DESIGN FOR DETERMINISTIC MONITORING OF DISTRIBUTED REAL-TIME SYSTEMS

5/9/00

Henrik Thane

Mälardalen Real-Time Research Centre
www.mrtc.mdh.se
henrik.thane@mdh.se

ABSTRACT

In order to test, or debug, a system we must *observe* its run-time behavior and deem how well the observations comply with the system requirements. There are two significant differences between debugging and testing of software for desktop computers and embedded real-time systems: (1) It is more difficult to observe embedded computer systems, simply because they are embedded, and that they thus have very few interfaces to the outside world, and (2) the actual act of observing a real-time systems or distributed real-time system can change their behavior.

Monitoring of sequential software is straightforward, but for distributed real-time systems it is more complicated, since race conditions with respect to order of access to shared resources occur naturally. Any intrusive observation, or probing, of the distributed real-time system affects the timing and consequently the outcome of the races.

In this paper we present a method for deterministic observations of single tasking, multi-tasking, and distributed real-time systems. This includes a description of what to observe, how to eliminate the disturbances caused by the actual act of observing, how to correlate observations, and how to reproduce them.

Keywords: Monitoring, testing, debugging, deterministic replay, reproducibility, determinism, testability, distributed real-time systems.

1 INTRODUCTION

In order to dynamically verify a system, e.g., to test or debug it, we must *observe* its run-time behavior and deem how well these observations comply with the system requirements. Fundamental in all physical sciences, as well as in testing of software, is the non-ambiguity, or determinism, of observations, and the ability to reproduce observations. Of equal importance is that the actual act of observation does not disturb, or intrude on, system behavior. If nonetheless the observations are intrusive then it is imperative that their effect can be calculated and compensated for. If we cannot, there are no guarantees that the observations are accurate or reproducible.

Race conditions with respect to order of access to shared resources occur naturally in multi-tasking real-time systems. Different inputs to the racing tasks 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 accesses.

Example 1-1.

Consider figures *1-1* and *1-2*. Assume that two tasks *A* and *B*, use a shared resource *X*, which they both do operations on, and that the resource *X* is protected by a semaphore *S*. Task *B* has higher priority than task *A*. Depending on the inputs, the execution time of task *A* will vary, which will result in different accesses to the shared resource:

(1) *Figure 1-1* illustrates a scenario, in which task *B* locks the semaphore, and enters the critical region before task *A*. Task *B* then preempts *A* and performs an operation on *X*. The new value of *X* is $B(X)$. The entire transaction will yield a value of *X* corresponding to $A(B(X))$.

(2) *Figure 1-2* illustrates a different scenario, in which task *A* terminates before task *B* is released, and thus performs an operation on *X* before *B*. The new value of *X* is $A(X)$. The entire transaction will yield a value of *X* corresponding to $B(A(X))$.

If we now add a probe to task *A*, in order to test its behavior, we may extend its execution time so that only scenario (1) is run. Consequently scenario (2) will never be executed during run-time, and if $B(A(X))$ is erroneous due to e.g., an error in task *A*, this will not be revealed during testing. If we later, after satisfactory testing, remove the probe in task *A*, scenario (2) may occur again and the erroneous calculation $B(A(X))$ may be executed, leading to a failure. This non-deterministic effect of intrusively observing a system is called the *probe-effect* [18][42] or the *Heisenberg uncertainty in software* [36][41].

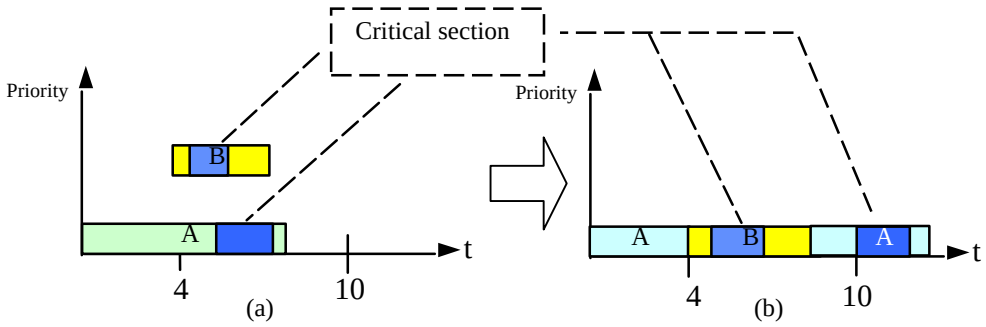


Figure 1-1. The releases of tasks A and B - figure (a). Where task B has higher priority than task A. Task B enters the critical section before A, when A has its worst case execution time. Figure (b) depicts the resulting execution, where A is preempted by B.

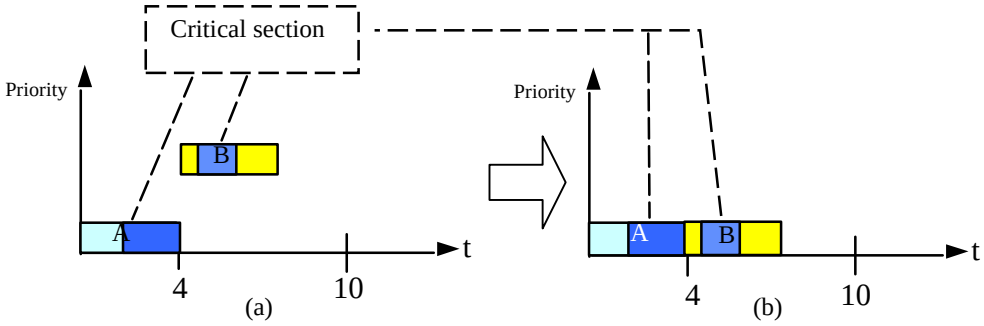
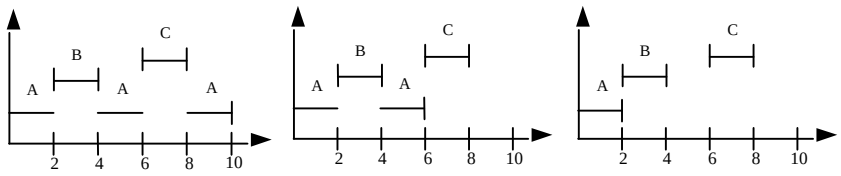


Figure 1-2. The releases of tasks A and B - figure (a). Where B has higher priority than task A. Due to shorter execution time task A starts and terminates before task B is released. Figure (b) depicts the resulting execution, where A precedes B.

Example 1-2.

Consider *Figure 1-3* which depicts the execution orderings of tasks during the Least Common Multiple (LCM) of the period times of the involved tasks *A*, *B* and *C*, based on a schedule generated by a static off-line scheduler and where later release time gives higher priority. Due to a varying execution time of task *A*, with a best case execution time (*BCET*) of 2 and a worst case execution time (*WCET*) of 6, we get three different scenarios, depicted in figures *1-3(a–c).* As exemplified below the execution of these different execution orderings will give different results.



(a) C_A=6 (b) C_A=4 (c) C_A=2

Figure 1-3 Three different execution order scenarios.

Assume in addition that all tasks call a common subroutine *f*(), that is by error non-reentrant, and that the tasks *A*, *B*, and *C* execute the program code in *Figure 1-4*. Task *A* is also in error by assigning 7 to *b*, when it should be 10. A critical point for the values computed by task *A* is indicated in the code for task *A*, by the preemption point.

```
int f(int a)
{
   int sum;

   sum =a+b;
   return(sum);
}
```

```
Task A:
int b; /* global*/
int ans;

/*1st assignenment in prg. */
b=7; /* error*/
…
…
/* Preemption point */
…
…
/*last assignment in prg.*/
ans = f(3);

...
```

```
Task B:
...
int ans;
...
b=10;
ans = f(2);

...
```

```
Task C:
...
int ans;
...
b=10;
ans = f(5);

...
```

Figure 1-4. The tasks A, B and C and the called function f()

The values calculated for task *A*, depending on which scenario is run, would thus be scenario (a) ans = 13 (correct), scenario (b) ans = 13 (correct), and scenario (c) ans = 10 (erroneous).

Hypothesize now, that we add a probe to task *A*, in order to test its behavior, and thus extend its execution time to always exceed its BCET. As a consequence scenario (c) will never be executed during run-time, and the error in task *A* will not be revealed during testing. If we later, after satisfactory testing, remove the probe in task *A*, scenario (c) can occur again and task *A* will fail. Thus giving rise to the probe effect.

Besides race-conditions, and the occurrence of the probe-effect in DRTS, there is also a difference between DRTS and sequential software with respect to control. Achieving deterministic observations of regular sequential programs is easy because in order to guarantee reproducibility we need only control the sequence of inputs and the start conditions [42]. 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 [54], or in the presence of intrusive probes. Reproducibility is essential when performing regression testing or cyclic debugging [53][56], 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 [33], or to show that no new errors have been introduced when correcting another error. However, trying to directly apply test techniques for sequential programs on distributed real-time systems is bound to lead to non-determinism and non-reproducibility, because control is only forced on the inputs, disregarding the significance of order and timing of the executing and communicating tasks.

Consequently, in order to facilitate reproducible monitoring of DRTS we must:

1. Reproduce the inputs with respect to contents, order, and timing
2. Reproduce the order and timing of the executing and communicating tasks.
3. Eliminate the probe-effect.

However, if deterministic monitoring is sufficient it is enough to only observe all entities with respect to contents, order and timing. A system can be defined as deterministic with respect to a certain set of behaviors if we can observe all necessary and sufficient conditions for the set of behaviors to occur. As for sequential software, it would be necessary to observe inputs and outputs in order to deterministically deem if the outputs comply with the requirements. If the control flow of the sequential program also depends on random number generators, we would have to observe these also for determinism. For a multitasking real-time systems, it would be necessary to observe contents, order and timing of all inputs, outputs, and executions of the involved tasks in order to deterministically deem if the system fulfills its requirements. For reproducibility however, it would also be necessary to control all necessary and sufficient conditions for a set of behaviors to deterministically occur. A system is partially reproducible if we can deterministically observe it but only control some of the necessary and sufficient conditions.

Reproducibility is a necessity when debugging, when regression testing [53], or when sufficient coverage during testing is sought [63][62][70].

Contributions

In this paper we present a software-based method for deterministic observations of single tasking, multi-tasking, and distributed real-time systems. This includes a description of what to observe, how to eliminate the disturbances caused by the actual act of observing, how to correlate observations between nodes, and how to reproduce the observations. We will give a taxonomy of different observation techniques, and discuss where, how and when these techniques should be applied for deterministic observations. We argue that it is essential to consider monitoring early in the design process, in order to achieve efficient and deterministic observations.

2 THE SYSTEM MODEL

We assume a distributed system consisting of a set of nodes. 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 an external process. We further assume the existence of a global synchronized time base [14][26] with a known precision d, meaning that no two nodes in the system have local clocks differing by more than d.

The software that runs on the distributed system consists of a set of concurrent tasks and interrupt routines, communicating by message passing or via shared memory, all governed by a real-time kernel. Tasks and interrupts may have functional and temporal side effects due to preemption, message passing and shared memory.

We assume that there exists a set of observers, which can observe/monitor the system behavior. These observers can be situated on different levels, ranging from dedicated nodes, which eavesdrop on the network, to programming language statements inside tasks that outputs significant information. These observers are fundamental for monitoring, testing and debugging of real-time systems and distributed real-time systems.

3 TERMINOLOGY

In this section we will introduce some basic vocabulary that we will be used in the paper.

3.1 Fault, error, and failure

Definition. A *failure* is the nonperformance or inability of the system or component to perform its intended function for a specified time under specified environmental conditions [71]. That is, an input, X, to the component, O, yields an output, $O(X)$, non-compliant with the specification.

Definition. An *error* is a design flaw, or a deviation from a desired or intended state [71]. That is, if we view the program as a state machine, an error (bug) is an unwanted state. We can also view an error as a corrupted data state, caused by the execution of an error (bug) but also due to e.g., physical electromagnetic radiation.

Definition. A *fault* is the adjudged (hypothesized) cause for an error [33]. Generally a failure is a fault, but not vice versa, since a fault does not necessarily lead to a failure.

The relation between the definitions of fault, error, and failure, is depicted in *Figure 3-1*.

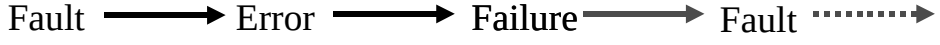


Figure 3-1. Cause consequence diagram of fault, error and failure.

3.1.1 Systematic and physical failures

Failures are usually divided into two categories:

- *Systematic failures* which are caused by specification or design flaws, i.e., behaviors that do not comply with the goals of the intended, designed and constructed system. Examples of contributing causes, are erroneous, ambiguous, or incomplete specifications, as well as incorrect assumptions about the target environment. Other examples are failures caused by design and implementation faults. Wear, degradation, corrosion, etc. do not cause these types of failures, all errors are built in from the beginning, and no new errors will be added after deployment.

  Definition. A systematic *failure* occurs if and only if:

  1) the location of an error is executed in the program,
  2) the execution of the error leads to an erroneous state, and
  3) the erroneous state is propagated to the output.

  This means, that if an error is not executed it will not cause a failure. If the effect of the execution of the error (infection) is indistinguishable from a correct system state it will not cause a failure. If the system's state is infected but not propagated to output there will be no failure.

- *Physical failures* which are the result of a violation upon the original design. Environmental disturbances, wear or degradation over time may cause such failures. Examples, are electromagnetic interference, alpha and beta radiation, etc.

  Definition. A physical *failure* occurs if and only if:

  1) the system state is corrupted or infected, and
  2) the erroneous state is propagated to the output.

  Fault-tolerance mechanisms usually try to prevent (1) by applying robust designs, and (2) by applying redundancy, etc.

3.2 Failure modes

Depending on the architecture of the system we can assume different degrees, and classes, of failure behavior. That is, certain types of failures are extremely improbable (impossible) in some systems, while in other systems it is very likely that they occur. For example, consider multitasking systems where we have to resolve access to shared resources by means of mutual exclusion. One approach is to make use of semaphores, and another to make use of separation in time. In the latter case deadlock situations are impossible, while in the previous case deadlocks certainly are possible. Using synchronization in time we thus eliminate an entire class of failures, and can therefore during testing eliminate the search for them.

Components can fail in different ways and the manner in which they fail can be categorized into failure modes. The failure modes are defined through the effects, as perceived by the component user. We are going to present categories, i.e., failure modes, (1 to 6) ranging from failure behavior that sequential programs, or single tasks in solitude, can experience, to failure behavior that is only significant in multitasking, distributed systems and real-time systems, where more than one task is competing for the same resources, e.g., processing power, memory, computer network, etc.

1. ***Sequential failure behavior*** (Clarke et. al. [10]):
    - *Control failures*, e.g., selecting the wrong branch in an if-then-else statement.
    - *Value failures*, e.g., assigning an incorrect value to a correct (intended) variable.
    - *Addressing failures*, e.g., assigning a correct (intended) value to an incorrect variable.
    - *Termination failures*, e.g., a loop statement failing to complete because the termination condition is never satisfied.
    - *Input failures, e.g.*, receiving an (undetected) erroneous value from a sensor.

Multitasking and real-time failure behavior

2. ***Ordering failures***, e.g., violations of precedence relations or mutual exclusion relations.
3. ***Synchronization failures***, i.e., ordering failures but also deadlocks.
4. ***Interleaving failures***, e.g., unwanted side effects caused by non-reentrant code, and shared data, in preemptively scheduled systems.
5. ***Timing failures***. This failure mode yields a correct result (value), although the procurement of the result is time-wise incorrect. For example, deadline violations, too early start of task, incorrect period time, too much jitter, too many interrupts (too short inter-arrival time between consecutive interrupt occurrences), etc.
6. ***Byzantine and arbitrary failures***. This failure mode is characterized by a non-assumption, meaning that there is no restriction what so ever with respect to which effects the component user may perceive. Therefore, the failure mode has been called malicious or fail-uncontrolled. This failure mode includes two-faced behavior, i.e. a component can output “X is true” to one component user, and “X is false” to another component user.

The above listed failure modes build up a hierarchy where byzantine failures are based on the weakest assumption (a non-assumption) on the behavior of the components and the infrastructure, and sequential failures are based on the strongest assumptions. Hence byzantine failures are the most severe and sequential failures the least severe failure mode. The byzantine failure mode covers all failures classified as timing failures, which in turn covers synchronization failures, and so on (see *Figure 3-2*).

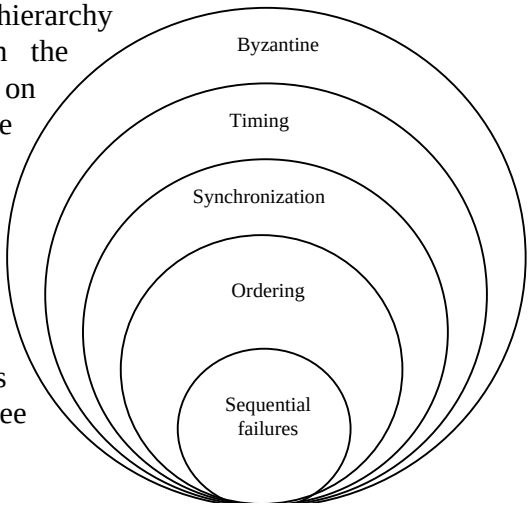


Figure 3-2. The relation between the failure modes.

The component user can also characterize the failure modes according to the point of view. A distinction can be made between primary failures, secondary failures and command failures (Leveson [71]):

- ***Primary failures***

  A primary failure is caused by an error in the software of the component so that its output does not meet the specification. This class includes sequential and byzantine failure modes, excluding sequential input failures.

- ***Secondary failures***

  A secondary failure occurs when the input to a component does not comply with the specification. This can happen when the component is used in an environment for which it is not designed, or when the output of a preceding task does not comply with the specifications of a succeeding task's input. This class includes interleaving failures, sequential input failure modes, as well as changed failure mode assumptions.

- ***Command failures***

  Command failures occur when a component delivers the correct result but at the wrong time or in the wrong order. This class covers timing failures, synchronization failures, ordering failures, as well as sequential termination failures.

The persistence of failures

The persistence of failures can be categorized into three groups:

- ***Transient failures***. Transient failures occur completely aperiodic, meaning that we cannot bound their inter-arrival time. They can appear once, and then never appear again. Typically, these types of failures are induced by electromagnetic interference, or radiation, which may lead to corruption of memory, or CPU registers – *bit-flips.* Transient failures are mostly physical failures.

- ***Intermittent failures***. The inter-arrival time of intermittent failures can be bounded with a minimum and/or maximum inter-arrival time. These types of failures typically take place when a component is on the verge of breaking down, for example, due to a glitch in a switch. Examples from the software world could be failures due to sporadic interrupts.

- ***Permanent failures***. A permanent failure that occurs, stays until removed (repaired). A permanent failure can be a damaged sensor, or typically for software, a systematic failure – caused by an error in a program, which stays there until removed.

3.3 Failure semantics

The above classification of failure modes is not restricted to individual instances of failures, but can be used to classify the failure behavior of components, which is called a component's failure semantics [48].

- ***Failure semantics***

  A component exhibits a given failure semantic if the probability of failure modes, which are not covered by the failure semantic, is sufficiently low.

If a given component is assumed to have synchronization failure semantics, then all individual failures of the component should be synchronization-, ordering-, or sequential failures. The possibility of more severe failures, like timing failures, should be sufficiently low. The failure semantic is a probabilistic specification of the failure modes a component may exhibit. The semantic has to be chosen in relation to the application requirements. In other words, the failure semantics defines the most severe failure mode a component should experience. Fault-tolerant systems are designed with the assumption that any component that fails will do so according to a given failure semantic. When we *test* a system we do so also with a certain failure semantic in mind. That is, we look for failures of a certain kind. For plain sequential programs we usually do not look for interleaving failures, or timing failures. However, if the component will be used in a multitasking or real-time system we certainly have to look for these types of failures.

3.4 Fault hypothesis

When a system is designed for fault-tolerance or when testing is performed it is always based on a *fault hypothesis*, which is simply the assumption that the system will behave according to a certain failure semantic.

This means that if a system is tested with a specific fault hypothesis, and a certain confidence in its reliability is achieved (*Figure 3-3*), then if we later assume a more severe fault hypothesis, the confidence in the achieved reliability decreases (*Figure 3-4*). For example, if we have tested a system, which has memory protection, and then remove the memory protection we cannot say anything about the achieved reliability with respect to that fault hypothesis. Changes of this type typically give rise to secondary failures.

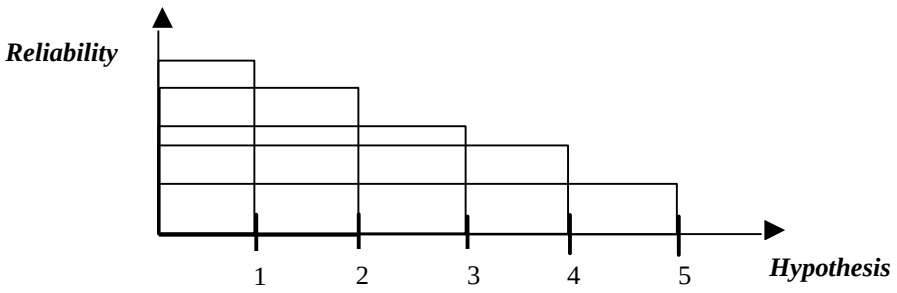


Figure 3-3. The achieved reliability for different fault hypothesis.

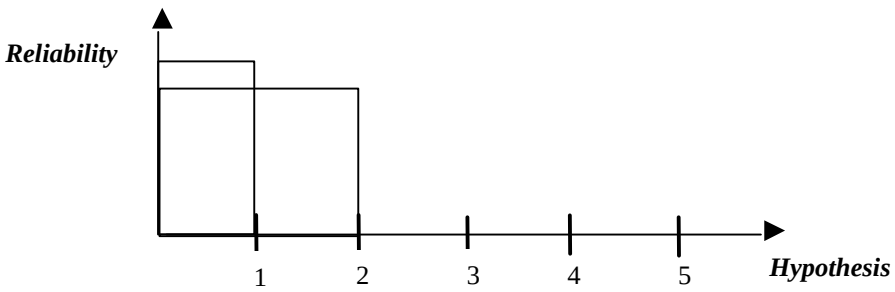


Figure 3-4. The confidence in the reliability for more severe fault hypothesis collapses when basic assumptions do not hold due to e.g., the removal of memory protection.

3.5 Determinism and reproducibility

Sequential programs are usually regarded as having deterministic behavior, that is, given the same initial state and inputs, the sequential program will consistently produce the same output on repeated executions, even in the presence of systematic errors. For example,

```
int SUM(int a, int b, int c)
{
        int s;
        s = a+b;
        printf("c=%d\n", c);
        return(s);
}
```

Given that the parameters *a* and *b* were equal on repeated calls to *SUM*(*a*,*b*,*c*) then the function would deterministically reproduce the sum of *a* and *b* – regardless of the value of *c*.

The determinism of a system with respect to an observed behavior can be defined as:

Definition. *Determinism*. A system is defined as deterministic if an observed behavior, *P*, is uniquely defined by an observed set of necessary and sufficient parameters/conditions, *O*.

Definition. *Partial Determinism*. A system is defined as partially deterministic if an observed behavior, *P*, is uniquely defined by a *known* set of necessary and sufficient parameters/conditions, *O*, but the observations are limited to a subset of *O*.

The implication of the definition of determinism is that if we have a function *f(a, b, c)* and the observed behavior, *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)* an infinite number of times and deterministically observe this behavior by observing the output of *f* and by observing *a* and *b*. The value of *c* is of no significance because it is not necessary for *P's* determinism. If we can also control, not only observe, the values of *a* and *b* we can also reproduce the observation of behavior *P*.

Definition. *Reproducibility*. A system is reproducible if it is deterministic with respect to a behavior *P*, and if it is possible to *control* the entire set of necessary and sufficient conditions, *O*.

Definition. *Partial reproducibility*. A system is partially reproducible if it is deterministic with respect to a behavior *P*, and if it is possible to *control* a subset of the necessary and sufficient conditions, *O*.

Hence, the relation is such that the property reproducibility is stronger than the property determinism, i.e., if some observations are reproducible they are deterministic, but not necessarily vice versa, thus:

Determinism Ì Partial reproducibility Ì Reproducibility

This is an important distinction to make, since the desired behavior, the fault hypothesis and the infrastructure dictates how many conditions/variables/factors we need to observe in order to guarantee determinism of observations, as well as how many conditions we must control for reproducibility of observations.

4 MONITORING

Research on testing of shared memory multiprocessor systems and distributed systems have been penetrated in some detail over the years. The focus has mainly been on monitoring, i.e. gathering of run-time information for debugging [47][66][67] and performance measurements [6].

The research issues have been:

- The intrusiveness (perturbation) of observations in software [18][36][42] and how to eliminate the perturbations using special hardware [68].
- How to deterministically reproduce the observations using mechanisms in software [12][35] [61] or mechanisms in hardware [68]
- The problem of defining a global state in distributed systems [16] using logical clocks [7][32] or synchronized physical clocks [26][28][48].

However, the number of references on research regarding monitoring for testing and debugging of single node real-time systems, and multiple node (distributed) real-time systems, that consistently handle time, distribution, interrupts, clock synchronization, and scheduling, dwindle fast (to zip).

The observational requirements for testing and debugging differ, in the amount and type of information required. The quintessential difference comes from the fact that testing is used for finding failures (or showing their absence), while debugging is used for finding the errors that cause the failures. Another difference is that testing can easily be automated, while debugging is essentially a manual task. For the verification of safety-critical software (failure rates of 10^{-4} to 10^{-9} failures/hour) it is necessary that the test process can be automated since the number of test cases required are enormous [5][72].

For testing of sequential programs it is usually sufficient to monitor inputs, and outputs via predefined interfaces of the programs, and based on that information deem, according to the specification, if a test run was successful or not. For (distributed) real-time systems we need also observe the timing and order of the executing and communicating tasks, since the outputs depend on these variables, and thus also the determinism of the observations.

To detect errors using debugging it is also necessary to monitor the internal behavior of the programs with respect to intermediate computed values, internal variables, and the control flow. For interactive debugging, in the classical sense of sequential programs, it is required that the control flow can be altered via manual or conditional breakpoints, or via traces, all in order to be able to increase the observability of the program. Consequently, for debugging of (distributed) real-time systems, we need to control the timing and order of the executing and communicating tasks, otherwise we cannot achieve deterministic debugging. However, the problem of defining a global state in distributed real-time systems, and break-pointing tasks on different nodes at exactly the same time, is a serious obstacle when debugging. Either we need to send *stop* or *continue* messages from one node to a set of other nodes with the problem of nonzero communication latencies, or we have a priori agreed upon times when the execution on the processors should be halted or resumed. In the latter case there is the problem of non-perfectly synchronized clocks, so the tasks may not halt or resume their execution at the same time. Most real-time systems are also driven by the

environment, so just because we breakpoint one task on one node, does not stop the external process.

When monitoring a DRTS there are some fundamental questions that must be answered:

- How to extract enough information from the system?
- How to eliminate the perturbations that the observations cause?
- How to correlate the observations, i.e., how to define a consistent global state?
- How to reproduce the observations?

We are now going to address each of these questions in turn.

4.1 How to collect sufficient information

The amount of monitoring needed in order to collect sufficient information is dependent on two basic factors:

- *What is the fault hypothesis*? That is, the more severe failure semantics, the more information we need to store and process in order to achieve deterministic observations. For sequential software it is sufficient to observe inputs and outputs. For multi-tasking systems we need also observe task execution orderings and their access to shared resources. For real-time systems we further need to observe the timing of the tasks. However, if we want to test a multitasking real-time system and assume sequential failure semantics we need only test tasks in solitude, since we can regard each task as a sequential program. The probability that the multitasking real-time system will only exhibit sequential failure behavior in practice is not very high though. It is therefore very important to chose a realistic fault-hypothesis and observe the system based on that fault hypothesis.
- *What is the a priori knowledge* of the system with respect to its behavior and attributes? The validity of the fault-hypothesis is based on the support that the environment and the infrastructure (real-time kernel, hardware, etc.) give the fault hypothesis.

  Does the system have memory protection? How does the system synchronize access to shared resources (time or semaphores)? Is the execution strategy time-triggered or event-triggered?

  For example, if the system is time-triggered and scheduled using e.g., strict periodic fixed priority scheduling or static scheduling, we know that the system will repeat its execution every LCM. For event-triggered systems we have no such general limit and might have to observe and store copious amounts of information.

4.1.1 Data flow, control flow, and resources

The actual information to be observed can be categorized into three groups:

- Data flow (internal and external)
- Control flow (execution and timing)
- Resources (memory and execution resources)

4.1.1.1 Data flow

- *Inputs* – determine for which input the task will execute, this is important if the actual input is not provided by a test oracle, but rather by an external process or an environment simulator.
- *Outputs* – what are the produced outputs via the predefined interfaces of the task?
- *Auxiliary outputs* – output intermediately computed values, or program state, which are not visible via the predefined interfaces. For sequential software these are commonly implemented using e.g., assertions, or even using printf statements in C. For distributed real-time systems the situation is more complex and we need to define the type of data we *additionally* need. Typically these are related to memory mapped I/O interfaces, for example received messages over the network, readings of A/D converters, readings of the local clock, etc. Because any additional outputs will require more memory, communication bandwidth, and execution time, we need to take these auxiliary outputs into account when designing and scheduling, in order to avoid the probe-effect. These auxiliary outputs could also be parameterized, i.e., we can during run-time switch between different auxiliary outputs, given of course that this parameterization is designed in such a way that the timing behavior is constant.
- *Inter-node messages* – Which are the messages that are passed between the system nodes?

4.1.1.2 Control flow

- *Inputs, outputs, auxiliary outputs, and inter-node messages*. At what time and in what order were the inputs received? At what time and in what order were the outputs produced?
- *Task switches* – which, when, and in what order are tasks starting, preempting, and finishing? We can make use of this information for deriving the synchronization and execution orderings between tasks. We can also make use of the timing information in order to deem if tasks start too early, too late, finish too late or finish too early. We can further measure the periodicity of the tasks and thus deem if jitter requirements are met. Making use of this timing information we can also measure the execution times of the tasks.
- *Interrupts* – which, when, how long, and in what order are interrupts interfering with tasks. Using this information we can judge how the interrupts interfere with the execution of the tasks. We can thus measure if basic assumptions of interrupt overhead are true.

- *Real-time kernel overhead*. What is the execution time of the real-time kernel. What are the latencies due to interrupt disable, that is when the kernel needs to perform atomic operations it usually disables all interference by interrupts, for how long time can the kernel block all interrupts?
- *Tick rate*. The tick rate is the frequency at which the real-time kernel is invoked, and at which new scheduling decisions are taken. However, the tick rate can vary due to global clock synchronization. That is, the inter arrival time between ticks might increase or decrease if the local clock is to slow or to too fast compared to the global time base.

4.1.1.3 Resources

- *Memory use* – stack use, etc. How much of the memory is used by the tasks, interrupt service routines, or the kernel?
- *CPU utilization*. How much of the CPU's calculating power is used?
- *Network utilization*. How much of the networks bandwidth is used?
- *The state of the real-time kernel*: Which tasks are waiting for their turn to execute (waiting queue, list, or table)? Which task is running? Which tasks are blocked?

It is very important to deem which information is necessary for monitoring of the system, because if you extract too much there will be a heavy performance and cost penalty. If you extract too little, the precision of the observations will be too coarse or simply non-deterministic for judging how and why the system behaved as it did. For determinism there is a least necessary level of observability required, i.e., we need to observe the necessary and sufficient parameters as defined in section 3.5. Any additional (useful) information surpassing the level of necessity for determinism will increase the precision of the observations. Think of inputs, and outputs for sequential software as the least necessary level for determinism, while debugging provides a higher level of observability (precision) since we can inspect the internal control flow and the contents of the variables.

4.2 Elimination of perturbations

After a decision on what entities to observe we need to decide on how to eliminate the probe-effect. There are basically three approaches (1) non-intrusive/passive hardware, (2) intrusive software instrumentation, and a hybrid (3) where the software instrumentation is minimized.

4.2.1 Hardware monitoring

A transparent non-intrusive approach towards monitoring, is the application of special hardware, e.g. hardware that allows buss sniffing, or non-intrusive access to memory via dual-port memories, etc., but also through the use of hardware CPU emulators, (Lauterbach et al. [34]). Hardware monitoring has been applied for performance measurements [4], execution monitoring of multiprocessor systems [38], and real-time systems [39][47][68]. Since the monitoring hardware is interfaced to the target system's hardware via the CPU socket (emulator) or via the data and address busses, it can observe the target system without interfering with its execution, and thus not introduce any probe-effects. The drawbacks are that the monitoring mechanisms must be very target specific and therefore very expensive, but also that the observations will be on a very low level of detail, since only the external interfaces of the microprocessors and shared resources such as dual-port memories, can be monitored. The ever-increasing integration of functionality in current general-purpose micro-controllers makes it correspondingly harder to observe the internal behavior of the micro controllers/CPUs, due to cache memory, on-chip memory, etc. Hardware monitoring must also be considered early in the design of the system since monitoring mechanisms will be difficult to integrate when the rest of the hardware configuration is set. Non-intrusive monitoring of distributed real-time systems also requires that we have dedicated monitoring hardware on each node, and that the nodes are interconnected via a dedicated monitoring network for data transfer and synchronization, in order to avoid the probe-effect. We need also establish a globally synchronized time base, relative which all observed events on the nodes can be correlated otherwise there can be no guarantees of the consistency between observations.

It can be argued that the cost for the monitoring hardware will only impact the development budget, not the production cost, since the monitoring hardware can be removed from the target system. Experience of software development has however shown that maintainability is a necessity also after deployment. The non-portability, the lack of scalability and the observations low level of detail severely limit the viability of the hardware approach. The current trend of making application specific hardware using FPGAs and VHDL [6] gives, however, an opportunity to conveniently integrate non-intrusive monitoring mechanisms in the hardware for single node systems.

4.2.2 Hybrid monitoring

In order to increase the level of abstraction and decrease the amount of information recorded, hybrid approaches to monitoring have been suggested. "Triggers" are implemented in software, which using a minimum number of instructions assists the hardware in recording significant events. Software triggers do for example, write to specific addresses that are memory mapped with the monitoring hardware, or use special co-processor instructions. When the monitoring software writes to these

addresses the hardware records the data passed on the data bus of the processor. Using this approach the limitations of hardware monitoring can be alleviated, although the cost and non-portability issues still remain. The monitoring instructions in the software must also be resource adequate, and remain in the target system in order to avoid the probe-effect. Hybrid performance monitoring of distributed systems have been covered by Haban et al. [22], and performance monitoring of multiprocessor systems by Mink et al. [45][52][20].

4.2.3 Software monitoring

Historically, the contributing motivations for using hardware, and hybrid, monitoring approaches have been the problem of predicting the perturbations caused by instrumenting software [15]. That is, any instrumentation of the software will require memory and execution time resources, while hardware can passively monitor the system with no interference. For software instrumentation there will be a probe-effect, if the probes are removed after satisfactory monitoring, or if the probes are added to a system that has already been shown, e.g., using scheduling theory, to always meet its deadlines. If the probes are not removed there will be a financial penalty due to the dedicated resources (memory, processing, bandwidth, etc.) or they might hamper the performance of the system.

In order to test and debug a system to satisfactory levels of reliability we fundamentally need to observe the system, and by including instrumentation code in the software (application and kernel), we can observe significantly more than possible with hardware approaches. Software monitoring of real-time systems have been covered by Chodrow et al [9], distributed systems by Joyce et al [24][44], and distributed real-time systems by Tokuda et al [66][51]. In general it is necessary to leave the software probes in the target system in order to eliminate the probe-effect. If the target system is a real-time system, which can be scheduled e.g., using fixed priority or static scheduling it is straightforward to analyze the effects that the probes have on the system. Just make the probes part of the design, i.e., allocate execution time and memory, and then make use of execution time analysis [50] and scheduling theory [1][40][69]. For monitoring of distributed real-time systems we need also to allocate communication bus bandwidth and account for the probes when making the global schedule. We need also establish a global time-base in order to correlate observations on different nodes.

4.2.3.1 Software probes

For software monitoring of distributed real-time systems we have identified the following four different types of probes, depending on where they are implemented:

- *Kernel-probes* – System and kernel level probes, monitor task switches, interrupt interference, etc. (*Figure 4-1*). These types of probes are typically not programmable by the application designer, but rather given as an infrastructure by the real-time kernel. In order to avoid the probe effect, these types of probes should be left permanently in the kernel, their contributing overhead must also be predictable, and minimized.

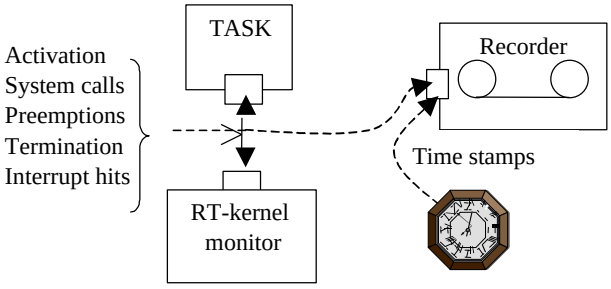


Figure 4-1. Kernel-probe mechanism.

- *Inline-probes* – Are task level probes that add auxiliary outputs to the task they instrument (*Figure 4-2*). These types of outputs are outputs that are regarded necessary from a monitoring, testing or debugging perspective, rather than from a functional application requirement perspective. As they are part of the application code they will also be covered in the estimations, or measurements of the execution times. This also means that we usually need to let them remain in the target system in order to eliminate the probe effect

```
        ….
        ….
printf(“red alart”);
        …
```

Figure 4-2. Inline- probe.

.

- *Probe-tasks* – Are tasks dedicated to collecting data from kernel-probes, inline-probes and other probe-tasks. As depicted in the *Figure 4-3*, a dedicated probe-task receives data from a set of tasks. All probe-tasks must be taken into account when designing and scheduling the system. These types of probes need also remain in the target system in order to avoid the probe effect. We will however soon elaborate on this, and show that there are certain circumstances that allow for these probes to be removed from the target system.

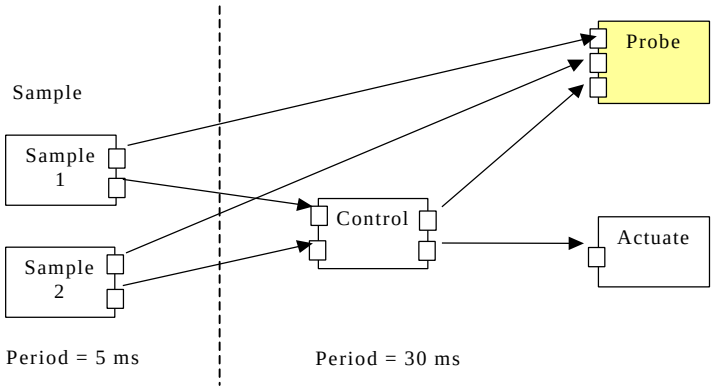


Figure 4-3. Probe-task receives data from other tasks.

- *Probe-nodes* – Are dedicated nodes that collect data from probe-tasks and are able to monitor communication busses (*Figure 4-4*). The probe-node can also analyze the collected data for performance estimations or for testing and debugging. These probe-nodes must also be taken into account when allocating resources for the system since they will require communication bus bandwidth, unless they passively eavesdrop on the network. These types of probes are however easier to remove from the target system since they usually are self-contained computing elements, and can thus be regarded as passively observing hardware monitoring elements, and consequently they can be removed with minimal interference.

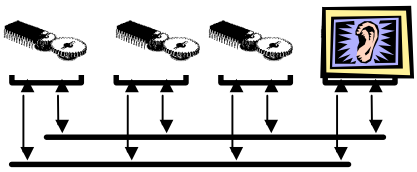


Figure 4-4. Probe-node.

4.2.3.2 Resource allocation

The prerequisites for avoiding the probe-effect when using the above-defined probes are the allocation of sufficient resources e.g., execution time, network bandwidth and memory. It is not very plausible that these resources will “pop” up in the right place during the test phase if they have not been taken into account during the design phase.

Eliminating the probes?

In most embedded systems the execution time (CPU speed) and memory are limited resources, mostly due to large production volumes, where per unit cost reduction is of significance. From an end-quality, and verification point of view, it is not hard to motivate the extra cost for dedicated probes. That is, you fundamentally need the probes for testing. If you do not have the probes you cannot assess the reliability, or find the errors. Accidents have however, shown that having non-functional code in the target system can be hazardous [71]. Some testing measures are sometimes also deemed so hazardous that they must by all means be eliminated from the target system. Examples include test procedures for train signaling systems, where the test procedures actually change the state of the signals, and consequently could an inadvertent execution of these procedures during runtime cause severe accidents.

So, is there any possibility for us to remove probes after satisfactory testing without introducing the probe-effect? For some execution strategies, e.g., statically scheduled real-time systems, probes can be removed without temporal side-effects if they are situated within *temporal firewalls* [56]. That is, as long as we do not change the start and completion times of tasks, and change their times of output (communication or access to shared resources), we can remove the probes (*Figure 4-5*). The probes can also be eliminated in fixed priority scheduled systems, if we make use of offsets and thus erect temporal firewalls, or if the probes have lower priority than the rest of the tasks in the system (*Figure 4-6*). In the latter case we must also guarantee that the monitoring probes cannot ever block a higher priority task.

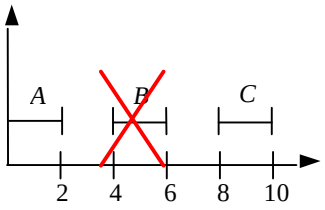


Figure 4-5. Probe task B can be removed due to fixed release times of A and C.

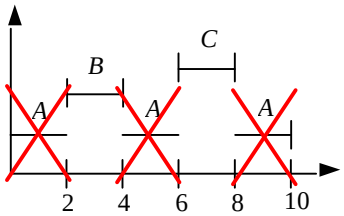


Figure 4-6. Low priority probe task A can be removed without side effects.

If it now would be possible to eliminate the probes, to what end could we use the spare resources (memory and time)? If we remove the probes and thus decrease the processor utilization, we could possibly use the spare resources for non-critical activities (soft real-time tasks) [13]. However, how do we guarantee that the non-critical software does not introduce new errors? In most micro-controllers we do not have memory protection schemes, and consequently can a soft real-time task wreak havoc in the memory space where the hard real-time tasks reside and operate. Could we use a cheaper and slower processor? It is not very likely that we could use a processor with different timing characteristics than the one tested, because all execution times and scheduling are based on the timing specifics of the target

processor. That is, if we change the processor we need to reschedule the entire system, and consequently retest the entire system again; thus gaining nothing.

Memory reduction

One possible benefit however, could be the reduction of memory use. If it can be shown that the removal of probes will not change the functional behavior of the system with respect to memory access, and memory side effects, and all the memory used by the probes have been allocated in a specific address space, we could remove this memory and thus save money.

4.3 Defining a global state

In order to correlate observations in the system we need to know their orderings, i.e., determine which observations are concurrent, and which precede and succeed a particular event. In single node systems or tightly coupled multiprocessor systems with a common clock this is not a problem, but for distributed systems without a common clock this is a significant problem. An ordering on each node can be established using the local clocks, but how can observations between nodes be correlated?

One approach is to establish a causal ordering between observed events, using for example logical clocks [32] derived from the messages passed between the nodes. However, this is not a viable solution if tasks on different nodes work on a common external process without exchanging messages, or when the duration between observed events is of significance. In such cases we need to establish a total ordering of the observed events in the system. This can be achieved by forming a synchronized global time base [14][26]. That is, we keep all local clocks synchronized to a specified precision ***d,*** meaning that no two nodes in the system have local clocks differing by more than *d*.

Figure 4-7 illustrates the local ticks in a distributed system with three nodes, all with tick rate ***P***, and synchronized to the precision *d.* There is no point in having $P \leq d$, because the precision *d* dictates the margin of error of clock readings, and thus a $P \leq d$ would result in overlaps of the *d* intervals during which the synchronized local ticks may occur [31].

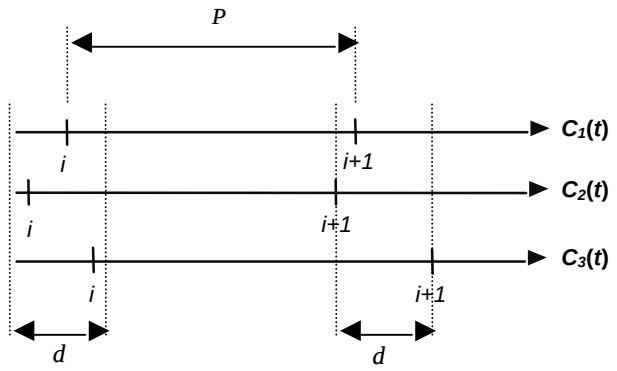


Figure 4-7. The occurrence of local ticks on three nodes

Consider *Figure 4-8,* illustrating two external events that all three nodes can observe, and which they all timestamp. Due to the sparse time base [28] and the precision *d*,

we end up with timestamps of the same event that differ by 1 time unit (i.e., P) while still complying with the precision of the global time base. This means that some nodes will consider events to be concurrent (i.e., having identical time stamps), while other nodes will assign distinct time stamps to the same events. This is illustrated in *Figure 4-8*, where node 2 will give the events *e1* and *e2* identical time stamps, while they will have difference 2 and 1 on nodes 1 and 3, respectively. That is, only events separated by more than $2P$ can be globally ordered. Due to the precision of the global clock synchronization there is thus a smallest possible granule of time defined by $2d$ for deterministic ordering events in the system, since tick overlaps are not acceptable, i.e., $2P > 2d$. Consequently the ultimate precision of the global state, i.e., the observed state, will be defined by the precision of the global clock synchronization.

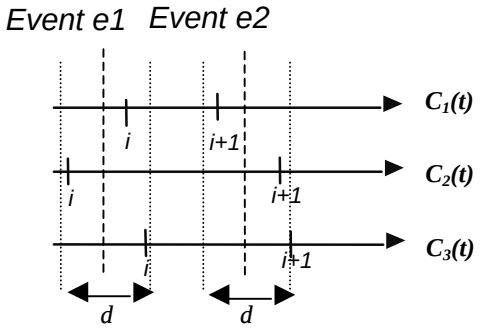


Figure 4-8. Effects of a sparse time base.

4.4 Reproduction of observations

In order to reproduce observations we must bring about the exact same circumstances as when the original observations were made. That is, for distributed real-time systems we need to reproduce the inputs, and as the behavior of a distributed real-time system depends on the orderings, and timing of the executing and communicating tasks, we need also reproduce that behavior in order to reproduce their outputs.

4.5 Reproducing inputs

For the reproduction of inputs it is common to use environment simulators [19][37]. The simulators are models of the surrounding environment, e.g., models of the hardware, or the user and user interface, that can simulate the environment's inputs and interactions with the target system, with respect to contents, order and timing.

Classically, the environment simulators have not focused on reproducing inputs to the system, but rather been necessities when the target hardware has not been available, due to concurrent development, or when the application domain has been safety-critical. For the verification of safety-critical systems it is necessary to produce very rare scenarios (10^{-9} occurrences/hour) that would be extremely difficult (or even dangerous) to produce even if the target system and target environment were available to the testers [73][57]. Examples are space applications, weapons systems, and medical treatment devices.

4.6 Reproduction of complete system behavior

When it comes to the reproduction of the state and outputs of single tasking RTS, multitasking RTS and DRTS, with respect to the orderings, and timing of the executing and communicating tasks, there are two approaches:

(1) *Off-line replays, i.e., record* the runtime behavior and examine it while replaying it off-line.

(2) *On-line reproduction*, i.e., rerun the system while controlling all necessary conditions.

4.6.1 Deterministic off-line replay

The first approach, on-line recording and off-line replay, is commonly referred to as deterministic replay in the literature and is used for debugging. The basic idea, is an equivalent of a tape recorder, or the black-box in airplanes, where significant events are recorded over a period of time during run-time, and then using this recording the systems behavior can be reproduced and examined off-line. The examinations can be of finer detail than the events recorded. For example, by recording the actual inputs to tasks we can off-line re-execute the tasks using a debugger and examine the internal behavior to a finer degree of detail than recorded. If we also record all synchronization and scheduling events, i.e., the task switches, we can also off-line examine the actual real-time behavior without having to run the system in real-time, and without introducing any probe-effect. We can thus deterministically replay the task executions, the task switches, and the system behavior over and over.

In a survey on the testability of distributed real-time systems Schütz [56] has identified three issues related to deterministic replay in general, which we briefly comment below:

Issue 1: *One can only replay what has previously been observed, and no guarantees that every significant system behavior will be observed accurately can be provided. Since replay takes place at the machine code level the amount of information required is usually large. All inputs and intermediate events, e.g. messages, must be kept.*

The amount and the necessary information required is of course a design issue, but it is not true that all inputs and intermediate messages must be recorded. The replay can as we have shown actually re-execute the tasks in the recorded event history. Only those inputs and messages which are not re-calculated, or re-sent, during the replay must be kept. This is specifically the case for RTS with periodic tasks, where we can make use of the knowledge of the schedule (precedence relations) and the duration before the schedule repeats it self (the LCM – the Least Common Multiple of the task period times.) In systems where deterministic replay has previously been employed, e.g., distributed systems [46] and concurrent programming (ADA) [61] this has not been the case. The restrictions, and predictability, inherent to scheduled RTS do therefore give us the great advantage of only recording the data that is not recalculated during replay.

Issue 2: *If a program has been modified (e.g., corrected) there are no guarantees that the old event history is still valid.*

If a program has been modified, the relative timing between racing tasks can change and thus the recorded history will not be valid. The timing differences can stem from a changed data flow, or that the actual execution time of the modified task has changed. In such cases it is likely that a new recording must be made. However, the probability of actually recording the sequence of events that pertain to the modification may be very low. This is an issue for regression testing [62][63], which we will discuss in section 4.6.2.

Issue 3: *The recording can only be replayed on the same hardware as the recording was made on.*

The event history can only be replayed on the target hardware. This is true to some extent, but should not be a problem if remote debugging is used. The replay could also be performed on the host computer if we have a hardware simulator, which could run the native instruction set of the target CPU. Another possibility would be to identify the actual high-level language statements where task switches or interrupts occurred, rather than trying to replay the exact machine code instructions, which of course are machine dependent. In the latter case we of course run into the problem of defining a unique state when differentiating between e.g., iterations in loops.

Related work

There are a few descriptions of deterministic replay mechanisms (related to real-time systems) in the literature:

- A deterministic replay method for concurrent Ada programs is presented by Tai et al. [61]. 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 orderings 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. Further, issues like unwanted side effects caused by preempting tasks are not considered. The granularity of the enforced rendezvous does not allow preemptions, or interrupts for that matter, to be replayed. It is unclear how the method can be extended to handle interrupts, and how it can be used in a distributed environment.

- Tsai et al. present a hardware monitoring and replay mechanism for real-time uniprocessors [68]. Their approach can replay significant events with respect to order, access to time, and asynchronous interrupts. The motivation for the hardware monitoring mechanism is to minimize the probe-effect, and thus make it suitable for real-time systems. Although it does minimizes the probe-effect, its overhead is not predictable, because their dual monitoring processing unit causes unpredictable interference on the target system by generating an interrupt for every event monitored [12]. They also record excessive details of the target processors execution, e.g., a 6 byte immediate AND instruction on a Motorola 68000 processor generates 265 bytes of recorded data. Their approach can reproduce asynchronous interrupts only if the target CPU has a dedicated hardware instruction counter. The used hardware approach is inherently target specific, and hard to adapt to other

systems. The system is designed for single processor systems and has no support for distributed real-time systems.

- The software-based approach *HMON* [12] is designed for the HARTS distributed (real-time) system multiprocessor architecture [59]. A general-purpose processor is dedicated to monitoring on each multiprocessor. The monitor can observe the target processors via shared memory. The target systems software is instrumented with monitoring routines, by means of modifying system service calls, interrupt service routines, and making use of a feature in the pSOS real-time kernel for monitoring task-switches. Shared variable references can also be monitored, as can programmer defined application specific events. The recorded events can then be replayed off-line in a debugger. In contrast to the hardware supported instruction counter as used by Tsai et al., they make use of a software based instructions counter, as introduced by Mellor-Crummey et. al. [43]. In conjunction with the program counter, the software instruction counter can be used to reproduce interrupt interferences on the tasks. The paper does not elaborate on this issue. Using the recorded event history, off-line debugging can be performed while still having interrupts and task switches occurring at the same machine code instruction as during run-time. Interrupt occurrences are guaranteed off-line by inserting trap instructions at the recorded program counter value. The paper lacks information on how they achieve a consistent global state, i.e., how the recorded events on different nodes can consistently be related to each other. As they claim that their approach is suitable for distributed real-time systems, the lack of a discussion concerning global time, clock synchronization, and the ordering of events, diminish an otherwise interesting approach. Their basic assumption about having a distributed system consisting of multiprocessor nodes makes their *software* approach less general. In fact, it makes it a hardware approach, because their target architecture is a shared memory multiprocessor, and their basic assumptions of non-interference are based on this shared memory and thus not applicable to distributed uniprocessors.

4.6.2 On-line reproduction

To facilitate reproducible execution on-line we must identify which execution orderings, or parts of execution orderings that can be enforced without introducing any probe effect.

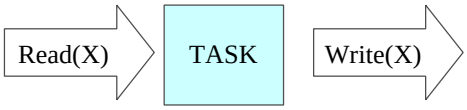
In order to reason about this we make some refinements on the original system model in chapter 2. We assume that the software that runs on the distributed system consists of a set of concurrent tasks, communicating by message passing. Functionally related and cooperating tasks, e.g., sample-calculate-actuate loops in control systems, are defined as transactions. The relationship between the cooperating tasks with respect to precedence (execution order), interactions (data-flow), and a period time typically define each transaction. The tasks are distributed over the nodes, typically with transactions that span several nodes, and with more than one task on each node. All synchronization is resolved before run-time and therefore no action is needed to enforce synchronization in the actual program code. Different release times and priorities guarantee mutual exclusion and precedence. The distributed system is globally scheduled, which results in a set of specific schedules for each node. At run-time we need only synchronize the local clocks to fulfill the global schedule [25].

Task model

We assume a fairly general task model that includes both preemptive scheduling of statically generated schedules [69] and fixed priority scheduling of strictly periodic tasks [1][40]:

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

Related to the task model we assume that the tasks may have functional and temporal side effects due to preemption, message passing and shared memory. Furthermore, we assume that data is sent at the termination of the sending task (not during its execution), and that received data is available when tasks start (and is made private in an atomic first operation of the task) [13][29].



On-line reproducibility

From the perspective of a single transaction, reproducible behavior 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 ordering failure semantics, that the jobs have no unwanted functional side effects via unspecified interfaces, otherwise we could miss such errors. Control over the execution times in other transactions can easily be achieved by incorporating delays in the jobs, or running dummies, as long as they stay within each job's execution time range [*BCET*, *WCET*].

For example, consider *Figure 4-9*, which illustrates the possible execution orderings of the schedule in *Table 4-1*. Assume now that task *C* and *A* belong to one transaction, and tasks *B*, *D* to another transaction. Assume further that task *C* uses the last five samples provided by task *A*. With respect to tasks *A* and *C* we can reproduce the different scenarios by running a dummy in place of task *B*. By varying the execution time of the dummy we can enforce the different scenarios.

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

(1) [39,60)

(2) [60,60]

(3) [121,121]

(4) (60,121]

(5) (60,121]

For a longer discussion on the factors for on-line reproducibility and determinism see Thane et al. [63][62].

Table 4-1. A job set for a schedule with a LCM of 400 ms.

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

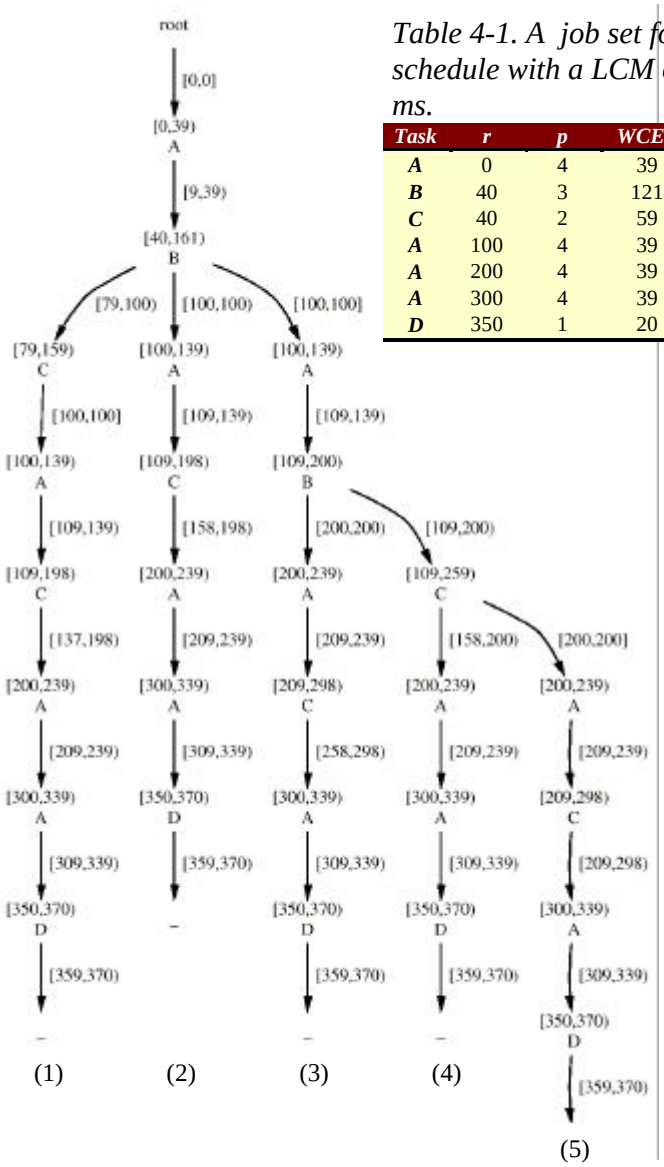


Figure 4-9. The resulting execution order scenarios for the job set in Table 6-1.

5 SUMMARY

We have in this paper presented a framework for monitoring single tasking, multi-tasking, and distributed real-time systems. This includes a description of what to observe, how to eliminate the disturbances caused by the actual act of observing (the probe effect), how to correlate observations (how to define a global state), and how to reproduce them. We have given a taxonomy of different observation techniques, and where, how and when these techniques should be applied to obtain deterministic observations. We have argued that it is essential to consider monitoring early in the design process in order to achieve efficient and deterministic observations. Software monitoring is also the preferable approach since it *scales* better than the hardware approaches. Software monitoring can compared to hardware monitoring also observe the system on many levels of abstraction while hardware monitoring is limited to observation of low level details.

6 REFERENCES

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

[2] Audsley N. C., Burns A., Richardson M.F., and Wellings A.J. *Hard Real-Time Scheduling: The Deadline Monotonic Approach.* Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software, pp. 127-132, Atlanta, Georgia, May, 1991

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

[4] Brantley W.C., MeAuliffe K.P. and Ngo T.A. *RP3 performance monitoring hardware*. In M. Simmons, R. Koskela, and I. Bucher, eds. *Instrumentation for Future Parallel Computing Systems*, pp. 35-45. Addison-Wesley, Reading, MA, 1989.

[5] Butler, R.W. and Finelli, G.B. *The infeasibility of quantifying the reliability of life-critical real-time software*. IEEE Transactions on Software Engineering, (19): 3-12, January, 1993.

[6] Calvez J.P., and Pasquier O. *Performance Monitoring and Assessment of Embedded HW/SW Systems.* Design Automation for Embedded Systems journal, 3:5-22, Kluwer A.P., 1998.

[7] Chandy K. M. and Lamport L. *Distributed snapshots: Determining global states of distributed systems*. ACM Trans. On Computing Systems, 3(1):63-75, February 1985.

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

[9] Chodrow S.E, Jahanian F., and Donner M. *Run-time monitoring of real-time systems*. In Proc. of IEEE 12th Real-Time Systems Symposium, San Antonio, TX, pp. 74-83, December 1991.

[10] Clarke S.J. and McDermid JA. *Software fault trees and weakest preconditions: a comparison and analysis.* Software Engineering Journal. 8(4):225-236, 1993.

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

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

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

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

[15] Ferrari D. *Consideration on the insularity of performance perturbations.* IEEE Trans. Software Engineering, SE-16(6):678-683, June, 1986.

[16] Fidge, C. *Fundamentals of distributed system observation*. IEEE Software, (13):77 – 83, November, 1996.

[17] G J. Myers. *The Art of Software Testing*. John Wiley and Sons. New York 1979.

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

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

[20] Gorlick M. M. *The flight recorder: An architectural aid for system monitoring*. In Proc. of ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, pp. 175-183, May 1991.

[21] Graham R. L. *Bounds on Multiprocessing Timing Anomolies*. SIAM journal of Applied Mathematics, 17(2), March, 1969.

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

[23] Hetzel B.. *The Complete Guide to Software Testing*. 2nd edition. QED Information Sciences, 1988.

[24] Joyce J., Lomow G., Slind K., and Unger B. *Monitoring distributed systems*. ACM Trans. On Computer Systems, 5(2):121-150, May 1987.

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

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

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

[28] Kopetz H. *Sparse time versus dense time in distributed real-time systems*. In the proceedings of the 12th International Conference on Distributed Computing Systems, pp. 460-467, 1992.

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

[30] Kopetz H.. *Event-Triggered versus Time-Triggered Real-Time Systems*. Lecture Notes in Computer Science, vol. 563, Springer Verlag, Berlin, 1991.

[31] Kopetz, H. and Kim, K. *Real-time temporal uncertainties in interactions among real-time objects.* Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems, Huntsville, AL, 1990.

[32] Lamport L. *Time, clock, and the ordering of events in a distributed systems*. Comm. Of ACM, (21):558-565: July 1978.

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

[34] Lauterbach emulators. Lauterbach GmbH Germany. http://www.lauterbach.com/.

[35] LeBlanc T. J. and Mellor-Crummey J. M. *Debugging parallel programs with instant replay*. IEEE Trans. on Computers, C-36(4):471-482, April 1987.

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

[37] Lee J.Y., Kang K.C.,Kim G.J., Kim H.J. Form the missing piece in effective real-time system specification and simulation. In proc. IEEE 4th Real-Time Technology and Applications Symposium, pp.155 – 164, June 1998.

[38] Liu A.C. and Parthasarathi R. *Hardware monitoring of a multiprocessor systems.* IEEE Micro, pp. 44-51, October 1989.

[39] Lozzerini B., Prete C. A., and Lopriore L. *A programmable debugging aid for real-time software development.* IEEE Micro, 6(3):34-42, June 1986.

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

[41] Malony A. D., Reed D. A., and Wijshoff H. A. G. *Performance measurement intrusion and perturbation analysis*. IEEE Trans. on Parallel and Distributed Systems 3(4):433-450, July 1992.

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

[43] Mellor-Crummey J. M. and LeBlanc T. J. *A software instruction counter*. In Proc. of 3d International Conference on Architectural Support for Programming Languages and Operating Systerns, Boston, pp. 78-86, April 1989

[44] Miller B.P., Macrander C., and Sechrest S. *A distributed programs monitor for Berkeley UNIX*. Software Practice and Experience, 16(2):183-200, February 1986.

[45] Mink K., Carpenter R., Nacht G., and Roberts J. *Multiprocessor performance measurement instrumentation*. IEEE Computer, 23(9):63-75, September 1990.

[46] Netzer R.H.B. and Xu Y. *Replaying Distributed Programs Without Message Logging.* In proc. 6th IEEE Int. Symposium on High Performance Distributed Computing. Pp. 137-147. August 1997.

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

[48] Poledna S. *Replica Determinism in Distributed Real-Time Systems: A Brief Survey*. Real-Times systems Journal, Kluwer A.P., (6):289-316, 1994.

[49] Powell D. *Failure Mode Assumptions and Assumption Coverage: In Proc. 22nd International Symposium on Fault-Tolerant Computing.* IEEE Computer Society Press, pp.386-395, July, 1992.

[50] Puschner P. and Koza C. *Calculating the maximum execution time of real-time programs*. Journal of Real-time systems, Kluwer A.P., 1(2):159-176, September, 1989.

[51] Raju S. C. V., Rajkumar R., and Jahanian F. *Monitoring timing constraints in distributed real-time systems*. In Proc. of IEEE 13th Real-Time Systems Symposium, Phoenix, AZ, pp. 57-67, December 1992.

[52] Reilly M. *Instrumentation for application performance tuning: The M3l system*. In Simmons M., Koskela R., and Bucher I., eds. Instrumentation for Future Parallel Computing Systems, pp. 143-158. Addison-Wesley, Reading, MA, 1989.

[53] Rothermel G. and Harrold M.J. *Analyzing regression test selection techniques*. IEEE trans. Software Engineering. 8(22):529-551. August 1996.

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

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

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

[57] Schütz W. *Real-Time Simulation in the Distributed Real-Time System MARS*. In proc. European Simulation Multiconference 1990, Erlangen, BRD, June 1990.

[58] Shimeall T. J. and Leveson N. G. *An empirical comparison of software fault-tolerance and fault elimination.* IEEE Transactions on Software Engineering, pp. 173-183, Feb. 1991.

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

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

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

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

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

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

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

[66] Tokuda H., Kotera M., and Mercer C.W. *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.

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

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

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

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

[71] Leveson N. G. *Safeware - System, Safety and Computers*. Addison Wesley 1995. ISBN 0-201-11972-2.

[72] Littlewood B. and Strigini L. Validation of Ultrahigh Dependability for Software-based Systems. Com. ACM, 11(36):69-80, November 1993.

[73] Parnas D.L., van Schouwen J., and Kwan S.P. *Evaluation of Safety-Critical Software*. Communication of the ACM, 6(33):636-648, June 1990.