Distributed Fault-Tolerant Real-Time Systems: The Mars Approach

Real-time process control has to be timely and has to be right. Here's how one system promotes these features through a number of key functions.

Hermann Kopetz
Andreas Damm
Christian Koza
Marco Mulazzani
Wolfgang Schwabl
Christoph Senft
Ralph Zainlinger

Technische Universitat Wien

Most computer systems for real-time process control must meet high standards of reliability, availability, and safety. In many of these real-time applications, the costs of a catastrophic system failure can exceed the initial investment in the computer and the controlled object. To prevent such failures, system design must guarantee performance as specified in the domains of both value and time during all anticipated operational situations. The computer system must also be designed to tolerate faults caused by environmental disturbances or a physical degradation of the hardware.

Distributed computer-system architectures have gained general acceptance in the area of real-time process control. These architectures offer a significant potential for fault tolerance and functional degradation as well as for testability and extensibility. However, many of the distributed computer-system architectures presently on the market[1] or proposed by the research community as academic prototypes (the V-Kernel,[2] Accent,[3] or Chorus[4]) do not support some key points that are essential to reliable control of real-time applications. These points are

- limited time validity of real-time data,
- predictable performance under peak load,
- fault tolerance, and
- maintainability and extensibility.

Consequently, we developed the Maintainable Real-Time System, a fault-tolerant distributed system for process control. The Mars project started in 1980 at the Technische Universitat Berlin. The first prototype appeared in 1984 and demonstrated the fundamental concepts of Mars. The second academic prototype developed at the Technische Universitat Wien in Vienna has been functional since the beginning of 1988. Its main feature is predictable performance under a specified peak load. Its industrial applications include rolling mills and railway-control systems in which the controlled system imposes hard deadlines.

This article presents the Mars approach to real-time process control, its architectural design and implementation, and one of its applications. But first, let's explore the characteristics of distributed real-time systems as background to this discussion.

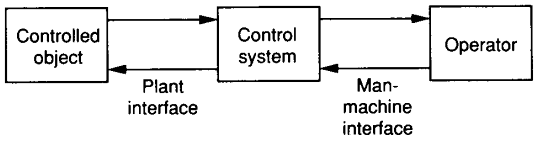


Figure 1. Real-time system.

Distributed real-time systems

In a real-time system, a controlled object (the controlled environment) and a control system (the computer) are connected via sensor- and actuator-based interfaces. The control system either accepts data from the sensors at regular intervals or is driven by events. It processes the data and outputs the results to the controlled object via the actuators. The output data influence the controlled object, and the sensors observe the effects, thus closing the loop as shown in Figure 1.

It is extremely important to avoid inconsistencies between the internal states of the control system, the control object, and the operator. The control system must respond to a stimulus from the control object within an interval dictated by the environment, called *response time*.

The system must guarantee this response time under extreme load and anticipated fault conditions. Typical systems respond in 1 millisecond to 1 second or more.

If a serious failure—either in the control system or the controlled object—closes down the plant, the system must shut down in a controlled, predetermined manner (fail-safe operation).

We considered the following characteristics to achieve the requirements of timeliness and high availability.

Correct versus valid information. In a real-time environment, one must access information in two domains: value and time. Information is *correct* if it corresponds with the intentions of the user. Information is *timely* if it is available within the intended interval of real time. Information qualifies as *valid* if it is both correct and timely.

In the nonreal-time world, we concern ourselves only with the value domain, that is, with correctness. The inclusion of the time domain in real-time systems adds a new dimension to the problem of providing valid information. The speed of processing in a given interval of real time (system performance) becomes an essential property.

Real-time versus archival data. A real-time control system performs real-time control functions and also collects data for archival purposes. We therefore distinguish between the two databases required for these purposes.

A *real-time database* consists of the set of data elements essential to instantaneous real-time control, operator display, alarm monitoring, and other real-time functions. An *archival database* includes the set of data elements required for archival purposes.

Major differences exist between these two databases from the point of view of time and fault tolerance.

The real-time database changes as time progresses, that is, the passage of real time invalidates the information in the database. Losing the real-time database suspends control of the environment.

After an element has been stored in the archival database, one cannot modify it again. It does not change as time progresses (it is not allowed to modify "history"). Losing the archival database does not immediately affect real-time control of the environment.

To correctly respond to an external or internal stimulus, the real-time database inside the control system must contain a valid (correct and timely) image of the external state of the environment.

Event-driven versus periodic systems. An event (state change) or a time signal initiates action in a real-time system. In *event-driven systems*, a state change in the environment or an external event (such as an interrupt) usually initiates activities spontaneously. In *periodic systems*, an equidistant time signal initiates all system activities at predefined points in time. The time period between related sequences of actions that are initiated by equidistant time signals is called a *duty cycle*.

If a system is driven by events, shielding it from faults in the environment can be very difficult. Spurious events can cause the initiation of more activity than the system was designed to handle. Any good design must contain mechanisms to protect the system from such conditions. One common hardware technique for the suppression of unwanted signals provides a low-pass filter.

In a periodic system, the response-time requirements of the given application under worst-case conditions determines the duty cycle (sample rate). System design guarantees that only a single event of a given type will be active within a given duty cycle. A periodic system has the advantage of implicit flow control and the protection of the system from (erroneous) overload conditions caused by a fault in the environment.

A maintainable system

This section outlines the architectural principles we followed in the design of Mars based on the principal considerations just described.

Design for peak load. In many real-time applications, one most urgently needs the services of the control system under peak-load conditions in which all stimuli (events) occur with maximum (but specified) frequencies. Consider the case of an air-traffic control system or a nuclear-reactor shutdown system. The systems must handle a peak-load condition without missing any hard real-time deadlines. If a system can do this, it can accommodate all low-load conditions automatically. Consequently, we designed hard real-time systems for peak loads.

Transaction orientation. Mars uses a transaction model to describe the activities of a real-time system. A *transaction* is the single execution of a specified set of tasks (generally in different nodes) between the stimulus and the corresponding response. We structured a transaction as an acyclic-directed graph with tasks that appear as nodes and messages that appear as arcs. If the corresponding response has to be produced within a given time interval after a stimulus, we have a *real-time* transaction.

A transaction transfers the system from one consistent state into another. In a distributed system, a transaction can decompose into a sequence of subtransactions consisting of executing tasks and communication phases between these subtransactions. An external event (generated by an external state transition) or an internal event (generated inside the computer system like a clock tick) can serve as the stimulus that initiates a transaction.

Network structure with clustering. The operational structure of a distributed, real-time control application consists of a set of *components* that form a network. These components include self-contained computers and the application *tasks*[5] that consume, process, produce, and interchange messages between components. The concept of *clustering* helps to manage the complexity of a large network of components. A Mars cluster is a subset of the network with a high functional connectivity. Clusters are the basic elements of our system architecture (Figure 2). Each cluster consists of several components interconnected by a synchronous, real-time Mars bus.

Global time. Distributed real-time systems require a common time base (called global time) of known synchronistic accuracy to measure the

- absolute time of an event occurrence,
- causal ordering of events,
- calculation of time intervals, and
- establishment of information consistency between the real-time database and the environment.

In Mars, the underlying architecture (operating system and hardware) provides a fault-tolerant, global time base called *system time.*[6]

Interprocessor communication. Mars introduces a new type of message for interprocessor communications called the *state message*. The semantics of a state message is similar to that of a global variable. A new version of a state message updates the previous version. State messages are not consumed when read. An arbitrary number of tasks can read a state message an arbitrary number of times. State messages exchange information about the state of the environment that has been observed at a given point in time and is assumed to hold for a certain interval of time. Since every change of state is an event, in principle, a set of (periodic) state messages can realize any information exchange.

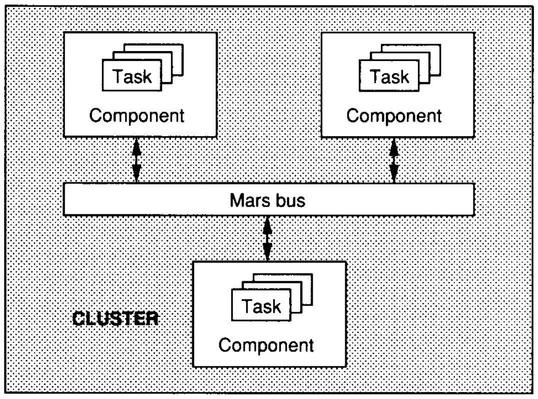


Figure 2. Mars cluster.

In real-time systems, the validity of information does not exclusively depend on its correctness in the value domain. This validity also depends on the timeliness of the information.[7] Each message has an attached *validity time*. As soon as the validity of a message expires, the operating system discards the message.

End-to-end protocols. Reliable communication can only occur with the knowledge and help of the application software residing at the endpoints of the communication system.[8] Dynamic time redundancy in the lower levels of the communication protocols increases communication reliability. However, this redundancy also leads to an uncontrolled increase in communication traffic under error and heavy-load conditions (for example, implementing Positive Acknowledge or Retransmission protocols). Real-time systems do not allow for a gain in communication reliability at the expense of unpredictability in communication delays. We feel that strict control over the communication traffic by the application software is necessary in real-time systems to meet the timing requirements. Mars provides merely an unacknowledged datagram[9] with a "no wait send" semantic as the communication service. That is, the sender does not wait until the receiver accepts the message. The (periodic) state messages realize implicit flow control between sender and receiver(s). One can derive the timing properties of the application from an analysis of the end-to-end protocols in the application software.

Dependability Analysis

Dependability[1] is the property of a system that allows reliance to be justifiably placed on the service it delivers. Dependability analysis concentrates on estimation and analysis of failures and their impact. Examples of quantitative measures are reliability and availability. Other measures depend on the type of application.

In Mars applications, two aspects are of special interest for dependability analysis. First, an *early dependability analysis* is preferable because of its cost-effectiveness. The later a design change becomes necessary, the more it costs. The dependability analysis requires close examination of the system, including failures and mutual dependencies of sensors, actuators, man-machine interface, operator, plant, and other subsystems.

The architecture shown in Figure A helps to structure the concepts, unify the methods and tools, and provide an easy framework for development. Expected long-term benefits include reusability, variability, and possible standardization of methods, interfaces, and tools. A layered architecture forms the underlying principle for the design and implementation of the Mars Reliability Predictor and Low-Cost Estimator (Marple).[2]

Marple fits perfectly into the concept of the contractual approach used in the Mars design-system environment. Figure B shows the interaction of the different programs and their layering according to the reference architecture.

One major difference exists between Marple and other tools for dependability analysis. Most tools require a model as their input. In contrast, Marple *generates* dependability models. It is a compiler, translating a general-purpose design language for

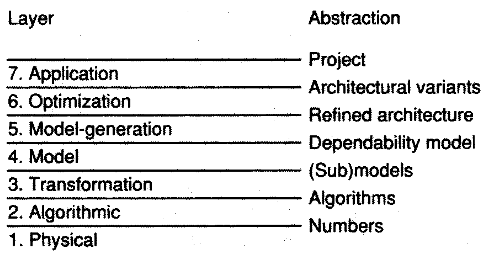


Figure A. Architectural layers and their abstractions.

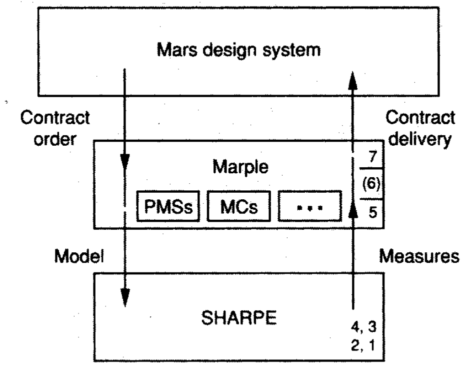


Figure B. Interaction of Marple with the design system and SHARPE.

TDMA media-access strategy. Since we designed Mars to master peak-load situations, its performance must not degrade because of variations in external-stimuli frequencies or message congestion on the real-time bus. The media access-delay time to the bus should be independent of bus activity. In Mars, a time-division, multiple-access strategy (TDMA) provides a deterministic, load-independent, and collision-free method for media access. The duty cycles of all tasks are synchronized in advance with the TDMA slots to optimize the system-response behavior.

Fault tolerance by active redundancy. The fault hypothesis in the Mars design covers permanent and transient physical faults in the components and on the real-time bus. Examples include transient faults caused by alpha particles or permanent faults caused by physical degradation of hardware components. Errors in the design and implementation of the software are not currently included in the present Mars fault hypothesis.

We assume that components possess self-checking properties[10] and fail silently, that is, that they either operate as intended or do not produce any results. The various inner-failure modes of a component thus reduce to a single-failure mode of the component from the standpoint of its environment: The component doesn't operate.

In Mars, fault tolerance relies on self-checking components that run with active redundancy. Fault tolerance also relies on multiple transmissions of messages on the real-time bus. Active redundancy better enhances reliability in hard real-time systems than passive redundancy because it has superior timing properties. As long as any one of a set of redundant, synchronized, self-checking components can operate, the required service can be maintained. Every message is transmitted n times, either in parallel over n buses or sequentially over a single bus (or a combination thereof). Therefore, the loss of $n - 1$ messages is tolerable. We have developed a special tool to assess the reliabilidistributed systems into corresponding reliability models. Marple concentrates on application-layer and model generation. The generated dependability models are then analyzed by the Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE),[3] which covers layers 4 to 1 of the reference architecture. Currently, the design is translated into SHARPE processor-memory-switch (PMS) models. The generation of models based on Markov chains (MCs) is under way. Future extensions may include generation of various other models, approximation methods, and/or usage of other tools for dependability analysis.

The application layer of Marple analyzes system designs. Marple receives a detailed description of the designed system in the form of a structured, text-oriented, specification language. The basic elements of this language are objects (such as clusters, sensors, operators, and tasks), information items (notions for the abstract entities of information exchange), and transactions (describing the functionality of objects) in a system-wide context. The power of the input language stems from the ability to combine these elements hierarchically in a flexible way.

The general description of the design is augmented with dependability data including

- redundancy,
- failure and repair distributions,
- failure modes of the elements,
- coverage values,
- cost functions of failures (in preparation), and
- repair dependencies (in preparation).

For more complicated subsystems, one can define submodels (reliability block diagrams, MCs, series-parallel graphs, and combined performance and reliability models). On the other hand, one can also use default values for the different types of elements of the system. Due to this general design language, Marple is independent of the Mars design-system environment in terms of such features as a specific database or user interface.

Our first experience in using Marple produced an interesting effect. Although Marple was intended as a reliability predictor, the term "unreliability predictor" proved more appropriate. Systems without massive redundancy of sensors, actuators, and computers expose poor reliability behavior due to their wide-spread dependencies. Feedback occurs immediately (not after two months of reliability analysis) that the actual design is unreliable. This information forces the designer to

- think about failures,
- decide which functions are important,
- analyze which parts are important, and
- determine where and to what extent redundancy should be applied.

Thus consideration of faults and their impact becomes an integral part of each design step.

References

1. A. Avizienis and J.C. Laprie, "Dependable Computing: From Concepts to Design Diversity," *Proc. IEEE*, Vol. 74, No. 5, May 1986, pp. 629-638.
2. M. Mulazzani, "An Open Layered Architecture for Dependability Analysis and Its Application," *Proc. 18th Fault-Tolerant Computing Symp.*, IEEE CS Press, Los Alamitos, Calif., June 1988, pp. 96-101.
3. R.A. Sahner and K.S. Trivedi, "Reliability Modeling Using SHARPE," *IEEE Trans. Reliability*, Vol. 36, No. 2, June 1987, pp. 186-193.

ty of a given Mars application. For more details, refer to the box entitled Dependability Analysis.

Maintainability and extensibility. In Mars, maintainability and extensibility result from the clustering of components. One can remove redundant components from a running cluster (for repair) and reintegrate them later. A Mars cluster can be configured with spare capacity in TDMA-bus access, messages, and CPU utilization. One can add new components without modification of the running components until the configuration limit is reached. If more extension is needed, existing components can expand into a cluster. This process converts one component in the original cluster to an interface component showing the same I/O behavior as the old component. This interface component forwards all messages to the added cluster (Figure 3). The additional cluster can be designed independently from the rest of the system as long as the I/O characteristics of the interface component remain unchanged.

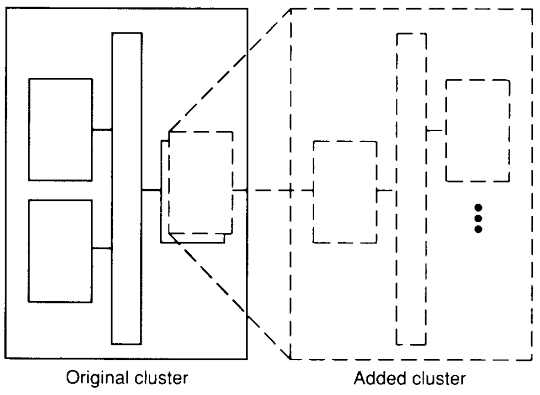


Figure 3. Expansion of a component into a cluster.

Mars operating system

We developed a new operating system to implement the Mars architecture and to guarantee a deterministic system behavior. An identical copy of the operating system runs locally and autonomously in each Mars component, which is a hardware/software unit as shown in Figure 4. The operating system consists of a small kernel and a set of system tasks.

The kernel consists of the entire code running in supervisor mode on the CPU. The kernel's primary goals are

- administering resources (CPU, memory, bus), and
- hiding all hardware details from the tasks.

The kernel provides its functionality via a set of defined system calls. The interfaces of the system calls—as well as major parts of the kernel—are written in the C programming language. Adapting the kernel alone lets Mars port to a new hardware environment. The kernel is responsible for the periodic execution of hard real-time (Hrt)—or time-critical—tasks according to a schedule calculated off line. (The section on timing analysis describes the off-line task and message scheduler.) The kernel also maintains global time and oversees the efficiency of message passing.

Real-time tasks. Hrt tasks are periodically scheduled and must terminate before a given deadline. Thus their reaction time and latency must be deterministic and known in advance. An off-line scheduler calculates the activation periods (duty cycle) based on the transaction specification during the system-design phase. The kernel executes these tasks according to the results of the off-line scheduler.

Most Hrt tasks are *application tasks*, but some of them are *system tasks*. System tasks perform specific, hardware-independent functions of the operating system. These functions include time synchronization and protocol conversions to and from RS-232 strings and Mars messages, for example. Privileged system calls are restricted to system tasks.

All tasks that are not subjected to strict deadlines are called *soft real-time tasks*. Normally an Srt is nonperiodic and utilizes the CPU idle time in low-load situations.

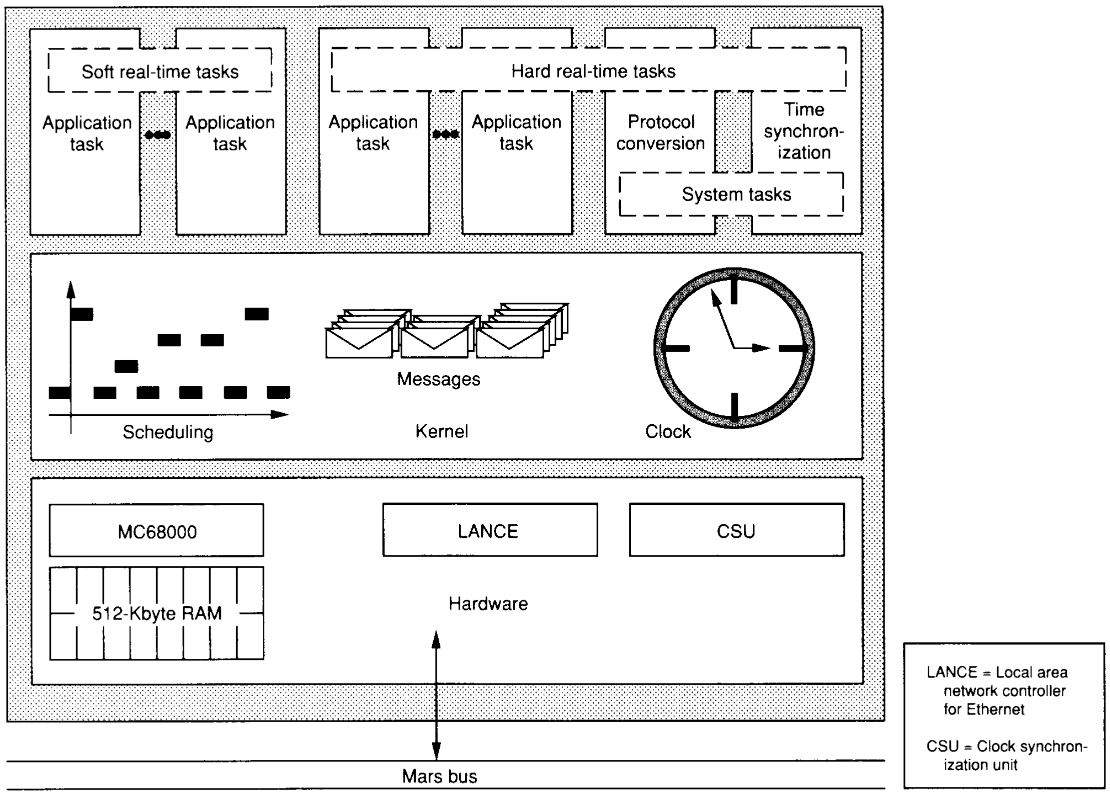


Figure 4. Structure of the current Mars component.

Clock Synchronization in Mars

Each Mars component has its own real-time clock with a resolution of 1 μs. Clock synchronization consists of two parts. *Internal synchronization* keeps all clocks within a cluster synchronized within a known constant, the internal synchronistic accuracy ΔD^{int}. *External synchronization* adjusts the clocks of a cluster to international atomic time. IAT is a physical time measure. It does not suffer from switching seconds due to irregularities in the rotation of the earth.

We based 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 incoming message. Each component records the time differences to the other components periodically. Based on this information, a correction term for the local clock is calculated with the *Fault-Tolerant Average Algorithm* (Fta). In the Fta, an ensemble of n clocks may contain up to k faulty clocks. The local clock differences d from clock i to clock j are sorted by value. The k lowest and the k highest values are discarded. The arithmetic average of the remaining values is the new correction for the local clock j. According to theory,[1] an upper bound for the internal synchronization Δ^{int} is given by

$$\frac{\Delta^{int}}{\epsilon+\xi} = \frac{n-2k}{n-3k}$$

where ξ is the maximal tolerable drift of the clocks during a resynchronization interval. The *reading error* ϵ is the measurement error in reading the time of one component by another component. Due to the cooperation between the clock synchronization unit (CSU) and the Mars bus-controller chip LANCE (see Figure 4), the reading error in Mars is bounded (with 4 μs) as explained later.

Figure C schematically shows the time-stamp mechanism for messages in Mars. The CPU places

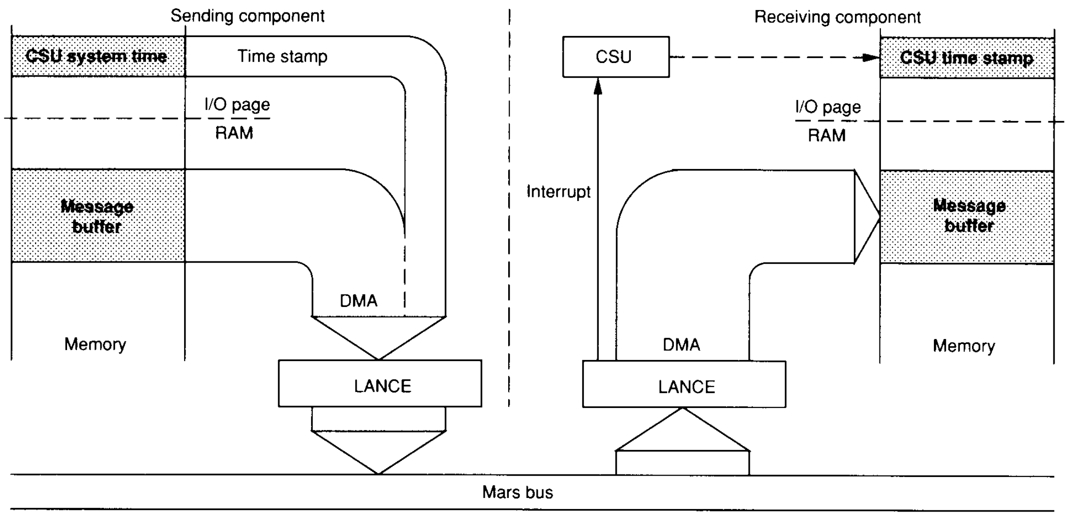


Figure C. Time-stamp mechanism in Mars.

Global time. The operating system maintains synchronized global time. The synchronization of local clocks is based on message exchange[6] and is supported in hardware by the clock synchronization unit (see the accompanying box on that topic). We developed this special very large scale integration unit especially for this project. Using global time helps to implement the following Mars features.

Validity. The system provides tasks with messages only if their validity time has not expired when the tasks are delivered. A task initiates the reading of a message, and the operating system checks to see whether the message is valid. The sender of a message must define a validity time. The operating system internally and automatically discards outdated messages. Thus, every task observes only valid messages.

waiting messages in a buffer, and LANCE starts. LANCE transmits the data to the Mars bus by direct memory access (DMA) after this access to the bus has been granted. LANCE can package several memory fragments continuously into one message. The last fragment of each message is a memory-mapped, real-time register of the CSU that is accessed at the moment of sending. At the receiver LANCE issues an interrupt immediately after a message arrives. This interrupt is directed to the CSU, which generates a time stamp. Afterwards, the CPU stores the CSU time stamp into the received message.

We use external synchronization to calibrate to IAT. Long-wave radio signals provide an economical access to world time, or universal time coordinated. Since UTC and IAT differ an integral number of seconds (in 1988, IAT − UTC = + 24.0 seconds) and the Bureau International de l'Heure publishes the time differences between UTC and IAT in advance, any UTC receiver provides a source for IAT.

Each Mars cluster contains a component with access to a time standard that measures the deviation between the cluster's time and the world time. The external clock-synchronization task broadcasts an appropriate rate correction that affects the speed of all internal clocks independently of corrections due to internal-clock synchronization.

An instantaneous change of the local clock(s) would lead to errors in running measurements and disturb the periodic schedules. So the CSU supports continuous time adjustment in hardware in multiples of 1 μs/s. Currently, we can achieve a typical clock synchronistic accuracy of better than 10 μs within a cluster consisting of eight components and 100 μs between two clusters.

References

1. H. Kopetz and W. Oshsenreiter, "Clock Synchronization in Distributed Real-time Systems," *IEEE Trans. Computers*, Vol. 36, No. 8, Aug. 1987, pp. 933-940.

Schedules. Task and message schedules are calculated off line and interpreted at runtime. The schedules are a periodic function of global time, and thus all tasks within a cluster are synchronized according to the communication and transaction description specified within the Mars design system.

Consistency. A real-time database consisting of valid messages naturally changes with time. The real-time database of a component is stored in message lists in the kernel. Any new message—regardless of origin—alters the database. However, the message lists in the kernel are updated solely during the clock-interrupt routine. Thus the changes of message states occur simultaneously in all running components. After such updates, the real-time database remains constant until the next occurrence of a clock-interrupt routine. Therefore, tasks operating in different components receive the same input if they read a message with the same name at the same time.

Self-checking features. As stated, self-checking components produce either correct results or no results at all. Ideally, all messages sent by a self-checking component are correct. The processor itself detects errors within a Mars component at the level of the operating system. The kernel must perform numerous checks to prevent erroneous messages from being sent to other components.

The mechanisms within the Mars operating system check both the correctness of information in the value domain and its validity in the time domain. Checks in the value domain include plausibility tests and time-redundant execution of tasks. In the time domain, these mechanisms check runtime limits, global time limits, and the timing behavior of the tasks with respect to the timing requirements of the controlled system.

When the operating system detects an error, it attempts to logically turn off the component regardless of whether the fault is transient or permanent or has occurred in hardware or software. The operating system fails silently to prevent error propagation within the cluster and to provide fault isolation. We conducted numerous experiments with specially designed, fault-insertion hardware to evaluate the self-checking features of Mars components

Message passing. Mars messages are sent as periodic real-time datagrams that each contain a validity time. These messages carry status information only. State-message semantics provide the following advantages for operating-system design.

Flow control. A periodic state message implicitly controls flow in the duty cycles of involved senders and receivers. The off-line scheduler synchronizes sending and receiving rates. Even in case of a fault (a sender is too fast or a receiver too slow), a buffer overflow in the operating system is impossible due to the overwriting of previous state messages by more recent instances.

Message redundancy. Backward-error-recovery protocols usually delay communication unpredictably. Thus these protocols can implement massive redundancy at the message level. Each message on the Mars bus is sent twice or more, depending on the fault hypothesis and the transient-failure probability of the bus.

The Mars bus is a Cheapernet version in which we measured an experimental message-loss probability of $1{:}10^5$. Sending each message twice decreases the message-loss probability to $1{:}10^{10}$ (in case of statistically independent failures), which is comparable to the failure rates of the component hardware.

Support of component redundancy and recovery. Two or more components running in active redundancy produce logically equivalent messages. Due to the state semantics of messages, only the most recent of two or more valid messages must be stored at the operating-system level. The filtering of redundant messages occurs within the kernel.

The operating system does not note the actual number of receivers or senders of a Mars message. Thus, one can insert redundant components into an operational system without any reconfiguration, modification, or notification of the running components.

As explained, every message contains a validity time to limit its lifetime. If a new component is added to a running system, it needs to fetch the real-time database. Since the message lifetime has a global upper bound, a new component collects messages only for this maximum period. After that, the recovery of the real-time database completes, and the new component can start its activity.

Efficiency issues. Any real-time system must guarantee reaction or transaction times. Therefore, it is reasonable (but not sufficient) to optimize certain operating-system procedures. This process requires the efficient handling of peak loads even at the expense of degrading the performance under average loads.

Interrupts. Only the real-time clock can interrupt the CPU. Other interrupt routines are disabled or used for time-stamping mechanisms by the CSU chip. The clock-interrupt routine periodically polls all peripherals—even the serial input-output chip for an RS-232 line. Thus the operating system is time rigid and deterministic in its kernel.

Path length. The application-independent overhead involved in message passing, measured in number of executed assembler instructions, is known as *path length*. The total path length for sending and receiving a message can be 20,000 to 50,000 in an Open Systems Interconnection protocol.[11] In Mars we keep the total path length as short as possible. When a task sends or receives a message, only a pointer to the message exchanges with the operating system. Any physical message copying is avoided. Otherwise the CPU would totally occupy itself with copying messages.

Message buffers. We kept constant the number of buffers needed within a component to overcome the buffer-allocation problem. We used the following method. All tasks must have preallocated message buffers before they can send or receive a message. A task must return one of its message buffers to the kernel when it receives a message. Similarly, it receives a new buffer when it sends a message.

Schedule switch. Process-control systems exhibit mutually exclusive phases of operation and control. For example, the start-up of an industrial plant can consist of a complicated procedure that is quite different from the control of a production line or an emergency stop of a machine. The overhead for control changes must be kept low, while the reaction time in emergency situations must be short.

In Mars, introducing a set of schedules solves this problem. All tasks for all schedules must reside within a component, but only one schedule activates at a given point in time. In case of a phase change or an emergency, a Mars message can trigger a simultaneous switch to another schedule in involved components. The reaction time to switch to a new schedule equals the time until the next clock-interrupt routine.

Timing analysis

One can guarantee the predictability of the time behavior of real-time systems only if the peak-load conditions of the system are known before runtime. This requirement necessitates using a static set of tasks for a component. From the aspect of timing analysis, such a static set of communicating periodic tasks are described by their duty cycle, their *maximal execution time* (Maxt), and the component in which they are executed. The designer specifies all these attributes through the Mars design system (see the box on the next page). This system provides all relevant data for timing analysis.

The TDMA slots of the Mars bus are assigned to the components in round-robin fashion. If a receiving task of a message resides on a component different from that of the sending task, that message must be broadcast (or exported) on the Mars bus. Each exported message requires a TDMA slot of the component running the sender task.

The tasks of a transaction in their entirety represent the implementation of a required stimulus-response action. The timing requirement of such a transaction is expressed by its *maximal response time* (Mart). A Mart includes

- the latency between the stimulus to the first task that uses this stimulus,
- a Maxt of the tasks involved,
- the communication times between these tasks, and

The Mars Design System

To control the design of Mars objects—such as transactions, clusters, components, tasks, and messages—we developed a special design methodology supporting both the creation of a Mars application and its evaluation. This computer-aided, real-time design methodology allows cost reduction and fault minimization in the development of critical and complex real-time applications.

The Mars design system does not view design activities and the support of a special design methodology as isolated problems. Respectively, the system does not provide isolated tools but rather integrates them into a *coherent design environment*. Our approach covers management aspects and life-cycle support of system development right from the beginning. These environments are termed *method-based* environments in the classification system developed by Dart et al.[1]

Figure D gives an overview of the entire tool system.[2] Its structure contains two dimensions:

- programming in the large and programming in the small, and
- design creation and evaluation.

The term *programming in the large* covers the phases from the specification of requirements and overall system design to the specification of the component level characterized by tasks and messages. Programming-in-the-large design steps manage design creation and its evaluation before coding takes place. *Programming in the small* is concerned with the inner construction of tasks, their implementation, and programming issues.

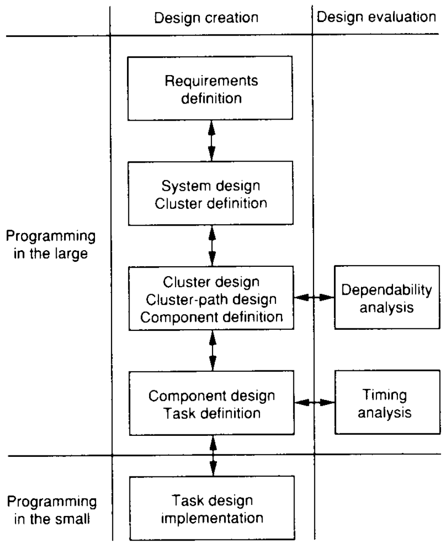

Design creation
Design evaluation
Requirements definition
System design Cluster definition
Programming in the large
Cluster design Cluster-path design Component definition
Dependability analysis
Component design Task definition
Timing analysis
Programming in the small
Task design implementation

Figure D. The tool system.

The second dimension distinguishes between *design-creation* tools and *design-evaluation* tools. Design-creation tools support the system analyst in the creation of a distributed real-time application. The activities in each step are detailed in Senft.[3] Evaluation tools analyze a given design and verify the proposed requirements. The dependability-analysis tool analyzes the system structure and possible failures. It computes measures for reliability, safety, and availability of the entire system (or system parts). The timing-analysis tool concentrates on the pre-runtime, static scheduling of the designed tasks and messages. If a schedule exists, the programmer has to code the tasks to meet specified execution times and interfaces. A key concept of the design environment is that the design-verification phases precede the coding activities.

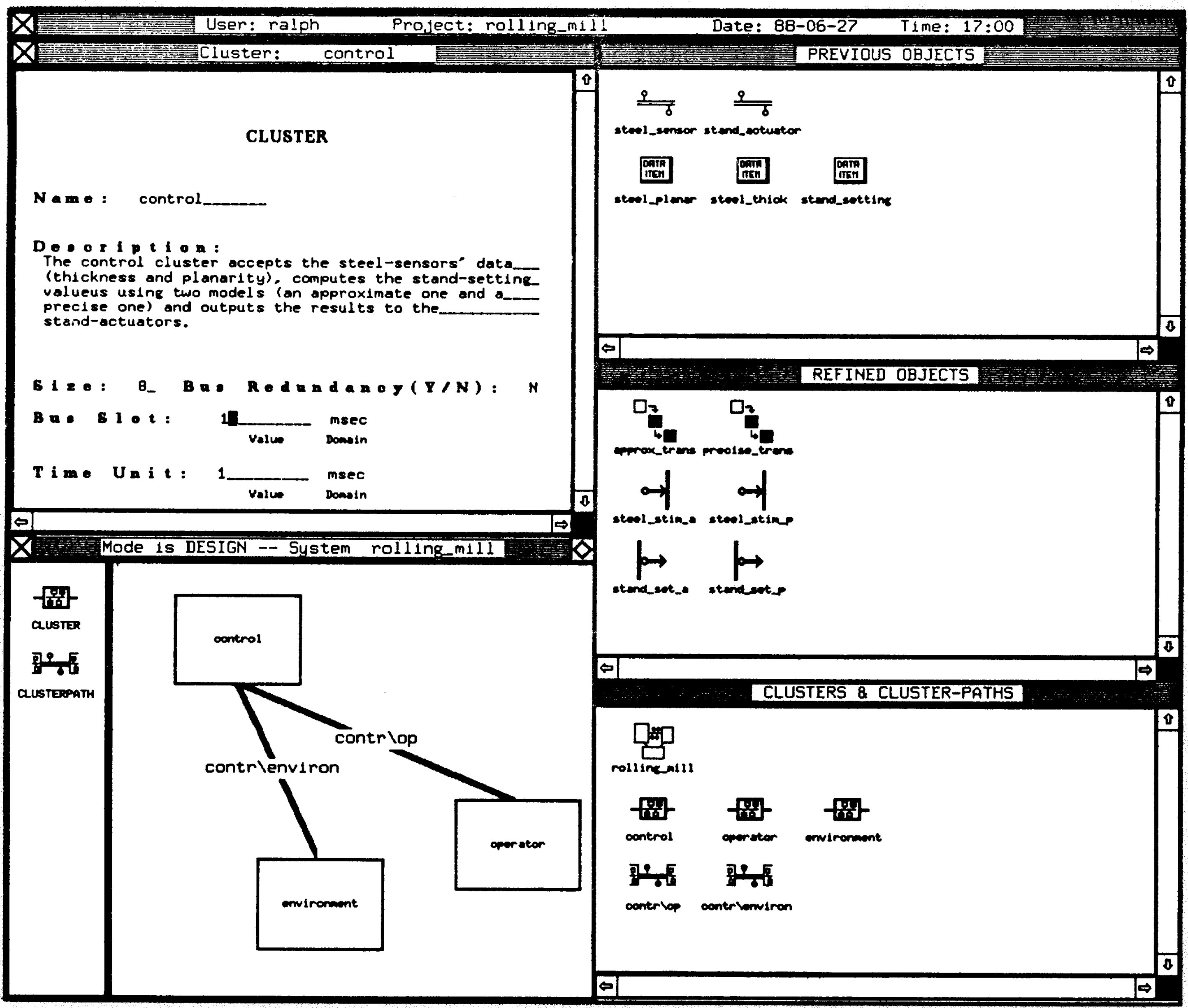
Each tool maintains its own local information base. The highly interactive design-creation tools must provide the engineers with efficient storing and retrieval mechanisms of the designed information as well as with consistency and integrity checks. Relational database systems offer these services. Actually, we used an extended relational database approach to manage large objects of variable length. For the first prototype, we employed the relational database management system DB++,[4] in connection with the Unix file system. (The current version comprises 103 relations, each containing an average of 5.2 domains.)

Strictly defined interfaces control data exchange between tools. The management structure of a contract[5] handles these interfaces, more explicitly the inputs, outputs, environmental data of a tool, and the management of the project members. The fundamental prerequisites for reasonable design consist of well-defined activities and clear assignment of responsibilities to members at all times. A tool and its local information are based on the principle of an information-hiding module; the contract is based on the principle of an abstract interface.[6]

In principle, each tool can execute on a different machine, thus supporting project members in various organizational or geographical entities. Tools (and project members) exchange information via contracts. The entire setup is Unix-based. Hence, contract passing is embedded in standard Unix mail and is transparent to the user.

We developed the highly interactive, design-creation tools under the X Window System. Their user interface is uniformly organized. Tiled windows allow engineers to concentrate on their design activities without having to waste time on window management and hidden-window searches.

Figure E illustrates the user interface of the sys-



CLUSTER

Name : control_______

Description :
The control cluster accepts the steel-sensors' data___
(thickness and planarity), computes the stand-setting_
valueus using two models (an approximate one and a____
precise one) and outputs the results to the___________
stand-actuators.

Size : 8_ Bus Redundancy(Y/N) : N

Bus Slot : 1________ msec
Value Domain

Time Unit : 1________ msec
Value Domain

Figure E. User interface of a Mars system-design tool.

tem design for a rolling mill controlled by Mars (discussed elsewhere in this article). The right side of the screen shows objects in an iconic representation. The upper right window contains objects—that is, transactions, spots, and data items resulting from a previous design step (in this case from the requirements-definition phase). We refined some of them to further subobjects (transactions to subtransactions), which are displayed in the middle right window. The lower right window displays new objects such as cluster and cluster paths that have been designed at the current design level. The left side of the screen is reserved for design and development in a textual and graphical manner. The window in the upper left half shows a textual representation of the currently treated object, while the window in the lower left part offers the following functions:

- definition and refinement of objects with a built-in graphics editor,
- display of several decomposition and relationship diagrams,
- graphical support for establishing relations between objects, and
- document preparation.

The interaction between designer and tool is mainly managed by dragging icons to the different windows. This process of combining an icon with a window invokes a predefined action. For example, moving a cluster icon into the text window displays an editable template for gathering the information relevant to a cluster.

References

1. S. Dart et al., "Software Development Environments," *Computer*, Vol. 20, No. 11, Nov. 1987, pp. 18-28.
2. C. Senft, "A Computer-Aided Design Environment for Distributed Realtime Systems," *Proc. IEEE Compeuro 88, System Design: Concepts, Methods and Tools*, IEEE CS Press, Los Alamitos, Calif., Apr. 1988, pp. 288-297.
3. C. Senft, "Remodel—a Realtime System Methodology on Design and Early Evaluation," *Proc. IFIP Conf. Distributed Processing*, Oct. 1987, pp. 305-321.
4. M. Agnew and R. Ward, "The DB++ Relational Database Management System," *Proc. European Unix Users Group (EUUG) Conf.*, Apr. 1986, pp. 1-15.
5. M. Dowson, "Integrated Project Support with Istar," *IEEE Software*, Vol. 4, No. 6, Nov. 1987, pp. 6-15.
6. D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1,053-1,058.

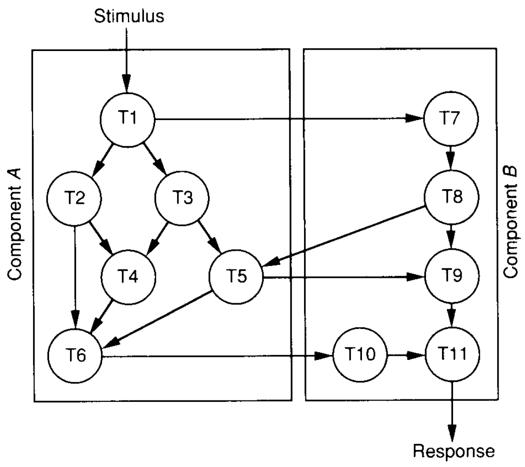


Figure 5. Communication structure of tasks T1 through T11 in a transaction.

- the time from the end of the last task until the response.

Here we assume that all tasks in a transaction are allocated to components within a single cluster.

As soon as the designer has refined the transactions to a task-message system and has estimated the Maxts of the tasks, the off-line task and bus scheduler calculates a preliminary schedule for the designed tasks. As the task implementation proceeds, a value derived from the actual source code using a source-level, execution-time analysis tool replaces the estimated Maxts. If no schedule for the task set can be found to guarantee the specified transaction times, the design needs revision.

Off-line task and bus scheduling. The schedules for tasks and messages are calculated by the off-line task and bus scheduler before runtime and stored in a runtime scheduling table. Figure 5 shows task transactions executed on components *A* and *B* and their internal and external communication. Figure 6 outlines a possible schedule for this example of a transaction. Because all tasks of components *A* and *B* have the same time period, both component cycles equal the period of the tasks.

For the bus schedule of the entire cluster, we assume a system with eight components. Only the TDMA slots assigned to components *A* and *B* are marked explicitly. The TDMA slots not available for components *A* and *B* are marked with the letter *X*. The schedule of the tasks within a component satisfies the precedence constraints according to internal message exchange. Figure 6 shows only the exported messages that are of interest for the bus schedule. The CPU schedule must be synchronized to the bus schedule to minimize the communication time between tasks of different components.

Because only one message can be sent per TDMA slot, exported messages have to be scheduled appropriately. For example, if TDMA-slot B_2 where task T8 sends its message were already used by a T7 message, the T8 message would have to be scheduled for another bus slot available for component B (B_3 in our example).

In a first attempt, we developed a two-pass scheduling algorithm. The first pass tries to minimize the communication times between the tasks allocated to the same component. Tasks executed on one component have a higher interconnectivity. We used a modified version of the CP/MISF (critical path/most immediate successor first) strategy[12] for this purpose.

In the second pass, a TDMA slot of its component is reserved for each exported message of a task. The schedules of all components shift to minimize the communication times between tasks of different components. Koza provides a detailed description of this two-pass sceduling algorithm.[13] This algorithm attempts to reduce the communication times in a cluster. Totalling the execution and communication times of the tasks involved in the transaction verifies whether the Mart of the transactions can be met.

To improve schedule generation, we developed an off-line algorithm that better accounts for the Mart of a transaction. This algorithm is based on a heuristic search strategy.[14] A heuristic function that calculates task urgency according to estimates of the time necessary to complete the transaction controls the inspected part of the search tree. Zhao, Ramamritham, and Stankovic describe an algorithm that works similarly on a problem that is quite different because it does not take precedence relationships between tasks into account.[15]

Source-level, execution-time analysis. Because the Maxt estimation of real-time tasks is critical, a special tool that derives the Maxt of a task from its source code supports designers. The estimation of the execution time of tasks requires bounded loops and prohibits recursions.[16] Maxt estimates must be very close to real Maxts. If the estimate is too high, excess time reserved for task execution unfavorably reduces CPU activity. Good Maxt estimations require program constructs that allow programmers to exploit knowledge about the execution of their algorithms.

Processor performance also affects the execution time of application tasks, and designers must consider this during timing analysis. On each clock tick a known amount of time is used to dispatch tasks and administer incoming and outgoing messages. The percentage of CPU time that is not available for application tasks is expressed as the CPU availability factor. This factor can vary according to hardware devices connected to the component and their generated DMA load. This factor also makes it possible for the scheduler to adapt the theoretical Maxt of each task in a component to an effective Maxt.

A typical Mars application

This section discusses the control of a rolling mill, which produces metal plates and bars.

Production control proceeds as follows. A sheet of steel passes through three pairs of rolls with a speed of about 10 meters/second (see Figure 7 on the next page). A sophisticated sensor system periodically measures the thickness and planarity of the surface of the steel. The raw sensor data are processed in node D (processing time, 2 milliseconds) to generate a periodic message

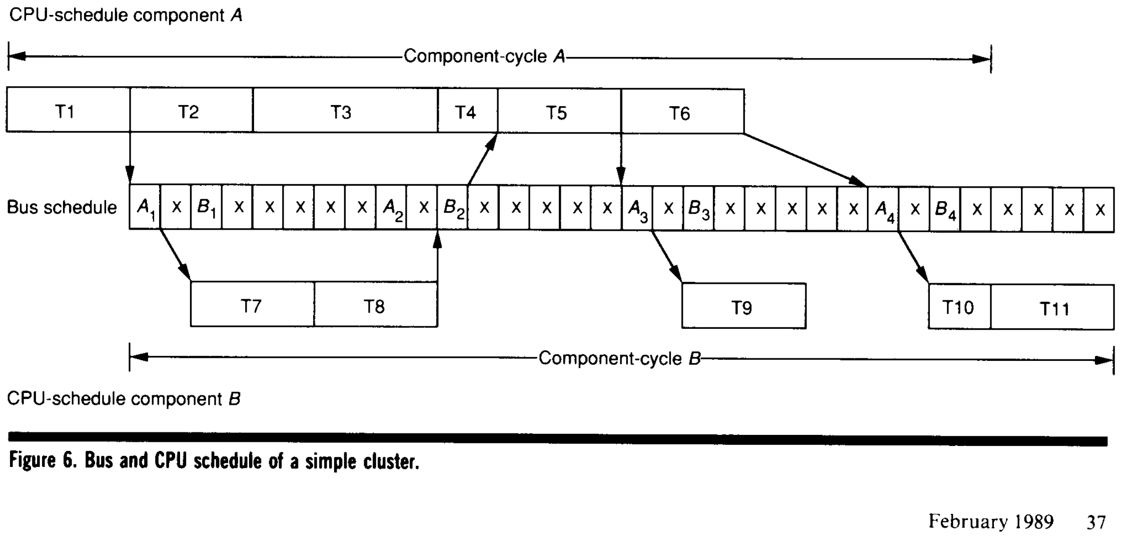


Figure 6. Bus and CPU schedule of a simple cluster.

M_1 describing the quality parameters of the product. This message is broadcast on the Mars bus. Nodes *G* and *H* accept M_1 from the bus and calculate the set-point vectors for the stands. Node *G* contains a simple model that always produces an approximate result message M_2 ahead of the deadline (say 10 ms). Node *H* contains a sophisticated model that produces a better result M_3 than node *F* but sometimes misses the deadline.

The results of both model calculations are broadcast on the bus. If the "better" result, M_3, has not arrived until the specified deadline (or does not arrive due to a failure of node *G*), nodes *A*, *B*, and *C* take the approximate result, M_2, from node *G* and position the stands accordingly. Furthermore, nodes *A*, *B*, and *C* collect operating data from each stand and distribute them on the Mars bus. The operator station *E* can access any real-time data from the Mars bus without interfering with the operation of the other components. It can also output the data in an appropriate format on a man-machine interface. Node *F* collects relevant information about the operation of the process and transmits it to other computers.

In this application, the real-time clock in node *D* triggers the most important time-critical transaction. It visits nodes *G* and *H* in parallel before delivering the results to nodes *A*, *B*, and *C* in parallel. The response time of this transaction has an important influence on the quality of rolling-mill control. Since this transaction is scheduled without any unnecessary delay, it takes 2 ms of processing in *D*, 1 ms for transport on the Mars bus to *F* and *G*, 10 ms of processing in *F* and *G*, and, finally, 1 ms for transport to *A*, *B*, and *C*—for a total of 14 ms. This time is about an order-of-magnitude faster than the response time achieved with a standard, central real-time computer. (A single computer has to perform all the parallel tasks in a timesharing mode that are assigned to eight nodes in the Mars distributed solution.) Exploiting the parallelism inherent in Mars' distributed architecture achieves a significant degree of its acceleration.

We configured the message formats, the CPU slots, and the TDMA protocol with spare capacity in the original design. As long as any change can be accommodated within this spare capacity, no modification of the schedules is required. If, however, a more powerful model is developed that cannot be processed on node *G* within the allocated time of 10 ms, that node can expand into a new cluster. This cluster increases the processing power manyfold without changing the interface (value and timing) to the original cluster. If the incorporation of fault tolerance is envisaged, additional slots on the TDMA bus for the redundant components must exist. As long as any of the redundant components is operating, the system provides the intended service on time.

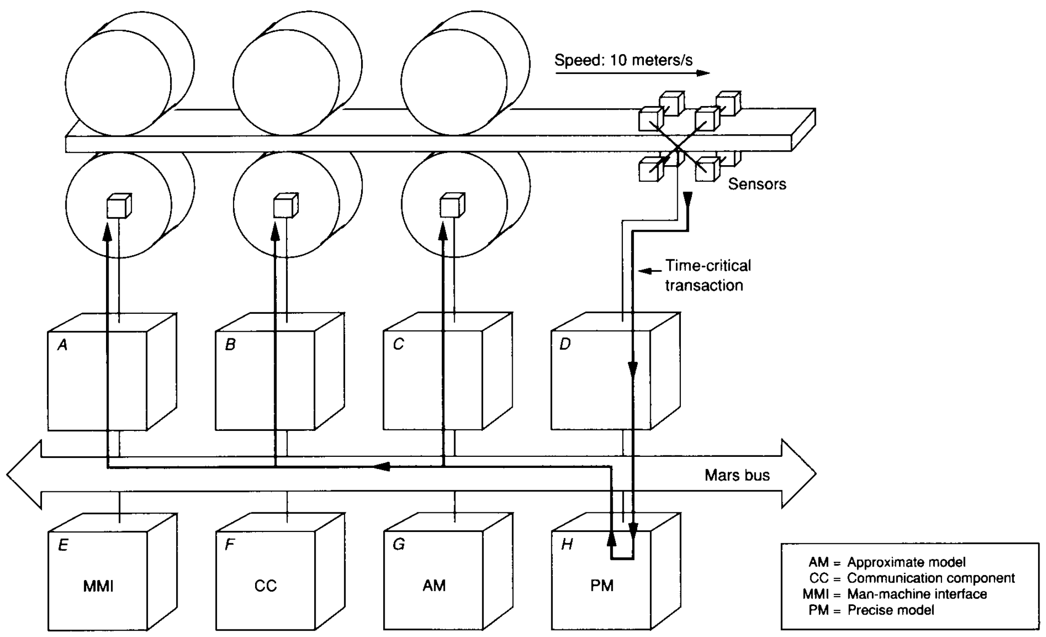


Figure 7. A rolling mill controlled by Mars.

The Mars project presents a solution to the problems of distributed, fault-tolerant, real-time systems. Mars copes with the needs of hard real-time systems in a number of ways. Each application for Mars is evaluated during its design and implementation phase with respect to runtime behavior to guarantee its transaction times. Users must know the frequency of all control procedures and involved reaction times in advance and realize them with an appropriate off-line schedule for the CPU and bus access. Each schedule is a periodic function in time, and the entire operating system is consequently driven only by time and not by events. Thus unexpected events cannot disturb the system behavior, which guarantees a time-rigid determinism.

The granularity of the system time is 1 μs. The system time is synchronized with an accuracy of 10 μs among the components of a Mars cluster. Scheduling decisions with a granularity of 8 ms achieve synchronism in the distributed system.

Mars messages have a fixed length and header format for standard interfacing to application tasks and intelligent process-control peripherals. Mars messages are state messages, that is, a more recent instance of a message updates the older one (comparable to global variables) to overcome flow-control and buffer-allocation problems. Mars messages have a validity time attached, after which the operating system automatically discards the message. Thus only valid messages arrive at the application, which guarantees short-term data integrity. The bus can transport a data volume of up to 1,000 messages per second.

Mars achieves fault tolerance by means of active redundancy (logically by sending each message twice and physically by having two or more components execute the same tasks). The Mars dependability analysis determines the degree of redundancy required. Each component has intrinsic self-checking mechanisms that safeguard the component's fail-stop behavior for fault isolation. Thus a component either sends correct information, or it is silent.

The cluster concept allows extensibility of the whole system. A logically equivalent interface to a new cluster can substitute for each component without modification of the rest of the system. One can add components for such passive observations as process monitoring or data logging to the bus without modification of other components.

Mars' current implementation serves experimental purposes. Presently, we have 16 available Mars hardware components that can be split into one-to-four Mars clusters. The purpose of this experimental system is to help evaluate the Mars architecture and to provide a testbed for future research in distributed fault-tolerant, real-time systems. Exploratory areas include self-checking methods, reliable broadcast, real-time databases, recovery strategies, time-synchronization algorithms, and consensus algorithms.

Acknowledgments

Grants from the Osterreichische Fond zur Forderung Wissenschaftlicher Forschung (the Austrian Fund for Support of Scientific Research) under contract number P6251 and in part by the Osterreichische Bundesministerium fur Wissenschaft und Forschung (the Austrian Federal Ministry for Science and Research) under contract number GZ 605.504/3-26/87 supported this project.

Many students contributed to the implementation of the details of the experimental Mars system. We thank Thomas Brustbauer, Thomas Fahringer, Gerhard Fohler, Ewald Halbedl, Heinz Kantz, Thomas Keil, Franz Lackinger, Manfred Pflugl, Peter Puschner, Johannes Reisinger, and Alexander Vrchoticky. We also thank Gunther Grunsteidl, Krithi Ramamritham, Johannes Reisinger, Karin Schneider, and Werner Schutz for their useful comments and criticisms on the early versions of this article.

References

1. "Product Guide for SCADA (Supervisory Control and Data Acquisition) Systems," *J. Modern Power Systems*, Jan. 1987, pp. 56-67.
2. D.R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, Vol. 1, No. 2, Apr. 1984, pp. 19-42.
3. R. Fitzgerald and R.F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *J. ACM Trans. Computer Systems*, May 1986, pp. 147-177.
4. H. Zimmermann et al., "Basic Concepts for the Support of Distributed Systems: The Chorus Approach," *Proc. 2nd Conf. Distributed Computing Systems*, Apr. 1981, pp. 60-66.
5. A.H. Kopetz and W. Merker, "The Architecture of MARS," *Proc. 15th Fault-Tolerant Computing Symp.*, June 1985, pp. 274-279.
6. H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans. Computers*, Vol. 36, No. 8, Aug. 1987, pp. 933-940.
7. H. Kopetz, "Design Principles for Fault-Tolerant Real-Time Systems," *Proc. 19th Hawaii Conf.*, Vol. II, IEEE CS Press, Los Alamitos, Calif., 1986, pp. 53-62.
8. J.H. Saltzer, D.P. Reed, and D.D. Clark, "End-to-End Arguments in System Design," *ACM Trans. Computer Systems*, Vol. 2, No. 4, Nov. 1984, pp. 277-288.
9. M. Sloman and J. Kramer, "Distributed Systems and Computer Networks," Prentice-Hall Series in Computer Science, Englewood Cliffs, N.J., p. 140.
10. R.D. Schlichting and F.B. Schneider, "Fail Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Trans. Computing Systems*, Vol. 1, No. 3, Aug. 1983, pp. 222-238.
11. L.C. Mitchell, "A Methodology for Predicting End-to-

End Responsiveness in a Local Area Network," *Tutorial Local Network Technology*, 2nd ed., W. Stallings, ed., IEEE CS Press, Washington D.C., 1985, pp. 320-328.

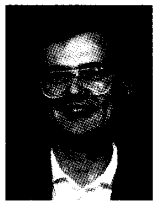
12. H. Kasahara and S. Narita, "Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System," *IEEE J. Robotics and Automation* Vol. 1, No. 2, June 1985, pp. 104-113.
13. C. Koza, "Scheduling of Hard Real-Time Tasks in the Fault-Tolerant Distributed Real-Time System MARS," *Proc. 4th IEEE Workshop Real-Time Operating Systems*, July 1987, pp. 31-36.
14. R.E. Korf, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, Vol. 27, 1985, pp. 97-109.
15. W. Zhao, K. Ramamritham, and J.A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Trans. Computers*, Vol. 36, No. 8, Aug. 1987, pp. 949-960.
16. E. Kligerman and A.D. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems," *IEEE Trans. Software Eng.*, Vol. 12, No. 9, Sept. 1986, pp. 941-949.

H. Kopetz

A. Damm

C. Koza

M. Mulazzani


W. Schwabl

C. Senft

R. Zainlinger

Hermann Kopetz is professor of computer science at the Institut fur Technische Informatik, Technical University of Vienna (TU Wien), where he directs research on the Mars project. He previously was a department manager in industrial process control and a professor for process-control computers at the Technical University of Berlin.

Kopetz received the Dr. Phil. degree in physics from the University of Vienna. He is a charter member of the Austrian Computer Society (OCG) and the Int'l Federation for Information Processing working group 10.4 on reliable computing. He is a senior member of the IEEE and the IEEE Computer Society, and a member of the ACM. He has served on the program committees of many international conferences and was the general chairman of the Fault-Tolerant Computing Symposium held in Vienna in 1986.

Andreas Damm is an assistant of computer science at the TU Wien. His main field of interest is the design of fault-tolerant, distributed, real-time operating systems and the experimental evaluation of error-detection mechanisms at the operating-system level. He received the Dipl. Ing. degree in computer science and the Dr. Techn. degree from the TU Wien. He is a member of the IEEE Computer Society, the ACM, and the OCG.

Christian Koza is an assistant of computer science at the TU Wien. His interests focus on real-time scheduling and execution-time analysis of programs. He received the Dipl. Ing. degree in computer science from the TU Wien. Koza is a member of the IEEE Computer Society and the ACM.

Marco Mulazzani was an assistant of computer science at the TU Wien until he recently joined the Alcatel-Elin Research Center in Vienna. His research interests concentrate on dependable computing and fault-tolerant systems. He received the Dipl. Ing. degree in mathematics and the Dr. Techn. degree from the TU Wien. He is a member of the IEEE Computer Society, the ACM, the OCG, and a charter member of the Unix User Group Austria.

Wolfgang Schwabl, an assistant of computer science at the TU Wien, previously worked on electronic-mail communication. His current interests are distributed real-time operating systems and clock synchronization. He received the Dipl. Ing. degree in computer science and the Dr. Techn. degree from the TU Wien. He is a member of the IEEE Computer Society, the ACM, the European Unix Users Group, the /usr/group, and a charter member and former president of the Unix User Group Austria.

Christoph Senft currently works as an assistant of computer science at the TU Wien. His research interests include software design concepts, methodologies, and their computer-aided support. He received the Dipl. Ing. degree in computer science and the Dr. Techn. degree from the TU Wien, and the Magister degree in economics from the University of Vienna. He is a member of the IEEE Computer Society, the ACM, the OCG, the EUUG, and a charter member of the Unix User Group Austria.

Ralph Zainlinger is employed as an assistant of computer science at the TU Wien. His current interests are software design methodologies and computer graphics. He received the Dipl. Ing. degree in computer science from the TU Wien and is a member of the OCG.

Questions concerning this article can be directed to Christian Koza, Institut fur Technische Informatik, Technische Universitat Wien, Treitistrasse 3/182, A-1040 Wien, Austria; Email: koza@vmars.uucp.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 156 **Medium** 157 **High** 158