Real-Time System Development: The Programming Model of MARS *

H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner,
J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, R. Zainlinger

Institut für Technische Informatik Technische Universität Wien
A-1040 Vienna, Austria E-mail: hk@vmars.tuwien.ac.at

Abstract

The systematic development of fault-tolerant real-time systems with guaranteed timeliness requires an appropriate system architecture and a rigorous design methodology. The first part of this paper describes those services of the architecture that help to simplify the work of the real-time programmer, taking MARS as an example. The second part deals with the 'programming in the large' activities, i.e., the systematic derivation of task timing parameters from the requirements specification. The following core section of the paper focuses on 'programming to meet a time budget' and presents the programming interface, the programming environment, and the testing and support tools. The architecture and most of the presented tools have been implemented and can be demonstrated on a fault-tolerant prototype application.

1 Introduction

Distributed real-time computer systems are replacing conventional mechanical or hydraulic control systems in many applications, e.g., flight control systems in airplanes, "drive by wire" systems in automobiles, and industrial process control systems. In addition to the specified functional capabilities, these applications demand predictable timeliness and specified levels of non-functional attributes, such as reliability, safety, and maintainability.

At present, the process of designing real-time systems is tedious and often unsystematic. Frequently the primary focus during the design is on the functional capabilities of the planned control system. Concerns about timeliness and dependability are usually deferred until the final testing phases, when all parts of the system have to be integrated. The application code implementing the specified transformations in the data domain is most often intertwined with the code for the *synchronization* of concurrent tasks and the code for error handling and recovery. As a consequence, it is very difficult to establish the timeliness of these systems by formal reasoning or by a constructive test methodology. Furthermore, minor changes in one part of the system can have a major effect on the timeliness of some other part.

We propose a system architecture—MARS [11]—which supports a strict separation of the issues of synchronization and timeliness, data transformation, and the dependability aspects (e.g., error detection, error handling and redundancy management). We denote the sequence of processing and communication steps between an observation of the environment and a response to the environment as a *real-time transaction.* During the design phase the real-time transactions are refined into a sequence of task executions and message exchanges. The task dependencies are analyzed and an execution time limit for each task is established, so that all transactions can be scheduled on the given hardware resources. In the programming phase, the application programmer can focus his attention on his primary task, i.e., writing correct application programs that meet a given time budget. Error detection, error handling, and redundancy management are services of the architecture.

In our view, such a separation of issues is only possible if the system architecture is time-triggered, i.e., all system activities are initiated as a consequence of the progression of real-time. Although the occurrence of events in the environment is outside the sphere of control of the computer system, the points in time when these events are to be recognized by the computer are predetermined in a time-triggered architecture. This is in contrast to event-triggered architectures, where the system activities are initiated as a consequence of the occurrence of external or internal events.

Event-triggered real-time architectures are assumed to provide a high degree of flexibility and have therefore received considerable attention in the literature (FTPP [4], ARTS [27], MAFT [7]). Because of their event triggered nature, however, an excessive number of possible behaviors must be analyzed in order to establish timeliness guarantees. Furthermore, the implementation of active redundancy by the replication of the components is hard because of the issue of replica determinism [28, 21]. DELTA4 [1] proposes the implementation of the rather complex "leader-follower" model to overcome the latter difficulty. Other event-triggered architectures, e.g., Spring [25], do not consider the issues of fault-tolerance at all.

This paper is organized as follows. In the next section we give a short overview of MARS and describe the services provided by the architecture, such as clock

*This work was supported in part by the ESPRIT Basic Research Project 3092 'Predictably Dependable Computing Systems', by the Austrian Science Foundation (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P8002-TEC and by the ÖNB (Österreichische Nationalbank) under project 4128.

synchronization, error detection, error handling, and redundancy management. Section 3 deals with the design activities which lead to the predictable timeliness and required dependability of an application. Section 4, the main focus of this paper, describes the programming model as seen by the application programmer. Finally, the topic of testability is covered in section 5 before the conclusions.

2 The Services of the Architecture

Although this paper deals with the *programming model* of MARS, it is necessary to describe the services of the architecture to understand why the programmer does *not* need to care about some aspects. The architecture relieves the designer from being concerned with the predictability of the communication and from dealing with fault tolerance.

On the architectural level a MARS System is a distributed computer system that consists of a number of autonomous, fail-silent node computers called *components* which are interconnected by a real-time network. The network consists of two replicated broadcast channels which are used in active redundancy. Each component is a self-contained computer with a local real-time clock and an interface (incoming and outgoing links) to the real-time network. It is controlled by the MARS Operating System and executes a set of application tasks. A task is a unit of computation that receives a set of messages, performs some calculations, and sends a set of messages. There is no explicit interaction between tasks inside a task's body. Communication among components (and tasks) is solely achieved by exchanging broadcast messages. Some of the components, the interface components, provide a connection to the intelligent instrumentation, i.e., the sensors and actuators in the environment. In order to handle component failures, a set of actively redundant components is combined to form a Fault-Tolerant Unit (FTU) which is active as long as at least one of its members is operational. The redundancy management is completely transparent to the user. In the rest of the section we describe the services of the architecture.

2.1 Clock Synchronization

The local clocks of all autonomous MARS components are synchronized both with each other and with the external time; the accuracy is guaranteed to be at most a few microseconds [14]. This *global time base* forms the basis for all services of the architecture and is used to attach timestamps to the observations of external events. These timestamps may be used by application tasks to determine the temporal (and possibly the causal) order of events occurring in the environment [10].

2.2 Predictable Component Interaction

In MARS there is no explicit synchronization between tasks (e.g., by using semaphores); instead, all component activities are implicitly synchronized using the global time. The tasks and messages are scheduled prior to the run time of the application in a way that guarantees that all deadlines will be met.

Pre-run-time scheduling of messages is possible because the timing behavior of the underlying network is predictable as a result of using a TDMA[1] protocol for communication among MARS components. TDMA not only provides a fixed communication bandwidth for each component, it also guarantees that at any point in time each component of the system knows which component has send access to the real-time network. This leads to a *temporal encapsulation* of the component. Because each operational component *always* broadcasts a message in its transmit slot, message losses or failures of sending components can be detected by the *receivers* of the messages.

To make the timing behavior of the communication predictable, messages are not retransmitted in case they are lost. Instead, a sufficient degree of communication reliability is achieved by masking redundancy. Each component sends each message over each of the two broadcast channels. Since we do not only have redundant communication channels but also redundant components, several identical messages are sent over the network. This allows us to maintain a reliable communication service in the presence of message losses, single component failures and a break down of one of the communication channels [3]. The operating system discards redundant copies of messages at the receiving components.

2.3 Distributed Redundancy Management

The smallest part of the system that a programmer must deal with is an FTU. An FTU is composed of two or three fail-silent components, depending on the required degree of fault tolerance. Two components of an FTU work in active redundancy. The optional third component, the so-called *shadow* component, performs the same computations as the other two, but does not send any messages on the network [13]. In the rest of the section we assume that an FTU always consists of three components. In case of a component failure the shadow component takes over the transmit slot of the failed component. The detection of a component crash and the reconfiguration of an FTU is supported by a distributed membership service [12, 3]. This service provides every operational component with consistent and timely knowledge about the set of active components.

The replicated components of an FTU not only have to produce *correct* results, they also have to create *identical* results in order to avoid inconsistencies in the system. This *replica determinism* is achieved if three conditions are fulfilled:

- Redundant components get the same inputs. This condition is fulfilled by the reliable multicast protocol described in section 2.2, and because messages arrive only at points of the sparse time base. On these points it is guaranteed that they can be handled consistently by the redundant components [10]. When reading inputs from the environment, an agreement protocol must be executed to guarantee that the replicated components get the same

[1] Time-Division Multiple Access

inputs (see section 4.1).

- The replicas perform the same computations since they execute identical schedules without unplanned preemption and the execution of tasks is triggered by the global time.
- The inner state of the replicas is identical. This follows from the above provided that their initial states are equal. If a failed component is re-integrated into the system, it must acquire the inner state from its redundant partner component. Because the predictability of the system's timing behavior must not be influenced by re-integration, re-integration actions are pre-planned. To realize this, each component periodically sends its inner state (in the form of the so-called 'h-state message') at predefined re-integration points on the real-time network, where it can be read by the redundant components.

2.4 Error Detection

Error detection is done at two levels, at the component level and at the system level [26, 13]: At the component level mechanisms to detect hardware failures are provided. The goal of these mechanisms is to make the component fail-silent, i.e., the component should create either correct results or no results at all. On the system level it is then sufficient to detect crash failures. This is done by the membership protocol as described above.

The most important mechanisms for error detection at the component level are:

- *Time redundant execution of application tasks.* The architecture supports the time-redundant execution of application tasks to handle transient faults. Each redundant instance of a task starts with the same inner state and performs the same operations on identical input messages and should therefore produce identical output messages. The equality of input messages is guaranteed by the pre-run-time scheduler and the operating system because the set of input messages does not change during the execution of the redundant instances of a task and input messages are not consumed when read. Any difference in the output messages indicates that a hardware fault has occurred. The two instances of the tasks are generated and scheduled automatically.
- *Checksums for messages.* The compiler generates code to provide each output message with a checksum and to verify the checksum of each input message. Therefore the whole path between the sender and the receiver of a message is protected against message corruption.
- *Time slice controller for the real-time network.* To prevent a faulty component from disturbing the message traffic on the real-time network, each component is provided with a *time slice controller* that controls access to the network. The time slice controller is a micro controller that works autonomously from the rest of the component and computes the transmit slots of the TDMA protocol in parallel with the operating system. Send access to the network is granted only if *both* the operating system *and* the time slice controller agree that the component is allowed to transmit a message.

2.5 Error Handling

The application programmer does not have to deal with error handling at the component level. There is only one action for error handling at the component level, and it is performed without the aid and interference of the programmer: the component is shut down immediately to satisfy the fail-silent property and to prevent propagation of errors.

On the system level, a component error leads to the reconfiguration of the corresponding FTU: The shadow component takes over the transmit slot of the failed component and the failed component tries to restart and re-integrate itself into the system as a shadow component. Before a component is re-integrated into the system, test procedures are performed to determine if the fault is permanent. If not, the component receives its initialization state (program images and schedules) from a dedicated *maintenance component.* It then listens to the network until it receives a history state message from a redundant component. At this point, the component is fully re-integrated into the system and becomes a shadow component (if there are two redundant active components) or an active one.

3 Designer's Concerns

While current practice of real-time programming must deal with fault-tolerance and synchronization aspects in a rather ad-hoc and application specific way, programming of a MARS real-time application corresponds much more to conventional software development. The issues of synchronization and timeliness are primarily dealt with at the system design level, not at the programming level.

3.1 The Design

At the top level, the designer identifies the different *modes of operation* and defines the possible *mode changes.* An airplane, for example, performs different system activities in the start, flight, and landing phases. Each mode as well as each mode change may consist of a set of co-operating real-time tasks [2]. The only significant difference between modes and mode changes is that modes are finally implemented by a set of *periodic* tasks while mode changes constitute a set of tasks that is executed only once. To cope with the inherent complexity of real-time system design in the domain of time, an object based design methodology is provided to the designer.

Real-Time Transactions

The most important design object is the *real-time transaction.* A real-time transaction refers to a set of co-operating real-time tasks fulfilling a particular system function. We distinguish between transactions dealing with *discrete* and transactions dealing with *continuous* processes.

Transactions describing discrete processes start with the observation of an external (discrete) event (e.g., "button pressed") and have to react (output a result to the environment) within a time interval dictated by the environment. We call this time interval MART, the maximum response time. The minimal time interval between two subsequent observations of

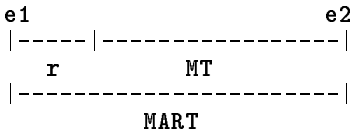

Figure 1: Relationship between Temporal Parameters

the same event must also be specified. The corresponding parameter is called MINT. MINT is often formulated as a load hypothesis defining the maximal load the system has to cope with.

Transactions describing continuous processes (e.g., controlling the temperature in a vessel) are typically characterized by the required control quality rather than the particular MART and MINT values. The timing parameters (e.g., the period) for the intended control quality are determined during design.

Apart from these attributes each transaction consists of an informal description of the functional behavior along with the input and output data. These data are modeled by so-called *data items* (e.g., temperature). Data items represent input data, output data, as well as data reflecting the internal state of the computer system (e.g., intermediate results of computations, history information).

Refining Real-Time Transactions

During design, transactions are refined into subtransactions. We present the input/output relationship of the newly obtained subtransactions in the form of an acyclic directed graph where the nodes correspond to subtransactions and the edges reflect the dependencies due to the input/output relations. This graph is called *precedence graph* (since it determines precedence relations between subtransactions). The refinement process can thus be interpreted as subsequently expanding nodes of the graph to new subgraphs. At the end of the transformation process each node is sufficiently refined and represents a single MARS task.

Refinement of Temporal Parameters

Since MARS transactions are executed periodically, the design process must also account for the derivation of a PERIOD and an MT (maximum execution time) value for each transaction. For continuous processes there is a number of well-known algorithms in control theory that help to determine these values from the description of the environment and the required control quality. For discrete processes we look at Figure 1 that presents the relationship between the relevant temporal parameters (`r` is the maximum reading latency, `e1` indicates the point in time where an event in the environment is observed by the sensors, and `e2` labels the point in time where the response of the transaction is passed to the environment).

`r` is determined by the polling rate of the transaction (i.e., the frequency with which the sensor is polled). We thus obtain the following conditions for the PERIOD of a transaction (see also [17]).

$$PERIOD \leq MINT \wedge PERIOD \leq MART - MT$$

As PERIOD not only depends on the predefined value for MINT but also on the (implementation dependent) MT of the transaction, the determination of PERIOD cannot necessarily be established by a simple top-down refinement strategy. Therefore, the methodology supports the iterative manipulation and variation of these temporal parameters. At the top levels of design, the MT value may be estimated by an experienced designer (*designer's judgement*), thus defining a preliminary bound for the PERIOD of the transaction. The complementary strategy (starting with the PERIOD and deriving a bound for MT) is also possible.

Designing FTUs

Apart from the manipulation of temporal parameters described above, the design methodology also supports the definition and description of FTUs. Simultaneously with the refinement of transactions the necessary FTUs may be established. The manual allocation of tasks to FTUs is optional, i.e., the designer may assign some tasks to particular components (e.g., tasks running on interface components) or he may leave it up to the off-line scheduler to make appropriate allocations.

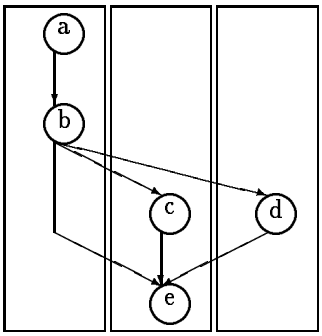

Figure 2: A Simple Precedence Graph Allocated to Three FTUs (Boxes)

3.2 Temporal Evaluation and Redesign

The input of the scheduler consists of the transactions (described by their corresponding precedence graphs) along with the (partial) task-to-FTU allocations. Figure 2 gives an example of a simple precedence graph. Tasks are depicted as circles, precedence constraints as arrows. The boxes surrounding tasks represent FTUs. All tasks are already allocated to FTUs.

Based on this information the scheduler tries to establish off-line a feasible schedule for each mode and each mode change. If such a schedule cannot be found, then an iterative redefinition of the temporal parameters or a partial redesign (e.g., introducing additional parallelism by splitting up a task into several parallel tasks) has to be carried out. This redesign takes place at an early stage of system development, i.e., before any coding activities have been initiated. If on the other hand the subsequent implementation of the tasks (see Section 4) reveals the need for a larger time interval than previously estimated, i.e., the programmer is not able to implement the tasks according to

the required MT, then again a redesign has to take place.

3.3 Ensuring Dependability

Dependability concerns influence many design decisions during system design. Therefore, dependability analysis should be an integral part of system design and should start as soon as possible in the system development cycle. Applied at early design phases, it provides coarse results that can be refined as more detailed design information is taken into account. These results provide the designer with useful information that guide him towards more appropriate design alternatives by pinpointing the most critical system parts at an early stage.

In MARS the results of the dependability analysis [6] influence the number of FTUs and the allocation of tasks to FTUs. It also determines the degree of redundancy of an FTU. For high dependability requirements an FTU consists of two actively redundant components and one shadow component. If dependability requirements are not stringent, the shadow component can be omitted. The length of the reserved CPU-time slot for a task determines the probability for the completion of the time-redundant task execution. An increase of this time slot therefore increases the error detection coverage, but also increases the response time of the task. Finally, a maintenance strategy that achieves a balance between maintenance costs and system dependability has to be determined.

4 The Programming Model of MARS

In the preceding sections we were concerned with the services of the architecture and the concerns during the design phase. This section describes the point of view of the application programmer who, given a specification of a task, has to produce an implementation that is correct in both the value and time domains.

In the first part of the section we describe the programming interface, and how it can be used to implement tasks that are correct in the value domain. In the second part, we discuss how the temporal correctness of a task can be already ascertained during the coding activities. We describe the concept of 'programming to meet a time budget' and discuss the impact of this concept on the work of the programmer. In the third part we present the programming environment that we have implemented to assist the programmer. This environment comprises a compiler, a tool for calculating the maximum execution time of programs, and an editor with special provisions for presenting information pertaining to the temporal behavior of the task.

4.1 The Programming Interface

MARS tasks are implemented using a dedicated high-level programming language called Modula/R [29] that has been designed for the purpose of supporting the development of MARS tasks running under the MARS operating system. The language is a variant of Modula-2 with extensions to support the task and communication structure of MARS and the development of programs meeting a time budget. Several features of Modula-2 that were deemed unnecessary for or even detrimental to our goals have been omitted, other features modified to better support our goals.

Modula-2 was chosen as the base language, because it is a relatively small, well-structured language with support for modularity and separate compilation, for which we could construct a compiler with reasonable effort. Besides, its strong type checking and emphasis on defensive programming support the detection of software errors due to programmer mistakes in the early testing phases. Modula/R continues this tradition by offering a number of compile-time checks, as well as the following means of run-time error detection:

- Automatic range-checking on variable assignments
- Automatic array bounds checking
- Automatic overflow checks on all arithmetic operations
- Loop iteration bounds checking.
- An `Assert` statement that can be used by the programmer to further improve error detection.

These run-time error detection mechanisms are left in the code during production use, where they can detect errors caused by transient or permanent hardware faults.

In the following, we detail how the interaction between programmer, compiler, design process, and operating system presents itself to the programmer.

The Task Structure

The basic units of computation in the MARS Design Systems are *tasks*. Tasks are executed periodically. They receive a set of messages, perform a calculation, and send a set of messages. Tasks that contain data structures that have to be initialized before the first invocation must have an acyclic *initialization part* that sets these structures to a defined value.

From the operating system's view, a task is implemented as a *team* (i.e., an address space) that consists of a set of *processes* (i.e., entry points). For the programmer, only two processes of each team are of concern: He has to program one process (the *init process*) that is scheduled once and initializes data structures, and a second process (the *cyclic process*) that is scheduled with the specified period and performs the actual real-time calculations.

In Modula/R a team is the top-level object (corresponding to the 'main' module in Modula-2). Every team can import objects (constants, types, variables, or procedures) from library *modules*, thus making use of separate compilation. Process definitions are contained within teams. They are akin to procedure definitions.

Communication via Messages

It is important to realize that the communication structure is already fixed during the design phase. The set of messages received by the task and the set of messages generated by the task are inputs to the programming phase.

Moreover, the static scheduling algorithm guarantees that all messages required for the execution of a task will indeed have arrived when the task is started, and that the messages to be generated by the task

will not be needed by other tasks before the task has finished and the messages have been disseminated to all receivers. It follows that the programmer does not have to be concerned with synchronization of the task with other tasks, as there are no synchronization points within the task. Instead, synchronization is between whole tasks and is implicit in the system's design.

Conventional message-passing systems usually provide the programmer with statements to send or receive a message, by specifying the name and/or type of the message, a receive or send buffer and possibly a timeout. Requiring the programmer to use such statements is a potential source of errors, as the correspondence between the name of the message, the type of the message, the buffer size, and even the timeout period has to be assured.

In our system, such statements are unnecessary, as the mapping of message names to types and message buffers is static, and communication occurs 'outside' the tasks' execution. Thus the programming language supports communication via messages in the following way:

- Message types are defined similarly to record types. As message types are objects global to the system (rather than local to a single team) it makes sense to declare them in a dedicated module. This module can be automatically generated from design data.
- Message variables are objects of the team. For each message variable the name of the corresponding message (i.e., the type name), and the address and size of the message buffer are automatically communicated to the operating system. A 'task skeleton' that contains these definitions can be automatically generated.
- The access to a message variable is granted to a process if the name of the variable is mentioned in the process header's *send* or *receive* list. The compiler recognizes accesses to the message variables and generates code to access the message buffers at run-time. An access to a message that is neither in the received nor in the send list constitutes an error that is reported by the compiler. Naturally, the send and receive lists also are part of the 'task skeleton'.

This approach has a number of advantages over more traditional forms of message communication:

- The synchronization problem is treated at an appropriate level, namely the design level. Implementing synchronization at the task level mingles the concerns of correctness on the task level and correctness at the task interaction level, increases the complexity, and aggravates the problem of validation.
- Message communication is non-blocking. This allows predictions about the maximal execution time of the program to be made taking only the code of the program and the (known) timing characteristics of the system into account.
- The message variable definitions can be generated automatically from design data.
- As messages are known to the compiler, it can check the correct usage of them.

Definition of History State

As explained in Section 2, MARS takes care of automatic re-integration of failed components. This is done by synchronizing the states of all tasks on the failed component with the state of their actively redundant replicas.

The simplest way to achieve this synchronization would be to copy the entire data space of all tasks to the component to be re-integrated. This approach, however, is wasteful, as the history state of a task is usually only a small part of its entire data space. It is therefore mandatory that those variables that actually comprise the history state are clearly identified [22].

The programmer can associate every global variable (i.e., every variable not local to a procedure) with a *storage attribute* that describes the semantics of the variable between executions of cyclic processes: Variables having the *history* attribute keep their values between two task executions. These variables are typically used for storing the state of a calculation that will be needed in the next iteration of the task. If the component fails and is reintegrated, the history variables are copied from the history state message of the redundant task. Variables not having the 'history' attribute are undefined upon start of every process in the team and become undefined again once the process terminates. They are typically used for intermediate results in calculations.

The compiler allocates all history variables of a team to adjacent memory locations and communicates the start address and length of this area to the operating system. The history state message is then formed by the operating system by broadcasting a message containing this area on the real-time network at predefined reintegration points.

Time as an Input

Tasks in real-time systems frequently need a notion of external time. Conventional systems usually provide a system call to access the notion of 'current time' with high precision. Unfortunately, this 'precision' is nonsensical, because imponderabilities of the underlying scheduling decisions restrict the usable part of the value to a certain granularity, which is much longer than the granularity of the clock[2]. This also brings about the problem of replica determinism: If two actively redundant executions of a task execute in parallel a certain difference in their clock readings is inevitable. If the tasks base decisions on the clock value they might take different decisions, thus violating the requirement of replica determinism.

As a consequence, MARS does away with the notion of an arbitrarily precise clock reading. The only notion of time available to a MARS task is the scheduled time of its invocation. This time is guaranteed to be identical for all actively redundant replicas of the task. Moreover, thanks to the pre-planned cyclic execution of tasks, these times can be made equidistant for any two successive executions of the same task. No system call is needed to access the invocation time, instead it is passed to the process as an argument. By

[2]You get an exact value of the clock 'now', but have only a vague idea just *when* 'now' is.

declaring the process to have a formal parameter of the appropriate type the programmer gains access to the invocation time.

Agreement Protocol for Environmental Inputs

There is one situation in which the programmer has to deal with replica determinism: A task that gets input from the environment, henceforth called *interface task*, cannot be sure that the corresponding interface task at the redundant component has received the same data. Redundant components may read different values, because events in the environment happen on a dense time-base [10]. If a state change in the environment occurs close to the reading of this state, one component may read the state before the change and the other the state after the change, no matter how tightly the clocks of the components are synchronized.

In this situation, replica determinism cannot be automatically maintained; it is possible, however, to handle this inconsistency within a single agreement task. This task is executed on all components which receive a message from the interface task. It must read *both* instances of the message and perform an agreement algorithm on them. Because the agreement tasks on redundant components read the same messages and perform the same algorithm on them, their outputs are identical and consistency is re-established.

Within each component cycle there must be at least one point where each task of a component has a well-defined history state that is identical to the history state of the corresponding task on the redundant component. As this is difficult to achieve for an interface task, such a task must not have a history state.

4.2 Programming to Meet a Time Budget

A programmer who has to implement a MARS task is given the functional specification and an execution time constraint—a so-called time budget—for that task. These data are passed to him from the higher levels of the design. He then has to implement a process that both adheres to the specification and does under no circumstances need more time to execute than has been allotted.

In order to aid implementation, the programming environment must provide the programmer with information about the temporal behavior of the code under development, especially about its worst case execution time. This raises the following needs:

- Since we have to guarantee that the processes terminate within the reserved time under all circumstances, including the worst case, an upper bound for the worst case execution time of each process must be calculated and checked against the available time resources.
- The calculated execution time bounds should be tight. Otherwise the maximum execution time may appear to be over the time budget, while in fact being below it. To aid the calculation of tight bounds the programmer must be able to express knowledge about execution constraints like infeasible paths (program paths that can actually never be executed) that cannot be derived from the code structure itself.
- It must be possible to display detailed information about the worst case execution time of processes and parts thereof to the programmer. This enables the programmer to choose strategies, algorithms, and optimizations that appropriately meet the given timing demands.

These three demands greatly influence the MARS programmer's environment. They significantly affect both the design of the programming language and the tool set and interface of the MARS task development environment. The rest of this section describes the temporal aspects of Modula/R and the task programming environment in more detail.

Programming Language and Temporal Predictability

The programming language defines the syntax and the semantics of the interface between the programmer on the one hand, and the compiler and static pre-run-time timing analysis tool, on the other hand. It is the carrier of all information that is available concerning the worst case execution time analysis of processes and thus determines the computability of execution time bounds and their quality. Information about the dynamic behavior of a program that cannot be expressed by the constructs of the language cannot be utilized by the timing analysis.

In order to adhere to the requirements imposed by the execution time analysis Modula/R has significant differences with Modula-2:

- Modula/R restricts the available constructs and programming techniques in such a way that the temporal behavior of all constructs can be predicted.
- Modula/R contains some extensions over Modula-2 that help to provide the timing analysis with information about a task's dynamic behavior that goes beyond the facts that can be concluded from the task's structure.

Restrictions of the Programming Language

Modula/R does not allow for any constructs that are unpredictable or not boundable in time. This has been demanded in [8, 19] to allow for a static, code based execution time analysis. It does not provide a goto-statement, prohibits any (directly or indirectly) recursive function or procedure calls, and does not allow dynamic data structures. Furthermore—and this is crucial—*all* loop constructs enforce that the programmer specify an upper bound for the number of iterations (see Figure 3). These bounds have to be given *in addition* to the loop conditions and are checked at run-time.

```
while x>y max 100 times
do
    -- loop body
endwhile;
```

Figure 3: Example of a Bounded Loop

Adding Information about the Dynamic Behavior

The described restrictions allow for the computation of *safe* execution time bounds, but may not be

sufficient to derive *tight* bounds. In order to facilitate the calculation of tight execution time bounds, Modula/R comprises constructs that enable the programmer to make application-specific knowledge about the control flow of a program available to the maximum execution time analysis. In the current version of Modula/R the *scope*, *marker*, and *loop sequence* constructs are available for that purpose.

The scope and marker construct give the programmer a means to state that a point in a loop will only be passed a certain number of times, possibly less often than the loop bound suggests. Loop sequences can be used to group a set of disjoint loops and state that the sum of iterations through all of these loops is bounded. A more detailed description of these constructs can be found in [19].

4.3 The Programming Environment

The real-time programmer has to develop software that meets the execution time constraints imposed by the earlier phases of system design. Therefore, detailed information about the timing behavior of a process is desirable during the implementation phase. The MARS programming environment [20, 18] is built such that it not only gives the programmer a very detailed documentation of a processes timing, but also allows an immediate observation of the effects of code changes on its overall execution time. These facilities are especially desirable when developing critical processes with tight time budgets.

From the programmer's view, the programming environment is a single editor that allows him to visualize and experiment with execution times of process parts and whole processes. In the "background" this environment also comprises a compiler, which does not only generate executable code but also basic timing information, and the timing analysis tool that computes worst case execution times for all process parts and whole processes. The editor, compiler, and the timing analysis tool exchange timing information via a so-called *timing tree*, which represents both the structure and the execution time information of the process under development.

The editor differs from other editors in that it consists of two co-operating parts, the *text editor* and the *time editor* (see Figure 4). The text editor is based on a segment-oriented approach, i.e., each program is developed and interpreted as a sequence of possibly nested segments, representing program constructs. Each of these segments is associated with a time window. All time windows together form the time editor.

Text segments allow the programmer to easily implement and experiment with different code variants, whereas the fields of the time editor additionally support the process of experimenting with different hypothetical time values of program parts. This technique, which we call *time editing* displays how changes in the worst case timing behavior of certain program parts affect the timing behavior and the worst case execution time of the whole task. The immediate visualization of and experimentation with execution times allow for a quick identification of code parts that consume too much time and for an efficient investigation and comparison of possible improvements, thus facilitating the development of tasks that meet their time budgets.

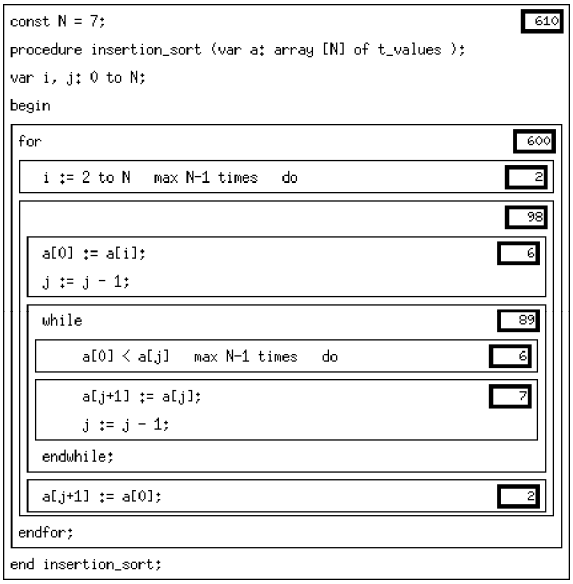

Figure 4: The Combined Text and Time Editor

5 Testing

Designing and programming an application is only one part of the total effort—it is also necessary to test the implemented application or parts thereof. It is well known that testing is a complex, time-consuming, and labor-intensive process, especially so for real-time software where both the functional and the temporal correctness are of equal concern.

We argue, however, that due to the unique properties of the MARS system, the complexity of testing is only slightly higher than the complexity of testing non-real-time software. This belief arises from the following observations:

1. In MARS, tests and test results are not the very first source of information on the system's temporal behavior, as is often the case with traditional real-time system design. Instead, the temporal behavior is *designed* early in the life-cycle, and needs "only" be double-checked through testing. Consequently, a separation of concerns between functional and temporal verification is achieved.
2. Different parts of MARS (e.g., tasks, components) are isolated from each other with respect to both function and timing ("temporal firewalls" [9]). Changes in the local temporal behavior of one part will not influence other parts (temporal encapsulation), provided these changes remain within bounds established by worst-case analysis. This is one of the consequences of designing the temporal behavior of an application and considerably eases the integration of (tested) system parts into the whole system.
3. The system perceives inputs only at predetermined points in time. This reduces the number of

environmental scenarios the system is able to discriminate and, therefore, the number of test cases needed to achieve a certain test coverage, i.e., the ratio of tested to anticipated real-world scenarios [24].

4. Because MARS is a time-triggered system it is relatively easy to ensure the reproducibility of tests and test results. This is often a major problem with other (event-triggered) distributed systems (see e.g., [16]).

We have developed a test methodology for MARS [23, 24] which takes advantage of the architectural properties and is integrated with the existing design methodology [15]. The test methodology consists of a number of test phases that summarily guarantee that an application can be tested adequately. It may be tailored to fit specific characteristics of a particular application.

The Task Test may be conducted either on the host system (i.e., the development system) or on the target system (i.e., a MARS system); it is primarily intended to test the functional correctness of an individual task. We expect that the Task Test will be mostly performed by the task programmer; it is also possible, however, to have an independent tester or test team do the Task Test. Executing the Task Test on the host system requires an emulation of the real-time operating system. For the MARS system, this would be relatively easy to implement because of its time-triggered nature.

A MARS task communicates with its environment only by receiving and sending messages (see Section 4). During the Task Test, input messages are read from files instead of being received, and output messages are written to files instead of being sent. In addition, a task may need certain temporal information; during the Task Test this information is read from a specified file or is computed at run-time by the operating system, just as in actual operation. With these facilities the tester can subject the task to various input message scenarios, and he can re-test the task after modifications and compare the test results. He may also simulate different temporal behavior of the task, e.g., experimenting with different task invocation rates, simulating task execution at a particular (absolute) point in time, or subjecting the task to unrealistic temporal behavior.

Despite this, there is little difference to testing a non-real-time task for the tester. This is because all the temporal information that a task can see is already mapped into the value domain, i.e., it is provided as time-stamps. The issue of test data generation is therefore similar to non-real-time software, except that values for the needed temporal information must be generated in addition to values for input messages. The test methodology does not prescribe a particular test data generation technique; instead we believe that many techniques (e.g., domain testing, functional testing [5], or path testing may be appropriately used, perhaps with modifications. This issue is left to the project manager to decide based on application characteristics.

In later test phases several tasks or the whole application are tested in conjunction. Such tests are conducted on the target system by the designer or an independent test team. To execute these tests it is necessary to set up a test bed. Apart from testing the correct cooperation between tasks (integration test) and from verifying the temporal behavior, the system can also be easily tested in situations which occur only very rarely in the real world, or in unsafe situations, without the need to actually create such situations in the real world. For more details on these test phases the reader is referred to [23, 24].

6 Conclusions

Real-time software development has to be concerned with both the correctness in the value domain and the correctness in the time domain. The dynamic interactions of the application software with the operating system, the communication protocols and the target hardware determine the behavior of a distributed system in the domain of time. If all these issues are treated simultaneously in an unstructured manner—as is the state of practice—the resulting software is complex, difficult to develop and even more difficult to verify.

In this paper we described a software development methodology and a distributed system architecture which enforces a strict separation of the correctness in the value domain and correctness in the time domain. Communication to other tasks and the environment is handled like reading from or writing to local memory. Testability is explicitly supported by the architecture. Error detection, error handling and the dynamic reintegration of repaired components are services of the architecture.

The system architecture and prototypes of most of the software development tools described in this paper have been implemented in an academic environment. A real-time application, the rolling ball, has been completed in order to evaluate the concepts experimentally. Many of the properties of the architecture described in this paper, such as the separation of scheduling from programming, the ease of programming, the testability, and the dynamic reintegration of repaired components can be demonstrated on this application.

References

[1] P. A. Barrett, A. M. Hilborne, P. Verissimo, L. Rodrigues, P. G. Bond, D. T. Seaton, and N. A. Speirs. The DELTA-4 extra performance architecture (XPA). In *Proc. 20th Int. Symposium on Fault-Tolerant Computing*, pages 481–488, Newcastle upon Tyne, U.K., June 1990.

[2] G. Fohler. Realizing changes of operational modes with pre run-time scheduled hard real-time systems. In *Proceedings of the Second International Workshop on Responsive Computer Systems*, Saitama, Japan, October 1992.

[3] G. Grünsteidl and H. Kopetz. A reliable multicast protocol for distributed real-time systems. In *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 19–24, Atlanta, GA, USA, May 1991. published by Pergamon Press, 1992.

[4] R. E. Harper, J. H. Lala, and J. J. Deyst. Fault tolerant parallel processor architecture overview. In *Proc. 18th Int. Symposium on Fault-Tolerant Computing*, pages 252–257, Tokyo, Japan, June 1988.

[5] W. E. Howden. *Functional Program Testing and Analysis*. Software Engineering and Technology. McGraw-Hill, New York, 1987.

[6] H. Kantz. Flexible handling of diverse dependability requirements in MARS. In *Proc. 10th Symposium on Reliable Distributed Systems*, pages 142–151, Pisa, Italy, Sep. 1991.

[7] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidural. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–404, Apr. 1988.

[8] E. Kligerman and A. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, Sep. 1986.

[9] H. Kopetz. Event-triggered versus time-triggered real-time systems. In A. Karshmer and J. Nehmer, editors, *Proc. Int. Workshop on Operating Systems of the 90s and Beyond*, Lecture Notes in Computer Science, Volume 563, pages 87–101, Berlin, Germany, 1991. Springer-Verlag.

[10] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *12th Int. Conf. on Distributed Computing Systems*, pages 460–467, Yokohama, Japan, June 1992.

[11] 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.

[12] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avizienis and J. C. Laprie, editors, *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 411–429. Springer-Verlag, 1991.

[13] H. Kopetz, H. Kantz, G. Grünsteidl, P. Puschner, and J. Reisinger. Tolerating transient faults in MARS. In *Proc. 20th Int. Symposium on Fault-Tolerant Computing*, pages 466–473, Newcastle upon Tyne, U.K., June 1990.

[14] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933–940, Aug. 1987.

[15] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. An engineering approach to hard real-time system design. In *Proc. of the Third European Software Engineering Conference, ESEC '91*, pages 166–188, Milano, Italy, Oct. 1991.

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

[17] A. K. Mok. The design of real-time programming systems based on process models. In *Proc. 5th Real-Time Systems Symposium*, pages 5–17, Dec. 1984.

[18] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5):35 – 44, Sep. 1992.

[19] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[20] P. Puschner and R. Zainlinger. Developing software with predictable timing behavior. In *Proc. 7th IEEE Workshop on Real-Time Operating Systems and Software*, pages 70–76, Charlottesville, VA, May 1990.

[21] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[22] Fred B. Schneider and Richard D. Schlichting. Towards fault tolerant process control software. In *Proc. 11th Int. Symposium on Fault-Tolerant Computing*, pages 48–55, Portland, Maine, USA, June 1981.

[23] W. Schütz. A test strategy for the distributed real-time system MARS. In *IEEE CompEuro 90, Computer Systems and Software Engineering*, pages 20–27, Tel Aviv, Israel, May 1990.

[24] W. Schütz. *The Testability of Distributed Real-Time Systems*. PhD thesis, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria, Feb. 1992.

[25] J. A. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for real-time operating systems. *IEEE Software*, pages 62–72, May 1991.

[26] A. Steininger and J. Reisinger. Integral design of hardware and operating system for a DCCS. In *Proc. 10th IFAC Workshop on Distributed Computer Control Systems*, Semmering, Austria, Sep. 1991.

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

[28] A. Tully and S. K. Shrivastava. Preventing state divergence in replicated distributed programs. In *Proc. 9th Symposium on Reliable Distributed Systems*, pages 104–113, Huntsville, Alabama, Oct. 1990.

[29] A. Vrchoticky. Modula/R language definition. Research Report 2/92, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, March 1992.