Real-Time Software Design - State of the Art and Future Challenges

by *Manas Saksena, Concordia University, and*
Bran Selic, ObjecTime Limited

In this article, a high-level overview of the state of the art of real-time software design is presented. In particular, attention is focused on standard techniques for dealing with the critical issues of concurrency and timeliness as well as the tools that support them. The various design styles that have evolved over time for constructing real-time software are also described. In conclusion, a brief review of the principal technological trends currently emerging in the field are described.

Dans cet article, on présente un survol des plus récents progrès technologiques accomplis dans le domaine de la conception de logiciels en temps réel. Une attention particulière est portée aux techniques normalisées servant à gérer les questions clés de la concurrence d'accès et de la synchronisation ainsi qu'aux outils utilisés pour leur mise en application. On décrit l'évolution des divers styles de conception des logiciels en temps réel. En conclusion, on présente un bref examen des principales nouvelles tendances technologiques dans ce domaine.

1. Introduction

In recent years, real-time computing has emerged as an important discipline in computer science and engineering. With ever-increasing computational power, more systems are being implemented in software to exploit the flexibility and sophistication afforded by software implementations. However, as real-time software becomes more complex, software design styles play an important role in software and system development.

Traditionally, the real-time community has been very reluctant to adopt new software design technologies. This is due in part to the high performance and stringent response times required for many such systems, which often compel developers to use very low-level techniques. Another factor that contributes to this conservative mentality is the high cost and safety requirements in many such systems. Thus, an approach that has proven to work is preferred over possibly better but less proven technologies.

Recent developments in modern computing technology, however, challenge both these traditional assumptions. Computers are becoming faster and software development tools are becoming more powerful. Also, as the demand for more sophisticated functionality increases and software becomes more complex, traditional low-level techniques can no longer keep up. Hence, for many real-time applications, the need to utilize more sophisticated state-of-the-art methodologies is critical to ensure that they meet safety requirements.

1.1 Characteristics of real-time systems

Broadly speaking, a real-time system is any system that responds in a timely manner. Thus, “time” is a resource of fundamental concern in real-time systems, and activities must be scheduled and executed to meet their timeliness requirements. Timeliness requirements (and systems) are often classified into hard real-time, where failure to meet a deadline is treated as catastrophic system failure; and soft real-time, where an occasional missed deadline may be tolerated.

Another fundamental characteristic of many real-time systems is that they are embedded systems, i.e., they are components of a larger system that interacts with the physical world. This is often the primary source of complexity in real-time systems. The physical world typically behaves in a non-deterministic manner, with events occurring asynchronously, concurrently, and in an unpredictable order. Whatever happens, the embedded real-time system must respond appropriately and in a timely fashion.

The concurrency of the physical world usually implies that real-time system software must also be concurrent. In fact, one of the primary requirements of many real-time systems is to synchronize multiple concurrent activities arising in the environment. Unfortunately, concurrency adds complexity to software design since it conflicts with the inherently serial, cause-and-effect flow of human reasoning.

Another salient characteristic of many real-time systems is dependability. Many real-time systems play a crucial role in their environments and failure to perform correctly may result in significant costs or jeopardize human safety. As a result, real-time systems must often be highly reliable (i.e., they must perform correctly), and available (i.e., they must operate continuously).

1.2 Overview of the Article

Due to space limitations, in this article we focus our attention to the concurrency and timeliness aspects of real-time systems. We begin with an overview of techniques to manage concurrency and timeliness issues in real-time systems. Then we present two software design styles and show how these issues are addressed in the two styles, and discuss the strengths and limitations of each. Next, we present an overview of the technologies and tools available to a real-time software developer. Finally, we conclude by identifying some future challenges.

2. Concurrency Management

One major concern in the design of real-time systems is the management of concurrency. Even in single processor systems, concurrency is often used to both simplify the software structure, and to improve responsiveness of the system to critical events. This form of concurrency is referred to as multi-tasking, and involves time multiplexing of many concurrent tasks on a single physical processor. However, while concurrency is a necessary tool to manage timely response in a system, it is also fraught with difficulties. Our inherent inability to properly reason about concurrency can lead to subtle bugs in software that are difficult to find and fix, and this makes it difficult to realize the necessary level of reliability. Concurrency also makes performance estimation more complex.

Consequently, special concurrency management techniques have been developed to ensure that concurrency is used in a controlled manner. These are based on three types of mechanisms:

- mechanisms for representing and tracking the progress of concurrent activities within a computer
- mechanisms for communicating between concurrent activities
- mechanisms for synchronizing the execution of multiple concurrent activities.

2.1 Abstractions for Concurrent Activities

The most common mechanism for representing concurrent activities is to provide each activity with a “virtual” processor that executes a thread of control and maintains its state during this execution. This virtual processor is typically referred to as a process. Normally, each process has its own address space that is logically distinct from the address space of other processes. Unfortunately, the overhead required to switch the

physical processor from one process to another is often prohibitive. It involves context switching, time-consuming swapping of register sets within the CPU that typically takes many tens, possibly hundreds, of microseconds even with modern high-speed processors.

To reduce this overhead, many operating systems provide the ability to include multiple lightweight threads within a single process. The threads within a process share the address space of that process. This reduces the overheads of context switching, but increases the likelihood of memory conflicts.

It is often possible to further reduce the context switching overhead by constructing even lighter-weight concurrency abstractions at the application level (i.e., above the operating system). This technique is quite common in very large systems with large numbers of concurrent threads. It relies on doing application-controlled context switching at points when there is little or no context that needs to be saved. This type of context switch does not require crossing the "heavyweight" boundary between the kernel-level and the user level of the processor and, therefore, multiple application-level threads may be multiplexed on a single operating system thread. These application-level threads are scheduled by an application-specific scheduler operating within the operating system thread.

Active object abstraction is an example of such application-specific concurrency management. An active object receives events from outside (typically placed by a sender in a queue), and responds to events in a run-to-completion manner. That is, new arriving events cannot preempt the processing of the event being handled. Once event processing completes, data and call stacks of the active object are empty and do not need to be saved during a context switch for the next event. Multiple active objects may be combined into a single operating system thread, sharing its data and call stacks. In this case, scheduling of active objects takes place only on completion of a previous event handling.

2.2 Communication Mechanisms

Data communication is usually based on either shared data or message passing. In a shared data paradigm, two or more threads can access shared data objects, and thus indirectly communicate with each other. Message passing, on the other hand, involves explicit exchange of data between two threads.

When shared data is used between threads, concurrent access can lead to inconsistent updates. Thus, the shared data must often be accessed in a critical section (i.e., in a logically atomic action), and a synchronization mechanism called mutual exclusion is used to achieve this. Mechanisms to achieve mutual exclusion are described in the next section. Shared memory communication is automatically available between threads sharing an address space. Even with processes, many operating systems will allow creation of shared memory regions between processes.

In message passing communication, the communication can be asynchronous or synchronous. In asynchronous communication, the sending activity does not synchronize with the receiver; instead it forwards information to some queue from where the receiver can retrieve it later. The sender and the receiver proceed independently of each other, and cannot make assumptions about each other's state. Synchronous communication includes synchronization between the sender and the receiver in addition to the exchange of information. In pure synchronous communication, the sender must rendezvous with the receiver to exchange information. Most operating systems support some form of message passing communication between processes or threads. Message passing between threads in a single process can also be built using shared memory and synchronization mechanisms described in the next section; this is often more efficient since the operating system is not involved in the communication.

2.3 Synchronization Mechanisms

Synchronization involves coordinating the execution of two or more activities in such a way that concurrency conflicts are avoided. Mutual exclusion is one form of such synchronization. It is relatively simple to achieve using mutex locks or semaphores; the problem with these mechanisms is that they are low-level abstractions that are independent of application semantics. This means that the semantic link between the application and the synchronization mechanism (e.g., the application-specific circumstances under which it is necessary to synchronize), must be constructed by the application programmer. Furthermore, these mechanisms are highly error prone since the synchronization code is typically spread throughout the application program, often making it difficult to understand and maintain.

A safer approach to managing such concurrency conflicts is to encapsulate shared data as private data of protected objects and making the data accessible through operations invoked on the object. In principle, at most one concurrent thread at a time is allowed to access the protected object. In this way, the synchronization code is encapsulated within the shared data objects, and different threads can safely invoke operations on the data object without worrying about concurrency conflicts.

More general forms of synchronization requirements may require a concurrent activity to wait for some event to occur before proceeding. In this case, the activity is blocked using synchronization abstractions like condition variables. The activity can be woken up by signaling on the condition variable. Normally these are used in conjunction with shared memory and mutual exclusion to solve synchronization problems.

3. Real-time Performance Analysis

Real-time performance analysis is an essential part of most real-time systems, and is used to estimate whether a system will meet its timeliness requirements.

3.1 Execution Time Estimation

To estimate the performance of a concurrent system, we must first be able to estimate the performance of a single sequential task. Execution time estimation bounds the running time of a single-threaded code fragment when running on a dedicated processor (i.e. in the absence of multi-tasking). The problem of estimating execution times is composed of two sub-problems: (1) identifying legal paths through the code, and (2) determining the execution time of each path. Accurate estimation of execution time bounds using analytical techniques is difficult for a number of reasons. First, in general, it is undecidable to determine all legal paths through a program. Second, the execution time for legal paths may be data dependent. Finally architectural features of modern CPUs, such as caches and pipelining complicate the determination of accurate execution time estimates due to global side effects. In practice, due to the limitations of analytical techniques, one must rely on measurements and extensive testing to accurately estimate execution time bounds.

3.2 Real-Time Scheduling

While analyzing execution times for an individual task is a necessary building block, it is by no means sufficient. When multi-tasking is used, we must also consider the scheduling of tasks on the CPU. The real-time scheduling problem for a single processor consists of deciding the order of execution of a set of tasks (concurrent threads) with certain known characteristics (e.g., periodicity, and execution times).

The simplest approach to real-time scheduling is to construct a schedule off line. At run-time, tasks are dispatched according to the pre-constructed schedule. Static real-time scheduling is particularly applicable when the majority of activities are based on periodic polling. A cyclic schedule can be created for a set of periodic real-time tasks. The traditional cyclic executive approach is a special case of such static scheduling.

Static cyclic scheduling is inadequate when events may arrive aperiodically. While a polling approach may be used, it leads to very inefficient CPU utilization. Instead, a run-time scheduling approach is used. The most common run-time scheduling mechanism is a preemptive-priority approach, in which tasks are dispatched at run-time according to their priorities, and at all times the highest priority task gets to run.

An off line analysis (or simulation) must complement a run-time scheduling scheme to ensure that timeliness requirements will be met at run-time. Schedulability analysis is used to determine whether a given sys-

tem will meet its timeliness requirements at run-time. There is a well-developed theory, based on preemptive priority scheduling and deterministic analysis that can be used to predict worst-case response times to events, given their maximum arrival rates, and maximum execution times, and a priority assignment of tasks.

Synchronization between tasks complicates schedulability analysis. When mutual exclusion is used between tasks, priority inversion can occur, where a low priority task can block the execution of a higher priority task when they share a critical section. Worse still, this priority inversion can be unbounded since an arbitrary number of intermediate priority tasks can preempt the low priority task. Fortunately, a number of priority inheritance protocols exist that allow us to bound priority inversions, and apply schedulability analysis. The basic idea behind these protocols is to bump up the priority of a low priority task when it is in a critical section to prevent intermediate priority tasks from preempting it.

More general forms of synchronization make the task of schedulability analysis even harder, and in general impossible. However, simple precedence synchronization is still analyzable using the deterministic analysis approach. Precedence synchronization is used in synchronous communication.

While schedulability analysis is now a mature technique, it is based on several assumptions such as known worst-case execution times, simple concurrency, synchronization, and communication models, etc. While a variety of performance analysis techniques may be used when some of these assumptions may not hold (e.g., model checking, stochastic performance analysis, simulation), it remains a hard problem.

4. Design Paradigms

In this section we present two design paradigms that may be used to develop real-time systems. We use the term design paradigm to represent a style of developing a real-time application and especially on how concurrent activities are represented and modeled.

Ideally, a design paradigm should be supported by a standard set of (preferably visual) notations, and formal semantics associated with such notations. Formal semantics allow a design specification to be simulated, analyzed for timeliness, as well as allow code generation to be automated. None of this is feasible if we want the full generality available in standard programming languages. Instead, given the specific demands of real-time systems, this generality is often sacrificed to get the benefits of automated analyzability, formal semantics, and code generation.

4.1 Time-Driven Style

This design style has evolved to model primary functional scenarios of a real-time system, and to make these designs easily analyzable for timeliness. As a result, it is based on a simple concurrency model. In this model, a task is activated by an event trigger, and, upon arrival of the event trigger, it performs some computation and then awaits the arrival of the next event-trigger. The event trigger may be connected to an external interrupt source, a periodic timer, signal from another task, etc. Often timers are the main event sources (to ensure predictable workload), and hence we call it the time-driven style. The communication between these tasks is based on the shared memory model through shared protected data objects.

The tasking model is often extended to permit a message passing communication between tasks, in addition to shared data. One way to accomplish this is to make each task read input signals when it is triggered, and produce output signals when it finishes execution. By connecting the output signal of a task with the input signal of another task, one may form directed task graphs based on producer-consumer relationships. These producer-consumer relationships between tasks impose ordering requirements on the execution of tasks, which may be achieved by a variety of means. In such a model, external input signals flow through a task graph before producing external output signals.

The time-driven style is naturally suitable for dealing with regular and recurring input signals, where the processing of such signals may flow through a number of processing steps (represented as tasks) before generating output signals. Digital signal processing and feedback control loops are two examples that fit this design style. The main strength of the time-driven style is that it maps directly to real-time scheduling models and thus such designs are highly amenable to schedulability analysis (and in some cases off line scheduling).

On the other hand, the tasking model is relatively simple and complex task behaviors cannot be described. For example, aspects of control (system initialization, mode changes, response to exceptions and failures, etc.) are not formally modeled, and hence not analyzable. The event-driven style, described in the next section, explicitly models these phenomena, and therefore making the designs more amenable to analysis and simulation.

4.2 Event Driven Style with Object-Oriented Models

In contrast to the time-driven style, the event-driven software style has evolved largely to deal with unpredictable asynchronous events. Thus, event-driven software is structured around event-handling code. An event triggers the appropriate event-handling code, and when the action is complete, the software enters a dormant state awaiting the next event. Since events may arrive while a previous event is being processed, the software also allows for events to be queued for future processing. The system must respond to asynchronous events in the external world, and the reaction must depend on the system state. Thus, the software behavior is often modeled as a finite state machine, where the arrival of an event triggers transitions in the state machine, and includes the event-handling code. An event-driven software system is often composed of a set of concurrent and communicating finite state machines.

Event-driven software can be combined with object-orientation for large complex systems. The active object concurrency abstraction is particularly suitable for representing such finite state machines. The finite state machine specification gives a behavioral specification of the active object. Thus, a system is composed of a set of active objects communicating with each other using message-passing mechanisms. Additional structuring mechanisms such as hierarchical decomposition and layering may be used to build complex systems.

The biggest strength of this design style, in comparison with the time-driven style, is that it has mechanisms to model complex system behavior arising often from control aspects. This is manifested in a more general behavioral specification of concurrent activities as manifested in the active objects. On the other hand, this generality results in a less clear understanding of the data-flows in a system, making it more difficult to perform schedulability analysis.

5. Real-time Technologies, Tools And Standards

An important ingredient of real-time software development is the necessity of tools and technologies to support the development process. In this section, we briefly look at some of the technologies, tools, and standards that are available to real-time developers.

5.1 Real-Time Operating Systems

While traditionally many real-time applications have been built without using an operating system, the use of a real-time operating system to better structure software is indispensable whenever an application manages multiple concurrent activities and devices.

A real-time operating system provides the abstractions for concurrent activities, and communication and synchronization mechanisms between concurrent activities. This greatly simplifies the writing of real-time software. Most real-time operating systems provided preemptive priority based scheduling. The management of priorities for concurrent activities to satisfy the timeliness requirements is left to the application.

Traditionally real-time operating systems have provided their own proprietary API's. However, the emergence of the POSIX standard has helped to standardize this with several OS vendors providing POSIX API support, and especially those that relate to the real-time extensions.

5.2 Design and Development Tools

Traditionally, design and development tools for real-time software have not been very sophisticated. This is, again, in part due to the intrinsic nature of real-time and embedded software. However, the industry is finally beginning to invest in development tools, and this is a welcome sign. For example, many vendors now provide an integrated development environment with tools for editing, compiling, profiling, monitoring, debugging, configuration management, etc. Many of these tools are cross-hosted, i.e., the development tools run on host platform (usually a Windows NT or Unix machine), but support communication with the real-time software environment on the target machine.

Some tools go beyond providing a programming environment by supporting model based real-time design. These, so called Computer-Aided Software Engineering (CASE) tools provide the ability to develop software using high-level models, and provide support throughout the development life-cycle from requirements analysis to design to implementation. Most of such tools support the event-driven, object-oriented software style, and there is some convergence towards using the newly developed Unified Modeling Language, and standardized by Object Management Group.

5.3 Performance Analysis Tools

Performance analysis tools help in the analysis of timeliness properties of real-time software. Ideally, such tools should provide early assessment of the feasibility of suitable designs based on estimates of run-time costs, and then support the analysis through the design cycle to the implementation, where real costs can be measured, and performance analysis validated against the run-time behavior. Unfortunately, this is a relatively weak area as far as tool development goes, although a few tools have come out that support schedulability analysis and simulation of models for timeliness properties.

6. The Future

In this paper we have looked at the problem of real-time software design, concentrating on two key aspects namely timeliness and concurrency. We also presented two design styles and showed how the two issues are addressed in them. Finally, we presented some of the technologies and the tools that are available to real-time software developers.

It is encouraging to see real-time software design emerge from its unduly conservative phase and finally adopting modern tools and techniques. There are clearly many areas of improvement. Some of the biggest deficiencies are in dealing with the timeliness requirements. For example, performance analysis tools are relatively primitive, and not integrated with the design and modeling environments. There is also a conflict between more generalized models for design and the ability to analyze them for timeliness. Many systems are soft real-time, and there is a clear need for tools that support the design of adaptive real-time systems whose performance degrades in predictable and controlled manner in overload situations.

Another area where we envision many challenges is in design heuristics. In developing complex real-time software, a number of design choices are available to the developer. In the absence of proper guidelines or heuristics, such design choices are made in ad-hoc manner, and can easily lead to inefficient designs. The trend towards higher level design abstractions and patterns that are closer to the problem domain exacerbate this problem, since it leads to many more choices in moving from the design abstractions to the implementation abstractions. For example, in an object-oriented design style, choices must be made with respect to how the design is mapped to operating system threads, how are priorities assigned to threads, etc.

As society moves towards a symbiotic dependency on computer systems, more and more of these systems will fall in the broad category of real-time systems. This means that the methods and techniques developed for traditional real-time software, including in particular techniques for fault tolerance and predictability, will become part of the standard knowledge base of most software engineers and developers. Furthermore, they will be supported by increasingly more sophisticated integrated suites of tools that will significantly increase the level of automation of real-time software development.

7. Further Reading

[1]. IEEE-CS Technical Committee on Real-Time Systems Home Page. Available at: http://cs-www.bu.edu/pub/ieee-rt/Home.html

This site contains links to various research groups, conferences, and vendors of commercial products.

[2]. Burns, A. and Wellings, A., “Real-Time Systems and Programming Languages,” (2nd ed.) Addison-Wesley, 1997.

This provides the most complete overview of the vast array of different mechanisms and practices developed specifically for real-time software.

[3]. Jalote, P., “Fault Tolerance in Distributed Systems”, Prentice Hall, 1994.

This contains an excellent review of various dependability techniques.

[4]. Klein, M., et al., “A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems,” Kluwer Academic Publishers, 1993.

An encyclopedic treatment of schedulability analysis techniques.

[5]. Object Management Group, “The Unified Modeling Language Specification (version 1.1),” OMG, Framingham, MA 1998.

The UML standard can be obtained from the site: http://www.omg.org

[6]. Selic, B., Gullekson, G., and Ward, P., “Real-Time Object-Oriented Modeling,”, John Wiley & Sons, 1994.

An example of the application of the object paradigm to complex real-time systems based on the event-driven style.

About the Authors

Manas Saksena received his Ph.D. from University of Maryland, College Park in 1994.

He is an Associate Professor in the Department of Computer Science at Concordia University, Montreal. He has done research in various aspects of real-time systems, including operating systems, scheduling, communication, and design. His current research interests are in various aspects related to design of real-time systems, a topic on which he has published a number of articles in various IEEE conferences and journals. Most recently, he has been working on schedulability analysis of real-time object-oriented designs, and developing a UML based design methodology for real-time systems.

Bran Selic is the Vice President of Advanced Technology at ObjecTime Limited.

He has over 25 years of experience in constructing large-scale real-time systems in a number of different disciplines including telecommunications, aerospace, and robotics. He is the principal author of the popular book, "Real-Time Object-Oriented Modeling" that describes how the object paradigm can be used effectively in real-time applications. Most recently, he has been active in the specification of the Unified Modeling Language (UML) standard for object-oriented analysis and design.