Software Engineering of Real-Time Operating Systems[*]

Alan Shaw
Kevin Jeffay
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

Technical Report 88-01-01
January 1988

Abstract

This paper is a slightly revised and updated version of a research proposal submitted to the National Science Foundation in fall of 1986 and funded in fall 1987. The research covers general topics in software engineering, such as programming-in-the-large, and specific areas in real-time processing. We are developing:

1. a module structuring and interface methodology. This is based on an acyclic graph model where the nodes represent modules (restricted to processes and abstract data types) and the edges denote procedure import and export relations. The scheme is simple, is applicable to "low-level" hardware/software interfaces as well as higher-level software abstractions, and is intended to permit the analysis of timing behavior.

* This research was supported in part by the National Science Foundation under grant CCR-8700435 and by the Washington Technology Center.

2. a prototype for real-time operating systems design, involving libraries of reusable components. We are identifying, designing, and building a variety of useful components and structures, such as interrupt-handlers, critical section locking mechanisms, message passing objects, timers, schedulers, input-output systems, cyclic executives, and producer-consumer applications.

3. a graphics front end for interacting with designs. This is essentially a graph editor that permits the construction, modification, and annotation of our module structures, including the association of modules with their code files. The design will be based on earlier experience with a first prototype.

Initially, we expect to study and construct modules for a uniform shared-memory multiprocessor environment. Some emphasis is being given to timing predictability and the exploitation of concurrency.

Software Engineering of Real-Time Operating Systems

1. The Problem and Our Approach

The research is concerned with methods and tools for generating *real-time* operating systems. This area of software systems has received relatively little research attention in the past in comparison with general purpose operating systems. The situation has changed in recent years, however, primarily because of the widespread applications of real-time systems, the international interest and activity in the Ada programming language, and the resulting recognition that the subject contained interesting, challenging, and important research problems.

Typical applications include guidance and control in transportation systems (for example, air traffic, avionics, train, and marine), robotics control software (for robots that perform manufacturing and testing operations), and communications systems (for example, telephone switching systems). Despite the existence of commercial real-time operating systems and tool-kits, most of these applications require that the user "hand-craft" or even write their own operating system to fit their needs. A major goal of this research is to simplify this task by providing the technology to easily produce reusable, robust, efficient, and timing predictable software.

Real-time systems have a number of features and requirements that distinguish them from more conventional software and applications. The most important is the existence of strict timing constraints; correct systems performance involves not only standard correctness notions for software and hardware (e.g. computing the right answers) but also *timing correctness* such as responding to external events within a specific time interval or meeting given start and completion times for various applications processes. Another major feature is *parallelism*; physical and logical concurrency is inherent in the applications and computer environments, and concurrent processing is viewed as one of the principal methods for improving timing performance. Other characteristics are the need for reliability and fault tolerance, stand-alone and applications-specific hardware/software configurations, and difficult but critical pre-testing and certification prior to delivery.

Several approaches to real-time operating system design have been identified. One set of methods, based on the notion of *interacting processes*, implements the generic operating system tasking model and uses high-level concurrency and synchronization constructs, such as processes, monitors, rendezvous, and condition variables. This model may be implemented within concurrent programming languages such as Ada [Ada 83] or Modula-1 [Wirth 77], where the run-time support for the language is essentially the operating system. An alternative method has been to provide a library of concurrency primitives that can be invoked from a conventional sequential programming language; this "tool-kit" idea is found in many commercial real-time development systems, such as iRMX 86 [Intel 82] and VAXElan [DEC 83]. A second set of methods, which we call *slice-based*, deals explicitly with processor scheduling and time, by decomposing tasks into atomic schedulable units termed slices, chunks, or strips [e.g. Orn et. al 75, Donner 84, Baker and Scallon 86]. Here, the operating system is typically a *cyclic executive* that employs static table-driven schedules to dispatch slices so that periodic and sporadic (i.e. aperiodic) timing constraints are met [Mok 83]. One goal of our real-time systems research is to combine the flexibility, elegance, and generality of the concurrent programming approach with the explicit scheduling, timing predictability, and efficiency of the lower-level slice methodology [Shaw 86].

Our proposed research involves the design and development of libraries of reusable software components and software structures, that can be configured to produce real-time operating systems for particular applications. The work covers topics in software engineering, programming-in-the-large, and real-time processing. Specifically, we are developing:

1. a module structuring, interconnection, and interface methodology. This is based on an acyclic graph model where the nodes represent modules (restricted to processes and abstract data types) and the edges denote procedure import and export relations.

2. a prototype for real-time operating systems design, involving libraries of reusable components. Here, we wish to identify, design, and build a variety of useful components and structures, such as interrupt handlers, critical section locking mechanisms, message passing, timers, schedulers, input-output systems, cyclic executives, and producer-consumer applications.

3. a graphics front end for interacting with designs. This is essentially a graph editor that permits the construction, modification, and annotation of our module structures, including the association of modules with their code files.

Initially, we expect to study and construct modules for a uniform shared-memory multiprocessor environment. We will be emphasizing timing predictability and the exploitation of concurrency. The research should be applicable not only to the real-time operating systems area but also to more general software engineering issues related to, for example, reusability, module definition methods, interconnection schemes, and naming problems.

The next section describes some related work. Section 3 then presents our research in more detail. Section 4 outlines our plan to accomplish this work.

2. Related Work

We briefly describe past work that is related to the proposed research or that has influenced it in the three areas listed in the last section.

Techniques for defining and interconnecting modules for real-time systems have been reported from several development and research projects, including the A-7 project by Parnas and his colleagues [e.g. Parnas et. al 85], the DARTS methodology [Gomaa 84], the Mascot method used in the U.K. [Simpson and Jackson 79, Mascot 80], and the GEM system [Schwan et. al 85]. Parnas distinguishes three types of software components and structures: modules (sets of related programs acting as a unit of work for a programmer or team), processes (a run-time unit which may involve parts of one or more modules), and programs that are related by a "uses" structure. Modules are organized in a tree hierarchy, and are based on Parnas' information hiding principle [Parnas 72]. It is argued that this modular approach to program design is a significant and effective design tool (when accompanied by documentation in the form of a module guide) and that it yields reusable software.

In DARTS, which is a complete software development methodology for real-time systems [Gomaa 84, 86], modules are used as the components of processes ("tasks") and as the means for implementing process communication and synchronization ("task interfaces", in Gomaa's terminology). A process is composed of an acyclic structure of modules; modules may also be shared across processes. The scheme was used successfully in two projects, a robot controller and a vision system, and has been particularly useful for incremental development and prototyping. The Mascot approach decomposes a system into processes, channels, and data repositories called "pools", where processes employ message-passing channels to communicate with each other and with input-output activities. This model, which appears attractive because of its cleanliness and simplicity, has apparently been the basis for a number of military systems in the U.K.

Historically, Parnas' research on modularization, information hiding, and hierarchies has influenced much recent software engineering work. It lead to the idea of "programming-in-the-large", defined in a widely referenced paper by [DeRemer and Kron 76]. For our research, the notions of module interfaces and interconnection languages [e.g. Tic 80] are most significant, where the interfaces of components are defined by the "resources" that the module *imports* (i.e. requires from other modules) and *exports* (i.e. makes available to other components). These ideas have also been implemented in contemporary programming languages, notably as part of the module in Mesa [e.g. Lampson and Redell 80], package in Ada [Ada 83], and module in Modula-2 [Wirth 82].

Many of the works cited above and in the Introduction also have proposed a variety of different components and structures for real-time operating systems. In addition to these, there are software mechanisms based on some relatively unknown but very useful synchronization primitives [Faulk and Parnas 83], abstractions implemented in interesting research systems such as Thoth [Cheriton et. al 79], and current developments for Ada run-time environments [e.g. Baker and Jeffay 87]. The design philosophy and some of the ideas in the FAMOS project from the mid 1970's are also related to our work, in particular, their structuring of a *family* of operating systems using common components [Habermann et. al 76]. Our goals are also similar to those of the Choices project which aims to build an extensible family of real-time operating systems using the ideas of class hierarchy and inheritance [Campbell et. al. 87]. While concurrency seems to be handled well, at least logically, few systems provide higher-level modules for explicitly dealing with time and assuring timing predictability.

The third area of our proposed research, i.e. the construction of interactive systems that permit a designer to edit and manipulate graphical representations of systems at various levels, has long been attractive but is still in its infancy [e.g. IEEE 85]. For defining and manipulating networks of program modules, many of the ideas in the Pegasus system seem particularly useful, especially their methods for refining and abstracting nodes and edges of the interconnection graph, for displaying the graph, and for associating the represented program code with the displayed elements [Moriconi and Hare 85]. The analog model implemented in the STILE system for programming real-time applications provides a particularly simple, but perhaps restricted, set of primitive entities (processes with "ports" for communications via "links") [Brown and Weide 84]; this graphical programming system, still under development, takes a data flow approach, and appears applicable to relatively small control problems. Another project involving a graphical system for interconnecting modules is reported in [DeMarco and Soceneantu 84]; the interest here lies mainly in their two-dimensional generalization of UNIX pipes.

3. Proposed Research

In order to handle the complexities of engineering real-time software, our philosophy is to take the simplest possible approach and model that is compatible with producing efficient, predictable, robust, maintainable, and reusable code. Thus, for example, we would like our software objects to be static, so that compile-time modules map directly to the run-time objects. Hierarchical structures of components are preferred so that top-down "divide and rule" strategies can be employed and to reflect module refinements and abstractions. For the most part, it will be assumed that code is permanently stored in main memory (e.g. no swapping); operating system issues related to storage management for dynamic, virtual memory systems will not be addressed because these systems do not normally lend themselves to timing predictability. Some emphasis will be placed on lower level kernel mechanisms in the operating system related to time control, scheduling, locking, and synchronization, since these facilities often determine real-time behavior. We have had some success in using our *flow expression* notation as a formal scheme for describing and understanding software architectures for concurrent systems [e.g. Shaw 78, Chi and Shaw 86] and would continue to use this method where appropriate.

3.1 Module Definitions, Interfaces, and Hierarchies

We assume only two kinds of components: *process* modules and *abstract data type* (ADT) modules. Processes are the active entities in the system and have their conventional meaning; each process corresponds to a sequential program with an associated real or virtual processor. An ADT is an encapsulated data object with an associated set of "public" access procedures; the object can *only* be accessed through these procedures. Initially, these objects will be static, with modules defining instances rather than classes; i.e. each instance of a process or ADT is in one-to-one correspondence with its code.

An ADT module *exports* its access procedures for possible use by other modules and may *import* ADT modules or particular procedures from ADT modules; if an entire module is imported, then all public procedures of that module are available for use by the importer. A process exports nothing and may import one or more procedures from ADTs. Process communication thus occurs through ADT modules. Procedure and module import/export is *the* module interface scheme that we propose to use in our work.

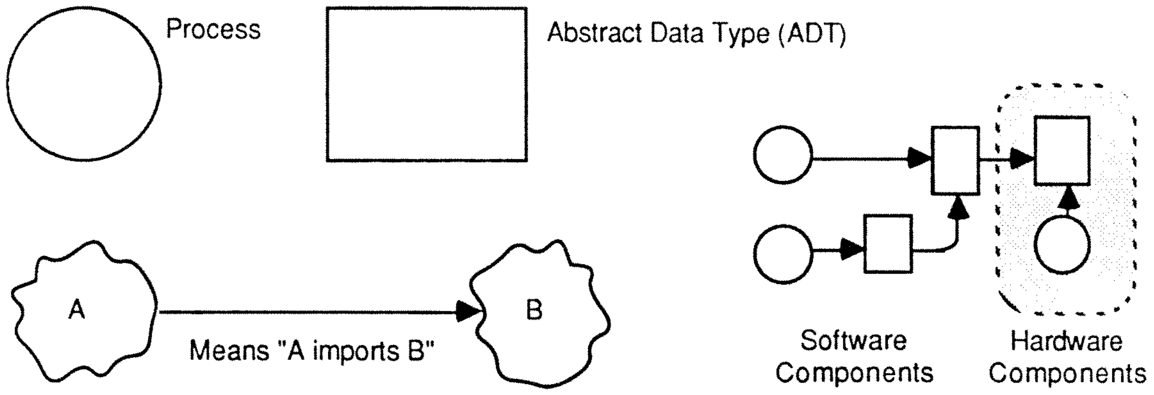
We hope that the module structure can be restricted to an acyclic directed graph, where the nodes are modules and the edges represent the "import" relation; i.e. an edge directed from module M_1 to M_2 indicates that M_1 imports M_2. Conceptually the lowest levels of the hierarchy will define the real or abstract machine architecture of the system. At the highest level, the "roots" of the hierarchy, will be the process modules. The structure is a hierarchy rather than a strict tree since we wish to permit sharing of modules.

These ideas are illustrated in the hypothetical designs of Figures 1 and 2. Figure 1 shows a simple cyclic executive process responsible for scheduling n User processes by giving each 1/n th of a time interval in a round-robin fashion. (In this case, our executive is identical to a simple time-sharing system scheduler.) We represent conceptually the unprotected machine by an ADT, and the protected clock and interrupt registers by separate process and ADT modules respectively. The interrupt from the hardware Clock process, labeled "Tick", is modeled as a procedure call to the Interrupt Handler ADT. Figure 2 contains a producer-consumer design with preemptive scheduling; at the highest level, the Buffer Manager ADT exports the abstractions of two procedures Get and Put as shown. Input-output handling is based roughly on the approach used in Modula-1.

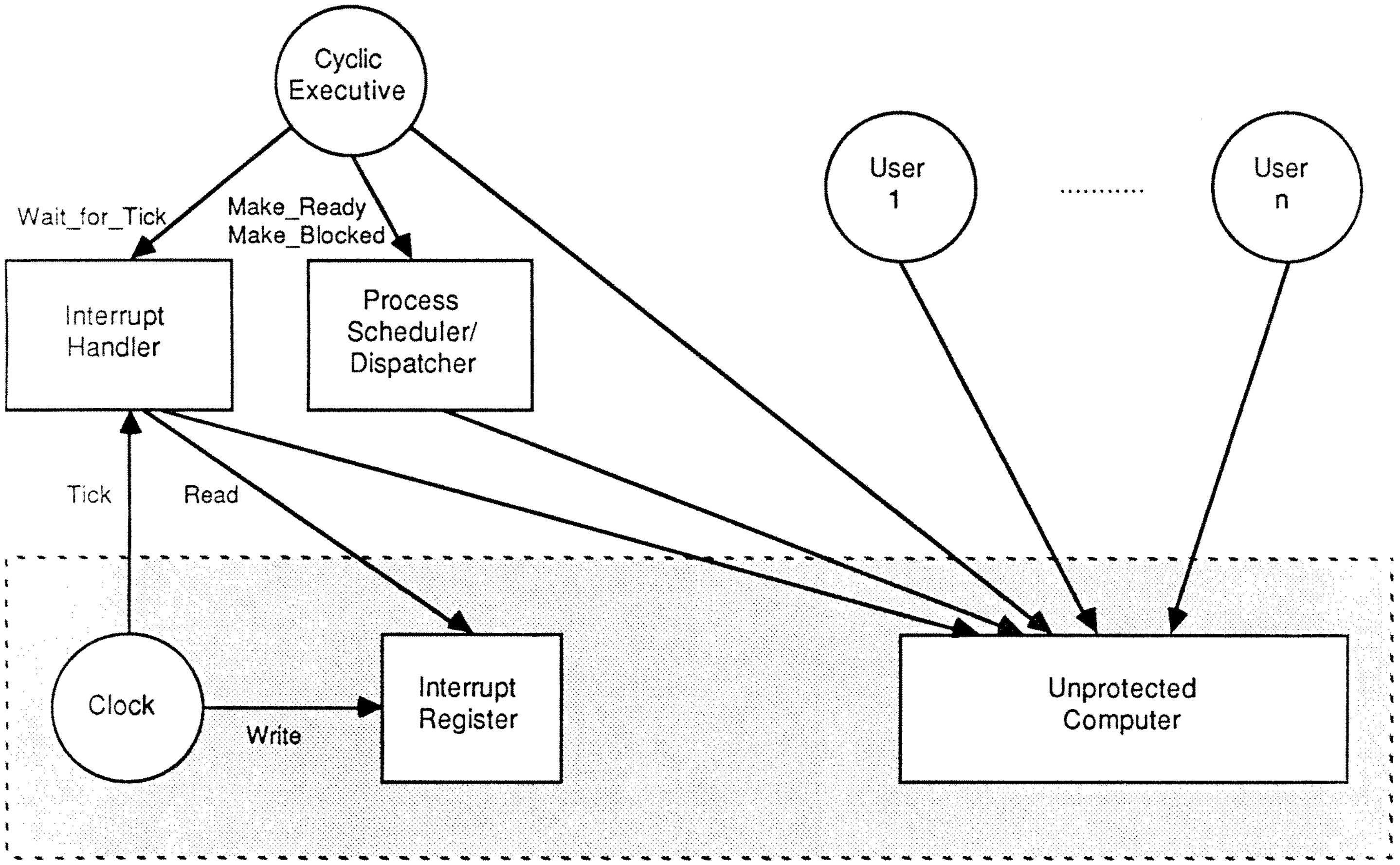
Based on some paper experiments and the analysis of some local software, we believe that this model for components is suitable for real-time operating systems and has the advantages of simplicity and flexibility. It is simpler than, for example, Parnas' more general scheme in that it combines processes, ADTs, modules, and the import/export relation. It appears more flexible than either the DARTS or Mascot methods in that the synchronization, locking, communications, and scheduling mechanisms can be arbitrarily defined through ADTs to suit the needs of the particular system. A particularly pleasing aspect is that the process/ADT components appear convenient and suitable for treating interfaces at the hardware architectural level - for defining "low-level" kernel software and for describing hardware interfaces connected to a kernel.

For our research, we propose to develop the above model in conjunction with the module designs and implementations described in the next subsection. Timing behavior will be incorporated by associating timing properties with the procedures of the ADTs and by synchronizing modules with various predictable software clocks. Methods for computing worst and best case execution times will be further studied, using the assertional techniques recently developed [Shaw 87]. We also hope to verify global system timing constraints [Jeffay 87].

Much research is also necessary on the appropriate abstractions for hardware features that directly impact software, such as input-output devices and processor-memory interfaces. The intent is to describe these features as processes or ADTs also, as in Figures 1 and 2. While we will not be concerned with the problems of dealing with several programming languages at the same time, it is important that the modules and structures be translatable into a contemporary language, such as Ada or Modula-2 or C, and that efficient code results. For example, many of the lower level procedures may be most efficiently implemented as macros and common procedure code may be shared such as in the two semaphore modules of Figure 2. As part of this translation, we also need to study naming problems, i.e. how to name modules, procedures, and parameters, and store and retrieve different *versions* of them. In the longer term, it may be useful to include dynamic objects (i.e. having process or ADT classes). (An even longer term research problem which we do *not* propose here is to develop some practical methods for expressing the *semantics* of modules.)

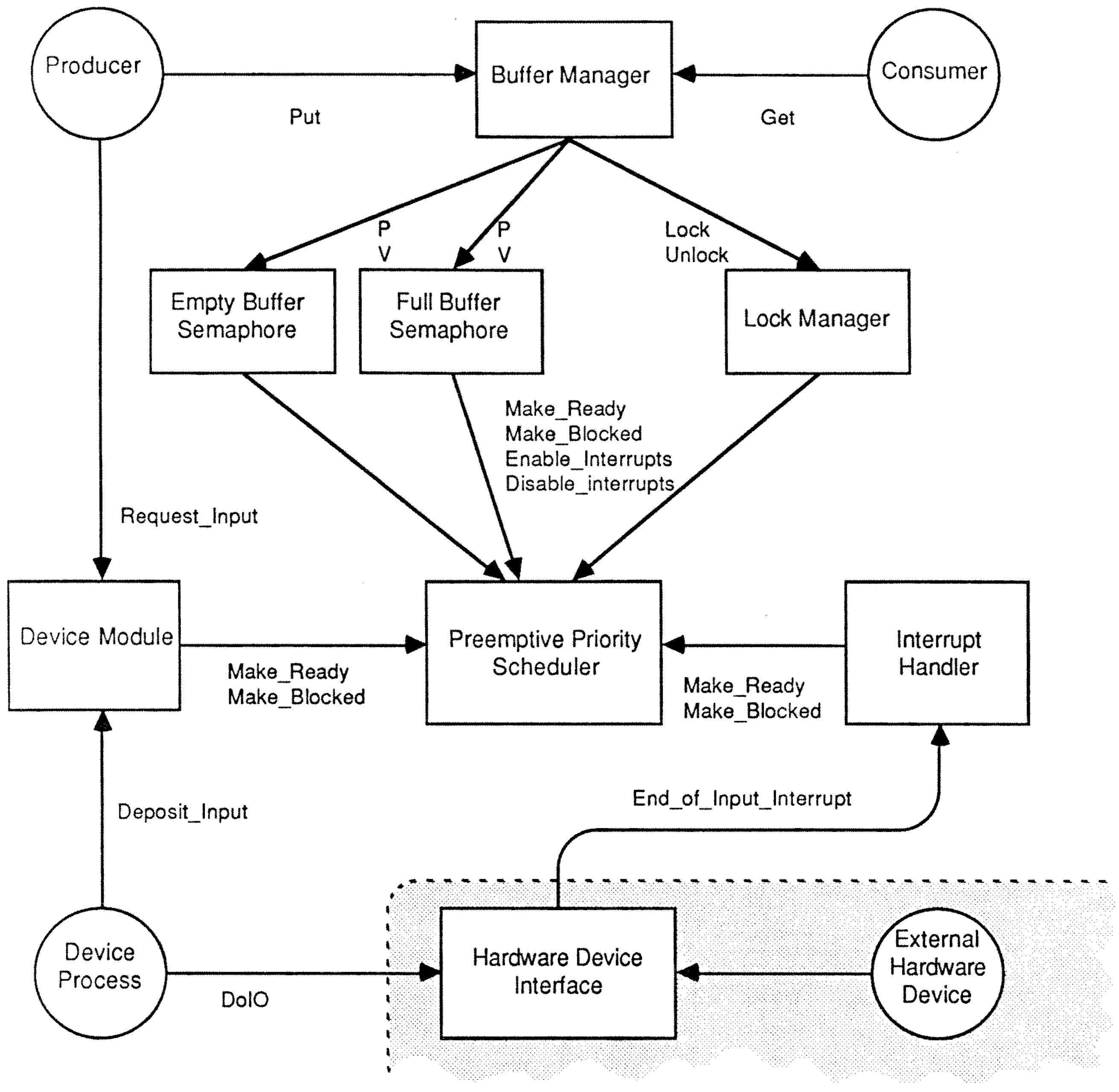


a) Graph Notation



b) Simple Cyclic Executive

Figure 1



Producer/Consumer with Input
Figure 2

3.2 Components and Structures for Real-Time Operating Systems

Our goal in this phase of the research is to design and construct a library of reusable software components and structures for shared memory multiprocessor configurations, based on the module definition and interconnection scheme described in the last section. The modules should permit fast prototyping and testing of different software architectures and parts for real-time operating systems. Within our current computing environment, we would use Sun or microVAX workstations with the C programming language and UNIX as the development system. The target system could be several Motorola 68000's and a large common main store connected via a common bus; flexible programmable signal generators, which produce periodic and sporadic real-time interrupts and data, would also be connected to the bus to permit realistic simulation of an arbitrary external world. (The signal generators could be identical to the processor boards used for the multiprocessor function.)

First, it is necessary to identify useful objects and encapsulate them as processes or ADTs. There is much experience in identifying similar kinds of objects at some of the higher levels of abstraction of systems; for example, most texts on general purpose operating systems contain descriptions of many components that are also applicable to the real-time area [e.g. Bic and Shaw 88]. Producer-consumer subsystems and a rudimentary file system are examples.

Detailed definitions of lower-level objects are not as common. One interesting and practical set of objects are those necessary to solve the critical section problem when strict timing constraints are also included. The Lock Manager ADT in Figure 2, which provides *lock* and *unlock* primitives to surround a critical section of code, could encapsulate one such set. We have designed seven different ADTs for this problem and convinced ourselves that any or all the seven may be used in a single shared-memory, multiprocessor system.

There are "spinning" (busy-wait) locks and "block" (suspend) locks that block their caller if the lock is in use; within these, there are locks which permit preemption in the critical section and those that do not. On a single machine in a multiprocessor configuration, there is also a need for both busy-wait and suspend locks, which provide for preemption; there is also a requirement for a single processor guaranteed immediate-entry lock (e.g., by disabling interrupts).

The interest and challenge is to define ADTs for these that have predictable and fast performance when concurrent access is made to the lock procedures. Aids in the form of hardware instructions are significant at this level. The possibility of starvation on a "test-and-set" loop is normally not acceptable in a hard real-time application and $O(n^2)$ time guarantees, where n is the number of processors, are often too slow. The *tinc*/*dect* instructions on the ICL2900 series computer [Keedy and Freisleben 85] appear to be a good hardware basis for solving this problem, but more study and testing is necessary. (*tinc* is an indivisible test and increment operation, and *dect* is a decrement followed by a test).

Another fundamental set of lower level objects are clocks and timers that are connected to, and abstractions of, hardware timers. In recent work, we have shown that a surprising (to us) number of different software clocks may be useful for organizing and programming real-time tasks [Shaw 86]. These include local (decentralized) and global clocks, clocks with fixed interval ticks and those with variable interval ticks, and timers implementing a broad range of tick granularities. We wish to develop these "logical" clock ideas further by incorporating them into subsystem structures where convenient (for constraining timing behavior), by comparing both ADT and process implementations, by linking them to hardware timers to assure real-time correctness, and by searching for other useful logical timing mechanisms. The use of these clocks is one way in which we hope to incorporate time as a first class object in programs and help achieve timing predictability.

Other components and structures that require careful definition, analysis, and construction are input-output subsystems, ADTs for process synchronization, preemptive and non-preemptive schedulers (dispatchers), complete run-time systems, cyclic executives, and interrupt-handler ADTs. Input-output is especially important since these modules connect directly to the external environment where the real-time constraints of a problem are ultimately determined. At least two common organizations need to be constructed and evaluated. Both have been used in general purpose operating systems. One, implemented in the Modula-1 language, associates a driver process with an input-output device, and user requests pass through a special device monitor (Figure 2 illustrates this approach). The second method just employs an encapsulated data object that offers input-output commands to user processes and implements them directly; this is used, for example, in Concurrent Pascal and Mesa. Also, interrupt handlers can be treated as either processes or ADTs.

Some general implementation issues were mentioned in Section 3.1. Others include how to interchange functionally-similar components easily, so that many variations of a system may be prototyped. For example, it should be easy to substitute a spinning lock ADT for a suspend lock version, without changing the code of the user of the ADT; a preprocessor may be written to handle this problem. We would also like to configure systems containing different scheduling strategies for their subsystems; parts may be preemptive while others are non-preemptive, parts may be scheduled according to static numerical priorities while others are scheduled based on time (e.g. deadline scheduling), and we wish to have a cyclic executive operating at the same time as a more conventional process scheduler.

3.3 A Graphics Editor For Software Architectures

The aim is to develop a graphical specification language for programming-in-the-large. By this, we mean a set of two dimensional representations and methods to edit these representations, that permit an interactive user to define and manipulate our software modules and their structures.

At the user interface level, we expect the appearances of designs to be similar to those in Figures 1 and 2. The system will be constructed within a multiple window environment so as to permit a multiplicity of views. It should be possible to display (parts of) several designs and/or parts of the same system simultaneously in different windows. Some refinement and abstraction facilities will also be developed, permitting module structures to be treated both as a unit and also in terms of its lower level constituents.

Not surprisingly, the editor will be based on an underlying graph model. Nodes correspond to process and ADT modules, and edges denote import/export relations. Because of our hierarchical organization principle, this graph is a directed *acyclic* graph (dag). Several attributes will be associated with the edges and nodes, including their graphic images and geometry, textual labels that identify the element, annotations that describe their function and properties, and source code that refers to program files. A long term storage system will be used to store and retrieve graphs and their attributes.

A prototype front end for a graph editor that is meant for this application has been constructed as part of Binding's Ph.D. research [Binding 87]. One purpose of this implementation is to test the features of a general purpose user interface tool kit that he has built on a Sun workstation using Unix. The graph editor design allows a user to create, delete, move, annotate, refine, and view our proposed images of nodes, edges, and graphs in a system with an arbitrary number of overlapping windows for multiple views. The Unix file system provides the long term storage. This system allowed some early experimentation with user interface issues, such as the iconic representations for modules and interconnected structures, menu and selection techniques for the editing operations, and good ways to display and manipulate textual information associated with the graph.

There are many research problems connected with this part of the project. These include some difficult front-end geometry and interface problems, such as how to remove, add, or copy a subsystem (a subgraph) in an existing structure, how to drag or move parts of the graph around a screen, and how to consistently reflect the results of geometric or layout editing when several views of parts of the same structure are being changed at the same time. At a higher level, it is important that consistency checks be made between the textual code of each module with its definition of interfaces and the graphical representations; the text should be consistent with the graph and vice versa. Naming, storing, and version problems mentioned in the last section are clearly the same here. In fact, an overall research goal is to maintain a different sort of consistency among the three phases of the research: inevitable changes in the model, in the way components are defined, and in the graphics front end must be accommodated and reflected in all three areas.

4. Research Plan and History

Our work on real-time systems started in summer 1984 with support from the Washington Technology Center and Boeing Aerospace Corporation. The first tasks involved defining the nature of real-time systems, and studying the software organizations and tools used in various commercial and development systems. Some time was also spent investigating a generic application that has all the features of the general problem area; here, we also built some simple prototype software and a simulator for a variety of real-time scheduling disciplines. The application is an air traffic monitoring system (ATM) that tracks and communicates the position of all aircraft in a given airspace volume. In the actual ATM, real-time inputs and outputs include radar signals and control, communications received when new aircraft enter the volume and sent when aircraft leave the volume, operator input for querying and controlling the system, and one or more screen displays showing aircraft tracks and various textual data. These research efforts were summarized in [Shaw et. al 85], under the headings: timing specifications in concurrent programs, operating systems and concurrent programming languages, programming-in-the-large, and simulator/prototype implementations.

We plan to proceed as follows with the proposed research described in the last section. All three parts (Section 3.1, 3.2, and 3.3) are mutually reinforcing and will be done in parallel. We will continue to test the completeness and convenience of our module definition and structuring scheme through paper experiments with real and hypothetical software. It is also necessary to precisely specify the scheme with some form of machine processable description language; a language that is simpler than but similar to a Modula-2 *definition module* or an Ada *package specification* will be designed and implemented.

At the same time that we actually construct some of the higher and lower level modules and structures mentioned in Section 3.2, we will have to establish some conventions for interfaces for C-language versions of process and ADT modules, conventions for naming library elements, and the translations into appropriate Unix file system implementations. A *lightweight* process kernel will be built so that target machine processes can be efficiently stored, suspended and made ready. The proposed graphics front end will initially be used for documenting designs and will be based on a second version of the graph editor. One part of the project will be to establish links between the code files for our modules and the graph representations.

Many parts of this proposal are directed towards general software engineering problems. The elements that make it specific to real-time operating systems are: process and ADT modules are static, no virtual memory is assumed, a simple hierarchic structuring scheme (a dag) is deemed sufficient, run-time code is mainly memory-resident, a simple but timing predictable file system is adequate, many clock and timing tools (modules) are proposed, some emphasis is given to low level but timing-critical components such as interrupt-handlers, hardware architecture features are included in the process/ADT model, and scheduling mechanisms are to be time-based, such as a cyclic executive, as well as priority-based.

5. References

[Ada 83] *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, U.S. Dept. of Defense, Jan. 1983.

[Baker and Jeffay 87] T.P. Baker and K. Jeffay. Corset and Lace, Adopting Ada Runtime Support to Real-Time Systems. *Proc. of the 1987 IEEE Real-Time Systems Symp.*, San Jose, CA, Dec. 1987, pp. 158-176.

[Baker and Scallon 85] T.P. Baker and G. Scallon. An architecture for real-time software systems. TR85-06-04, Dept. of Computer Science, Univ. of Washington, Seattle, WA, June 1985. (also published in *IEEE Software*, May 1986).

[Bic and Shaw 88] L. Bic and A. Shaw. *The Logical Design of Operating Systems - 2nd Edition.* Prentice-Hall, 1988.

[Binding 87] C. Binding. The specification and implementation of a user interface management system based on a uniform output model. TR87-10-02, Dept. of Computer Science, Univ. of Washington, Seattle, WA, October 1987. (Ph.D. dissertation)

[Brown and Weide 84] M.E. Brown and B.W. Weide. Automating process-to-processor mapping under real-time constraints. *Proc. IEEE 1984 Real-Time Systems Symp.*, Austin, TX, December 1984.

[Campbell et. al. 87] R. Campbell, G. Johnston, and V. Ruzzo. Choices (Class Hierarchical Open Database for Custom Embedded Systems). Operating Systems Review, Vol. 21, No. 3, July 1987, pp. 9-17.

[Cheriton et. al 79] D. Cheriton, M. Malcolm, L. Melen, and G. Sager. Thoth, a portable real-time operating system. *Comm. ACM* 22, 2 (Feb. 1979), 105-115.

[Chi and Shaw 86] U.H. Chi and A. Shaw. Using flow expressions to specify timing constraints in concurrent programs. TR86-05-03, Dept. of Computer Science, Univ. of Washington, Seattle, WA, May 1986.

[DEC 83] *VAXElan Technical Summary.* Digital Equipment Corporation, Concord, Mass., 1983.

[DeMarco and Soceneantu 84] SYNCRO: a dataflow command shell for the Lilith/Modula computer. *Proc. 7th Int. Conf. on Soft. Eng.*, IEEE Computer Society Press, 1984, pp. 207-213.

[DeRemer and Kron 76] F. DeRemer and H. Kron. Programming-in-the-large vs. programming-in-the-small. *IEEE Trans. on Soft. Eng.* 2, 2 (June 1976), 80-86.

[Donner 84] M.D. Donner, Control of walking: local control and real-time systems. CMU-CS-84-121, Dept. of Computer Science, Carnegie-Mellon Univ., May 1984. (Ph.D. dissertation).

[Faulk and Parnas 83] S.R. Faulk and D.L. Parnas. On the uses of synchronization in hard-real-time systems. *Proceedings of the IEEE Real-Time Systems Symposium*, Arlington, VA, 1983.

[Gomaa 84] H. Gomaa. A software design method for real-time systems. *Comm. ACM* 27, 9 (Sept. 1984), 938-949.

[Gomaa 86] H. Gomaa. Software development of real-time systems. *Comm. ACM* 29, 7 (July 1986), 657-668.

[Habermann et. al 76] A.N. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Comm ACM* 19, 5 (May 1976), 266-272.

[IEEE 85] Visual Programming. Special Issue, *IEEE Computer*, August 1985.

[Intel 82] *Introduction to the iRMX 86 Operating System*. Intel Corporation, Santa Clara, CA, 1982.

[Jeffay 87] K. Jeffay. Concurrent Programming with Time. TR87-10-03, Dept. of Computer Science, Univ. of Washington, Seattle, WA, Oct. 1987.

[Keedy and Freisleben 85] J.L. Keedy and B. Freisleben. On the efficient use of semaphore primitives. *Information Processing Letters* 21, 1985, 199-205.

[Lampson and Redell 80] B.W. Lampson and D.W. Redell. Experience with processes and monitors in Mesa, *Comm. ACM* 23, February 1980, 105-117.

[Mascot 80] Mascot Supplier Association. *The Official Handbook of Mascot*. Computing Standards Section, Royal Signals and Radar Establishment, Worcestershire, UK, Dec. 1980.

[Mok 83] A.K. Mok. Fundamental design problems for distributed systems for the hard real-time environment. MIT/LCS/TR-299, MIT, 1983. (Ph.D. dissertation).

[Moriconi and Hare 85] M. Moriconi and D. Hare. Visualizing program designs through PegaSys. In [IEEE 85], 72-85.

[Orn et. al 75] S.M. Ornstein, W.R. Crowther, R.D. Kraley, A.M. Bressler, and F.E. Heart. Pluribus - a reliable multiprocessor. *Proc* AFIPS 1975, *Conf.*, 1975, pp. 551-559.

[Parnas 72] D. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM* 15, 12 (Dec. 1972), 1053-1058.

[Parnas et. al 85] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Trans. on Soft. Eng.* 11, 3 (March 1985), 259-266.

[Schwan et. al 85] K. Schwan, T. Bihari, B. Weide, and G. Taulbee. GEM: operating system primitives for robotics and real-time control systems. OSU-CISRC-TR-85-4, Computer and Information Science Research Center, The Ohio State Univ., Columbus, Ohio, Feb. 1985.

[Shaw 78] A. Shaw. Software descriptions with flow expressions. *IEEE Trans. on Soft. Eng.* 4, May 1978, 242-254.

[Shaw et. al 85] A. Shaw, C. Binding, W-L. Hu, and K. Jeffay. Research in real-time systems. TR85-12-05, Dept. of Computer Science, Univ. of Washington, Seattle, WA, Dec. 1985 (Presented at the IEEE Real-Time Systems Workshop, Boston, Feb. 1986).

[Shaw 86] A. Shaw. Software clocks, concurrent programming, and slice-based scheduling. *Proc. of the 1986 IEEE Real-Time Systems Symp.*, New Orleans, LA, December 1986, pp. 14-18.

[Shaw 87] A. Shaw. Reasoning about time in higher-level language software. TR87-08-05, Dept. of Computer Science, Univ of Washington, Seattle, WA, Aug. 1987. (To appear in *IEEE Trans. on Software Engineering.*)

[Simpson and Jackson 79] H.R. Simpson and K.L. Jackson. Process synchronization in Mascot. *Comput. J.* 22, 4 (1979).

[Tic 80] W. Tichy. Software development control based on system structure description. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Jan. 1980.

[Wirth 77] N. Wirth. Toward a discipline of real-time programming. *Comm. ACM* 20, 8 (Aug. 1977), 577-583.

[Wirth 83] N. Wirth. *Programming in Modula-2*, 2nd edition. Springer-Verlag, NY, 1983.