Testing a Distributed Real-Time System — The MARS Approach [1]

Werner Schütz

Institut für Technische Informatik
Technische Universität Wien
Austria

schuetz@vmars.uucp

Research Report Nr. 11/89

July 1989

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

A real-time system is a (computer) system which must meet not only functional requirements, but also time requirements, often also called "deadlines". A hard real-time system is a real-time system where these deadlines definitely have to be met under all anticipated fault and load conditions; any failure to do so would drastically increase the punitive cost function, thus leading to huge financial losses, loss of human life (accidents), or other socially unacceptable consequences (e.g. environmental pollution). Therefore, design and implementation of a hard real-time system have to *guarantee* that all deadlines will always be met. MARS is such a system that strides to achieve this goal.

This report is concerned with the development of a strategy for (system and) software testing within the context of hard real-time systems. There are two main challenges to real-time software testing, one fundamental and one practical: There are not only functional properties that need to be tested, but also timing properties; and real-time systems are usually developed on a (powerful) host system, but eventually executed on a target system, sometimes a custom-designed piece of hardware, providing only the most essential operating system functions.

All testing activities are structured into a set of test phases which are derived from the existing system design phases supported by the prototype MARS Design Environment. Suggestions are made on how to provide automated support for conducting tests in all test phases, while keeping in mind the necessity of integrating these support tools with the MARS Design Environment. The test support tools should also be able to collect and save data to be used later for software reliability modeling. A second line of investigation is studying the problem of real-time simulation and proposing methods to develop, implement, and execute an environment simulator for generating on-line test inputs for system (or sub-system) tests.

Further studies and prototype implementations are planned in order to assess the feasibility of the ideas presented in this report.

Contents

1 Introduction **3**

2 Real-Time Software Testing: A Survey **5**

3 The Properties of MARS and their Impact on Testing **8**

3.1 MARS System Characteristics 8

3.2 Exploring MARS Features for Testing 10

3.3 Objectives for a MARS Test Strategy 13

4 A Test Strategy for MARS **15**

4.1 Overview 15

4.2 Task Test 16

4.3 Cluster Test 17

4.4 Interface Test 20

4.5 System Test 20

4.6 Field Test 21

4.7 Managing the Test Process 21

5 Real-Time Simulation **22**

5.1 Modeling and Simulation 22

5.2 What does "Real-Time Simulation" mean? 23

5.3 Objectives of Using Real-Time Simulation in MARS 25

6 Modeling and Simulation in MARS **27**

6.1 Deriving the Specification for the Simulation Software 27

6.2 Developing the Simulation Software 29
6.3 Executing the Simulation Software 30

7 Supporting the Test Process **32**
7.1 Test Data Generation 32
7.2 Support for Test Execution 34
7.3 Test Result Evaluation 36

8 Conclusion **37**

Chapter 1

Introduction

A *real-time system* can be defined very generally as a (computer) system which is required by its specification to adhere to certain time constraints. 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.

Traditionally, the term "real-time" more often than not has been used in the sense of "as fast as possible, in order to keep up with the passage of real (i.e. external w.r.t. the system) time". This is not surprising since the main deficiency of early computers was their slow speed of execution (besides the lack of main memory). Therefore the main problem of early real-time systems was to design time efficient algorithms and implement them in a time efficient way. This approach resulted in event-driven systems [Kop89] where every external event initiates some activities.

Due to the enormous advances in execution speed and memory size during the last decade, the goal became to construct increasingly more complex and more safety critical real-time systems. It was then realized that speed of execution is not the primary concern, but to be able to *guarantee* adherence to all real-time requirements is. Execution speed then may or may not be a secondary concern, depending on the complexity of algorithms and the exact nature of time constraints. Thus, a new usage of "real-time" emerged, meaning "in synchronization with external time, exactly in time".

Event-driven systems are not suitable to guarantee real-time requirements because of the potential overload caused when more external events occur than were anticipated by the designers, the main problem being spurious events triggered by internal or environmental fault conditions. Event-driven systems usually provide facilities for a human operator to intervene in situations like this, but if the required response time becomes too short for a human to react this approach is no longer

feasible.

In the course of our work on the *MA*intainable *R*eal-Time *S*ystem (*MARS*) [Kop88,Kop89] we have found it very advantageous to clearly distinguish between these two notions of "real-time". This has lead to the following definitions:

A *hard real-time requirement* imposes a strict deadline for execution onto a certain activity of a real-time system; this deadline must be guaranteed to be met even under anticipated fault conditions and/or (specified) peak load. A *soft real-time requirement* on the other hand does not impose such a strict deadline; it either specifies no deadline at all if it is a non real-time requirement, or it is admissible not to meet the specified deadline occasionally or in some instances. This usually results in some type of cost, e.g. the user of the system becoming impatient, but this cost has been judged to be acceptable.

Now, one can further define a *soft real-time system* (also often called "on-line system") as a system whose specification does not impose any hard real-time requirements on it. Correspondingly, a *hard real-time system* is a system at least a part of which is subject to hard real-time requirements.

In MARS, emphasis is placed on the hard real-time aspects; therefore, MARS is essentially a hard real-time system that is designed to guarantee strict deadlines. In the remainder of this report the term "real-time" will be used in the sense of "hard real-time", except where explicitly noted otherwise.

This report concentrates on the problem of testing, verifying, and validating real-time software, and offers some solutions in the context of the MARS system. Chapter 2 gives an overview of current practice in real-time software testing, while chapter 3 summarizes the most important principles of MARS and how the software test process can benefit from them. Chapter 4 then proposes some system architectures that support test case execution and evaluation, as well as how they can be included in a general test strategy. It is a widely accepted practice to use an environment simulator to test a real-time system; therefore, as a "subtopic", chapters 5 and 6 deal with real-time simulation and the problem of how to construct and use an environment simulator, respectively. Finally, chapter 7 discusses possibilities of integrating all this into the existing MARS Design Environment [Sen88b].

Chapter 2

Real-Time Software Testing: A Survey

For real-time software development, usually two types of computer systems are used [Gla80,Gom86,DeM87]: A *host system*, a conventional computer system, is employed to construct the real-time software. Besides a standard operating system and any desired tools for software development and testing, it also includes a cross compiler or a cross assembler to generate code for the target system, and often also an instruction level simulator to simulate the execution of the target system on the host. This *target system* is the system where the real-time software is eventually executed on. Since the target is often subject to stringent space, weight, and/or cost requirements (e.g. embedded system, satellite control system, ...), the target is specifically tailored to executing the real-time software. Thus, it provides only the most essential operating system functions in order to be time efficient, and there are no software development tools available. For the same reason, special purpose or even custom-made hardware is often part of the target system.

Therefore, the typical phases of real-time software testing can be summarized as follows [Gla80]:

1. *Testing on the host system* is used to first eliminate errors from the source code, then to unit test all the software modules. Only functional tests can be conducted, since the results of any performance or timing tests would not be significant for the target system.

2. *Testing on the host system with the help of an instruction level simulator* allows performing further tests on the host. An instruction level simulator is a software package that emulates the target system's machine instructions. By using such a simulator it is possible to also test the functionality of program pieces written in target assembler language.

3. *Testing on the target system* involves loading the real-time operating system and all the application software onto the target system for final check-out. At this stage it is possible to also test the performance and timing characteristics of the software. It is very common to use an "environment simulator" (see also [Gom86] and [DeM87]) for producing inputs to the target system which are relevant for certain scenarios and for recording the reactions of the target system. However, this is often the only support available for target system testing.

The most important drawback of this test strategy is the virtual non-existence of any test support on the target system, and even for the existing support (instruction level simulator and environment simulator) 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 [Gla80] 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 recommendations have been followed in a recently proposed software development strategy for real-time systems [Gom86]. Again it is recommended that as much software testing as possible be done on the host system. Consequently, all unit tests are performed on the host system, and emulating the target system's operating system primitives enables the host system to support some limited integration testing, whose purpose is "to test the logical correctness of the interfaces between tasks" [Gom86]. Testing on the host system is automated by the use of so-called "test scripts". Each test script contains a set of (user) inputs that defines a specific test case, and it can be stored and automatically executed. In a second test phase, the application software is moved to the target system, where integration testing is completed, and system testing, performance, and time behavior testing is performed. Again, an environment simulator is used that runs on an external system, so as not to interfere with the software under test in the target system. Testing on the target system can be automated by having the environment simulator execute test scripts.

Very recently, a completely different approach to real-time system development has been suggested [Tar88]. It is argued there that target computers are gradually becoming powerful enough to be able to support both real-time software development through the use of suitable (probably simplified) Software Engineering Environments and real-time software execution. This would remove some disadvantages of the host/target approach, most notably the need of design and programming personnel to be knowledgeable of two different machine architectures, and the need to develop special targeting tools and target simulators (cf. "instruction level simulator" above). On the other hand, an operating system that supports both the execution of the real-time application and the development environment is needed or is at least highly desirable. From the MARS standpoint, we do not consider this a promising or useful approach (see section 3.1, page 10).

Chapter 3

The Properties of MARS and their Impact on Testing

3.1 MARS System Characteristics

As already mentioned in the Introduction, MARS 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, synchronous) system where all activities are triggered by the passage of (real) time.

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, either via the MARS bus or internally (within the same component).

Besides the physical realization of a MARS system (which is the ultimate product of a system development effort), there is also a MARS Design Environment [Sen88a,Sen87] that supports system (and software) development and evaluation. Its most important characteristic is that it supports the specification and refinement of time information throughout the life cycle, from requirements definition to task implementation.

Within this Design Environment, system functions are specified semi-formally as

a set of *transactions* each of which consists of a set of attributes. The most important attributes are the "stimulus" describing when or how a transaction is activated, the "response" describing the required reaction of the transaction after it has been activated, an informal description of the transaction's functionality (i.e. how to compute the response given the stimulus), and various time attributes, such as the maximum time allowed between stimulus and response, or the maximum frequency with which the stimulus may occur. During design, these transactions are iteratively refined into subtransactions until they are simple enough to be implemented as MARS tasks and to be executed on a single component.

In order to be able to guarantee that all deadlines are met, several techniques are used in MARS: First of all, 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 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 taken into account concurrently [Koz87]. Furthermore, it is required that access to the MARS bus is deterministic, so as to avoid queueing delays. A Time-Division Multiple Access protocol is used for this purpose; this guarantees that at any point in time exactly one component of a cluster is granted send-access to the bus. Finally, it is a prerequisite for the above 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 [Kop87].

As of July 1989, a new integrated circuit board for MARS components is under development which will include at least three Ethernet interfaces, two for the redundant MARS bus and one or two for the (redundant?) Interface Bus. As mentioned above, the MARS bus is used for exchanging messages within a single cluster. The Interface Bus will enable a cluster to communicate with its environment, either another MARS cluster, or the controlled object (the physical process) via appropriate peripheral devices. It should be noted that in general not every component of a cluster will be connected to the Interface Bus, and that a particular cluster might use more than one Interface Bus. As an example, two components of a cluster might be responsible for communicating with a neighboring cluster using one Interface Bus, while another component is responsible for controlling some peripheral devices, thus communicating with the physical process using a second, independent, Interface Bus. It is planned to use standardized MARS messages on the Interface Bus as well; a MARS task need then not be aware of the nature (type) of its communication partner. However, the variety of available peripheral devices requires to handle this design decision somewhat flexibly.

More information on MARS can be found in [Kop85,Kop88,Kop89].

As an aside, it should be noted here that MARS is clearly an example of the "traditional" host/target approach. Despite some arguments brought forth in [Tar88] (see also page 7), we consider it very unlikely that this will change any time in the near future. While it is certainly true that microprocessors are becoming increasingly powerful, there are some reasons why we do not consider the host/target approach to be obsolete:

1. Usually (also in the case of MARS), the functions demanded from the operating systems of the target and the host systems, respectively, are quite different: The host system needs to support time-sharing, multiple users, as well as a wealth of other activities besides real-time software development. The target system on the other hand must support the timely execution of hard real-time tasks. It is specifically designed to be small and efficient, such that as much of the resources as possible is available for the application tasks.

   Putting all these functions into one operating system would not only increase its functionality but also, and more importantly, greatly increase its *complexity*. As a consequence, it would be practically impossible to gain any level of confidence in, or even guarantee, the correct behavior of the operating system. Since the application software depends on the operating system for execution, this directly conflicts with our stated goal of guaranteeing certain aspects of application behavior.

2. In our experience, it is not possible to use a "*simple* Software Engineering Environment" for real-time system development, as suggested in [Tar88]. On the contrary, the necessity to handle real-time requirements consistently throughout the life cycle adds further complexity to the Design Environment, in comparison with traditional SEE's. In addition, user-friendly interfaces, such as terminals with graphic capabilities, consume even more resources which few if any current microprocessors are able to provide.

3. The target machines will eventually be embedded in some physical environment that they are intended to control, which is possibly not even accessible for humans. Thus all development information must at least be stored on a separate machine; therefore the host system is still required.

3.2 Exploring MARS Features for Testing

In this section we will briefly discuss how the properties of MARS mentioned in section 3.1 relate to the testing of a MARS system. We expect the following benefits as a result of system characteristics:

- It will be easier to check if all system requirements have been tested since all these requirements are available from the Design Environment; even some computer-aided checks seem possible.

- A common problem in testing concurrent programs is reproducibility. The most common approach to achieve reproducibility is *serialization* [Tai85,Wei88]. A serialization is generated by a *random scheduler* that "merges" the bodies of all tasks of the concurrent program into a single sequential program [Wei88]. 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, we do not have this problem. Due to off-line scheduling, the sequence of tasks executing on the same component and their interleaving is pre-determined. 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 [Kop87] and all synchronization constructs (in MARS: message passing) are scheduled to occur at time intervals larger than maximum clock inaccuracy. Therefore, the sequence of synchronization constructs is unique, and since all shared data are only accessed within synchronization constructs (message passing) all possible serializations are equivalent [Wei88], 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).

- Since MARS is time-driven, all occurring stimuli are detected only at the end of some (finite-length) time interval. The order in which these stimuli arrived is of no importance. If we assume that a maximum of n stimuli (events) can arrive (occur) during a given time interval, we can compute the number of possible control paths that exist due to the potential presence or absence of each of these n stimuli, $CP_t(n)$ and $CP_e(n)$, respectively. We also assume that there is a fixed order in which the stimuli detected by the time-driven system at the end of this time interval are serviced. Then, for time-driven systems, $CP_t(n) = 2^n = \sum_{k=0}^{n} \binom{n}{k}$ and $CP_e(n) = \sum_{k=0}^{n} \binom{n}{k} k!$ for event-driven systems. $CP_e(n)$ is actually 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 3.1 compares $CP_t(n)$ and $CP_e(n)$ for some (small) values of n. Note especially the logarithmic scale of the Y-axis!

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.

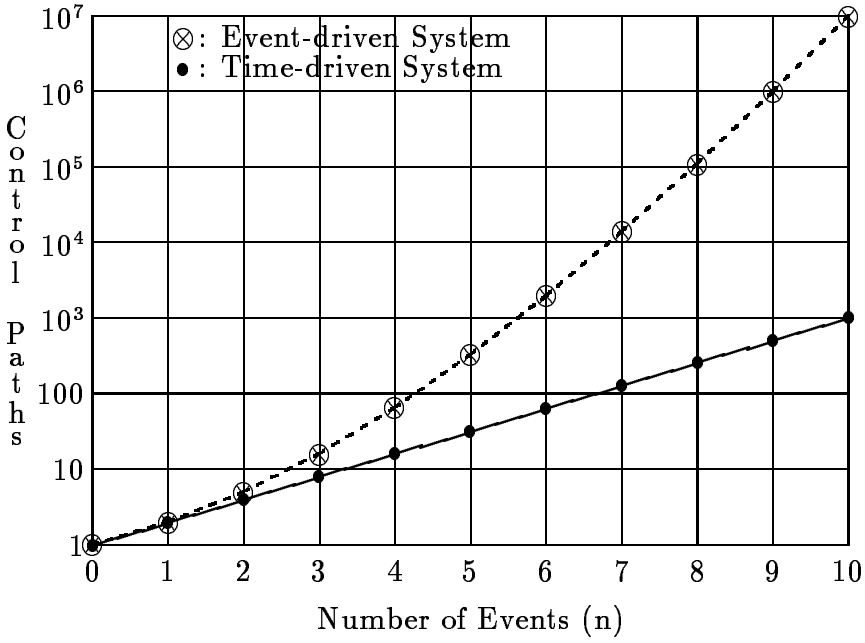


Figure 3.1: Test Effort for Time-Driven vs. Event-Driven Systems

3.3 Objectives for a MARS Test Strategy

In accordance with the physical and the software (transaction) structure of a MARS system, there are (at least) three phases of software test: task, cluster, and system test. The objectives of testing differ with the phase in which it is conducted.

At the task level, the most important objectives are threefold: First, functional testing of each individual task is required. The need for functional testing should be obvious, and furthermore it is relatively easy to perform because it can be done independently for each task and because each task in itself is relatively simple.

Second, a preliminary interface test can be applied by checking the message handling of each task and comparing it with information stored in the design environment's database. Here, the objectives are to verify the syntactic correctness of all the messages a task sends and receives, and to make sure that each task reads all the messages it is supposed to receive as well as generates all the messages it is supposed to send.

Third, all tasks that execute on the same component and communicate with each other by internal messages should be tested together, thus providing a preliminary integration test. A cluster schedule is required in order to determine the correct order of task execution if this order cannot be determined uniquely from the data flow. The objective is to verify that message passing between these tasks is (functionally) correct. This cannot be easily achieved in a later test phase on the target system because internal messages are not transmitted on the MARS bus and thus cannot be observed.

At the cluster and system level, it is first necessary to check the implementation for completeness and to see if the interaction between tasks is valid, i.e. both with respect to the design and its decomposition (functional correctness) and with respect to timing properties (real execution time of tasks, response times of transactions, etc.). All this should already be guaranteed by design, and therefore the objective of testing is in this phase to double-check if the (functional and timing) requirements derived during the design phases have indeed been met by the implementation. In addition, the measured task execution times can be used to assess the quality of the computed maximum execution time [Pus90].

Afterwards, it is expected, that testing rather quickly proceeds to "higher-level" objectives. These include determining the overall quality of the control algorithms, executing the system with real-life inputs, investigating system behavior under stressful environmental conditions or fault situations, and demonstrating peak-load performance and the handling of other critical or exceptional situations.

In addition, it is important to keep in mind that it is increasingly recognized [Gel88,Las89] that testing involves more than just executing test cases; it also comprises organizational, planning, analysis, design, implementation, and maintenance

activities. It is a complex task that should be carried out in parallel to software development and that should cover the whole life cycle. (This is, however, hardly a new idea [Adr82,How82].) Therefore any Software Engineering Environment must support activities like planning and controlling the test process and the effort (resources) devoted to it, developing test cases, checking for test completeness (coverage), executing test cases, updating test cases in the event of requirement changes, re-execution of test cases after modifications, and evaluating test results.

Chapter 4

A Test Strategy for MARS

4.1 Overview

This chapter proposes a general test strategy for real-time software under the MARS system. The strategy consists of five test phases, which will be described in the order they are most likely to be performed, assuming that testing proceeds bottom-up, i.e. low-level objects are tested before higher-level objects. If necessary (e.g. error detection and correction), it is of course possible to go back to a previous test phase or a design activity; this will not be explicitly mentioned in the following discussion. The test phases that will be described in the subsequent sections are:

- Task Test
- Cluster Test
- Interface Test
- System Test
- Field Test

Although it seems that one level of the hardware hierarchy, namely the component, is not covered by an associated test phase, from the software standpoint, all objects are included: a group of tasks together forms a (cluster) transaction with associated external (w.r.t. the cluster) stimulus and response. A component merely executes a collection of tasks which may or may not exchange information and which may or may not belong to the same cluster transaction. Furthermore, the Task Test has already removed a large number of errors, the maximum execution time of each task has been computed, and a valid cluster schedule exists which takes these maximum execution times into account. Considering this, it is not expected

that a dedicated Component Test would yield any significant results that cannot be obtained otherwise. Therefore it has been decided that testing can immediately proceed to the next-higher architecture level. This also significantly reduces the total (i.e. seen over the whole life cycle) effort required for test data generation, and there is no need to build yet another test-bed. In addition, it is much more useful to test a complete transaction than an "arbitrary" set of tasks. This approach also simplifies test data generation.

4.2 Task Test

During Task Test, every MARS task is tested individually. It 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.

According to the objectives of Task Testing (see above), the emphasis in this phase is placed on functional testing (see for instance [How86,How87]). Special care should be taken to exploit the unique properties of MARS tasks. For example, since both the number of loop iterations is bounded by some finite known number and recursions are prohibited, the number of potential control paths through a given MARS task is finite. This allows the implementation of complete path testing.

As a second step, a preliminary interface test can be performed. Since all communication between tasks is done by message transfer, a tool is conceivable that statically compares information from the design database and information derived from the task's source code, in order to check that first all messages that the task is required to process are indeed read, second all messages that the task must send are produced by the task, and third all message usage is syntactically correct.

Lastly, in a preliminary integration test, all tasks that are allocated to the same component and communicate with each other by internal MARS messages are tested as a unit, for the reasons mentioned above.

In this way, all the objectives for task testing as mentioned above can be fulfilled. For executing MARS tasks on the host system, it is necessary to build a special library for MARS system calls, most importantly for the message passing primitives. Most of these must have different functionality than the original MARS system calls, e.g. reading (writing) from (to) a file or a pipe instead of the MARS bus. In addition, other tools seem suited to aid the programmer/tester, such as a tool to support test case generation and coverage analysis (to see if all possible control paths have been executed yet).

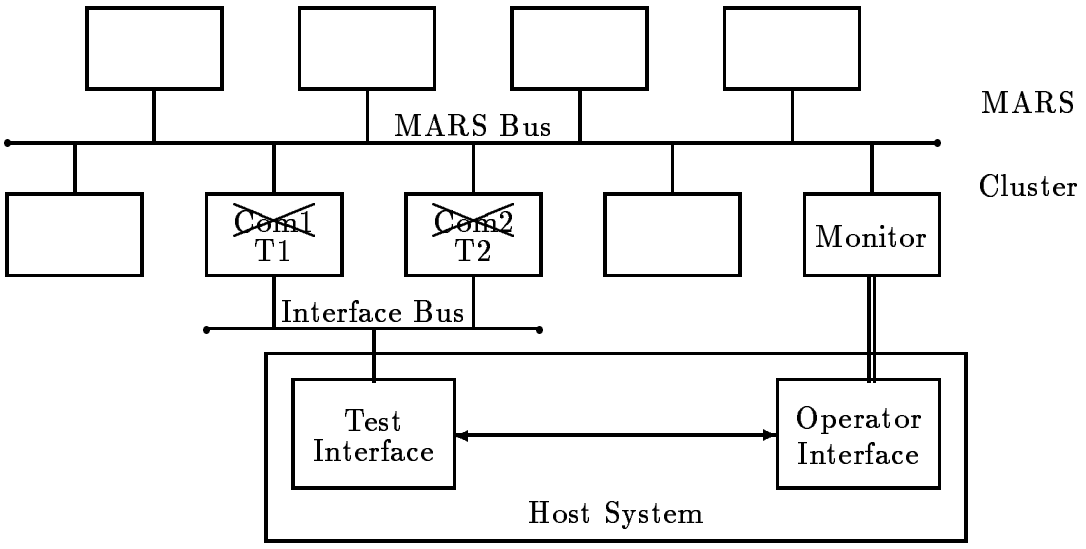
4.3 Cluster Test

There will be 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. We postpone a discussion of the closed-loop Cluster Test (see section 6.3).

The Cluster Test will be conducted on the target system. For the open-loop Cluster Test, the object under test is a MARS cluster from the physical point of view and any subset (except the empty one) of the set of cluster transactions to be implemented on this cluster from the software standpoint. The objectives are to check both the functional correctness (semantic integration test) and the timeliness of task (and component) interaction, and to verify cluster behavior in the presence of anticipated faults (message loss, component loss). The Cluster Test is intended to be employed as early in the system life cycle as possible. It is required that a *cluster schedule* has been computed and that an allocation of transactions and tasks to the components of the cluster has been done. Then, after all the tasks of a given transaction have been programmed and task-tested, this transaction can be down-loaded into the MARS cluster and Cluster Test can begin with respect to this transaction.

As other transactions become available, they can readily be added to the cluster under test until the entire cluster becomes operational. Then, more "higher-level" test objectives can be gradually emphasized. These include regarding the cluster as a black box in order to verify its reaction on and its behavior towards its environment, using 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).

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. Since a MARS cluster communicates with its environment (here: the interface components) by the exchange of messages, 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).



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

Figure 4.1: Test System for the open-loop Cluster Test

Figure 4.1 depicts the proposed test system architecture: The interface between the test system and the tested system is the MARS bus. This choice has been made in order to facilitate fault injection into the cluster by the test components (see below). Each component that is responsible for communication with the cluster's environment ("communication component") is replaced by a test component which does not receive external messages from the outside, but derives them from some other source (e.g. reading them from a file). In its TDMA slot it sends the same messages the original communication component would have sent.

In addition, all test components can listen to all messages transmitted over the MARS bus and are thus able to monitor the responses of the cluster in reaction to the stimuli they provide. In order to avoid duplication of effort, this function could be provided by a single component, the *monitor component*. An operator interface is in development that will have a similar functionality, and it could be used for this purpose. Of course, if resource utilization is not too heavy, all test components and the monitor component can be merged into a single "test driver/monitor" component. This would require that the MARS Operating System Kernel makes available a "test mode" where privileged components are allowed to send messages in more than one TDMA slot.

The monitor component can not only hear the final (response) message of a transaction but also all intermediate messages; therefore, the component (and pre-

sumably the task) where an error first occurred can be identified. As far as testing the time behavior is concerned, every monitored message is time-stamped; therefore it is possible to determine response times from the collected results. It is planned to introduce so-called "component state messages", by which each component of a cluster periodically broadcasts its internal state, for other components or the operator to use. These "component state messages" can be used to report task execution times, so that the real execution time of each task invocation is also available for the tester.

The basic advantage of this scheme is that it is possible to observe very closely the activities of the cluster under test, both by monitoring the traffic on the MARS bus and by using the "component state messages". 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, could 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. Again, this would require that the MARS Operating System Kernel provides a "test mode" allowing privileged components to send messages in any TDMA slot. There is also a system message directing a component to enter a given internal state; this feature can be used to set initial component states for a test run or to simulate the occurrence of an error during test execution. Lastly, test components are "automatically" synchronized with the cluster under test, thus avoiding any problems that may result from inaccurate synchronization between the test system and the tested system.

The main problem with the Cluster Test is that those (parts of) MARS tasks that actually perform external communication cannot be tested with this test system architecture. This is the reason why the test system architecture for the closed-loop Cluster Test will be slightly different (see section 6.3).

Another problem are MARS tasks that do not handle any external communication, but are assigned to and are executed on an interface component. The problem here is that these tasks belong to the interface component when viewed from the physical structure of a MARS system, but belong to the "interior" of the cluster when viewed from the software structure. It is not practical to test these tasks as part of the Interface Test since they are unrelated to the peripheral devices. Executing such tasks on the test components is also not a general solution because these tasks' timing requirements might conflict with those of the test system software. As a first solution to the problem, we are proposing a rule that only tasks which are actually responsible for external communication may be allocated to interface components.

4.4 Interface Test

Besides the set of clusters, a MARS system also consists of one or more peripheral devices that are connected to the clusters by the Interface Bus(es) and enable communication with the environment of the MARS system, i.e. the physical process. Since the interface between a cluster and its peripheral devices are MARS messages on the Interface Bus, some protocol and/or data format conversions will in general be necessary.

Let us call the set of all peripheral devices that are connected to a single Interface Bus a *device group*. Each device group is tested by an Interface Test independently from other device groups. 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 analog signals. In general, the Interface Test will involve construction of a *test-bed* (hardware and software) for the particular device group. The objective of the Interface Test is then to verify that this device group 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.

It may be possible to devise standard Interface Test methods for particular classes of applications or for particular classes of peripheral devices. However, within the scope of this report, we will neither try to develop an Interface Test method, nor identify any such classes.

4.5 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 whole system is tested with the help of a test-bed that simulates the system's environment, i.e. it is tested in closed-loop mode (usually). The architecture of the test-bed is identical to that of the closed-loop Cluster Test (see section 6.3). 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.

4.6 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 production environment and to conduct the customer acceptance test. User and operator training is also possible in this phase.

This configuration can also be used during software maintenance, for installing a new or upgraded release of the application software. The new software version is installed and executed on the redundant components of the system, but with their output disabled. Normal operation is continued with the old software version executing on the primary components. However, fault tolerance will be limited during this period. After determining that the new software version performs as expected, output of the redundant components is enabled and the new software version is also installed on the primary components. Thus, in most cases system operation will not be interrupted by the upgrade (neglecting faults occurring during this period).

4.7 Managing the Test Process

It should be noted that the test strategy described in the previous sections is the most comprehensive of all possible variations. There will be no requirement in the system development life-cycle that *all* these test phases are *always* mandatory. Instead, it is the responsibility of the project management to select and adapt the needed test phases and methods according to application requirements, and (more importantly!) to choose the effort and the amount of resources to be put into each selected method. This selection may be done independently for each cluster of the system.

It is also important to keep in mind that achieving a good *design* must be the first goal: An overly complex cluster should be split into multiple clusters, thereby possibly reducing the total test effort. By careful and conscious application of these principles it should be possible to achieve a level of test effort that is not only technically sound, but also economically viable.

These considerations lead to the conclusion that it is necessary to integrate testing and test management with the design activities and the project management. This issue is discussed further in chapter 7.

Chapter 5

Real-Time Simulation

Especially in the development of real-time systems, it is common practice to use a so-called "environment simulator" [Gla80,DeM87] for the purpose of driving a test run. The main advantages of this scheme are that it enables more realistic test runs, possibly by using real-life data, and that it eliminates the need to generate large amounts of input data, thus enabling longer and more extensive test runs. For closed-loop testing 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. 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.

Thus, 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. In the following two chapters we will therefore take a closer look at conducting tests in the closed-loop mode. Chapter 5 gives an overview on the state-of-the-art of real-time simulation, while chapter 6 tries to apply these concepts within the context of the MARS system design principles.

5.1 Modeling and Simulation

Simulation can be very generally defined as a method to investigate the properties or the behavior of a real (physical) system by building a second system which is a *model* of the original system [Sch85]. Experimentation is then performed with the model, and results can be applied to the original system. A similar definition is provided by the "VDI-Richtlinie 3633" [Ber87] where simulation is the construction of a model of a dynamic process in order to obtain results applicable to reality.

In the context of Computer Science, (computer) simulation is also more narrowly defined as "experiments run in a computer" [Dud85] or "experimentation

with computer models" [Van83]. This is a special case of the general definition, implying that the model system is implemented as a computer program and results are obtained by executing this program.

From the above definitions it is apparent that modeling and simulation are closely related. A model is a representation or a mapping of one system by (to) another [Ame85]. Constructing a model that can be used to perform the desired experiments usually proceeds in two steps [Sch82,Van83]: First, the real (physical) system is analyzed, relevant system properties are identified, and unimportant system properties are abstracted from. The decision which system properties are important and which are not is guided by the intentions of the modeler, i.e. the purpose of the model, or the problems one wants to investigate with the model. The result of this step is a conceptual or abstract model of the system which is formulated in some convenient notation, either formally (e.g. graphs, petri nets, queueing theory, etc.) or informally.

The second step is to "implement" the abstract model in order to get a model with which the desired experiments or measurements can be performed. Again depending on the purpose of the model and on practicality, several types of models are possible: analytical (mathematical) models (e.g. a set of equations), physical models (e.g. flight simulators), or computer models. Manipulating an analytical model by analytical methods, either exact or approximated, is not considered simulation, instead "the model is solved". In all other cases, experimenting with the model is termed "simulation".

To be able to construct computer models, it is necessary to describe the possible state transitions, along with the conditions under which they occur. The computer is then used to replay the sequence of state transition a given model (and its corresponding system) is subjected to. Computer models can also be regarded as special types of physical models [Sch85], with the computer being the physical system.

5.2 What does "Real-Time Simulation" mean?

The term "real-time simulation" appeared in the literature at least as early as 1966 [Sag66]. From the beginning, this term has been meant to denote a computer simulation where simulation time has to be kept synchronous to real (external) time or, more generally, to the system time of the original system being simulated [Hau84]. This is always necessary when a simulation does not model a closed system, but when there is instead some interaction of the simulation with some external system, e.g. parts of the original system not being modeled.

This is also consistent with our view of model construction above: When the computer model is intended to be used as part of a larger system, it is not per-

missible to abstract model time from real or system time; instead, these have to be synchronous. In traditional simulation, it is often the case that time is one of the first properties to be abstracted from. There may either not even be a notion of simulation time (static models), or simulation time may proceed at an arbitrary rate, faster than real time when the model is used for projections and forecasts, slower than real time when the underlying model is very complex (e.g. simulating elementary particle behavior), or it may even proceed at different rates at different points in (simulation) time (e.g. variable step integration techniques, discrete event simulation). It should also be noted that it is often *required* that simulation time is faster than real time. For example, simulation of operator command effects is often a feature of large real-time systems controlling some production process or plant. In this case, clearly, simulation time has to be faster than real time in order to achieve the desired benefits.

While the meaning of "real-time simulation" has always been (at least informally) well understood, the main problem during the early years, however, was that computers were not powerful enough to allow the real-time execution of simulation programs of any realistic complexity. Thus the first necessity was to reduce program run time. Research first focused on algorithmic issues, most notably the design of fast numeric integration algorithms since solving differential equations is almost always necessary for simulating dynamic processes. It was quickly recognized that implicit integration methods are unsuitable for real-time simulation because the number of iterations required to achieve the desired accuracy is often only known at run-time [Sag66]. Explicit integration methods are clearly preferable, and research on the derivation of explicit integration operators, their costs in terms of execution time and accuracy, their domain of applicability, etc. continues until this day [Gea77,Har85,Kel88].

Another approach to speed up execution time was the use of analog computers, which led to the construction of (special-purpose) hybrid computers. Those parts of the simulation program that could not be implemented efficiently enough on a digital computer were transferred to the analog machine. Examples include the simulation of electric power generation equipment [Bry76], car and jet engines [Led76], and flight simulators [Tom76].

During the past few years, digital computers became more powerful in terms of execution speed, and thus were increasingly used also for simulation since they provide more accurate and precise results in comparison with analog computers [Ame85]. However, users became increasingly ambitious, too, and wanted to run ever more complex simulation programs. Thus the drive to decrease program execution time continued, and the solution was to introduce multi-processor systems which enable parallel processing. Examples include general-purpose simulation systems [Hel82,Fey83] and flight simulators [Kod85].

Due to this emphasis on the need to speed up execution time, "real-time simu-

lation" often seems to be interpreted that "execution needs to be as fast as possible in order to have a chance to keep up with real time". This is sometimes explicitly acknowledged [Tro83], but often the term "real-time simulation" is simply left undefined [Fey83,Cho84]. It also seems that the goal is sometimes just to speed up execution to such a degree that interactive simulation becomes possible – a gross misuse of the term "real-time". No method for deriving and specifying real-time requirements, and to guarantee that these requirements are met during execution, has been described in the literature.

5.3 Objectives of Using Real-Time Simulation in MARS

It has already been discussed that the prime motivation for investigating real-time simulation is its use for testing purposes in the MARS system. The motivation is that it is generally both cheaper and safer to use a simulated environment than the real environment, especially in a phase where confidence in the real-time software is still low. In addition, it is often much easier to obtain system responses and other intermediate test results from a simulated environment than from the real environment. Thus, simulation provides the only ability to test system behavior 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.

It is necessary to build a model of the environment of the cluster or the system that needs to be tested. This model is therefore subjected to a set of functional and timing requirements which are determined by and derived from the original environment being modeled, 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.

To use real-time simulation for test purposes has been suggested before: [Led76] describes a real-time simulation of a diesel engine which is used to facilitate testing of controllers designed for this engine. Presumably, these controllers consist of a combination of hardware and software. Thus, this is an example where a physical system is tested by a (software) simulation of its environment. [Hau84] proposes a scheme very similar to the MARS scheme that uses a model of a process to test the control system for this process. This model must of course be simulated in real time. Here, a software system is tested by a software simulation of its environment.

However, none of these two examples take into account how to achieve real-time execution of their models, or how to guarantee that all timing requirements are met. It is the second objective of the MARS approach to provide an integrated, structured

method to derive the real-time requirements for the simulated environment (model construction) and to implement and execute the developed model such that these requirements are met (model implementation and test execution).

A second purpose for using real-time simulation is *user or operator training.* Flight simulators are probably the most well-known example for this type of systems; they are built exclusively for training purposes. In MARS we envision the approach that the same simulation software and the same (hardware) architecture (cf. "Cluster Test") than for testing can be used for training purposes.

Run-time error detection is yet another application of real-time simulation. [Stu82] proposes a scheme where system components of a process control system are simulated by a software component and error detection is performed by a comparison of the results of the simulation and the original system component. Again, however, there is no information of how the real-time execution of the simulation software is enforced or guaranteed, if at all. In MARS, a similar approach seems possible, although some special measures probably need to be taken.

Finally, it is often the case that knowledge of some internal or intermediate process variables is required for process control, especially for optimizing process control. These process variables may or may not be measured readily. In the case that they are difficult to measure, software simulation of the process is often used to obtain the necessary values [Kel87,Kel88]. This means that simulation is used as a substitute for measurement; required input parameters are computed instead of being measured. From the MARS standpoint, however, this type of simulation is part of the application software and must be designed and implemented in conjunction with application development. It will not be discussed further in this report, although software reuse might be possible.

Of course, there is also room for *non-real-time simulation* within the design and implementation of a MARS system. For example, a process simulation to enable the operator to predict future system states and to predict the effects of operator commands might be included as an aid for optimal process control. This simulation has to be executed faster than real time, typically by a factor of 30 to 50 [Kel85]. Again, this is part of the application problem and will not be considered further here. As a second example, simulation can also be used in the early phases of system development, such as for developing an optimal control algorithm. In this case, too, the user is interested that simulation runs are executed as fast as possible. Any suitable simulation package can be included in the MARS Design Environment and be used as a tool of the Requirements Definition Phase [Sen88a]. This aspect, however, is also beyond the scope of this report.

Chapter 6

Modeling and Simulation in MARS

This chapter is concerned with some suggestions on how to derive the specification for the simulation software (i.e. modeling), how to develop the simulation software, and how to finally execute the simulation software (i.e. doing the simulation). The focus is on suggesting a way to integrate the simulation software life-cycle into the existing MARS Design Environment, so that simulator construction can proceed in synchrony with overall system development. The general approach is to reuse existing (design and management) tools as much as appropriate.

6.1 Deriving the Specification for the Simulation Software

Before one can actually begin to write the simulation software, one must somehow develop a model of the object to be simulated (the environment, in this case). While there is probably no single universally applicable way of deriving 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*. The reason for this is the desire to use the MARS design tools also for developing the necessary test beds, consisting of environment simulation software. As mentioned above, the simulation software is also real-time software if it is intended to be used for test purposes; therefore, the same criteria with respect to software development apply than for the application software: One needs a Design Environment which supports the specification and refinement of time attributes during design, and which guarantees that all timing requirements will be met during execution. The MARS Design Environment meets these requirements, and is, of course, readily available for this purpose.

Thus the specification of an environment simulator will consists of a set of transactions. There are two sources of information which must be considered during modeling: First, a description of the environment to be simulated is needed; this environment is either (a part of) the physical machine (the process) that is going to be controlled by the MARS application, or some other cluster of the MARS application, or a mixture of both. This depends on whether the object whose environment is simulated is a single MARS cluster or the complete MARS system. Second, a description of the object that will be driven by this environment simulator is also needed. This second description will in many case implicitly include a lot of information about its environment, e.g. about the assumed behavior of its environment; therefore, these two sources of information will be partly redundant. However, in general one needs both of them.

In theory, a description of the environment that is to be simulated alone is sufficient to derive a simulation software specification. Such an approach would be more appealing since development of the environment simulator and of the application software could proceed independently from each other. The simulation software has to mimic the functionality of the real environment, but it also has to adhere to its timing properties. Problems will arise when a function of the environment is implemented in hardware (e.g. in the physical machine or process), and thus is very fast. In some cases, the time intervals involved may be so short that the function is performed essentially "instantly" and that no possible software implementation is able to satisfy the associated time requirements. Luckily, there is a way out: It is not really necessary to complete such a function within its "real" time, but one needs only to make sure that the function is completed before the application software requires its results. Thus it is possible to check at what period the application software needs the results of each such function and to derive the time requirements for the application software from this information. This information is contained in the specification of the object under test, and that is the reason why both descriptions mentioned above are necessary. The application software (which runs in the object under test) will still not perceive any difference between its real and its simulated environment, despite this relaxation of time properties. The drawback, however, is that the application software and the simulation software are mutually dependent on each other, which means that changes in the application software (especially in its time requirements) may have to be reflected in the simulation software.

Integration into the existing MARS Design Strategy is handled by the use of contracts. As soon as an object which will be tested with the help of an environment simulation is specified, management directs someone to design and implement the simulation software for this object and creates a contract to this purpose. This contract contains all the pertinent information, as outlined above. Thus application software and simulation software development can proceed essentially in parallel, but management must assure that any changes in the requirements of the applica-

tion software will be propagated to simulation development, perhaps by "update contracts".

The first step of simulator development is now to model the simulator, i.e. to derive a set of transactions specifying the simulator. A tool that extracts the relevant information from the contract and generates a first version of the model specification is conceivable. In a second step, this set of transactions probably has to be fine-tuned and manipulated by using the standard MARS design tools, until finally arriving at a set of transactions specifying a MARS cluster (or a MARS system, for very complex simulations).

Deriving the environment simulator specification from the application specification has an additional benefit: It indirectly "tests" (validates) the application specification. What we mean by this statement is the following: If (after simulator implementation) the behavior of the simulated environment agrees with the behavior of the real environment, confidence increases that the simulator specification — and thus the application specification, too — is correct.

It should also be noted that the application designer often specifies a (virtual) cluster describing the external environment and/or the behavior of the operator of the proposed system. This is done at the highest level of refinement, and might also be of some help to the simulation designer, at least as far as modeling the physical process environment is concerned.

6.2 Developing the Simulation Software

Once the appropriate set of transactions has been derived and serves as the requirements of the simulation software, simulator development is in the same state as any other application software development. Using the appropriate MARS design tools, cluster design, component design, and task design and implementation are performed. (It is assumed here that the simulator will be implemented as a MARS cluster.) A suitable CPU and bus schedule is calculated by the appropriate timing analysis tools.

In principle, simulation software development can proceed in parallel to application software development. However, due to the (possible) interdependence between application requirements and simulator requirements described in the previous section, there is an increased likelyhood of changing simulator requirements during the first phase of application development. Therefore, it is probably a good idea to delay simulation software development until requirements become more stable, in order to avoid duplication of effort.

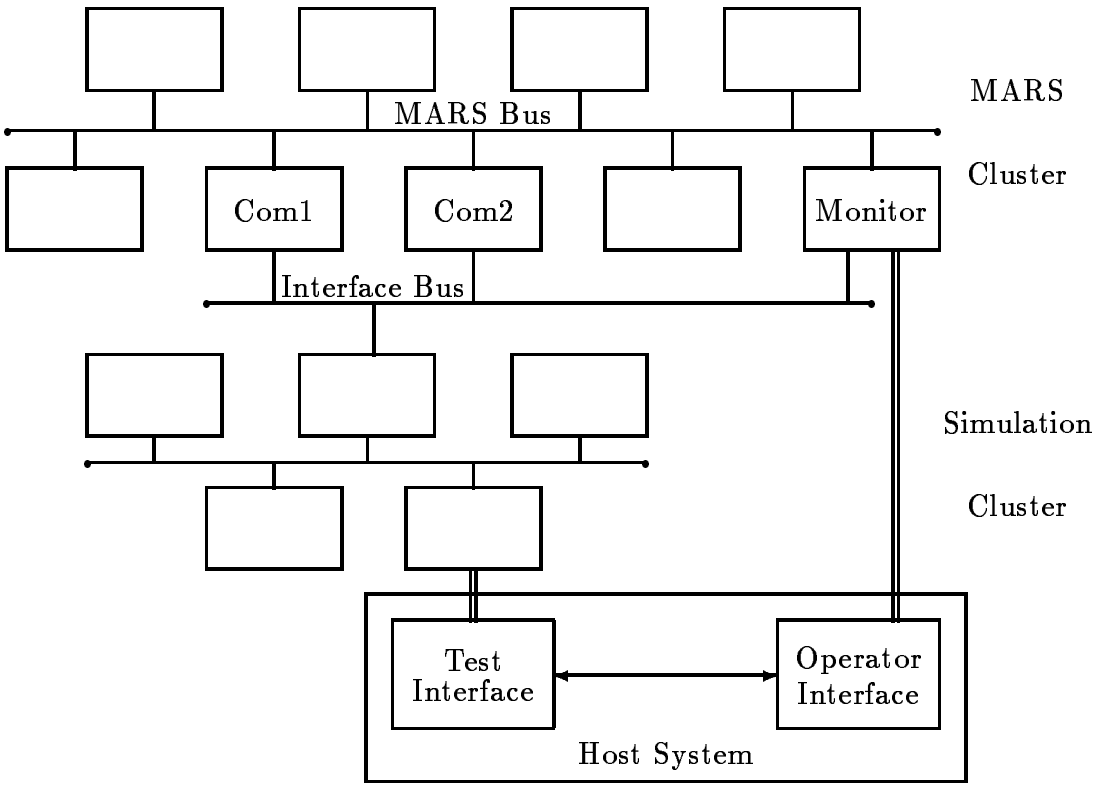
What remains to be addressed, is of course the problem of testing the simulation software itself. Clearly, it is not practical to implement yet another simulator

which simulates the environment of the first simulator; it would be identical to the application system anyway. The suggested approach is to use the Task Test and the open-loop Cluster Test just as in application software testing. If at all possible real-life data from the real environment (i.e. the physical process) should be used for this Cluster Test. Afterwards, the simulator is declared ready for its original purpose: aid in running tests. However, one has to keep in mind that any failure of a test run is due to *either* an error in the application cluster under test *or* an error in the simulation software. Again, genuine data can hopefully be obtained from the real environment in order to allow speedy detection of the error location.

6.3 Executing the Simulation Software

It seems reasonable to assume that execution of an environment simulator will consume more CPU power than execution of the test software without simulation. So far, this test software has been assumed to be relatively simple: What it has to do is to send precomputed test data (input messages) at the appropriate (predetermined as well) point in time via the appropriate input channel to the cluster under test, and to receive and store the responses (output messages) of the cluster under test ("open-loop testing"). Thus, methods have to be devised to ensure that adequate computing resources are present in the test system.

Luckily, this is easy to achieve in MARS: The test components of the open-loop test system are expanded into a simulation cluster. The application and the simulation clusters communicate via the Interface Bus. 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 behavior is influenced (not even indirectly by having to receive and discard messages that would not be sent under normal operating conditions) by the other's presence. Figure 6.1 shows the proposed architecture for a closed-loop Cluster Test.



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

Figure 6.1: Test System for the Cluster Test with Environment Simulation

Chapter 7

Supporting the Test Process

This chapter deals with the support that must be provided for the process of testing, and how this support is integrated into the existing MARS Design Environment. Items to address are test data generation, test execution, and evaluation of test results. Anything concerning the construction of environment simulators can be found in the previous chapter.

7.1 Test Data Generation

The most simple test strategy would be to have a simple loop of test data generation, test execution, and result evaluation, which is continued until it is somehow determined that enough testing has been done. This approach is not only simple, but also simplistic. For example, it is almost impossible to get any information on the coverage that has been achieved by the tests executed up to a particular point in time, and thus to determine or estimate the quality of the testing done so far, or to decide how much more effort to invest in testing.

As a first observation, test case design (i.e. test data generation) can and should begin much earlier than the point in time where one is ready to actually execute some tests. Test data generation can start as soon as the object it is intended to test is specified, i.e. in parallel with the design, refinement, or implementation of that object. The advantage is that test data generation is not biased by information on the implementation of the object, on the other hand not all the required test case can be generated at this time, since the object can only be considered as a "black box". Table 7.1 shows which kinds of test cases can be prepared at which phases of the MARS software life-cycle.

This approach makes it necessary, of course, to store test cases for later use in a so-called *test case data base*. In addition, it should be possible to add further test

Design Phase	Testing Activity
Requirements Definition	generate test cases to check consistency with system requirements (black-box testing)
System Design/Cluster Def.	for each cluster, generate test cases to check consistency with cluster requirements (black-box testing)
Cluster Design/Component Def.	for each component, generate test cases to check consistency with component requirements (black-box testing)
Component Design/Task Def.	for each task, generate test cases to check consistency with task requirements (black-box testing)
Task Design/Implementation	for each task, generate a set of test cases for structural (white-box) testing, e.g. path testing

Table 7.1: Design Phase versus Testing Activity

cases at any time during or after the design of each object, in order to allow the inclusion of "white box" test cases.

Management of test case generation is also done by contracts: When some object has been specified, a contract directing a developer to design and/or implement this object is sent out. In parallel, a contract directing the same or (preferably!) another person to generate test cases for this object is issued. These two contracts contain essentially the same information: the specification of the particular object. The tester can now begin to develop test cases for this object, as well as expected results of each test case (if possible); the results of this activity are stored in the test case data base associated with this object. In addition, it is very important that any specification changes that are made during design or implementation are propagated to the tester by so-called "specification update contracts", so that all test cases developed so far can be updated. Alternatively, the tools of the Design Environment could force the contract client (the requester) to design test cases for the object whose implementation he requests. He will use these test cases later on

for evaluating the quality of the delivered object implementation.

The content and format of the test cases will generally be dependent on the application, the test phase, and on whether the test case is intended to be executed in open-loop or closed-loop test mode.

7.2 Support for Test Execution

To support the actual test execution, we suggest the design of a *test specification language* that allows the tester to select and specify all the pertinent parameters for an intended test run. This will only be applicable to the Cluster Test and all higher-level test phases; for the Task Test, we suggest a less formal approach. In this case it is more important that testing is closely integrated into the programming-in-the-small environment, so that the programmer can perform the activity cycle of code generation, code execution, and error correction as efficiently as possible.

In all subsequent test phases, however, executing a test run is much more complicated; therefore, stronger and more formal support is desirable. In general, there are three kinds of information that it must be possible to specify by the proposed test specification language: Test system architecture and test run preparation, the test data to be used, and the test results to be collected.

Among the first group, the following items are needed:

- The test phase and whether the test run will be in open-loop or closed-loop mode.
- The object under test, where to find its source code, and how to down-load the source code in the target system.
- If the test mode is closed-loop, the environment simulator to be used, where to find its source code, and how to down-load the source code in the test system.
- If operator inputs are possible, will the real operator interface be used (thus enabling real interference from the tester) or are these inputs also prespecified like ordinary input data?
- Will this test be run in real time or in some debug mode, such as slow motion, or with some break-points enabled?

The second group consists chiefly of two items:

- The desired initial state of the object under test (which includes the start time of the test run) and of the environment simulator, if appropriate.

- If the test mode is open-loop, the test data that should be used. These could be a prespecified set of test data, or they could be generated "on the fly".

In the third group it is again possible to specify two different items:

- The results that should be sent to the operator interface in order to allow an on-line monitoring of the test run. (Note that this might change subsequently due to either simulated or real operator commands.)

- The results (i.e. messages) that should be recorded for later evaluation, and where these should be stored. It should be possible to qualify this information by some time interval, in order to facilitate a more detailed result monitoring than usual during certain critical time intervals.

A "precompiler" is then invoked that interprets the above mentioned information and prepares a *test script* (cf. [Gom86]). In doing so, it will get from the Design Environment allocation, scheduling, and (internal and external) message format information. Execution of the test script will then perform the following activities:

1. Starting the target and the test system;

2. Down-loading the required object code files;

3. Wait for the target and the test system to synchronize;

4. Initialize the state of the target system, the test system, and the operator interface;

5. Allocate all needed resources, especially for recording the results of the test run; and

6. Starting the test run.

Management of test execution is of course again done by contracts. When an object is ready to be subjected to a certain test phase, a *test contract* is sent to the tester who is responsible first for setting up the required test system architecture and second for conducting all the tests previously specified and possibly some additional ones, depending on the test plan. He does so by using the tools mentioned above, stores all the test results, and evaluates them. In his reply contract, he indicates either failure(s) or success.

7.3 Test Result Evaluation

The types of evaluation that the test results are subjected to depend first of all on the test phase they have been collected in, but more heavily on the current application problem. Therefore it is suggested to only provide the possibility to integrate custom-designed or application-specific evaluation tools at any time into an *open tool box*.

There are, however, a few tools that are universally useful, such as automatic comparison of test results with expected results, determination of mean response times or task execution times and their variance from the test results, or graphical display of result variables or internal states versus time for visual evaluation. Means and variances of task execution times are interesting first for assessing the quality of the computed maximum execution times, then to determine the probability that a task can be executed twice in its allocated time slot, a technique proposed to enhance transient error coverage. We also intend to ensure that test results contain all data needed for software reliability modeling. What data in particular are necessary for this purpose is still under investigation.

Chapter 8

Conclusion

The purpose of this report is to propose a first version of an integrated test strategy for hard real-time systems, including a survey of the relevant literature. The available literature has not been all too helpful, and many ideas presented in this report have originated in "free association" although the MARS design philosophy and the existing prototype of the MARS Design Environment acted as constraints.

To see how realistic some of these ideas are, a number of further studies and prototype implementations are planned. The highest priority is to provide all the necessary functions for performing open-loop Cluster Tests, with emphasis on the lower-level functions, while providing fully automated support for test execution has lower priority. Equally important is a feasibility study on how to transform the specification of an object into a specification of a model of this object's environment. Implementation of a prototype tool to aid in this transformation process is also planned. Support for test data generation will only be provided to the extent it is needed for the tasks mentioned above, while full support will have to wait. After the prototypes have reached an acceptable level of maturity, the contract handling of these new tools will be implemented, thus providing integration into the existing Design Environment.

Finally, it is strongly suggested that any further study of the problems associated with Task Testing is done in conjunction with the work on an environment 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 environment.

Bibliography

[Adr82] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[Ame85] W. Ameling. Methoden und Aspekte zur Planung und Analyse technischer Systeme. In D. Möller, Editor, *Proceeding 3. Symposium Simulationstechnik (Informatik-Fachberichte 109)*, pages 87–103, Springer Verlag, Bad Münster, BRD, 1985.

[Ber87] W. Berndt, K. Brantner, H. G. Thome, and B. Wieneke-Toutaoui. Modellebenen. In J. Halin, Editor, *Proceeding 4. Symposium Simulationstechnik (Informatik-Fachberichte 150)*, pages 103–118, Springer Verlag, Zürich, CH, 1987.

[Bry76] G. W. Bryce, T. R. Foord, D. J. Murray-Smith, and P. W. Agnew. Hybrid Simulation of Water-Turbine Governors. In R. Crosbie and J. Hay, Editors, *Toward Real-Time Simulation: Languages, Models, and Systems (Part 1)*, pages 35–44, Simulation Councils Proceedings Series, Vol. 6, No. 1, La Jolla, CA, June 1976.

[Cho84] C. K. Cho, E. K. Lin, and C. L. Jen. On Performance Evaluation of Multiprocessor Systems for Real-Time Simulation. In E. Comer, Editor, *Proc. 17th Annual Simulation Symposium*, pages 209–225, IEEE Computer Society Press, Tampa, Florida, 1984.

[DeM87] 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.

[Dud85] E. J. Dudewicz and Z. A. Karian. Modern Design of Simulation Experiments. In E. J. Dudewicz and Z. A. Karian, Editors, *IEEE Tutorial: Modern Design and Analysis of Discrete-Event Computer Simulations*, pages 186–190, IEEE Computer Society Press, Washington, D.C., 1985.

[Fey83] S. Feyock and W. R. Collins. ADA and Multi-Microprocessor Real-Time Simulation. In L. Holbrook, Editor, *Proc. 16th Annual Simulation Symposium*, pages 211–228, IEEE Computer Society Press, Tampa, Florida, 1983.

[Gea77] C. W. Gear. Simulation: Conflicts between Real-Time and Software. In J. Rice, Editor, *Mathematical Software III*, Academic Press, New York, 1977.

[Gel88] D. Gelperin and B. Hetzel. The Growth of Software Testing. *Communications of the ACM*, 31(6):687–695, June 1988.

[Gla80] R. L. Glass. Real-Time: The "Lost World" of Software Debugging and Testing. *Communications of the ACM*, 23(5):264–271, May 1980.

[Gom86] H. Gomaa. Software Development of Real-Time Systems. *Communications of the ACM*, 29(7):657–668, July 1986.

[Har85] T. T. Hartley and G. O. Beale. Integration Operator Design for Real-Time Digital Simulation. *IEEE Transactions on Industrial Electronics*, 32(4):393–398, Nov. 1985.

[Hau84] G. Hauser. Test von Steuerungssoftware mit Hilfe der Realzeitsimulation. In F. Breitenecker and W. Kleinert, Editors, *Proceeding 2. Symposium Simulationstechnik (Informatik-Fachberichte 85)*, pages 495–499, Springer Verlag, Wien, 1984.

[Hel82] K. Hellmold. Multiprozessorsystem für die parallele Simulation von Zeitdiskreten Systemen. In M. Goller, Editor, *Proceeding 1. Symposium Simulationstechnik (Informatik-Fachberichte 56)*, pages 227–233, Springer Verlag, Erlangen, BRD, 1982.

[How82] W. E. Howden. Life-Cycle Software Validation. *IEEE Computer*, 15(2):71–78, Feb. 1982.

[How86] W. E. Howden. A Functional Approach to Program Testing and Analysis. *IEEE Transactions on Software Engineering*, SE-12(10):997–1005, Oct. 1986.

[How87] W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, 1987.

[Kel85] H. B. Keller. Unterstützung der Prozeßführung im nuklear-chemischen Bereich durch den Einsatz der Simulationstechnik. In D. Möller, Editor, *Proceeding 3. Symposium Simulationstechnik (Informatik-Fachberichte 109)*, pages 453–458, Springer Verlag, Bad Münster, BRD, 1985.

[Kel87] H. B. Keller. Algorithmische und problemstrukturelle Parallelität – Ansätze zur verteilten Simulation komplexer dynamischer Systeme. In J. Halin, Editor, *Proceeding 4. Symposium Simulationstechnik (Informatik-Fachberichte 150)*, pages 286–295, Springer Verlag, Zürich, CH, 1987.

[Kel88] H. B. Keller. Verteilte/modulare Echtzeitsimulation komplexer Systeme. In W. Ameling, Editor, *Proceeding 5. Symposium Simulationstechnik (Informatik-Fachberichte 179)*, pages 73–83, Springer Verlag, Aachen, BRD, 1988.

[Kod85] R. Kodweiß. Software Konzept für Echtzeit-Simulation. In D. Möller, Editor, *Proceeding 3. Symposium Simulationstechnik (Informatik-Fachberichte 109)*, pages 251–253, Springer Verlag, Bad Münster, BRD, 1985.

[Kop85] H. Kopetz and W. Merker. The Architecture of MARS. In *Proc. 15th Int. Symposium on Fault-Tolerant Computing*, pages 274–279, Ann Arbor, Michigan, June 1985.

[Kop87] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, 36(8):933–940, Aug. 1987.

[Kop88] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. MARS: Ein fehlertolerantes, verteiltes Echtzeitsystem. *Informationstechnik (it)*, 30(3):197–208, June 1988.

[Kop89] 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.

[Koz87] 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.

[Las89] J. Laski. Testing in the Program Development Cycle. *IEE Software Engineering Journal*, 4(2):95–106, Mar. 1989.

[Led76] J. D. Ledger, A. J. Fisher, and G. C. Barney. Real-Time Hybrid Simulation of a Turbocharged Diesel Engine. In R. Crosbie and J. Hay, Editors, *Toward Real-Time Simulation: Languages, Models, and Systems (Part 1)*, pages 45–50, Simulation Councils Proceedings Series, Vol. 6, No. 1, La Jolla, CA, June 1976.

[Pus90] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 2(1), 1990.

[Sag66] A. P. Sage and S. L. Smith. Real-Time Digital Simulation for Systems Control. *Proceedings of the IEEE*, 54(12):1802–1812, Dec. 1966.

[Sch82] B. Schmidt. Systemanalyse und Modellbildung. In M. Goller, Editor, *Proceeding 1. Symposium Simulationstechnik (Informatik-Fachberichte 56)*, pages 106–115, Springer Verlag, Erlangen, BRD, 1982.

[Sch85] B. Schmidt. "Was tut man, wenn man simuliert?" Versuch einer Begriffsbestimmung. In D. Möller, Editor, *Proceeding 3. Symposium Simulationstechnik (Informatik-Fachberichte 109)*, pages 104–111, Springer Verlag, Bad Münster, BRD, 1985.

[Sen87] Ch. Senft. Remodel A Realtime System Methodology on Design and Early Evaluation. In *Proc. IFIP Conference on Distributed Processing*, pages 305–321, Oct. 1987.

[Sen88a] 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.

[Sen88b] Ch. Senft. *Realzeit im Entwurf verteilter Prozeßrechnersoftware*. PhD Thesis, Technisch Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria, Oct. 1988.

[Stu82] K. H. Sturm. Simulation von Systemkomponenten in Realzeitsystemen. In M. Goller, Editor, *Proceeding 1. Symposium Simulationstechnik (Informatik-Fachberichte 56)*, pages 494–504, Springer Verlag, Erlangen, BRD, 1982.

[Tai85] K. C. Tai. On Testing Concurrent Programs. In *Proceedings of COMPSAC*, pages 310–317, Oct. 1985.

[Tar88] J. E. Tardy. The Host/Target Approach to Embedded System Development is Becoming Obsolete. *ACM SIGSOFT Software Engineering Notes*, 13(4):45–51, Oct. 1988.

[Tom76] B. N. Tomlinson. Hybrid Computing for Real-Time Flight Simulation. In R. Crosbie and J. Hay, Editors, *Toward Real-Time Simulation: Languages, Models, and Systems (Part 2)*, pages 105–114, Simulation Councils Proceedings Series, Vol. 6, No. 2, La Jolla, CA, Dec. 1976.

[Tro83] K. C. Trott and F. K. Frantz. A Detailed Interactive Simulation System for Developing Command and Control Systems. In L. Holbrook, Editor, *Proc. 16th Annual Simulation Symposium*, pages 11–31, IEEE Computer Society Press, Tampa, Florida, 1983.

[Van83] G. C. Vansteenkiste. Simulation Methodology for Improved Process Interaction. In W. Ameling, Editor, *Proceeding First European Simulation Congress ESC 83 (Informatik-Fachberichte 71)*, pages 32–43, Springer Verlag, Aachen, BRD, 1983.

[Wei88] 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.