since our focus is on handling multiple execution scenarios.

Assumptions

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

- A feasible global schedule, including probes.
- Kernel-probes on each node, which monitor task-switches. This information is sent to probe-tasks so that they can identify execution orderings from the task-switch information and correlate it to run test cases.
- A set of in-line probes and probe-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.

Test 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 which execution ordering is run during $[0,J^{MAX}]$.
3. Map test case and output onto the correct execution ordering, based on observation.
4. Repeat 2-3 until sought coverage is achieved.

Coverage

Depending on what we want to test: the entire system, multiple use-cases (running over several nodes), single use-cases (running over several nodes), multiple use-cases (single node), single use-cases (single node) or parts of use-cases, we need different levels of coverage.

- *Distributed coverage***.** When we test use-cases that run over more than one node we need to consider the global EOG, introduced in Section 4.7. When testing the system we can combine observations of local execution scenarios on the different nodes to identify the executed global scenario.
- *Node coverage*. When we test use-cases that run on a single node, we simply follow the test strategy. However the coverage needs to be defined, and related to the use-case(s) tested. A use-case may run several times during a $[0,J^{MAX}]$ period, and it might therefore be necessary to give special attention to which instance gets which input.

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 use-case this can be achieved by controlling the execution times of preceding and preempting jobs that belong to other use-cases, these have no functional relation to the use-case under test. This of course only works if we adhere to fault hypothesis (1), that the jobs have no unwanted functional side effects, otherwise we would miss such errors. Control over the execution times in other use-cases 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].

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. Given that testability is an important system characteristic, we can use this measure, as a new optimization criterion in the heuristic search used to generate static schedules [Xu1991], thereby generating schedules that are more testable.

6 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 Distributed Real-Time Systems (DRTSs). The results can be summed up to:

- We have provided a method for finding all the possible execution scenarios for a DRTS with preemption, interrupts and jitter.
- We have proposed a test 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 DRTSs.

- The number of execution orderings is an objective measure of the testability of DRTS, which can be used as a new scheduling optimization criteria for generating static schedules that are easier to test.
- Jitter increases the number of execution scenarios, and jitter reduction techniques should thus be used when possible to increase testability.
- A positive side effect of the execution order analysis is that we get exact response-times for the jobs, even when interrupts afflict the system.

Future pursuits include to experimentally validate the usefulness of the presented results, extend the testing methodology, devise testability increasing design rules for DRTS, employ the technique for exact calculations of response-times in fixed priority scheduled systems, and to investigate the benefits of using the testability measure as a new heuristics in the generation of highly testable static schedules.

7 REFERENCES

[Audsley1991] N. C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. *Hard Real-Time Scheduling: The Deadline Monotonic Approach.* IFAC/IFIP Worshop, Atlanta, Georgia, pp. 127-132, May, 1991

[Audsley1995] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell. *Fixed Priority Pre-emptive Scheduling: A Historical Perspective.* Real-Time Systems, The Int. Journal of Time-Critical Computing Systems, Vol. 8(2/3), March/May, 1995.

[Beizer1990] B. Beizer. *Software testing techniques.* Van Nostrand Reinhold, 1990.

[Calvez1998] J.P. Calvez, and O. Pasquier. *Performance Monitoring and Assessment of Embedded HW/SW Systems.* In Design Automation for Embedded Systems, vol. 3, pp. 5-22, 1998.

[DeMillo1987] R. A. DeMillo, W.M. McCracken, R.J. Martin, and J.F. Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings Publ. Co., 1987.

[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.

[Eriksson1996a] C. Eriksson, J. Mäki-Turja, K. Post, M. Gustafsson, J. Gustafsson, K. Sandström, and E. Brorsson. *An Overvies of RTT: A design Framework for Real-Time Systems.* Journal of Parallel and Distributed Computing, August, 1996.

[Eriksson1996b] C. Eriksson, H.Thane and M. Gustafsson. *A Communication Protocol for Hard and Soft Real-Time Systems.* In the proceedings of the 8th Euromicro Real-Time Workshop, L'Aquila Italy, June, 1996.

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

[Glass1980] R. L. Glass. *Real-time: The "lost world" of software debugging and testing.* Communications of the ACM, vol. 23(5), pp. 264-271, May, 1980.

[Haban1990] D. Haban and D. Wybranietz. *A Hyrbid monitor for behavior and performance analysis of distributed systems.* IEEE Trans. Software Engineering, 16(2), pp. 197-211, Feb., 1990.

[Hwang1995] G.H Hwang, K.C Tai and T.L Huang. *Reachability Testing: An Approach to Testing Concurrent Software.* Int. Journal of Software Engineering and Knowledge Engineering, vol. 5(4), pp. 493-510, 1995.

[Joseph1986] M. Joseph and P. Pandya. *Finding response times in a real-time system.* The Computer Journal – British Computer Society, 29(5), pp.390-395, October, 1986.

[Kopetz1987] H. Kopetz. *Clock Synchronisation in Distributed Real-Time Systems*. IEEE Trans. Computers, Aug. 1987.

[Kopetz1989] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*. IEEE Micro, no. 9, pp. 25- 40, 1989.

[Kopetz1994] H. Kopetz and H Grünsteidl. *TTP - A Protocol for Fault-Tolerant Real-Time Systems*. IEEE Computer, January, 1994.

[Lamport1978] L. Lamport. *Time, Clocks and the ordering of Events in a Distributed System.* Communications of the ACM, vol. 21, July, 1978.

[Laprie1992] J.C. Laprie. *Dependability: Basic Concepts and Associated Terminology*. Dependable Computing and Fault-Tolerant Systems, vol. 5, Springer Verlag, 1992.

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

[Liu1973] C. L. Lui and J. W. Layland. *Scheduling Algorithms for multiprogramming in a hard real-time environment*. Journal of the ACM 20(1), 1973.

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

[Plattner1984] B. Plattner. *Real-time execution monitoring.* IEEE Trans. Software Engineering, 10(6), pp. 756-764, Nov., 1984.

[Rubus1995] Rubus OS. *Real-Time Operating System, Tutorial*. Arcticus Systems AB Datavägen 9A, 175 62 Järfälla, Sweden 1995.

[Rushby1995a] J. Rushby. *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.

[Rushby1995b] J. Rushby. *Formal methods and their Role in the Certification of Critical Systems*. 12th Annual CSR Workshop, Bruges 12-15 September 1995. Proceedings, pp. 2-42. Springer. ISBN 3-540-76034-2.

[Sandström1998] K. Sandström, C. Eriksson, and G. Fohler. *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*. In proceedings of the 5th Int. Conference on Real-Time Computing Systems and Applications (RTCSA'98). October 1998, Japan.

[Schütz1994] W. Schütz. *Fundamental Issues in Testing Distributed Real-Time Systems*. Real-Time Systems, vol. 7(2), pp. 129-157, 1994.

[Shin1991] K. G. Shin. *HARTS: A distributed real-time architecture.* IEEE Computer, 24(5), pp. 25-35, May, 1991.

[Sifakis1996] J. Sifakis and S. Yovine. *Compositional specification of timed systems.* In Proc. 13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96, Grenoble, France, Lecture Notes in Computer Science 1046, pp. 347-359. Springer Verlag, February 1996.

[Sommerville1992] I. Sommerville. *Software Engineering.* Addison-Wesley, 1992. ISBN 0-201-56529-3.

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

[Thane1999] H. Thane. *Towards predictable instrumentation of distributed real-time systems.* Technical report, Mälardalen Real-Time Research Centre, Dept. Computer Engineering, Mälardalen University, 1999.

[Tindell1995] K. W. Tindell, A. Burns, and A.J. Wellings. *Analysis of Hard Real-Time Communications.* Journal of Real-Time Systems, vol. 9(2), pp.147-171, September 1995.

[Tokuda1988] H. Tokuda, M. Kotera, and C.W. Mercer. *A Real-Time Monitor for a Distributed Real-Time Operating System*. In proc. of ACM Workshop on Parallel and Distributed Debugging, Madison, WI, pp. 68-77, May, 1988.

[Tsai1990] J.P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging.* IEEE Trans. on Software Eng. vol. 16, pp. 897 - 916, 1990.

[Tsai1996] J.P. Tsai, Y.-D. Bi, S. Yang, and R. Smith. *Distributed Real-Time System: Monitoring, Visualization, Debugging, and Analysis.* Wiley-Interscience, 1996. ISBN 0-471-16007-5.

[Xu1990] J. Xu and D. Parnas. *Scheduling processes with release times, deadlines, precedence, and exclusion, relations*. IEEE Trans. on Software Eng. vol. 16, pp. 360-369, 1990.

[Xu1998] J. Xu and D. Parnas. *Priority Scheduling Versus Pre-Run-Time Scheduling*. In Proc. IFAC Real Time Programming, Shantou, Guangdong Province, P.R China, 1998

[Yang1992] R-D Yang and C-G Chung. *Path analysis testing of concurrent programs*. Information and software technology. vol. 34(1), January 1992.