The Programmer's View 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. We propose a system with strict separation of the issues of synchronization, dependability aspects and data transformation. Dependability aspects (error handling and redundancy management) are handled by the architecture. Synchronization and programming in the large is handled at the design level. The programmer only has to be concerned with a sequential program for which he has to meet a so-called "time budget". The architecture and many 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 [6]—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).

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 (ARTS [15], MAFT [4]). 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 [11]. DELTA4 [1] proposes the implementation of the rather complex "leader-follower" model to overcome the latter difficulty. Other event-triggered architectures, e.g., Spring [13], 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 the architecture and the separation of design issues from programming. The following two sections describe the tasks of designer and programmer in more detail.

2 Separation of Concerns

In our proposed system architecture a strict separation of the issues of synchronization and timeliness, data transformation, and the dependability aspects is supported. The designer is responsible for handling these issues as long as they concern the inter-task relationship, the programmer handles them at the task level, and the system architecture provides the base services for both of them. In the following we will outline both the MARS architecture and the concerns of the designer and the programmer, in order to give the essentials to read the rest of the paper.

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 [14]. Each component is a self-contained computer with a local real-time clock and an interface to the real-time network. It is controlled by the MARS Operating System [10] 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 among tasks inside a task's body. Communication among components (and tasks) is solely achieved by exchanging broadcast messages.

In order to handle both network and component failures, *active redundancy* is used. All messages are sent at redundant broadcast channels and the tasks are executed at each component of a so-called *Fault-Tolerant Unit* (FTU) [3, 7]. The active redundancy of processing and communication resources as well as error detection, error handling, and redundancy management are *services of the architecture* and completely transparent to the application programmer. The architecture relieves the designer from being concerned with fault tolerance issues.

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. In this process 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. The schedule is generated off-line for verification of the estimated execution time bounds.

Synchronization and communication are handled by message exchanges among tasks. No synchronization or communication primitives other than messages are used.

In the *programming phase*, the application programmer can focus on his primary activity, i.e., writing correct application tasks that meet a given time budget. He is not concerned with task synchronization. The task gets input messages and must produce output messages within a given time. Communication with other tasks is handled like reading from or writing to local memory.

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.

Top Level 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 cooperating 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 cooperating 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 the same event must also be specified. The corresponding parameter is called MINT. MINT is often formulated as a load hypothesis defining the maximum 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.

Since MARS transactions are executed periodically, the design process also accounts for the derivation of a period and an MT (maximum execution time) value for each transaction.

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 FTUs (e.g., tasks running on interface components) or he may leave it up to the off-line scheduler to make appropriate allocations.

Temporal Evaluation and Redesign

The input of the off-line scheduler consists of the transactions (described by their corresponding precedence graphs) along with the (partial) task-to-FTU allocations.

Based on this information the scheduler tries to establish 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 budget 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.

4 The Programming Interface

The design phase ultimately decomposes each transaction into a set of cooperating tasks. Each of these tasks consists of a startup part for the initialization of its inner state and a main part that is executed periodically after initialization. The periodic actions of all tasks follow one pattern: The task receives a set of messages, performs a calculation, and finally sends a set of messages. No communication or synchronization takes place 'inside' the task's body, i.e., while it performs its calculations. Therefore a task's specification, as produced by the design phase, consists only of the following:

- the set of messages received by the task,
- the set of messages produced by the task,
- the variables that form the inner state of the task,
- the functional specification of the calculations performed by the task,
- and a maximum execution time for the task.

The programmer does not have to be concerned with synchronization, as this concern is handled at the design level. Instead, he can concentrate on producing correct sequential code for each given task. This separation has the following advantages:

- The synchronization problem is treated at an appropriate level, namely the design level, thus reducing complexity and facilitating validation.
- Message communication is non-blocking, making the execution of a task independent of the progress of other tasks in the system. This greatly facilitates predictions of the maximum execution time of a task.

To implement the tasks, the programmer uses a variant of Modula-2, called Modula/R [16]. The language allows the calculation of the maximum execution time of a task and supports MARS's specific task structure and message communication. Specifically, message naming, message typing, and buffer manipulation are implemented by the compiler. To further aid the programmer we

- provide a tool that uses design data to automatically generate message type definitions and a 'task skeleton' that contains the correct input- and output declarations for all the messages sent and received by the task,
- perform numerous compile-time and run-time checks to help the programmer to detect errors.

Programming Language and Temporal Predictability

The programming language establishes the interface between the programmer on the one hand and the programming and evaluation tools on the other hand. It is the carrier of all information available to the worst case execution time analysis of tasks and thus influences the computability of execution time bounds and their quality. The needs of the exeuction time analysis influenced the design of Modula/R:

- Modula/R restricts the available constructs and programming techniques such that the temporal behavior of constructs can be predicted. E.g., it does not allow the use of recursions and gotos, and all loops have to be bounded [5] in order to apply calculation rules as found in [12].
- Modula/R contains extensions to Modula-2 that help to provide the timing analysis with knowledge about a task's dynamic behavior that goes beyond the information that can be extracted from the structure of the task's source code [9].

The Programming Environment

The real-time programmer has to write tasks that meet the execution time constraints imposed by the earlier design steps. Therefore, detailed information about the worst case timing behavior of a task is desirable during the whole programming phase. The programmer shall not only see how long the execution of a whole task takes in the worst case, but he shall have access to execution time information of every part of a program. Only this comprehensive information allows him to identify those parts of a tasks that need too much time and have to be changed.

The MARS programming environment [8] not only gives the programmer a very detailed documentation of a task's timing. The so-called *time editing* feature also allows an immediate observation of the effects of planned code changes on a task's overall execution time, before actually implementing them. One can replace execution times that have been calculated for certain code parts by hypothetical values—the execution times expected for these parts after the change—and compute the hypothetical execution time for the whole task for the assumed values.

The permanent feedback about a timing during programming greatly facilitates the development of time critical tasks. This is especially the case when the available time budgets are tight.

5 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 enforce a strict separation of the issues of system design and task implementation. Communication to other tasks and the environment is handled like reading from or writing to local memory. 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 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, 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.

Acknowledgements

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.

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*, Atlanta, GA, USA, May 1991.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] J. Reisinger. Time driven operating systems – a case study on the MARS kernel. In *Proc. 5th ACM SIGOPS European Workshop*, Le Mont Saint-Michel, France, Sep. 1992.

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

[12] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.

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

[14] 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.

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

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