A Survey of MARS

Wolfgang Schwabl, Johannes Reisinger, and Günter Grünsteidl

Institut für Technische Informatik
Technische Universität Wien
Austria

schwabl@vmars, reising@vmars
gruen@vmars

Research Report Nr. 16/89

October 31, 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

Abstract

Providing a reliable service in a continuously running real-time system is problematic. MARS (MAintainable Real-time System) offers correct and timely responsiveness by design and ease of maintenance by design of a new fault-tolerant distributed real-time system with active redundancy.

Several unconventional techniques are implemented in the operating system of MARS. It is based on a time rigid scheduling, it avoids interrupts and has no dynamic memory allocation. All operating system concepts that may lead to an unexpected run time delay or deadlock are kept out of MARS. Contrary, MARS has new important services within its operating system. A fault-tolerant global time scale with a precision of a few microseconds allows the synchronous execution of distributed cooperating tasks. Tasks communicate through standardized messages with limited life time. The reliable message transport service tolerates transient message losses using redundant datagrams.

MARS strongest feature is to interpret each message as a state value, i.e. a newer value updates an older one. This gives notable advantages for the operating system and totally avoids queuing of events. Active redundant components produce the same messages. Due to the state semantics of messages redundant messages are masked out easily. State messages can be read arbitrarily often, but they never can be consumed. Thus multiple or redundant tasks may read the same message. Scheduling in MARS is a combination of time rigid CPU and message bus scheduling. Periodic tasks can be pre-scheduled such that their CPU slot will finish just when access to the communication media will be granted. Consequently, all transactions can be specified and their processing time can be checked during the system's design phase.

Several methods for fault-detection are included in MARS to achieve a fail-silent behavior of system components. All these features of MARS have been implemented and tested experimentally at the Technical University in Vienna. We strongly believe that such a deterministic system design is a feasible technique for engineering distributed hard real-time computers and applications.

1 Introduction

Real-time systems are characterized by a tight interaction of a computer system with the real world. The effects of a real-time computer's reaction have to take place within a specified interval in time. When the timely responsiveness of a computer systems becomes a critical factor to the safety of human life or to substantial commercial expenses, then such a system is called an *hard real-time* system. Examples for hard real-time systems are computers controlling nuclear power plants, railway control systems, aircraft control systems, and air traffic control systems. The reliability of such hard real-time systems must be high enough to suffice the challenges of timely responsiveness even under wicked circumstances, but the maintenance in a continuously running system is problematic.

1.1 Problems with the State-of-the-Art

Traditional real-time systems arbitrate computational power to different stimuli of the controlled object using a priority scheme. The different priorities are calculated using the mathematics of queuing theory, which often assumes a random distribution (Poisson distribution) of events in time. Requests which cannot be served immediately are queued for later service. If several requests arise simultaneously then the latest requests will probably become irrelevant before being processed or even worse, the associated deadlines will be missed and in consequence a catastrophe may occur. The accident at Three Mile Island has shown that the alarms processed by the reactor control computer had been 2 hours behind the real situation. Queuing theory can predict the average responsiveness of computers, but it cannot guarantee a maximum response time in case of peak load. Traditional real-time systems based on queuing of events are not adequate for hard real-time control.

Commonly, static priority scheme are used to determine the precedence of processing simultaneously occurring events. Alarms have often priority over normal operation. A static priority scheme cannot master every operational situation correctly. Consider e. g. an aircraft control system, where a fire alarm rises during takeoff. The normal operation task to stabilize the flight is as important as the task to extinguish the fire. No task can have priority over the other one. This problem cannot be solved by using static priorities. Dynamic priorities schemes are supposed to be better than static schemes. Nevertheless, it is a big problem to guarantee that a real-time system using a dynamic priority scheme can meet all deadlines under all circumstances. We feel better if a real-time system can prove its timely responsiveness by design.

Computer failures are a fact of live. Real-time computers cannot reboot during a critical situation, e. g. the landing approach of an aircraft. There is an essential need for fault-tolerant computing in hard real-time systems. Distributed systems can be more reliable than centralized systems, due to their inherent property of autonomous operation of system components. Fault-tolerant real-time computers consist of system components cooperating in *active redundancy*, i. e. more than one component execute the same tasks simultaneously. If one system components fails then still the result of a redundant component will be available instantly. It is a good strategy to switch off faulty system components automatically in order to prevent further failures caused by the first failure. A system component with such a behavior is called to be *fail-silent*.

In a system with fail-silent components the number of operational system components decreases with time. This is acceptable for systems with short mission times, e. g. the duration of a flight operation, provided that enough redundant system components are operating at the start. Systems with long mission times, e. g. an air traffic control system or a reactor control system, face the problem of continuously reintegration of repaired system components. We do not know how conventional real-time systems allow maintenance without interrupt of transactions being processed and especially, how repaired system components recover consistently without affecting the timely responsiveness of the whole control system.

1.2 Assumptions about Real-Time Systems

A hard real-time system must respond to a stimulus from the control object within an interval dictated by the environment, called the response time. This maximum response time must be guaranteed even under peak load conditions and under wicked fault conditions. But first it must be defined what a peak load situation is and which faults are assumed, otherwise it will make no sense to guarantee a response time.

When several stimuli arise simultaneously from the environment then a high system load is generated. In order to formalize the meaning of "peak load", each stimulus has to be expressed quantitatively. For each stimulus a maximum response time must be determined as well as how often the stimulus may arise in the worst case, i. e. the maximum frequency of the stimulus. Peak load is defined to be a situation where all possible stimuli arise simultaneously with their maximum frequency. It can be quite tedious to define a peak load situation, but it is essential for application analysis and system predictability. We assume that such a specification of peak load for hard real-time systems is possible.

If a serious failure occurs, either in the control system or the control object, such that further continuation of the operation is not possible, then the system must switch to a fail-safe mode. We assume that a fail-safe mode of the control object is specified during the system design phase and it is up to the application how to reach a fail-safe mode.

In the design of hard real-time systems it is necessary to consider the value aspect and the timing aspect of the system's behavior simultaneously. In our model of real-time systems the activity of a system can be described by a real-time transaction. A real-time transaction refers to the execution of a set of correlated actions between a stimulus from the environment and the corresponding response indicating the completion of this set of actions. Real-time transactions can be used to express the external requirements as well as the different levels of detail during the system refinement. The value aspect of real-time transactions is considered by evaluation of consistency constraints during system design. The timing aspect is conducive to guarantee the responsiveness of the real-time system by design and to check the correct system operation during run time. Within the model of a real-time transaction a formal and practical evaluation of correct system behavior is achievable.

A real-time control system has to perform real-time control functions and to collect data for archival purposes. We therefore distinguish between two data bases needed for these two different purposes.

- The set of all data elements which is needed for the instantaneous real-time control, operator display, alarm monitoring and other real-time functions is called the *real-time data base*.

- The set of all data elements which is required for purposes other than direct real-time control is called the *archival data base*.

The real-time data base changes as time progresses. The information in the real-time data base is invalidated by the passage of real-time. If we loose the real-time data base, the control of the environment is interrupted thus possibly leading to a catastrophe. After an element has been stored in the archival data base, it may not be modified any more. The information in the archival data base does not change as time progresses. If we loose the archival data base, there is no immediate effect on the real-time control of the environment.

Experience has shown that in most applications the real-time data base is much smaller than the archival data base. The archival data base can be kept on a file store outside the real-time system. Access to the real-time data base is critical, therefore it cannot be stored within an arbitrary file system. We think the

real-time data base must be kept within the main memories of a real-time system without a complicated file system organization. For the real-time data base we prefer a generic organization with direct access to data items indexed by semantic names of the data.

Most state of the art real-time systems are distributed, i. e. they consist of a set of autonomous components connected by a local area network. The communication between the components is realized solely by the exchange of messages. There are a numbers of international groups working to standardize the real-time communication system, e. g. the MAP group [Rod89]. The most common communication system found in real-time applications is a bus or a ring system controlled either by a token, time division multiple access (TDMA) or carrier sense multiple access (CSMA) protocol. For the sake of predictability and efficiency simple protocols are favorable for use in real-time systems. Disconnecting and joining components are easy in stateless communication protocols. Therefore we prefer the simplest protocol for real-time communication, i. e. messages are exchanged by using the basic datagram broadcast service of the local area network. We assume that checksum mechanisms inside a local area network controller are reliable enough to detect transmission errors. We do not assume that messages are sent reliably, messages can get lost.

2 Concepts for MARS

2.1 The MARS Architecture

We solved many problems of real-time systems by designing the architecture of MARS (MAintainable Real-time System). This is a fault-tolerant distributed system with active redundancy which provides correct and timely responsiveness by design as well as ease of maintenance [Kop89a].

MARS uses a transaction model to describe the activities of a real-time system. A transaction is a sequence of interrelated actions transferring the system from one state to another. Viewed on a lower level, an action itself may be a transaction constituted of more primitive actions. A transaction is triggered by a stimulus and it produces a response. The maximum time interval between stimulus and response is dictated by the environment.

A MARS system consists of a set of MARS clusters, the controlled object(s), a development and a maintenance computer system. Each cluster is composed of several components interconnected by a synchronous real-time bus (see also figure 1). From the point of view of one cluster all other clusters as well as the

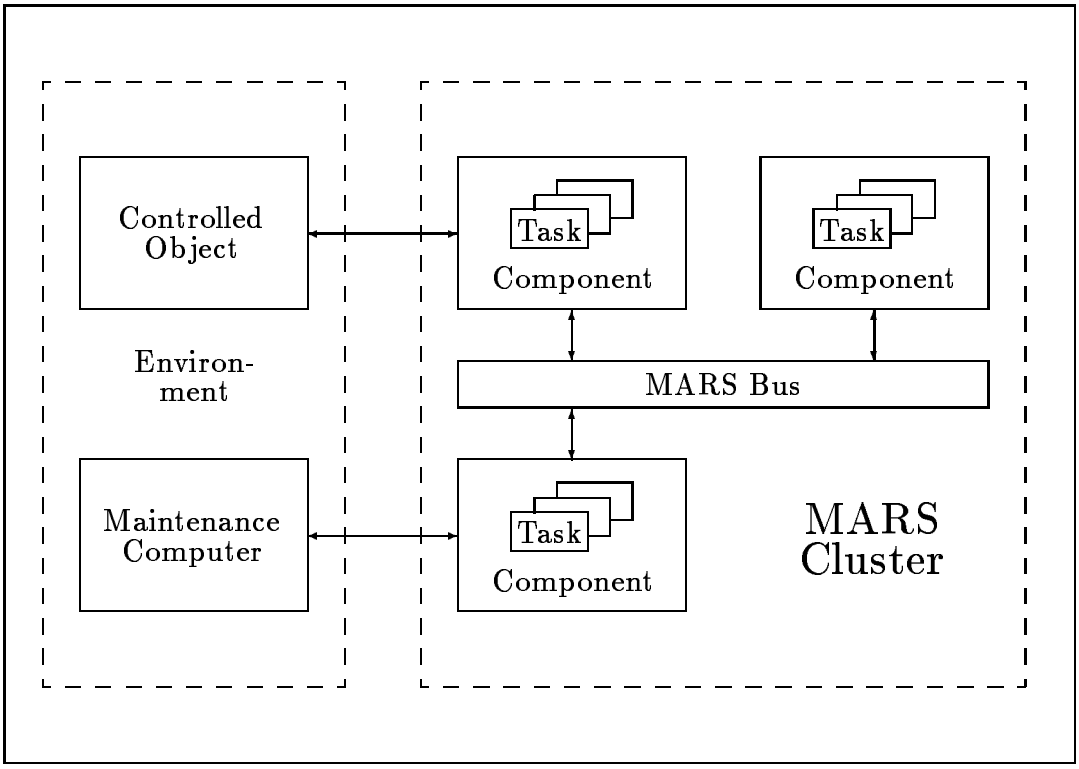

Figure 1: A MARS System

controlled object and other computer systems are considered to be the environment. The interfacing to the environment is done in dedicated components. A component is a self-contained computer, including the application software. It is a hardware/software unit of given functionality and performance. A set of real-time tasks and an identical copy of the MARS operating system are executed on each component [Dam89]. To achieve a deterministic system behavior all operating system concepts that may lead to an unexpected run time delay or deadlock are kept out of MARS. MARS has no dynamic resource allocation and interrupts are totally avoided except the clock interrupt.

Communication among the tasks and components is realized by the exchange of messages, which are broadcast as datagrams. MARS strongest feature is to interpret each message as a state value, i. e. a newer value updates an older one. This gives notable advantages for the control system and totally avoids queuing of events. Each message has a validity time attached. As soon as the validity expires, the message is discarded by the operating system. This measure is taken to avoid inconsistencies in the system by processing outdated and thus invalid

messages. Moreover, recovery of components becomes especially easy since the maximum validity time of all messages is exactly the time to wait until all state messages have been recovered.

All MARS components have access to a global time which has a known synchronization accuracy of a few microseconds. The global time is provided by the architecture and implemented by aid of an VLSI clock synchronization unit which is synchronized with a fault-tolerant clock synchronization algorithm. The global time is the basis for a time rigid scheduling of task, the conflict free access to the bus as well as other synchronous and thus deterministic activities in the system.

When a computer failure occurs then appropriate fault-detection mechanisms in the operating system isolate the failure within a smallest replaceable unit, which is in our architecture the MARS component. To reduce the complexity of combination of failures we consider only components with fail-silent operation, i. e. every SRU works correctly or works not at all. The methods for fault-detection included in MARS to achieve a fail-silent behavior of system components are described later. The fault-tolerance of MARS is based on fail-silent components running in active redundancy and the multiple sending of messages on the real-time bus. If one component fails, then the results of the redundant component are still available. After maintenance of a failed component it can recover silently with no drawback to the environment while the processing is done by another instance of the component.

All these features of MARS have been implemented and tested experimentally at the Technical University in Vienna. We strongly believe that such a deterministic system design is a feasible technique for engineering distributed hard real-time computers and applications.

2.2 Task Model

The task model in MARS corresponds to the lowest level of a real-time transaction. A stimulus of the transaction is an input to the task and the output is the response of this transaction. Each transaction can occur repetitively therefore a MARS task has to be cyclic in nature. The maximum frequency of a transaction determines the period of the task. Tasks are executed with this period in order to guarantee timely responsiveness even in peak load situations.

In each cycle a MARS task performs the sequence input – computation – output (see figure 2). The initialization part is executed only at the first invocation of the task.

Recovery can be subsummized in the problem of how to get the values of the local variables, which reflect the inner state of a task. If a task does not have an

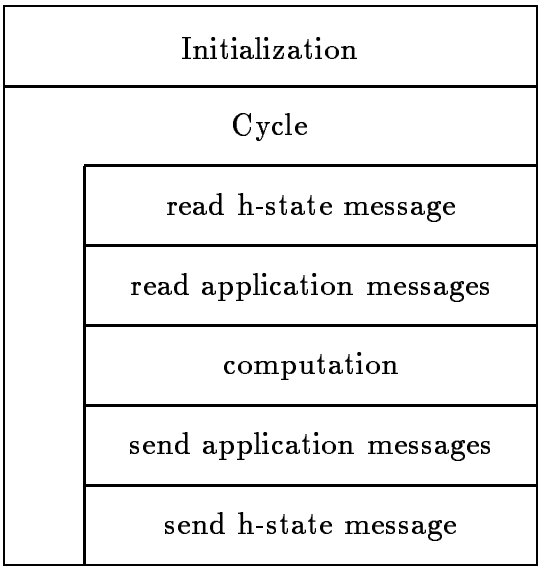

Figure 2: Task model of MARS

inner state then recovery can be done easily. Recovery of tasks owing an inner state is a little bit more complicated.

MARS tasks are made stateless by using a special scheme. During design the inner state of a task is specified. When a task with an inner state suspends execution, its inner state is written into a message and sent at the bus. At the beginning of the next cycle the task restores its inner state out of this message.

2.3 State Message Approach

In general, there are two ways to do the message passing between tasks: by event messages or by state messages. Event messages represent the traditional message semantics. They are created due to an event in the controlled object or in the system itself. Event messages correspond to the queuing of events. There are some drawbacks when using event messages in hard real-time systems:

Flow Control: In a real-time system the computational speed is dictated by the environment. A receiver cannot slow down a sender. Flow control of messages is a very difficult task in event-driven real-time systems. It has to be guaranteed during the design of the application that each component can process every incoming message within a predefined amount of time, even if all messages arrive with their maximum frequency.

Clogging: If some events occur nearly concurrently, a great amount of messages has to be sent within a short time span, what can lead to the clogging of the network or to large delays in message transmission.

Schedule Conflicts: It is not possible to schedule tasks statically in the case of an event driven system, since the execution of a task is triggered by the arrival of a message and not by the time. However, dynamic scheduling is not well suited for real-time systems (see chapter 2.4).

In MARS the periodic state-message approach is used, which implicitly solves the problems mentioned above. A message contains the state of a part of the environment or of a part of the system. Each message is provided with a name and a validity time. The name of the message corresponds to the semantic meaning of the data. Messages with the same name contain the same state, possibly at different points in time. After the validity time of a message is expired, it is discarded by the operating system. Each time a message is received by a component, it updates the currently valid message (if any exists) with the same name. Explicit flow control is unnecessary. Tasks are scheduled such that they can process even the maximum frequency of events. Processing of expired information is prevented, because an older message is replaced by a newer one each time when a newer message is received.

We have no problem to handle the loss of a redundant message or the duplication of messages. The only effect of loosing a single instant of a message is, that it does not overwrite its redundant copy; duplication of a message causes overwriting of a message by itself. This concept also facilitates the task of implementing redundant components. From the point of view of other components, the messages from a redundant component are treated like the duplication of messages from a non-redundant component. Therefore, no component has to know whether any of the components in the cluster (including itself) works in redundancy.

Since the state of each part of the environment or the system, which has to be known some time is sent periodically at the MARS bus, the bus traffic will not increase in the case of peak load, therefore clogging is avoided.

2.4 Static Scheduling

In any computer system there are basically two ways to do the scheduling: dynamically or statically. In the dynamic approach the scheduling decisions are made at run time whereas in the static approach they are made at compile time.

When using dynamic scheduling, each scheduling decision is made during run time according to the state of the system (system load, task priorities, mutual ex-

clusion requirements,...) and the state of the environment (external interrupts). This method is well suited for most computer systems since it makes a high processor utilization possible. However, the aim of hard real time systems is not to achieve high average performance but to meet all deadlines. To guarantee this, all possible high-load situations have to be analyzed during the design phase of the system. Provided with this knowledge, it is possible to build feasible schedules before run-time. There are advantages of this method for hard real-time systems:

No CPU time needed for scheduling: Since scheduling is performed before run-time, the amount of time needed to find the schedule is of no importance. In systems with dynamic scheduling strategies a compromise between the quality of the schedule and the computation overhead has to be made.

Optimized Scheduling: The schedules of the components of a distributed system can be mutually adapted, because complete knowledge of each component is available at a central instance (the pre-run-time scheduler).

Guaranteed Execution Time: The designer must analyze the application carefully. The maximum execution time of each task has to be determined, the communication structure between the tasks has to be analyzed. Design failures (e.g. the missing of deadlines under high load conditions) can be detected. The computation power of the system can be well adapted to the application.

On the contrary, in case of dynamic scheduling, the communication structure and run-time of the application also could be analyzed before run-time. Nevertheless, it has to be proofed that *all* deadlines have to be met in *every* possible sequence of the tasks, whereas in the static approach it has to be proofed only for *one* sequence.

Since the major design goal of MARS is to guarantee to meet all deadlines, the static scheduling method was chosen, in spite of its poor average processor utilization. *A proof for a correct real-time schedule is achieved by construction.*

2.5 Schedule Switch

A single static schedule cannot cope with all operational situations. Imagine some states of a system:

- Initialization of the system.
- Phase of normal operation.

- Some other normal operation modes (e.g. reduced operation during the night).
- Some emergency modes.

The method of static scheduling seems not to be well suited to handle these situations. Therefore MARS uses a rather simple but powerful mechanism to put the system into different states of operation: the *schedule switch.*

During the design phase the set of possible operation modes has to be built. In some cases it may be necessary to consider operation modes which handle two or more emergency situations concurrently (e.g. a fire during the take-off of a plane). In the next step a schedule for each of these modes has to be found (if this is not possible, the system has to be redesigned). The schedule switch concept introduces an additional level of complexity into the pre-run-time scheduler: not only the schedules for all situations have to be computed independently, but also the transitions between the different schedules have to be taken into account by the scheduler. There are different kinds of scheduling switches:

Immediate schedule switch: Switching to the new schedule is performed immediately after the request to switch. Because it cannot be guaranteed to continue all transactions after switching, the immediate scheduling switch should be performed only in emergency cases which will lead to a safe state of the system E.g. in the case of an emergency stop of a machine carrying on of normal operation need not be guaranteed. *Even if the cost incident to stopping operation is high, the decreased probability to loose human life in the case of immediate reaction overrides this disadvantage.*

Consistent schedule switch: Switching to the new schedule is performed only at predefined points in time. All currently active transactions will be continued after the switch, consistency between the tasks is maintained. An example for a consistent scheduling switch is the switch-over from day to night (reduced) operation. Tasks, which perform transactions, which are the same during day and night are not disturbed (they are scheduled in the same way during day and night), other tasks are substituted or eliminated in the new schedule.

Schedule switches are initiated by the arrival of messages to synchronize the points in time when they occur, Because messages are received simultaneously in all components, the switch occurs at the same point in time at all components.

2.6 Bus Scheduling

To be able to guarantee that all tasks meet their deadlines, it must be guaranteed that each task gets its input values in time. In MARS there is only one type of input values: standardized MARS messages. There are three types of messages in MARS, each type corresponds to the kind of distribution of the message in the system:

Internal Messages can be read only by tasks residing at the same component. An internal message is readable immediately after writing it, therefore it is sufficient to schedule the tasks in the right order (the task which produces the message before the tasks which consume the message).

Environmental Messages are produced or consumed by the environment. Since the environment can not be assumed to work correctly in all cases, the loss of an external message must be taken into account when designing the application.

Cluster Messages are produced by a task and can be read by every task in the same cluster. Because each bus has a limited transmission capacity, the protocol has to guarantee that each task receives its input messages in time.

The MARS bus is realized by a Cheapernet (like an Ethernet) cable. The medium access protocol is TDMA instead of the normally used CSMA/CD. Since TDMA guarantees each component to have access to the medium at equidistant points in time (the period, when a component may send a message is called its 'TDMA slot'), there is a constant number of messages, which each component may send in a time interval.

In conformance to the static task schedule there is a static bus schedule to avoid competing of the tasks of a component for the TDMA slots. Since the bus scheduling heavily influences the task scheduling and vice versa, both actions are made concurrently and they are supported by the same tool, the 'MARS pre-run-time scheduler' [Foh89].

3 Achieving Fault-Tolerance

3.1 Fault Detection

The fault tolerance of the MARS system is based on fail-silent components. It is assumed that a component produces either correct results or no results at all. Obviously, some mechanisms have to be implemented to achieve such a behavior.

Node failures can be partitioned into two classes: failures occurring during run-time of the operating system and failures occurring during run-time of application programs. Each time a new application is developed, the application programs have to be designed from scratch, whereas the system software is designed independently and used in each application. Therefore, from the point of view of the costs incurred much more effort can be made to increase the reliability of the operating system than the one of the application tasks. On the other hand, detection of faults inside the operating system is much more difficult than detection of faults in application programs. We face the following problems:

- The fault detection routines themselves are part of the operating system, therefore being a point of weakness in the system.
- Failures in interrupt handling routines are very difficult to detect, since they are detached from the normal control flow. They do not have input or output values, instead of this they work on global data structures or they interact with peripherals.
- Because of their interaction with peripheral chips, the results of operating system routines can not be easily checked, since every write operation to a peripheral chip is a '*result*'.

Operating System Fault Detection

The following mechanisms are used in the MARS operating system to increase the self-checking coverage [Dam88]:

- Usage of robust storage structures[1] to check the consistency of the system data structures during runtime of the operating system [Tay85]. Errors in the structure of lists, tables, trees,... can be detected.
- Massive usage of assertions for variables and parameters of functions. 'Firewalls' are built by checking the value ranges of variables.
- Plausibility tests for the results of system routines.
- Hardware mechanisms:
  - Parity bits for the memory can detect failures in the main memory and at the processor bus.

[1] A storage structure is called robust if some (specified) number of changes made to structural information can be detected

- Voltage and temperature control prevents operating of the hardware outside of its specification.
- Watchdog timer for access to the MARS bus. A simple circuit guarantees that each component has access to the bus only in its own TDMA slot.

Application Software Fault Detection

The usage of mechanisms like robust storage structures, assertions or plausibility tests will heavily increase the costs for application software development. However, unlike operating system routines, application tasks can be executed in redundancy without great effort. We use the following mechanisms to detect failures, which occurred during run-time of application tasks:

Run time control of tasks: During the scheduling phase of the application an upper bound for the worst case execution time (MAXT) of each task is calculated [Pus89], which becomes the length of the slot of the task. If a tasks exceeds its time limit, a severe error has occurred.

Time redundant execution of tasks: In most cases the actual execution time of a task will be significantly smaller than the calculated MAXT. The MARS operating system uses the period between the termination of a task and the end of its slot to execute the task a second time. Because the structure of each MARS task has the form input – computation – output, it is rather easy to provide the same input values to both instances of the task and to compare the output messages. Although this method does not guarantee to compute *each* instance of a task twice, a high percentage of tasks will be computed in redundancy, therefore the self-checking coverage of the components will increase without performance drawbacks. To be able to detect failures in the code segment of a task, each instance of the task will work on another copy of the core image. To handle permanent faults within the processor, the two core images are not exactly the same. The compiler can be forced to produce code with permuted registers, permuted program blocks, permuted data structures, ...

Code segment checking: The code segment of each task is provided with a checksum, which is checked periodically in the idle loop of the processor.

3.2 Fault Handling

3.2.1 Recovery

Whenever a component has failed, it shuts down and tries to restart to avoid disturbing of other components by its erroneous behavior. The startup routine tries to determine the cause of the failure. First, the component checks if there is a permanent hardware error. Extensive testing of ROM, RAM, the processor and the peripheral chips is performed. In addition, date and time of the last component errors are stored in a non-volatile RAM. If the rate of component failures exceeds a predefined number, the component is also assumed to have a permanent fault.

If the failure is assumed to be transient, the next step is to determine if the failure occurred in the code segment (by calculating the code segment checksums). When the code is assumed to be faulty, it is recovered by reading messages from a maintenance component, which periodically sends the code segments of each component at the MARS bus.

Given a correct code, the component tries to reintegrate itself into the cluster: After starting clock synchronization (see chapter 3.4) and membership service (see chapter 3.3), each task has to recover its inner state. Since each component is provided with a redundant one, this is a rather simple task. At the end of each cycle, each task puts its inner state into a MARS message, which is sent at the bus. At the beginning of its cycle, each task reads its own inner state message (or the one of the redundant component, whichever is available). Therefore the programmer need not take care of recovery phases of the tasks (see also chapter 2.2). After each task has restored its inner state, the component starts to work.

3.2.2 Reconfiguration

The time between the shutdown of a component and between start of full operation is critical to the reliability of the system (especially after the occurrence of a permanent hardware failure), because a single failure can crash the whole system during this period. There has to be a mechanism to prevent this. *MARS uses the concept of shadow components.*

During normal operation, the shadow component works in redundancy with two other active redundant cluster components (see figure 3). Differing from the other components, there is no TDMA slot assigned to the shadow component. It performs exactly the same actions as the two other components, except that it does not send any message at the MARS bus. In case of a failure of an active component – detected by the membership protocol – the TDMA slot of the faulty component will be assigned to the spare. This switchover is made immediately,

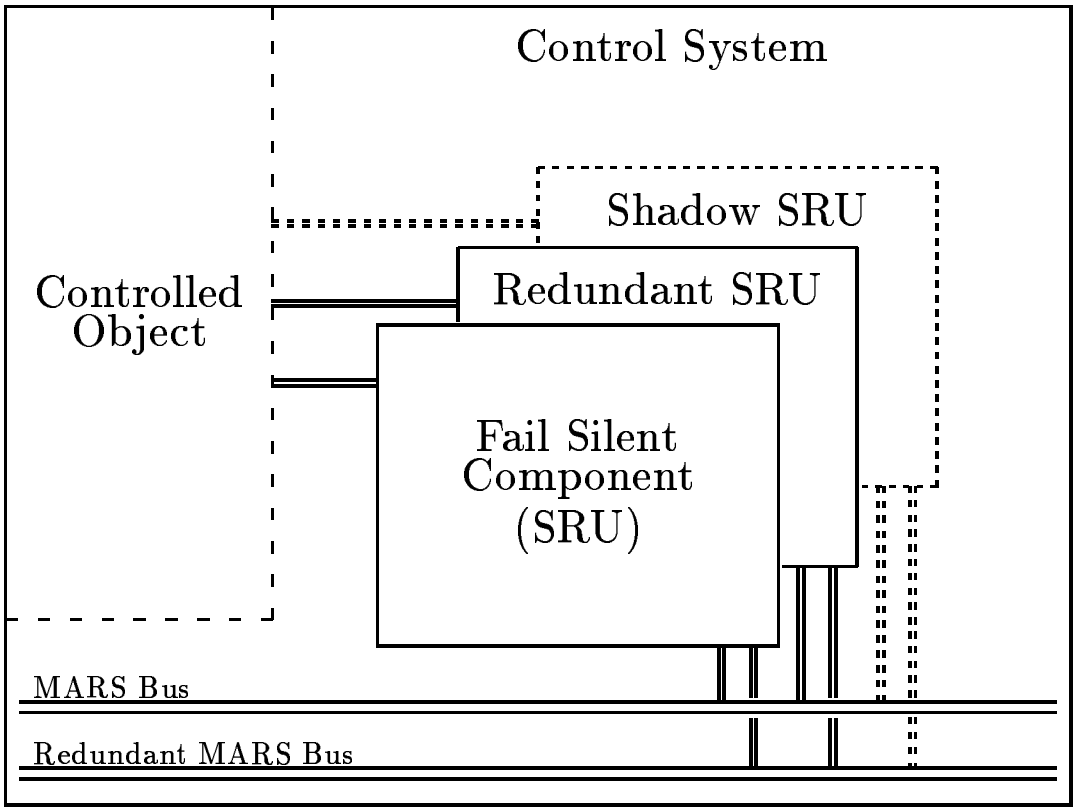

Figure 3: Component redundancy in MARS

because the internal states of the shadow component and the active components are equal.

3.3 Membership Service

In a distributed real-time systems knowledge about the operational state of the components is of significant importance, e.g. for maintenance activities, handling of component failures, etc.

In order to improve the self-checking coverage of components a component should turn itself off, if it is incapable of sending or receiving messages. A component has to have knowledge about the traffic on the bus and the behavior of the other components to detect this kind of faults. Without this knowledge a component which does not hear any traffic on the bus can never decide if there is no traffic, or the sender or the sender's interface to the bus or its interface has failed.

Therefore, in MARS a membership service is realized. It provides in a dis-

tributed real-time system knowledge about the operational state of all components [Kop89b]. The membership service operates on a system level where only two different operational states of a component are distinguished. Either a component operates correctly (active component), i.e. it operates according to its specification, or it stops its activity (inactive node), i.e. it terminates its service. This corresponds to the behavior of a fail-silent component as used in MARS.

The set of active components, called the membership set, will change over time. An active component may depart from the membership set (e.g. component failure, shut down itself) while an inactive node may join the membership set (e.g. repaired component) at a given point in time. The aim of the membership service is to guarantee that every active component at any point in time has a consistent and timely knowledge about the membership set in the recent past. In MARS the age of this information is two TDMA cycles, which is typically in the order of some milliseconds.

The realization of the membership service uses the advantage that each message sent has the semantic of a life sign inherently. Additionally, whenever a component broadcasts a message it includes information in the message header, whether it has received the last messages sent or not. Owing to the fact that all messages are broadcast messages each component gets these informations sent by all other components. By collecting and analyzing this information each component is able to determine the membership set and to examine if it is responsible for a message loss and therefore has to turn itself off.

3.4 Clock Synchronization in MARS

Each MARS component has its own real-time clock with a resolution of 1 μs. The fault-tolerant clock synchronization consists of two parts [Kop87]:

- The *internal synchronization* keeps clocks synchronized within a known constant.
- The *external synchronization* measures the difference between internal time and external time and it adjusts the drift rate of all clocks to match the progress of time of an international time scale, e. g. the Universal Time Coordinated (UTC).

We based the internal synchronization on normal message passing to avoid special hardware links for time signal propagation. Each message contains the time stamp of the sender's clock. The receiving component attaches the time stamp of the receiver's clock to each message. The time stamping is done by

a special VLSI chip as soon as the associated interrupt of the network interface arises. The time stamping mechanism is independent of the CPU load as well as the access protocol to the local area network. This method has a measurement error in clock comparison smaller than 4 μs, which is far better than methods implemented purely in software.

Based on this information, a correction term for the local clock is calculated with the *Fault-Tolerant Average Algorithm*. The clock differences measured at a particular component are sorted by value. The k lowest and the k highest values are discarded in order to tolerate up to k faulty clock values. The arithmetic average of the remaining values is the correction term for the clock of this particular component.

An instantaneous change of the local clock(s) would lead to errors in running measurements and disturb the time rigid scheduling. Our VLSI chip supports continuous time adjustment in hardware in multiples of 1 μs/s. Currently, we achieve a typical clock synchronization accuracy of better than 10 μs within a MARS cluster.

4 Outlook

In the past an experimental MARS system of 16 components has been established, where the basic architecture of MARS has been implemented. The old MARS components consist of a single board 68000 CPU with 512kB RAM, ROM, the VLSI clock synchronization unit, one Cheapernet interface and two serial lines. This serves as a testbed for experiments in the area of real-time and fault-tolerant systems. Currently, a new hardware is being developed which incorporates all mechanisms of fault-detection and fault-tolerance described in this paper. A double CPU architecture improves the performance of MARS significantly. One CPU is dedicated to the communication and all distributed system services, such as the clock synchronization or the membership protocol. The other CPU is free for the real-time application. The particular hardware chosen is a 68070 CPU and a future version of the MARS component based on 88000 CPUs is under consideration.

5 Conclusion

The design of hard real-time systems with guaranteed performance in order to meet all critical deadlines of the hard real-time transactions is a challenging and difficult task. If the transactions are scheduled dynamically at run time, it is very

difficult to guarantee that the specified deadlines can be met under peak load conditions. On the other hand, if the transactions are scheduled at compile time considering the worst load scenario, a very low utilization of system resources will have to be tolerated. It can be hoped that the improvement in the price performance ratio of the hardware will take away some of the economic pressure for high utilization in the future. A compromise between static and dynamic mechanism, e. g. the concept of a schedule switch, will increase the resource utilization to an acceptable level while still guaranteeing that all hard deadlines can be met.

Fast and efficient primitives in hard real-time systems are desirable but they certainly cannot guarantee the timely system responsiveness under all circumstances. By the design of MARS we have shown that a deterministic approach to fault-tolerant real-time systems is feasible. MARS allows ease of maintenance through a clear concept of smallest replaceable components, redundancy, and loosely coupled communication. The bounded validity of real-time data as well as the time rigid scheduling for tasks and communication provides the predictability of system activity which is a prerequisite for smooth reintegration of repaired components. Maintenance in a continuously running MARS system is no problem and the reliability of MARS can be made high enough to meet the stringent requirements of hard real-time control system with long mission times.

References

[Dam88] A. Damm. *Experimental Evaluation of Error-Detection- and Self-Checking Coverage of Components of a Distributred Real-Time System.* PhD Thesis, Technisch Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria, Oct. 1988.

[Dam89] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The Real-Time Operating System of MARS. *ACM SIGOPS Operating Systems Review*, 23(3):141–157, July 1989.

[Foh89] G. Fohler and C. Koza. Heuristic Scheduling for Distributed Real-Time Systems. Research Report 6/89, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, April 1989.

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

[Kop89a] 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.

[Kop89b] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System. In *Int. Working Conference on Dependable Computing for Critical Applications*, pages 167–174, Santa Barbara, CA, USA, Aug. 1989.

[Pus89] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[Rod89] M. G. Rodd and F. Deravi. *Communication Systems for Industrial Automation.* Prentice Hall, 1989.

[Tay85] D. J. Taylor and J. P. Black. Guidelines for Storage Structure Error Correction. In *Proc. 15th Int. Symposium on Fault-Tolerant Computing*, pages 20–22, Ann Arbor, Michigan, June 1985.