A Test Strategy for the Distributed Real-Time System MARS [1]

Werner Schütz

Institut für Technische Informatik
Technische Universität Wien
Austria

schuetz@vmars.uucp

Research Report Nr. 1/90

January 1990

Accepted for IEEE CompEuro '90,
Tel Aviv, Israel, May 1990

Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien
Austria

Phone: (43) 222-58801-8181
Fax: (43) 222-569149

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

Abstract

This paper addresses the topic of testing real-time systems within the framework of the distributed real-time system MARS (MAintainable Real-time System). After a brief survey of the available literature and an overview of the background of this work, a proposed test strategy for MARS is described, and two architectures for test-beds that allow actual test execution in two different test modes are derived. The specification, development, and implementation of one specific test tool, the environment simulator, is then considered and its use in the proposed test strategy is discussed. The principles described in this paper are currently applied to the construction of an experimental application for MARS.

1 Introduction

A *real-time system* is a (computer) system which is required by its specification to adhere not only to functional requirements, but also to time requirements, often also called "timing constraints" or "deadlines". These requirements are typically of the form "complete this computation by 3 o'clock" or "react to this event within x seconds". Typical applications include all those where the system must interact tightly with some outside physical object (other than a human user or operator), e.g. industrial process control.

A common characteristic of real-time applications [1] is that the required deadlines absolutely have to be met under all anticipated fault and load conditions; any failure to do so may drastically increase the punitive cost function, thus leading to huge financial losses, loss of human life (accidents), or other unacceptable consequences (e.g. environmental pollution). Therefore, design and implementation of a real-time system have to *guarantee* that all deadlines will always be met.

Verification and *validation* refer to all activities for determining whether the implementation of a system satisfies certain properties or conditions and for reaching confidence in a system's ability to deliver the specified service, respectively [2]. It is becoming increasingly accepted that verification and validation have to be carried out during the whole system life cycle, in parallel with actual development (see e.g. [3]). Interest in automated software development support has increased recently [4] because this promises to raise the manageability and efficiency of these activities.

Within a software development system, then, both development and verification/validation activities have to be supported, thus leading e.g. to the distinction between D(evelopment)-path and Q(uality Control)-path in [3]. In the MARS Design System (see section 3) the "D-path" (i.e. the set of all design and development activities and their interaction) is rather well established. Therefore, an effort was initiated to investigate the "Q-path".

One of the most common and most important verification/validation activities is *testing*, and it was thus chosen as a starting point for our work. During testing a system is executed and provided with input values with the intention of finding and subsequently correcting errors [2].

In comparison with "traditional" (non-real-time) testing, there are two main challenges to real-time software testing, one fundamental and one practical:

- Not only functional properties must be tested, but also timing properties.
- Real-time systems are usually developed on a *host system*, but eventually executed on a *target system*, sometimes a custom-designed piece of hardware, specifically tailored to efficiently execute the real-time software.

The rest of this paper is organized as follows: Section 2 gives a brief summary of the state-of-the-art of real-time system testing and presents the conclusions which guided our work. Section 3 then summarizes the principles of MARS and section 4 presents some preliminary *general considerations* on the special problems, benefits, and the required effort of testing time-driven real-time systems in comparison with testing event-driven real-time systems. A structured test process for MARS is proposed in section 5. Section 6 addresses the *practical question* how to conduct tests and discusses appropriate test bed architectures. Addressing the question of integrating automated support for test preparation, execution, and evaluation into the Design System is the subject of ongoing work. However, section 7 briefly discusses a method to integrate support for one special type of test tool, i.e. environment simulation.

2 Real-Time Testing: A Survey

Due to the distinction between host and target system, there are typically at least two phases of real-time software testing [5,6,7]:

1. *Testing on the host system* is used for (functional) unit testing and preliminary integration testing.
2. *Testing on the target system* involves completing the integration test and performing the system test. At this stage it is possible to also test the performance and timing characteristics of the software.

During target system testing, it is very common to use an "environment simulator" [5,7], a device that simulates the behaviour of the physical environment of the target system. The reason is that it is both cheaper and safer to use a simulated environment than the real one, especially since confidence in the real-time software is still low. Simulation provides the only possibility to test system behaviour in those "rare event situations" where system dependability is of utmost importance (e.g. emergency handling of a chemical process). Often these situations are neither easily nor safely obtained from the real environment. Other advantages include: (1) it is often much easier to obtain system responses and other intermediate test results from a simulated environment than from the real environment (measurement problem), (2) it is easier to run test with *realistic* inputs, and (3) the need to generate large amounts of input data is eliminated, thus enabling longer and more extensive test runs. However, an environment simulator is often the only support available for target system testing.

The most important drawback [5] of this test strategy is the virtual non-existence of any test support on the target system, and even for the existing support the

interaction between tester and system in most cases takes place at the machine language level, which severly effects productivity. The most important measures proposed in [5] to remedy this situation are:

- Use only high-level programming languages for implementation, so that all unit tests can be easily executed on the host system.
- Conduct as much testing as possible on the host, e.g. use a software environment simulator on the host to perform integration testing.
- Establish communication links between host and target for fast error correction, recompilation, and re-execution after an error has been detected.
- Provide more testing and debug tools for the target system which above all interact with the tester on the level of the *source language*.

Some of these concerns have also been addressed in a recently proposed software development strategy for real-time systems [6].

From the above, the following conclusions were drawn to guide our work: First, environment simulation is a well-accepted and useful test tool, and our proposed test strategy should enable the use of such a tool. Second, the main shortcomings are in the area of supporting target system testing. Thus this became one of the main areas of investigation.

3 An Overview of MARS

MARS (*MA*intainable *R*eal-time *S*ystem) [8] is a system designed to support hard real-time applications and to guarantee that all deadlines are met under all anticipated load and fault conditions. To achieve this, MARS is a *periodic (time-driven)* system where all activities are triggered by the passage of (real) time. External events (stimuli) are detected only at the end of some (finite-length) time interval. This contrasts to *event-driven* systems where each external event is detected upon arrival (interrupt) and handled immediately.

A MARS system consists of a set of *clusters* that may be interconnected in an arbitrary topology depending on application requirements. What subset of the application tasks is assigned to a given cluster is informally determined by requiring that these tasks perform (highly) related functions; it is the designers' responsibility to make reasonable choices. Each cluster consists of a set of *components* interconnected by an Ethernet-type real-time local area network (*MARS bus*). Each component is a self-contained computer that runs an identical copy of the MARS

Operating System and a set of system and application *tasks*. Tasks may interact with each other solely by the exchange of standardized MARS messages.

The so-called "Interface Components" are also connected to a second bus, the *Interface Bus*. This enables a cluster to communicate with its environment, which is either another MARS cluster or the controlled object (the physical process), via appropriate peripheral devices. Standardized MARS messages are used on the Interface Bus as well; a MARS task needs then not be aware of the nature of its communication partner.

In order to guarantee that all deadlines are met, several techniques are used: The system is designed for peak load, i.e. for the case of all stimuli occurring with their maximum frequency. The resource requirements for this case are determined, and resource allocation (CPU, bus access) is done according to these worst-case (or maximum-demand) situations. For CPU scheduling, the maximum execution time of each task [9] is reserved and a schedule is calculated off-line. Since all tasks are executed periodically, the schedule is also periodic. This is done for each component. In order to optimize response times, CPU and bus scheduling can be carried out concurrently [10]. Furthermore, access to the MARS bus must be deterministic; a Time-Division Multiple Access protocol is used in order to avoid queueing delays and message collisions. This guarantees that at any point in time exactly one component of a cluster is granted send-access to the bus. Finally, a prerequisite for the above is that all components have access to a common global time-base. Therefore, a synchronization protocol has been implemented that uses special hardware to keep components' clocks synchronized [11].

Besides the physical MARS system (i.e. the target system), there is also a MARS Design System [12,13] that supports system (and software) development and evaluation. Its most important characteristic is that it allows the specification and refinement of time information throughout the life cycle, from requirements definition to task implementation.

The following development phases are supported by the Design System: Basic system requirements are collected and described in the *Requirements Definition* phase. During *System Design and Cluster Definition* the system is decomposed into a set of MARS clusters and their interconnections. Each cluster is then refined into a set of components in the *Cluster Design and Component Definition* phase. The purpose of *Component Design and Task Definition* is to describe the functionality of each component by a collection of tasks. Lastly, all tasks are implemented in the *Task Implementation* phase using the programming language C.

Within this Design System, system functions are specified as a set of *transactions* each of which consists of a set of attributes. Attributes are the "stimulus" describing when or how a transaction is activated, the "response" describing the

required reaction of the transaction, an informal description of the transaction's functionality, and various time attributes (maximum time between stimulus and response, maximum stimulus arrival rate, etc.). Transactions are used both to specify system behaviour and to describe the functions of subsystems (e.g. clusters). Thus, during design, transactions are iteratively refined into subtransactions until they become simple enough to be implemented as MARS tasks.

4 Exploring MARS Features for Testing

In this section some general theoretical considerations concerning reproducibility of test results and the required test effort will be presented.

A common problem in testing concurrent programs is reproducibility. The most common approach to achieve reproducibility is *serialization* [14,15]. A serialization is generated by a *random scheduler* that "merges" the bodies of all tasks of the concurrent program into a single sequential program [15]. 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.

In MARS, this is not an issue: Due to off-line scheduling, the sequence of tasks executing on the same component and their interleaving is predetermined. The same cannot be said for tasks executing on different components because of the bounded, but still unknown difference of local clocks. However, the upper bound for clock inaccuracy is very small [11] and all synchronization constructs (in MARS: message passing) are scheduled to occur at time intervals larger than the maximum clock inaccuracy. Since all shared data are only accessed within synchronization constructs (message passing) *and* the sequence of synchronization constructs is unique, it follows that all possible serializations are equivalent (as has been proven in [14]), i.e. they yield the same result on a given input.

For any test input that is defined in the value *and* in the time domain, test runs must then deliver the same results when repeated. (They, too, are "deterministic" in that sense.) Thus, *regression testing*, i.e. repeating all prior test runs after an error has been corrected, becomes a viable strategy.

Related to above, it will be easier to predict the correct (intended) outcome of a given test run and thus to evaluate test results because the reaction of the system does not depend upon the exact point in time when a stimulus occurred (which would be the case in an event-driven system).

Let us now compare the test effort of a time-driven and an event-driven system: Assume that a maximum of n stimuli (events) can arrive (occur) during a given time interval, which are detected and serviced immediately by the event-driven

system, and which are detected and serviced in some fixed order at the end of this time interval by the time-driven system. The number of possible control paths that exist due to the potential presence or absence of each of these n stimuli is then for time-driven systems: $CP_t(n) = 2^n = \sum_{k=0}^{n} \binom{n}{k}$ and for event-driven systems: $CP_e(n) = \sum_{k=0}^{n} \binom{n}{k} k!$

Because of the deterministic task execution due to off-line scheduling $CP_t(n)$ reflects the actual number of different control paths, while $CP_e(n)$ is really a very 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 events occur before the first one has been completely serviced; it is *very optimistic* because a newly arriving event can interrupt the service of the previous event 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 compares $CP_t(n)$ and $CP_e(n)$ for some (small) values of n. Note especially the logarithmic scale of the Y-axis!

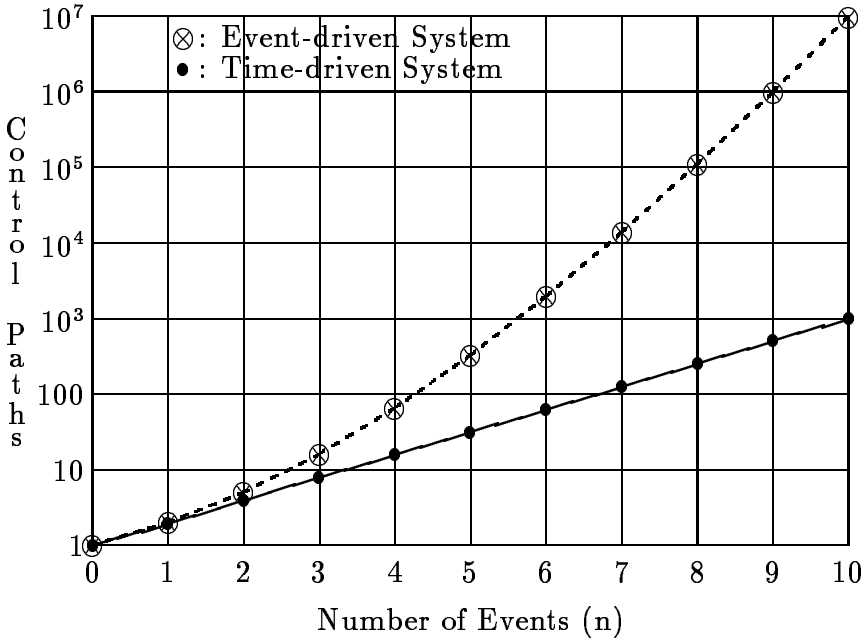


Figure 1: Test Effort for Time-Driven vs. Event-Driven Systems

In general, a system also executes different control paths depending on system state, algorithm or application structure, or specific input data. Let $CP(0)$ denote the number of these potential control paths that are independent of the existence of external stimuli. If each of these paths can be combined with any of the control paths due to external stimuli, then the total number of control paths is $CP(0) \cdot CP_t(n)$

and $CP(0) \cdot CP_e(n)$, respectively.

Typically, each control path is tested with a predetermined, fixed, (and necessarily finite) number of test cases. Thus, $CP_t(n)$ and $CP_e(n)$ are directly proportional to the total test effort required by a time-driven and an event-driven system, respectively. Therefore, the number of test cases required to test all possible control paths is much lower for time-driven systems than for event-driven systems. An equivalent level of testing can be achieved with less effort and resource expenditure. Or conversely, a time-driven system can be tested more thoroughly than an event-driven system with the same invested effort.

5 A Test Strategy for MARS

This section proposes a test strategy for real-time software under the MARS system. All testing activities are structured into a set of test phases which are derived from the design phases supported by the prototype MARS Design System.

5.1 Task Test

The objectives of the Task Test are (1) functional testing (see for instance [16]) of each individual task and (2) performing a preliminary interface (integration) test by statically checking the message handling of each task and comparing it with information from the Design System.

The Task Test is conducted entirely on the host system, with the advantage that rapid switching from executing test runs to program editing (error correction) and vice versa is possible. Further studies of the problems associated with Task Testing and implementation of support tools will be done in conjunction with the work on support for programming-in-the-small, because this test phase is very different in characteristic, and also rather independent, from all the other test phases. Furthermore, it is important that the task programmer is supplied with a consistent and easy-to-use programming-in-the-small tool set.

5.2 Cluster Test

The Cluster Test will be conducted on the target system. There are two variants of the Cluster Test, the open-loop Cluster Test and the closed-loop Cluster Test. In the open-loop mode, test data generation is done off-line, the resulting sequence of input messages is sent to the cluster under test at predetermined points in time, and the responses of the cluster are observed and stored for later evaluation. During closed-

loop testing, the outputs (responses) of the cluster under test are fed to a model of its environment which dynamically computes the reaction of the environment on the cluster's output which in turn is used as the new input to the cluster under test, thus closing the loop.

For the open-loop Cluster Test, the objectives are to check both the functional correctness (semantic integration test) and the timeliness of task (and component) interaction, and to verify cluster behaviour in the presence of anticipated faults (message loss, component loss).

Later, more "higher-level" test objectives can be gradually emphasized. These include extended test runs with highly realistic inputs, evaluating the quality of the (control) algorithms the cluster implements, and observing and evaluating the reaction of the cluster on both hypothesized and unforeseen errors in its environment (robustness test). These tests are usually performed in the closed-loop mode.

In order to actually execute test runs, a special test system has to be constructed that acts like the environment of the cluster under test from this cluster's point of view. The test system must both provide input messages for the cluster under test and accept output messages from this cluster. The test system must also be a real-time system because a cluster's environment is also a real-time system in a very general sense and because the tester must have control over the time of message arrival and also needs information on the time of message output (response time). See section 6 for a description of the test system architecture for open-loop and closed-loop testing.

5.3 Interface Test

Besides the set of clusters, a MARS system also consists of peripheral devices that connect the system to its environment via Interface Buses. Peripheral devices are tested by an Interface Test. Unfortunately, it is neither possible to develop a standard test method nor (consequently) test support for the Interface Test. The reason is the existing variety of peripheral devices, some of which might not even expect or produce digital messages, but analogue signals.

In general, the Interface Test will involve construction of a *test-bed* (hardware and software) for the particular device. The objective of the Interface Test is then to verify that this device correctly converts the MARS-specific into the process (application)-specific data format and vice versa, and that it transmits these data within the specified time requirements.

5.4 System Test

The System Test can begin after the Cluster Test has been completed for all the clusters of the MARS system. The objectives of the System Test are to check that *interactions between clusters* are correct and that the whole system functions according to specification. The System Test is (usually) performed in closed-loop mode. The architecture of the test system is identical to that of the closed-loop Cluster Test (see section 6). The System Test is equivalent to the closed-loop Cluster Test if the system consists of just one cluster.

In addition, user and operator training can be performed in this configuration.

5.5 Field Test

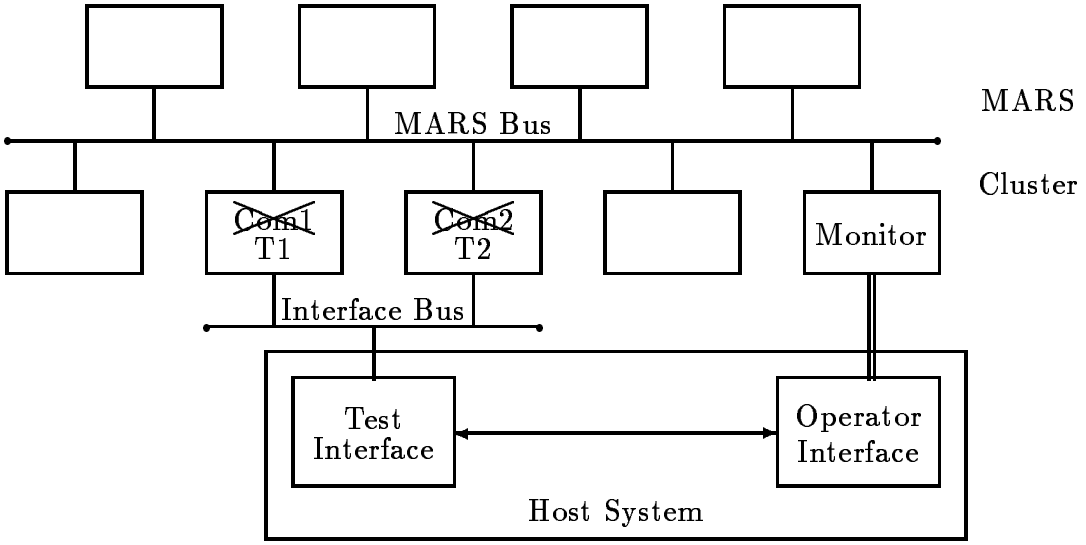
During Field Test, the system is tested within its real environment and with its real peripheral devices. Here, the system is integrated with its interface components. The purpose is to test the system in its operational environment and to conduct the customer acceptance test. User and operator training is also possible in this phase.

6 Test Bed Architectures for Target System Testing

In this section we will describe the architectures of the test systems for target system testing; they can be used both in the Cluster Test and the System Test phases. The principal aim of these architectures is to ensure that the behaviour of the tested cluster (system) (esp. its timing properties) remain unchanged and undisturbed by the act of testing. The following figures illustrate open-loop and closed-loop testing, respectively, for the Cluster Test; however, this is easily adapted for conducting the System Test.

6.1 Open-Loop Testing

Figure 2 depicts the proposed test system architecture for open-loop tests: The interface between the test system and the tested system is the MARS bus. Each interface component is replaced by a test component which does not receive messages from the environment, but derives them from some other source (e.g. reading them from a file), and sends them on the MARS bus as the original interface component would have done it.



Com1, Com2 ... MARS interface components
T1, T2 ... Test components replacing interface components
Monitor ... MARS component monitoring bus traffic

Figure 2: Test System for the open-loop Cluster Test

It is a feature of the MARS architecture that passive components (i.e. components that do not send) can be easily added to a cluster without altering the cluster's time behaviour. Thus, a *monitor component* listens to all messages transmitted over the MARS bus and thereby monitors the responses of the cluster in reaction to the stimuli the test components provide. Via the monitor component, outputs can be transmitted to the host system and be stored for later evaluation.

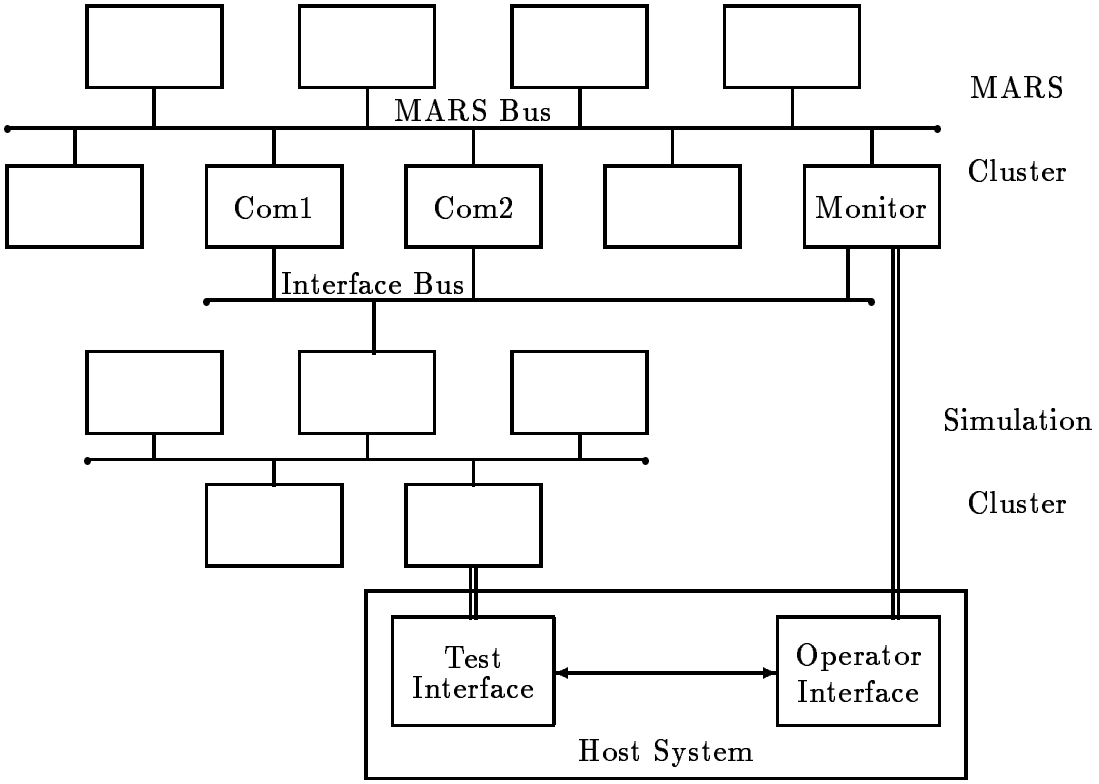
The monitor component can not only hear the final (response) message of a transaction but also all intermediate messages; therefore, the component (and presumably the task) where an error first occurred can be identified. As far as testing the time behaviour is concerned, every monitored message is time-stamped; therefore it is possible to determine response times from the collected results.

Using this scheme, the activities of the cluster under test can be observed very closely. Furthermore, it is possible to very subtly influence the execution of the cluster under test. Invalid input messages, either in the value or the time domain, can easily be simulated and used to evaluate the robustness of the cluster. A test component can send a "garbage" message in one or more chosen TDMA slots, thus sabotaging one or all messages from a given component and thus simulating message or component loss.

6.2 Closed-Loop Testing

The main problem with open-loop testing is that those (parts of) MARS tasks that actually perform external communication cannot be tested with this architecture because they are executed on the interface components. Therefore, the test bed architecture for closed-loop tests has been designed such that it does not have this limitation.

Closed-loop testing requires first construction of a model of the cluster's (system's) environment, then execution of this model under real-time constraints (see the next section 7).



Com1, Com2 ... MARS interface components
Monitor ... MARS component monitoring bus traffic

Figure 3: Test System for the Cluster Test with Environment Simulation

From these considerations, the architecture of the closed-loop test system is easily derived (see figure 3). The environment simulation is executed on a separate cluster, the "simulation cluster", and communicates with the application cluster via the Interface Bus. The application cluster remains totally unchanged (apart from

adding the monitor component). The simulation cluster has its own MARS bus to exchange any messages needed by the simulation algorithm. Thus neither the application cluster nor the simulation cluster is aware of the other's messages, and neither cluster's functional or timing behaviour is influenced by the other's presence.

7 Environment Simulation

As has already been discussed, one needs an "environment simulator" for performing test runs in the closed-loop mode. Its advantages have already been described in section 2. This section describes a suggested method for specifying and implementing an environment simulator (which is of course application-dependent).

To be as realistic as possible, the reaction of the environment has to be available within the same time interval as if the real environment was used, such that the cluster or system under test does not perceive any difference between being driven by its real environment or the simulated environment. Any such difference would invalidate the test runs. The software for simulating the environment is therefore real-time software as well and must be executed under real-time constraints by a real-time system.

While there is probably no single universally applicable way of constructing such a model, it is not unreasonable to demand that the *results* of this modeling process be specified in terms of the usual *MARS transactions*. There are two sources of information which must be considered: a description of the environment to be simulated and a description of the cluster or system that will be driven by the completed environment simulator. This second description will in many cases implicitly include a lot of information about its environment, e.g. about the assumed behaviour of its environment; therefore, these two sources of information will be partly redundant. However, in general one needs both of them.

After the environment simulation has been specified, the existing MARS design tools can then be used for developing the necessary environment simulation software. The role of this software is that of *application-dependent test support* software. Thus, an environment simulator is implemented as a set of MARS tasks and is finally executed during test runs on a separate simulation cluster (see section 6).

An Example

Let us consider an example in which ways a control system specification and that of an environment model are related and how a transformation from the former to the latter can be performed. This example is taken from an experimental application

[17] whose development is currently under way. The aim of this application is to balance a ball on a flat plane such that it follows a predefined course. With the help of a video camera the coordinates of the ball are input every 40 milli-seconds, and the required inclination of the plane is computed and output to a set of servo motors.

Viewed from the highest level of abstraction, the system behaviour can be specified by the following transaction (which is defined by the set of its attributes).

name: Control_Ball

description: This transaction is responsible for controlling the movements of the ball. It reads the observed coordinates of the ball and computes commands for the servo motors to adjust the inclination of the plane, based on the knowledge of the required course the ball should follow. Any known time delays (e.g. due to the age of the input data or due to the inertia of the servo motors) must be taken into account.

stimulus:

name: Coordinates

description: The stimulus contains two numbers, X_coord and Y_coord, i.e. the coordinates of the ball observed by the video camera. The system of coordinates has been chosen in such a way that coordinates are always positive (as long as the ball is on the plane). Thus the transaction "Control_Ball" is triggered whenever there is a valid stimulus message.

stimulus condition: $(0 \leq \text{X_coord} \leq \text{X_MAX})$ AND $(0 \leq \text{Y_coord} \leq \text{Y_MAX})$

response:

name: Inclination

description: The response contains the desired values for the inclination of the plane (one for each dimension). The plane is set to this inclination by the servo motors. This inclination is maintained until a new response message is received.

minimal interval time: 40ms

maximum response time: 40ms

The function of the environment simulation must then be to perform the inverse transformation, and its behaviour can be specified by this transaction:

name: Simulate_Ball

description: This transaction simulates the movements of the ball. It reads the inclination of the plane and predicts the coordinates of the ball at the next observation point, using the appropriate equations of motion.

stimulus:

name: Inclination

description: (see above)

stimulus condition: TRUE

response:

name: Coordinates

description: (See above, except that the values of the coordinates are not the observed position, but the predicted position of the ball.)

minimal interval time: 40ms

maximum response time: 40ms

8 Conclusion

In this paper we have compared time-driven and event-driven real-time systems from the viewpoint of testing, particularly with respect to reproducibility of test runs and to required test effort. Then a test strategy for our real-time system MARS has been presented. We have specifically discussed the problem of running target system tests without altering the timing behaviour of the tested system. Two test bed architectures for achieving this have been described.

The deficiencies of current test strategies for real-time software (see section 2) have been addressed as follows: MARS tasks are programmed in the C programming language, so that low-level tests can be easily conducted on the host system. Whereas techniques for host system testing have not yet been developed, target system testing is supported by the two test system architectures. These include communication links between host and target for improved turn-around times. With the proposed techniques it is not possible to monitor the target system on the source language level, instead debugging on the level of (concurrent) tasks and messages is facilitated.

Finally, we have considered the development and implementation of one particular application-dependent test tool, the environment simulator. Integrating further test support into the Design System is the subject of ongoing work. Possible activities include planning and controlling the test process and the effort (resources) devoted to it, developing test cases, generating test data, checking for test complete-

ness (coverage), executing test cases, updating test cases in the event of requirement changes, re-execution of test cases after modifications, and evaluating test results.

The current status of the MARS project is as follows: An operating system has been designed and implemented that specifically supports the special features of MARS. A prototype Design System has also been implemented. An experimental MARS system is available, and efforts are now underway to construct an experimental application for demonstration purposes (see above). Within the scope of this experiment it is planned to use the techniques described in this paper for testing the software, especially the construction of an environment simulator will be pursued in order to gain practical experience with this proposed technique.

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

References

[1] J. A. Stankovic. Misconceptions about Real-Time Computing. *IEEE Computer*, 21(10):10–19, Oct. 1988.

[2] 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.

[3] G. Chroust. Application Development Project Support (ADPS). *ACM SIGSOFT Software Engineering Notes*, 14(5):83–104, July 1989.

[4] J. A. McDermid, editor. *Integrated Project Support Environments.* Peter Peregrinus Ltd, 1986.

[5] R. L. Glass. Real-Time: The "Lost World" of Software Debugging and Testing. *Communications of the ACM*, 23(5):264–271, May 1980.

[6] H. Gomaa. Software Development of Real-Time Systems. *Communications of the ACM*, 29(7):657–668, July 1986.

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

[8] 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.

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

[10] Ch. Koza. Scheduling of Hard Real-Time Tasks in the Fault-Tolerant Distributed Real-Time System MARS. In *Proc. 4th IEEE Workshop Real-Time Operating Systems*, pages 31–36, July 1987.

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

[12] Ch. Senft. A Computer-Aided Design Environment for Distributed Realtime Systems. In *IEEE Compeuro 88, System Design: Concepts, Methods and Tools*, pages 288–297, Brussels, Belgium, Apr. 1988.

[13] Ch. Senft and R. Zainlinger. A Graphical Design Environment for Distributed Real-Time Systems. In *Proc. 22nd Annual Hawaii International Conference on System Sciences, Vol. II*, pages 871–880, Hawaii, Jan. 1989.

[14] K. C. Tai. On Testing Concurrent Programs. In *Proceedings of COMPSAC*, pages 310–317, Oct. 1985.

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

[16] W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, 1987.

[17] H. Kopetz and 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.