On the Testability of Distributed Real-Time Systems *

Werner Schütz

Institut f. Techn. Informatik
Technical University of Vienna
A-1040 Vienna, Austria

Abstract

In our work on real-time systems we found it useful to distinguish between event-triggered and time-triggered systems. The goal of this paper is to compare them with respect to their testability. To facilitate the comparison, three aspects of testability are introduced: Test Coverage, Observability, and Controllability. After a brief review of the problems of testing concurrent programs, we compare the two system architectures with respect to each of these three aspects.

The results of the comparisons favour time-triggered systems in all three aspects, most notably with respect to Controllability and Test Coverage. Thus, time-triggered systems are inherently more testable than event-triggered systems.

Finally, a series of experiments is summarized that shows that it is in fact easy to achieve reproducible testing of time-triggered systems.

1 Introduction

Testing is a form of verification which involves exercising a system and supplying it with valued inputs [10]. Testing safety-critical real-time systems is becoming ever more complex and expensive. It has been repeatedly reported that testing such a system typically consumes in the order of 50% of the total project costs [13,1]. This is due first to the more stringent demands placed on a system and its verification procedures in the context of safety-critical applications, secondly to the distributed architecture of modern systems.

Testing distributed programs is inherently more complex than testing sequential programs, mainly because of synchronization and communication requirements between cooperating processes, whose actions may depend on the actions of other processes and on the time and duration of these actions. Testing real-time programs adds yet another dimension to the problem, because the concern is now not only with the functional correctness, but also with the timeliness of the results produced.

It is therefore important to design a system in such a way as to facilitate its testing as much as possible. In this paper we introduce the notion of *system testability*. Testability is a qualitative measure of the ease with which a system can be tested. The following aspects of testability have been identified:

- *Test Coverage:* The Test Coverage indicates how many of the anticipated real-world scenarios can be covered by corresponding test scenarios. Ideally, this coverage would be 100%, in reality there are significant limitations due to the combinatorial explosion of possible event combinations.
- *Observability:* Observability is important for determining whether the system under test performs correctly or not. There are two aspects. First, one must be able to observe every significant event generated by the environment and – much more important and difficult – to determine the correct ordering and timing of events. Secondly, it is necessary to observe the actions and outputs of the system during testing, but without disturbing its time behaviour.
- *Controllability:* During testing, one must be able to control the execution of the system so that it is possible to reproduce arbitrary test scenarios such that the system's reaction is deterministic.

Testability is to a great extent determined by the properties of the underlying system architecture. Therefore, in this paper we want to assess the differences in the testability of time-triggered and event-triggered architectures that are introduced in Section 2. As a starting point of this comparison we briefly review the problems of testing concurrent programs in Section 3. In Sections 4, 5, and 6 the two types of

*This work was supported in part by the ESPRIT Basic Research Project 3092 "Predictably Dependable Computer Systems"

systems are compared with respect to Observability, Controllability, and Test Coverage, respectively. We note that Observability and Controllability correspond to two well-known problems of testing concurrent programs, namely the probe effect and the reproducibility problem, respectively. Finally, Section 7 summarizes a series of experiments to show that it is in fact easy to achieve reproducible testing of time-triggered systems.

2 The System Model

The system model used throughout this paper is as follows: A distributed system consists of a number of *nodes* which are interconnected by a *network*. Each node is a self-contained computer, consisting of a CPU, local memory, access to the network, a local clock, and other (optional) peripheral devices.

The software running on a distributed system consists of a set of (parallel) *processes* which communicate by passing *messages* to one another. Each node may execute more than one process, but no process has direct access to the data of another process, regardless whether this other process resides on the same node or not.

Based on [6], we distinguish between two types of systems: *Event-triggered (ET) systems* are characterized by the fact that all their actions (computations, execution of a communication protocol, or interactions with the environment) are initiated by the observation of some event (e.g. receiving an external input, receiving a message from another process, interrupt, ...). Usually, an ET system does not provide a global time to the application processes, i.e. the clocks in different nodes of the system are not synchronized.

In contrast, *time-triggered (TT) systems* initiate all their actions exclusively at predefined points in time. Thus, TT systems require access to a global time base which can be implemented by synchronizing the clocks of all nodes. Modern VLSI technology allows a relatively low-cost implementation of clock synchronization [8].

Based on this global time base, a TT system must use TT computations, TT communication protocols, and TT observations (i.e. interactions with the environment) [6]. All of those start at certain points in time which are known in advance. In a real-time (ET or TT) system, it is also necessary to know the *duration* of each of these actions, in order to be able to correctly implement all timeliness requirements. Therefore, in a TT real-time system it is also known in advance when each of the system's actions ends, usually based on some kind of analytic worst-case analysis [15].

A TT computation is then a piece of code that is executed between two well-known points in time by a specific process using only resources local to the node where this process resides. A TT communication protocol transmits a message from one process to one or several other processes (which may reside in nodes different from the node of the sending process). It starts at a specified time and guarantees message delivery before the end-time, provided the specified fault-hypothesis (e.g. single fault tolerance) is not violated. This means that if the fault hypothesis holds, then message delivery will be successfully completed before the end-time. One of the most important consequences of this approach is that it allows error detection by the receiver(s) [6].

Lastly, a TT observation is a special case of communication protocol that is concerned with the interaction between a system and its environment. The environment communicates with the system by some type of input (e.g. messages, signals, ...). The environment is usually not under the control of the system, and the system's and the environment's notions of "time" are usually not synchronized. Thus, the arrival of any input presents itself as an event, which is detected and serviced immediately by an ET system. A TT system, however, does not react immediately, but instead polls the state of the environment periodically, and only then detects all inputs which have arrived since the last polling action and services them in some (fixed) order.

It now makes sense to organize the actions of a TT system in such a way that a particular action is initiated only after all its preconditions have been fulfilled. For example, a computation starts only after all its inputs have been delivered and received, or a communication is initiated only after other actions have had the opportunity to complete the computation of the intended message. Thus, a TT system may also be thought of as executing a sequence of computation and communication steps.

It is interesting to note that a TT system can be interpreted as a specialization of a state machine [16]. The outputs of a TT system are not completely determined by the sequence of requests it processes, but only by the combination of requests sequences and the time between requests.

3 Testing Concurrent Systems

As a starting point, we briefly summarize the problems posed by testing concurrent (both distributed and uniprocessor) systems, whether they are real-time or not. All known approaches assume either ET systems

or systems outside the scope of the above definitions (see Section 2), e.g. systems with shared memory. For a more detailed summary of the state-of-the art in this field the reader is referred to [12].

It is generally recognized that testing concurrent programs is harder than testing sequential programs. The most important reasons are the following [12]:

- the "probe effect",
- non-reproducible behaviour of the system,
- increased complexity, and
- the lack of a synchronized global clock.

The last two points will not be discussed explicitly in this paper. Complexity is increased because errors can occur not only in each individual (sequential) process, but also in process interaction or synchronization, and because these interaction and synchronization requirements make a concurrent program much harder to understand, to write, and ultimately, to test than a sequential program. For the two system architectures that we want to compare in this paper, ET and TT systems, the situation is the same.

The availability of a synchronized global clock is one of the main characteristics of TT systems which distinguishes them from ET systems. Therefore, this aspect will be implicit in all further comparisons between ET and TT systems.

In the subsequent sections we will discuss the probe effect and the reproducibility problem in detail.

4 The Probe Effect

The probe effect [12] is a name for the fact that the behaviour of a system may be changed by attempting to observe this behaviour. It is thus the main impediment to achieving *observability*. It depends on the method chosen for data collection (monitoring) during testing. Monitoring may be *intrusive* [11] or *interfering* [23]. While the probe effect is not present or is (justifiably) ignored in non-real-time sequential programs, it becomes much more stringent in concurrent and/or real-time programs. The main concern is possible interference with the relative timing between processes, which may either prevent certain timing or synchronization related errors from occurring, or may introduce new errors which would not occur without the probe.

The probe effect can be addressed in a number of ways [12]: It can be *ignored* in the hope that the probe effect will in reality not or only rarely appear, it can be *minimized* by implementing "sufficiently" efficient monitoring operations, and it can be *avoided*, either by applications which allow hiding it behind logical time [9], or by leaving all monitoring and testing support in the system during its productive operation, or by employing dedicated hardware for monitoring.

In real-time systems, where timing constraints are imposed upon the system behaviour, we cannot, in general, be content with ignoring or minimizing the probe effect. (There may be applications that allow this, though.) Furthermore, hiding the probe effect behind logical time also works only for a particular class of applications. Logical time [9] can be employed to establish a unique (partial or total) *order* of events in a distributed system. If the externally perceivable system behaviour depends solely on the order of event occurrences, additional output or debugging statements may be inserted into one or more processes with some freedom, as long as these or similar modifications do not change the event ordering. Thus, although the timing of individual processes may be changed, there is no observable effect and "the probe effect is hidden behind logical time".

Unfortunately, the behaviour of many real-time systems depends not only on the order of events, but also on the elapsed real time *between* events. Therefore, this approach will not work for real-time systems in general. This leaves the possibilities of either using dedicated hardware, or integrating all monitoring and test support with the system (e.g. by incorporating it into the system calls), or a combination of both.

Several such monitoring facilities have been developed during the last years. Most of them use dedicated hardware which hooks into a single node, typically into the inter-node bus, thus passively listening to the traffic between the processor, the local memory, and other parts of the node [11,14,23]. This strategy is in principle suitable for both uniprocessor and distributed systems, although the overhead and complexity increase with the number of nodes in the system.

Monitoring on a per-node basis can in principle be applied to both ET and TT systems. However, TT systems offer some alternative flexibility as follows: Since the start and end times of each TT computation are known, it is possible to e.g. insert additional output statements for debugging purposes into a TT computation. If the TT computation still finishes before its end time, the probe effect can be avoided, although it might still occur when the computation accesses the global time (see Section 5). The known start and end times thus serve as "temporal fire walls" which protect the rest of the system from being affected by local changes of the temporal behaviour. (A similar idea are the "time fences" in the ARTS system [22].)

For distributed systems, an alternative or additional approach is to use dedicated nodes for monitoring the inter-node communication. The advantage is that no specialized (customised) monitoring hardware is necessary, and that special monitoring functions can be relatively easily implemented on the monitor nodes. The disadvantage is that information about the internals of a process or a node cannot be collected easily. This can be countered by a suitable test procedure on a per-node or per-process basis (see above). Provided that the presence or absence of monitor nodes is transparent to the other nodes, inter-node traffic monitoring can be applied with both ET and TT systems.

As we have seen, most of this discussion applies equally to ET and TT systems. In real-time systems, the probe effect has to be avoided, regardless of the architecture. However, TT systems provide slightly more flexibility in choosing monitoring mechanisms on a per-process basis.

5 Achieving Reproducibility

Testing sequential programs relies heavily on the assumption that results are reproducible, i.e. a program will compute the same results when supplied with the same inputs. With concurrent, i.e. either parallel or distributed, programs this is no longer the case. However, reproducibility is still considered valuable, for instance because it allows regression testing, i.e. (automatic) re-testing after modifications or maintenance. In order to achieve reproducibility it is necessary to control the execution of the system. Thus, *controllability* is primarily related to the effort of achieving reproducible behaviour.

For arguing about the reproducibility problem, it is first necessary to explore the reasons for non-reproducible behaviour. The behaviour of a distributed system may be non-reproducible because of nondeterminism in its hardware, software, or operating system. Possible reasons are [12]:

- the presence of concurrent activities leading to *races*, whose outcome is determined by the CPU load, the network traffic, non-determinism in the communication protocol, the probe effect (see above), and possibly other factors. It is in general not possible to know about these influences and their effects beforehand, therefore the behaviour of the system is not predictable. Slight changes in one of these factors may change the outcome of such a race, which may in turn lead to a drastically different system behaviour.
- the presence of non-deterministic statements in the application software, e.g. the use of random numbers.

In this paper we are not considering non-deterministic statements since these would pose similar problems in sequential systems. The source of non-determinism which we are concerned about is thus the lack of *sufficient a-priori knowledge* about the type and timing of events. On a more detailed level, the following factors contributing to non-reproducible behaviour will be considered in the rest of this paper: (We have already concluded that the probe effect must be avoided in real-time systems.)

- Synchronization of processes, given by the order of synchronisation events or the order of executing synchronization constructs.
- Access to the system's notion of "time", i.e. by accessing a clock or a timer.
- Asynchronous interrupts.

In the rest of this section, the reproducibility problem is discussed for ET and TT systems separately; in addition, for ET systems, it is useful to distinguish between language-based and implementation-based approaches [21].

5.1 ET Systems: Language-Based Approaches

The characteristic of language-based approaches is that different execution scenarios (sequences) are identified based on the concurrent program (or its specification). Reproducibility is then achieved by enforcing the identified scenarios during execution. A *serialization* is generated by a *random scheduler* that "merges" the bodies of all processes of the concurrent program into a single sequential program [24]. Instead of testing the concurrent program, a set of serializations is tested.

The question remains, however, how to adequately choose this set of serializations from all possible serializations, especially since an appropriate choice may depend on the actual input data. On the other hand, many serializations may be equivalent. As a special case, the following holds [24]:

> *If* all shared data are only accessed within synchronization constructs *and* the sequence of synchronization constructs is unique, *then* all possible serializations are equivalent, i.e. they yield the same result on a given input.

While the first precondition is fulfilled for all the systems considered in this paper (processes do not have access to the internals of another process, and data are only shared via message passing – see Section 2), the second condition must be enforced.

To this author's knowledge, Per Brinch Hansen was the first to address this problem [2,3]. His method relies on the atomicity of monitor calls. A sequence of monitor calls is constructed to satisfy a particular testing criterion. This sequence is executed by a number of test processes such that every monitor call is issued at some predefined and unique point in (logical) time. Time is simulated by a clock process such that the real-time interval between consecutive ticks is about an order of magnitude longer than the duration of the slowest monitor operation. This method is described for a uniprocessor system, but has been applied to a distributed system as well.

In more recent work, Tai [18,21] characterizes a test case as (a collection of) inputs and a *synchronization sequence*, a sequence of synchronization events that uniquely determines the results(s) of a concurrent program execution with some input. A synchronization sequence is valid with a particular input if the program specification allows this particular synchronization sequence for this input, otherwise it is invalid with this input. Finally, the program is executed while simultaneously trying to enforce the synchronization sequence ("forced execution"). The program is said to contain a synchronization error if either a deadlock results, or a valid synchronization sequence cannot be enforced, or an invalid synchronization sequence can be enforced.

The procedure to reproduce a given synchronization sequence depends on the exact nature of the synchronization events. Solutions have been developed for programs using SEND/RECEIVE primitives and timers [19] (for both single processor and distributed systems), and for concurrent Ada programs communicating by rendezvous [20,21]. All solutions rely on source code transformations, i.e. on inserting additional synchronization statements into the source code.

Because of the additional synchronization requirements, a "forced execution" usually introduces a probe effect. This does not affect the computation since this additional synchronization is necessary to enforce a particular order of events. However, the result of a computation may also depend on the execution *duration*, especially in real-time systems where e.g. the result depends on some perceived time difference, or where a different computation is performed in case a timeout is detected. It is important to recognize that access to "time" is in this case a significant event. Access to a "time server" is the only way a process can gain knowledge about time, thus these accesses have to be performed under the control of the "forced executions" mechanism, which means that time has to be simulated. This is addressed by Tai's work [19].

Tai's work allows deterministic execution of real-time systems both on the host and the target system [20], provided the system is not required to handle asynchronous interrupts. Such interrupts may occur *during* the execution of a basic source code statement, therefore a source code transformation strategy is unable to enforce all possible situations.

5.2 ET Systems: Implementation-Based Approaches

The characteristic of implementation-based approaches is that an attempt is made to collect all missing information during an execution of the system. Subsequently, this information is used to reproduce or *replay* this particular execution. All the work in this area is based on *event histories* or traces. This approach requires identifying all significant events and monitoring and recording them during the execution of a distributed system. The result is an event history or trace that can be used to control the re-execution of the system, with the goal of replaying the original execution. [12] summarizes these techniques and contains further pointers to relevant literature. Note that not all the systems described there provide a replay capability.

An event history can be used in two ways: First, to replay the same program with the intention of getting more information about the program behaviour. This can be done by e.g. running each sequential process in the system through an interactive debugger. This does introduce a probe effect, but it can influence only the duration of process execution because their relative timing (synchronization) is enforced by the event history. Secondly, to replay a modified program with the intention of checking the correction of an error or of experimenting with different algorithms.

[23] describes an integrated monitoring and replay mechanism which can handle replay of significant events in the right order, replay of access to time, and replay of asynchronous interrupts. It is suitable for real-time systems because the monitoring phase avoids the probe effect. Replay, however, may be slower than real execution.

Other monitoring approaches avoiding the probe effect have been described above [11,14]; these do not address replay.

Significant events may include synchronization events, access to time, and asynchronous interrupts, depending on the type of system. Thus, although all factors contributing to nondeterminism (see above)

can be handled, there are a few disadvantages to these approaches:

- Special, dedicated hardware has to be used.
- The amount of information required for replay is usually large, since replay takes place at a low level (machine level). In addition to the inputs, intermediate events (e.g. messages) have to be kept as well.
- One can only replay what one has previously observed. There is no guarantee that every significant system behaviour will be observed eventually.
- If a given event history is used to control the execution of a modified (corrected) program, it is unknown in advance, if this event history is valid for the new program.
- Event histories can only be used to replay the execution on the same system on which the events have previously been monitored and recorded, i.e. the target system.

5.3 TT Systems

In contrast, reproducibility of TT systems is essentially *free* because there are no uncertainties or missing information concerning their behaviour. We will consider the three impediments to reproducible behaviour identified above in turn.

It is evident from the discussion of system properties in Section 2 that all possible serializations of a program which executes on a TT system are indeed equivalent. The start and end times of all system actions are completely pre-determined. All resource conflicts or precedence constraints can be resolved before run-time, resulting in a suitable choice for these times. For each node, there is then a unique sequence of system actions, and no other run-time synchronization is needed.

The only "synchronization construct" is a TT communication protocol which is the only way to pass shared data between processes. Again, there exists one and only one synchronization sequence which is determined by the order of the end-times of all communication protocol invocations. If two or more communication protocols are active (execute) simultaneously, it is assumed that the network either handles them serially or provides independent paths.

A TT computation may access the global time at some point during its execution, e.g. by using a special system call. It is still an open research issue how to make access to time deterministic. In the following, two extreme possibilities are discussed:

First, one could rely on a deterministic behaviour in time of the underlying hardware. Since we have already decided that the probe effect must be avoided in real-time systems, a TT computation is started at a particular point in time and will reach the call to access the timer after a fixed number of instructions, given it is supplied with the same input data and starts computation from the same system state. If the execution time of an instruction is constant if it is executed on the same data, then access to a timer will also be deterministic and reproducible. This is not to say that any existing systems, including MARS [5], do in fact completely fulfill these assumptions. It might, however, be possible to construct such systems. As a disadvantage, low-level fault-tolerance mechanisms, such as instruction retry or memory error correction, must probably be sacrificed because then the execution time would depend on the number of faults that actually occurred.

Secondly, the call to the timer could return the start-time of the current TT computation always, no matter at what point during the computation it is actually issued. Access to time is not "arbitrarily precise", but the precision is limited to a time grid with a particular granularity ("scheduling grid"). It is not yet known if this approach is too restrictive or if it is suitable for a large class of real-time applications. However, we note that (within limits of reasonableness) it is possible to control the granularity of the scheduling grid by appropriately designing the duration of TT computations.

Asynchronous interrupts are the usual implementation of events in ET systems. As such, they are unnecessary in TT systems because there are no (different kinds of) explicit events. Instead, "events" are recognized by combining time and the (implicit) knowledge about the ordering of all system actions in time. Being a direct representation of events, asynchronous interrupts are thus inconsistent with the definition of TT systems.

According to the properties of TT observations, a test run in a TT system is then uniquely characterized by (a sequence of) input data consisting of both *values* and the *time* when these values are expected. When testing a TT system, one can thus concentrate on developing meaningful test cases; this is also the strong point of the language-based approaches.

6 Test Effort

The last aspect of comparing ET and TT systems is related to the Test Coverage. Preliminary work in this area can be found in [17]. In considering the advantages of TT over ET systems with respect to the Test Coverage, there are several aspects.

The first aspect has already been mentioned in conjunction with the reproducibility argument (see Section 5). This is the fact that there is one and only one synchronization sequence in a TT system. Apart from enabling deterministic execution, the other consequences are that

- no representative or "important" synchronization sequences have to be selected for testing, which would necessarily reduce the Test Coverage. Instead, efforts can focus on the only possible synchronization sequence.
- Even if all possible synchronization sequences could be tested, it would then be necessary to generate test cases for each of them, so that all synchronization sequences can be tested *adequately*. In addition, one has to determine whether a particular input sequence may give rise to one or more synchronization sequences, and one has to enforce each possible synchronization sequence in the latter case. In TT systems, the issues of synchronization sequence and test data generation are independent, thus separating these concerns. All the available effort can be directed towards creation of test cases for a single synchronization sequence; as a result, one can reasonably expect a more thoroughly tested system.

The second aspect relates to the way in which ET and TT systems perceive the states of their environments (see Section 2). Let us consider the actions of a TT and an ET system during a time interval equal to the polling period of the TT system. Assume that a maximum of n independent inputs can arrive (occur) during this time interval. A TT system detects the presence or absence of each of the possible inputs at the end of this time interval. At this time the order in which these inputs occurred is no longer of significance. (To ensure this is indeed the case, it is the designer's responsibility to choose an appropriate polling frequency in accordance with the application requirements.) The number of states of the environment that a TT system is able to perceive, and equivalently the number of possible control paths that are executed in response to the observed state, is then: $CP_T(n) = 2^n = \sum_{k=0}^{n} \binom{n}{k}$

For ET systems, on the other hand, the order of inputs is not transparent, so the number of environment states that may be detected by an ET system at the end of the same time interval is: $CP_E(n) = \sum_{k=0}^{n} \binom{n}{k} k!$

In addition, it is important to note that $CP_T(n)$ reflects the actual number of different control paths, while $CP_E(n)$ is really an optimistic lower bound; it is a *lower bound* because it does not take into account the additional effort and complexity of the cases where one (or more) further inputs occur before the first one has been completely serviced; it is *optimistic* because a newly arriving input can interrupt the service of the previous input at an arbitrary point in the latter's service routine, thus leading to a "combinatorial explosion" of the number of possible control paths. Figure 1 shows the ratio of $CP_E(n)$ over $CP_T(n)$ for some (small) values of n. Note especially the logarithmic scale of the Y-axis!

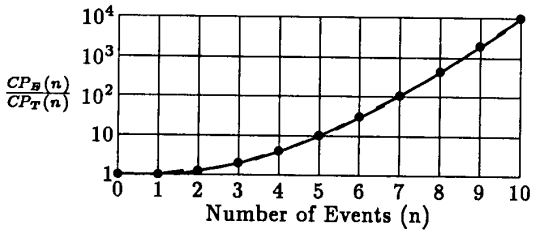

Figure 1: Test Effort Increase of Event-Triggered over Time-Triggered Systems

This curve is proportional to the lower bound of the Test Coverage loss of an ET system with respect to a TT system, if we assume that the budget allocated to testing is the same for both types of system. This is because testing has to cover a much larger number of states of the system environment in an ET system than in a TT system. Even for very small values of n, the Test Coverage of a TT system is better because all inputs in a TT system are recognized only at well-defined points during system execution, i.e. at well-defined system states. This is not reflected in Figure 1 because only a lower bound for $CP_E(n)$ can be given.

An alternative interpretation is that the function shown in Figure 1 indicates the factor by which the test budget and test resources have to be increased for an ET system in order to achieve an equivalent level of Test Coverage as for a corresponding TT system.

Finally, the properties of TT observations (see Section 2) ease the task of actually executing test runs. It is not necessary to deliver a particular input to the system interface at a certain point in time with very

small tolerances, instead test inputs can be applied at any time between two scheduled observations without affecting the behaviour of the TT system. Moreover, if there is more than one input, their relative order is insignificant. Thus, even if a process changes its internal handling of messages, the way of presenting the test inputs to the system can remain unaltered. There is a link to reproducibility insofar, as the amount of information in a test case which is necessary to make it reproducible is reduced in comparison with a reproducible test case for ET systems. The degree of independence between test cases and system implementation is increased, therefore less effort for maintaining and updating test cases can be expected.

7 An Example: Reproducible Test Runs in MARS

In this section we summarize the experiences gained from a number of experiments which were conducted in order to evaluate the feasibility of reproducible test runs. These experiments and their evaluation are based on a MARS system [5] executing the recently developed "Rolling Ball" application [7]. We first describe this application in some detail, then give the most important results.

7.1 The "Rolling Ball" Application

The "Rolling Ball" application [7] is a real-time control problem whose aim is to balance a ball on a flat surface (plane) in such a way that the ball follows a predefined course. The position of the ball is periodically (40 ms) sent to the MARS system which then computes commands for a set of servo motors to set the inclination of the plane. A two-dimensional orthogonal coordinate system on the plane is used.

Figure 2 shows the configuration for the experiments. For brevity, the connection to the host computer for loading the MARS components has been omitted.

The individual parts of this system cooperate as follows: Since the video camera could not be connected directly to a MARS component, a PC has to be used as an intermediary. The PC reads a picture from the camera every 40 ms, processes it, and sends data (i.e. one of coordinate system initialization, course initialization, or ball position) over a serial line to the Video component.

The Video component processes all the initialization data, then, as long as it continues to receive ball positions, it computes the current status of the ball (consisting of acceleration, velocity, and position, each

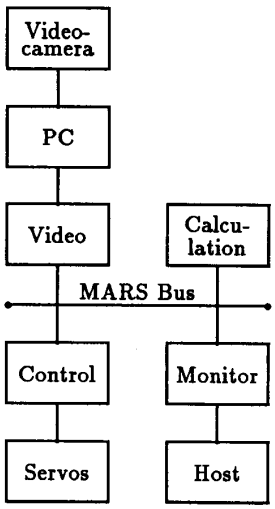

Figure 2: Experiment Configuration

for both the x- and the y-axis), and sends this status to the Calculation component.

The Calculation component uses this information and the description of the required course (also received from the Video component) to calculate the accelerations for the x- and the y-axis, respectively, which are required to keep the ball on the defined course (or bring it back to its course). These acceleration values are monitored by the Monitor component (see below) and are used to determine the experiment results.

Finally, the Control component computes the angles that the plane must be set to in order to achieve these accelerations. These angles are then sent to the servo motors, thus closing the control loop.

The Monitor component is not part of the original application; it has been added for observing and recording the experiment results. The Monitor periodically (i.e. each 40 ms) tries to receive a message from the Calculation component. If there is a message, its contents, consisting of a receiving time stamp and the two acceleration values, are output to the Monitor component's serial line interface. Otherwise, the string "no message" is output. The serial line is connected to a host computer where the experiment results are stored permanently. During all the test runs it was generally the case that the output was recorded completely, as intended. Only occasionally did the Monitor (or the host – it is undecidable which part of the system is responsible) miss the output of one iteration. Checks of the recorded time stamps detected this.

7.2 Results

For the purpose of these experiments, the "test data" used are data recorded during previous executions of this application. Each test case contains enough data for about 22 seconds of system execution. The goal of the experiments was to repeat test runs with the same input data and to compare sequences of the monitored outputs (i.e., the acceleration values) obtained by different test runs. If these sequences are identical, we have achieved reproducibility.

In a first round, the test data were input via the serial line, i.e. by the PC, just as if they had been derived directly from the video camera. Here, part of each test run (the input presentation) was under the control of an ET system, namely the PC. The time of data input is not controllable, there might be additional delays on the serial line, and reproducibility could not be achieved. The results of test runs always started to differ a certain (sometimes very small) time after the start of the test runs, the maximum was about 10 seconds. Thus, only the initial phase turned out to be reproducible.

Therefore, in a second round, the setup was changed in order to remove all uncertainties concerning the time of input arrival. The test data are input to the system by replacing the Video component with a test driver [17]. For each iteration of the application (i.e. every 40 ms), the test driver selects the appropriate values from the sequence of test data and puts them into a message which is then sent to the rest of the system.

The test driver is really a modified Video task which simulates receiving input data from the serial line. Because of limitations in the current version of MARS (there is no disk support or other stable storage) it is necessary to keep the test data in an initialized data structure of the test driver. Therefore, the test driver has to be edited and compiled anew for each set of test data. However, these modifications turned out to be relatively trivial.

This changed setup was designed to have a test run completely under the control of the TT system. When using it, the recorded test results of all test repetitions are identical. This means that the sequence of acceleration values is identical for the whole test run. The associated time stamps do differ by approximately 100 μs maximum; this is, however, in the order of the accuracy of clock synchronization and is not taken into account when comparing results of test runs.

The results of these experiments support the test system architecture for open-loop tests proposed in [17]. It is indeed possible to achieve reproducible test runs with this proposed architecture.

8 Conclusion

In this paper we have compared two types of architectures, event-triggered and time-triggered systems, with respect to their testability. To this end, three aspects of testability have been introduced: Test Coverage, Observability, and Controllability. Observability is a necessary prerequisite for efficient testing for all types of systems, but TT systems offer a slightly wider choice of observation mechanisms which at the same time avoid the occurrence of a probe effect. The comparison is in favour of time-triggered systems with respect to each of the other two aspects. Thus, time-triggered systems are inherently more testable than event-triggered systems.

Time-triggered systems, such as MARS [5], have been designed to allow predictable and deterministic execution of real-time software. However, these same properties have turned out to significantly enhance the testability of such systems as an additional benefit. In this paper we were only concerned with the testability aspect. A comparison of ET and TT systems with respect to other aspects can be found in [4].

The evaluation of the two system architectures has been done with a strong eye towards the requirements of hard real-time systems. However, if the comparison were made in the context of non-real-time systems, the arguments leading to the evaluation result would not change and the result of the comparison would be the same. Only the importance generally attributed to arguments of this kind is usually smaller in non-real-time systems. ET systems are cheaper and easier to build and there is more experience with them. It makes sense to use them, at least for applications where safety or predictable temporal behaviour are not of concern.

What has been compared are two extreme cases of possible system architectures. In between there may be other types of systems, such as event-triggered systems with global time, to which some, but not all, arguments in this paper equally apply.

As a final point, a comparison with hardware is in order: Hardware design has integrated functional design with methods that allow testing and evaluation of the designed product. Such work has been ongoing for some time and is known as "design for testability". The techniques developed in this field rely on the fact that a modern piece of computing hardware is, in effect, a time-triggered system: A single clock signal drives the functioning of the rest of the system, and interrupts (events) are not allowed to occur at arbitrary points in time, but are noticed only at well-defined points during execution (e.g. between machine instructions). This is generally regarded as the single most useful abstraction leading from the analog to

the digital world. Time-triggered systems extend this useful abstraction to the level of the interface between operating system and application.

Acknowledgements

The author would like to thank the members of the MARS research group, in particular H. Kopetz, G. Grünsteidl, H. Kantz, P. Puschner, J. Reisinger, A. Vrchoticky, and R. Zainlinger, for stimulating discussions and helpful comments on an earlier version of this paper.

References

[1] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 1983.

[2] P. Brinch Hansen. Testing a Multiprogramming System. *Software — Practice and Experience*, 3(4):145–150, Dec. 1973.

[3] P. Brinch Hansen. Reproducible Testing of Monitors. *Software — Practice and Experience*, 8(4):721–729, Dec. 1978.

[4] H. Kopetz. Event-Triggered versus Time-Triggered Real-Time Systems. *Lecture Notes in Computer Science*, 1991 (to appear).

[5] 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*, 9(1):25–40, Feb. 1989.

[6] H. Kopetz and K. Kim. Temporal Uncertainties in Interactions among Real-Time Objects. In *Proc. 9th Symposium on Reliable Distributed Systems*, pages 165–174, Huntsville, AL, Oct. 1990.

[7] H. Kopetz, R. Noisser, et al. The Rolling Ball: A Real-Time Control Problem. Research Report 2/90, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, 1990.

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

[9] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] J. C. Laprie. Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 1–28. BSP Professional Books, Oxford, 1989.

[11] D. C. Marinescu, Jr. J. E. Lumpp, T. L. Casavant, and H. J. Siegel. A Model for Monitoring and Debugging Parallel and Distributed Software. In *Proc. 13th Ann. Int. Computer Software and Application Conference (COMPSAC '89)*, pages 81–88, Orlando, Florida, Oct. 1989.

[12] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, Dec. 1989.

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

[14] B. Plattner. Real-Time Execution Monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764, Nov. 1984.

[15] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[16] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[17] W. Schütz. A Test Strategy for the Distributed Real-Time System MARS. In *IEEE CompEuro 90, Computer Systems and Software Engineering*, pages 20–27, Tel Aviv, Israel, May 1990.

[18] K. C. Tai. On Testing Concurrent Programs. In *Proc. 9th Ann. Int. Computer Software and Application Conference (COMPSAC '85)*, pages 310–317, Chicago, Illinois, Oct. 1985.

[19] K. C. Tai and S. Ahuja. Reproducible Testing of Communication Software. In *Proc. 11th Ann. Int. Computer Software and Application Conference (COMPSAC '87)*, pages 331–337, Tokyo, Japan, Oct. 1987.

[20] K. C. Tai, R. H. Carver, and E. E. Obaid. Deterministic Execution Debugging of Concurrent Ada Programs. In *Proc. 13th Ann. Int. Computer Software and Application Conference (COMPSAC '89)*, pages 102–109, Orlando, Florida, 1989.

[21] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Transactions on Software Engineering*, SE-17(1):45–63, Jan. 1991.

[22] H. Tokuda and C. W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM SIGOPS Operating Systems Review*, 23(3):29–53, July 1989.

[23] 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 Transactions on Software Engineering*, SE-16(8):897–916, Aug. 1990.

[24] S. N. Weiss. A Formal Framework for the Study of Concurrent Program Testing. In *Proc. Second Workshop on Software Testing, Verification, and Analysis*, pages 106–113, Banff, Canada, July 1988.