

Building a Real-time Kernel: First Steps in Validating a Pure Process/Adt Model

H. REBECCA CALLISON AND ALAN C. SHAW

Department of Computer Science FR-35, University of Washington, Seattle, WA 98195, U.S.A.

SUMMARY

A model based on simple, unadorned processes and abstract data types has been proposed for the design and implementation of real-time systems. We describe our generally successful experiences in using this model for the construction of a real-time operating system kernel containing services for process control, synchronization, time and input-output. Our work and results focus on the usefulness of the scheme for designing modules at hardware/software interfaces, for predicting deterministic timing behaviour, and for software development.

KEY WORDS Abstract data types Kernel Operating systems Processes Real-time systems Timing

INTRODUCTION

Our purpose is the initial testing and validation of a pure process/abstract data type (adt) model for the development of real-time systems.

A number of requirements distinguish real-time systems from other computing applications, particularly:

1. Predictable timing performance (to guarantee time-critical system activities).
2. Concurrency control (to monitor and control external physical processes that are inherently parallel).
3. Reliable, uninterrupted operation (to satisfy the demand for continuous service that characterizes many real-time systems).

The need for predictable timing performance and for reliability often derives from concerns for the safety of humans interacting with the system and the safety of property under the control of the system. These safety concerns lead to stringent demands for logical and temporal correctness.

In earlier work^{1,2} a simple process/adt model for the design of real-time systems was proposed. The model is essentially a scheme for defining, structuring and connecting modules. We build on fundamental concepts of contemporary software engineering, e.g. [References 3, 4 and 5](#), adapted to the requirements of real-time systems.

One long range goal of this research is the development of a prototype operating system family, where particular systems are composed from libraries of reusable

components. Our goals are similar to those of the Choices project,⁶ which aims to define a family of operating system capabilities as an object-oriented hierarchy extending down through the interface level of the hardware architecture. We did not use the more general object-oriented approach of Choices, which provides object classes, inheritance and dynamic instances, because it appeared that a simpler programming model would be sufficient and would better serve the specific predictability needs of real-time computing. Similarly, current language-based approaches incorporating the concept of modules for programming-in-the-large⁷⁻⁹ seem too general to be applied effectively to real-time problems, primarily because they do not lead to predictable performance.

To validate our ideas, we have designed and constructed a real-time kernel. There are three results and contributions of this research. First, we conclude that a *simple, unadorned* process/adt model offers an effective and tractable approach to real-time system design and implementation. A second result is that the model works particularly well at the difficult *hardware/software* interface. Last, our experiments indicate that the model provides a useful framework for analysing the *deterministic timing** behaviour of software.

THE PROCESS/ADT MODEL

The pure process/adt model provides two primitive types for the description of systems: *process* and *abstract data type (adt)* modules.^{1,2}

Processes are the active entities of the system. The presence of multiple processes connotes physical and/or logical concurrency; each process is a sequential program that executes concurrently and possibly interacts with other processes of the system. Adts are passive. They generally encapsulate data and provide public procedures for manipulation of data. They may be used to manage resources and to implement mechanisms for process interaction. Processes invoke procedures exported by adts. Adts may, in turn, use the procedures exported by other adts.

A process/adt model for a system consists of all the processes and adts in the system as well as the particular procedures called by each process and adt; it may be represented as a directed graph as illustrated in Figure 1. An arrow drawn from module A to module B indicates that A imports some service of B. The particular procedures used by A are indicated by a text annotation associated with the arrow.

Each entity of a process/adt graph is interpreted to be an instance of the module-type. Multiple instances of a single process or adt are allowed; these are assumed to be identical copies which may share code.

The use of the model is restricted to a static environment consistent with real-time predictability requirements. All programs are memory resident. Processes and data structures are either statically declared in source code or are created during an initialization phase that precedes real-time execution of the system.

The example of Figure 1 is the keyboard interface section of the kernel. There are three active entities in this subsystem: the *application process* which requests the next character input at the keyboard by invoking `KeyboardADT.getchar`; a `keyboard_in-`

*By deterministic timing we mean exact execution time or a tightly bounded time. We differentiate deterministic timing behaviour from stochastic timing behaviour, which uses statistical measures, such as average program performance.

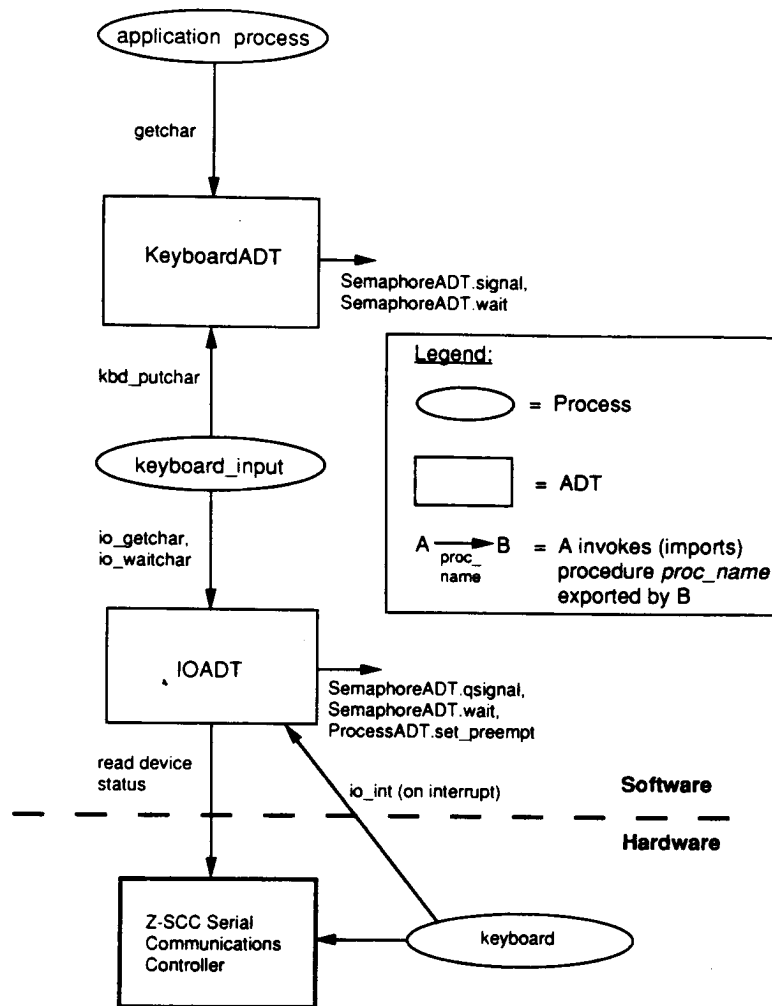


Figure 1. The process/adt graphical representation

put process which accepts and buffers characters received by the keyboard interrupt handler of the IOADT; and the *keyboard* itself which is a *hardware* process. The keyboard is modelled as a process to reflect the active nature of keyboard input.

There are also three adts in this subsystem, two implemented in software and one in hardware. The IOADT provides a single character buffer between the hardware device and keyboard_input. The KeyboardADT manages more extensive buffering for the user. Both software adts invoke the wait/signal services of the SemaphoreADT, described in the 'Kernel design' section below, to synchronize access to the resources managed. These calls may cause the process to block in the kernel awaiting resource allocation. Z-SCC status information, including the last received keyboard input, is read and written under the control of the system software, so this passive entity is modelled as an adt.

OVERVIEW OF THE VALIDATION EXPERIMENT

We chose to build a kernel rather than some higher level software because it allowed us (1) to investigate the flexibility of the model at the interface between hardware and software where much real-time behaviour is ultimately determined and (2) to develop a testbed for further research in real-time systems. In this section we describe the functions of the kernel, the target environment in which it executes, our development environment and a brief history of the development process.

Functions of the kernel

The kernel provides synchronization and timing services for lightweight processes*executing in a single processor. The services provided by the kernel include:

1. Process management (scheduling).
2. Process synchronization based on binary semaphores.
3. Timing services, including process delays and time-outs.
4. Character level terminal IO.

Wait ('P') and signal ('V') operations are provided on binary semaphores. A time-out may be associated with a wait operation; the waiting process is then reactivated on the semaphore signal or on expiration of the time-out period, whichever comes first. The signal operation has two semantic options: a classic signal, which may invoke the scheduler, and a quick signal, available to interrupt handlers, which defers possible rescheduling until the interrupt handler has been exited.

Time management services also include calendar and real time (timer interrupt count) and process delays. As with semaphore time-outs, delays for an absolute interval or until a particular time of day may be requested. The semantics of the delay guarantee that a waiting process will be reached for execution within 1.5 ms of the expiration of the time-out period in this implementation. (This approach differs from other delay operations, e.g. Ada's delay, which guarantee no maximum bound on the delay.) The exact time at which the readied process will be dispatched depends on the priority and CPU demands of other ready processes. The scheduling discipline employed by the kernel uses static priorities with process pre-emption.

The IO package provides character-level IO services which have compatible calling sequences with the C standard procedures `getchar` and `printf`. Commands to clear the screen and to position the cursor at a specific screen location are also available.

In keeping strictly with the philosophy of the model, *all* kernel services are accessed through procedures exported by kernel adts.

Development environment

Our implementation system, shown in [Figure 2](#), consists of the target system, a development host, network file server, and interconnecting LAN.

The execution hardware for the kernel is a diskless Sun-2. The major components of the target system are a 10MHz MC68010 processor with 2 MBytes of RAM

*Processes executing in a single address space.

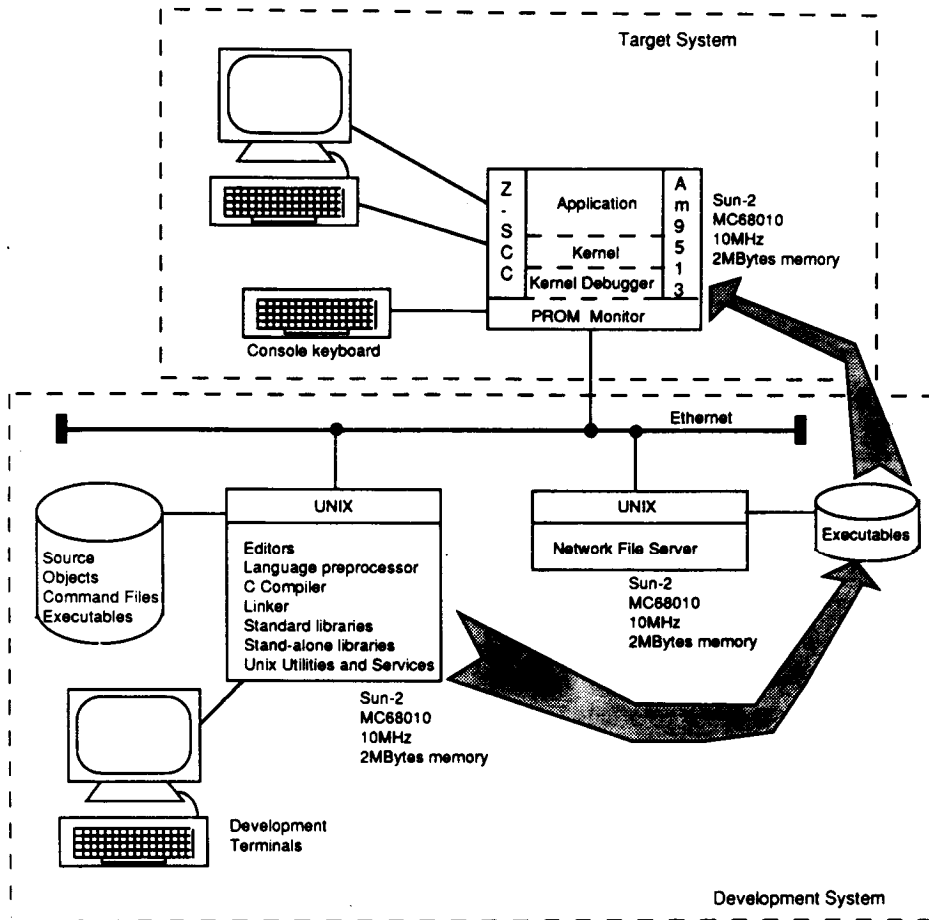


Figure 2. The kernel environment

memory and console keyboard, a Z8530 Z-S-C Serial Communications Controller interfacing a second keyboard and a CRT through a full-duplex 9600 bps serial channel, an Am9513 timer and an Ethernet interface.

The MC68010 has a relatively simple architecture and apparently simple timing behaviour.¹⁰ Except for instructions cached in a two-instruction loop buffer in very restricted circumstances, all memory references for instructions and data access physical memory. There are no cache operations, and our kernel implements no virtual memory support. These features simplify the timing prediction problem significantly.

The Am9513 timer chip provides five timers that are programmable sources of periodic and one-shot interval interrupts.¹¹ Timer interrupts drive a 10 ms processing cycle in the kernel. The keyboard and screen accessed through the Z-S-C are used as standard input and output devices.¹²

The target environment also includes some additional software. The Sun PROM monitor¹³ is permanently resident in the target system and executes concurrently

with the real-time application. It fields an unmaskable interrupt at 2 ms intervals and performs background control functions such as monitoring the console terminal keyboard. Kadb, a kernel mode debugger available with Sun UNIX, may optionally be loaded along with the applications software.

A Sun-2 workstation running Sun UNIX 4.2 is the development host for the target system software. Both the kernel and applications software are written in C with limited use of assembly language for interrupt handlers and for access to processor state information. The executable (load) module for the target system is built on the development system and stored on a public partition of the network disc known to the PROM monitor of the target system. The diskless target system is booted from the network disk via the LAN in response to commands entered at the console keyboard of the target system.

Procedure level and emulation mode debugging are performed on the development host using source level UNIX debuggers (e.g. DBX) and other UNIX facilities to improve debugging productivity. (See the 'Reconfiguration experiment' section below for a discussion of the emulation mode kernel).

History of the development

This particular real-time kernel began as a class project for a graduate course in real-time systems in the winter of 1988.¹⁴ A preliminary version of the kernel was built and used for a digital watch application in the spring of 1988.

The process/adts model was not used for design or implementation of the initial kernel. We decided to re-design and re-implement the kernel adhering strictly to the process/adts model. This activity took place in autumn and winter of the 1988-1989 academic year. The design described below satisfies this goal: all interactions with the kernel are accomplished via calls to exported procedures of kernel adts, all components of the kernel are designed as either processes or adts and there is no global data. Performance testing was carried out in the last year. Work is ongoing to extend the kernel and applications.

The kernel implementation consists of 2050 source lines of code excluding comments. It occupies 13.4 Kbytes of memory: 10.4 Kbytes for program instructions, 3 Kbytes for data structures.

KERNEL DESIGN

The kernel has the layered architecture depicted in [Figure 3](#). A layer of services available only to modules within the kernel isolates the details of the hardware environment from the remainder of the kernel. Application processes access kernel services through calls to procedures exported by higher level adts. In general, the availability of a service to an entity at another level of the hierarchy is implied by a shared horizontal boundary in the figure. Where indicated by arrows crossing a vertical boundary, interaction between entities within a single layer also occurs.

Kernel initialization

The kernel has two phases of execution: initialization and real-time execution ([Figure 4](#)). Initialization begins automatically when a stand-alone application is

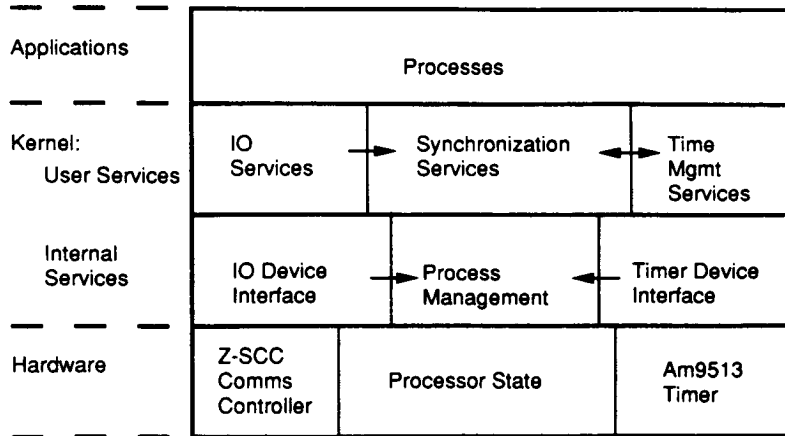


Figure 3. Layered kernel architecture

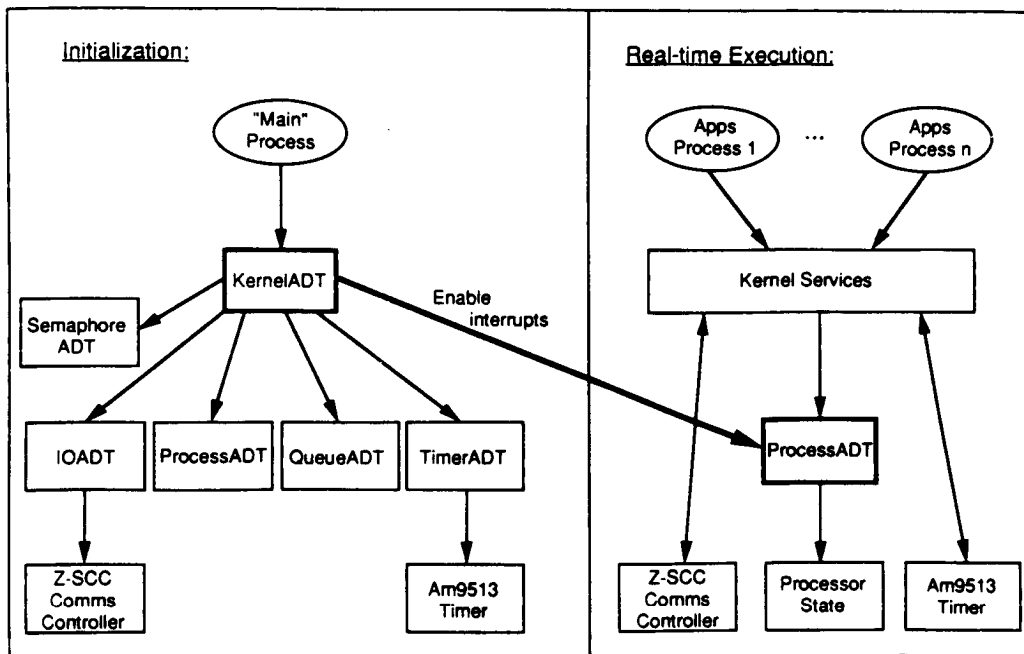


Figure 4. Kernel execution phases

loaded with the kernel into the target system. This phase is controlled by the KernelADT, which is active only during initialization, in response to requests from the application main process (C language main()).

Initialization has three steps: (1) initialization of kernel data structures and kernel

processes, (2) initialization of application processes, and (3) initialization and activation of external interfaces. Step 1 is undertaken when the application calls the `KernelADT.InitKernel` procedure. On completion of static kernel initialization, the application code must issue calls to `KernelADT.InitProcess`, one call per user process, to set up execution of application processes. When all user processes have been defined, the application then calls `KernelADT.InitExecution` to indicate that a transition to the real-time execution phase is desired. The `KernelADT` then activates the timer and IO devices, enables interrupts, and attempts to continue execution as the system idle process (`idle_process`). (The `idle_process` is the lowest priority process in the system. It requests no kernel services and is always ready to execute when all other processes are blocked.) As soon as interrupts are enabled, the transition to real-time execution phase occurs. The scheduling mechanisms embedded within the `ProcessADT` assume the function of system controller, allocating execution time as appropriate among ready processes.

Real-time execution

The detailed process/adts graph for the real-time execution phase appears as [Figure 5](#). The modules within the kernel fall rather naturally into four categories: externally visible service adts, hardware/software interface adts, kernel processes and kernel utility adts.

The service adts provide user applications with access to resources managed by the kernel. As shown in the Figure, services are categorized as: IO (Keyboard ADT, ScreenADT), synchronization (SemaphoreADT), and time management (TimerADT). There is a publicly visible interface to each service in the form of a set of exported interface procedures. Below this controlled public interface, the adts may interact with other kernel services and/or hardware to ensure that the service promised to the application is provided. These interactions give rise to a second level of exported procedures which are available to kernel modules only.

The hardware/software interface adts isolate the details of the hardware implementation from the application and the higher layers of the kernel. The `IOADT` encapsulates the interface to the Z-SCC, a controller which provides access to both keyboard and screen. The `TimerADT` includes the interface to the interval timer. The `ProcessADT` encapsulates processor state and manages the allocation of the CPU resource among competing processes.

The *external* interface of each of these adts, the actual interface between hardware and software, is dictated by the particular hardware configuration. Typically, each such interface is modelled as an optional interface to a hardware *process*, which captures the active role of interrupt driven devices in scheduling of system resources, and interfaces to one or more adts which encapsulate the control interface(s) accessible to the software. The *internal* interface, those procedures visible to higher level kernel software, comprises procedures which convey the essential operation of the device without exposing the details of the realization. The use of the model to describe both hardware and software interfaces proved particularly beneficial for debugging, when hardware interfaces were easily replaced with emulation software described in the 'Reconfiguration experiment' section below.

The kernel processes (i.e. `keyboard_input`, `screen_output` and `timer_process`) manage devices which operate asynchronously with respect to the application software.

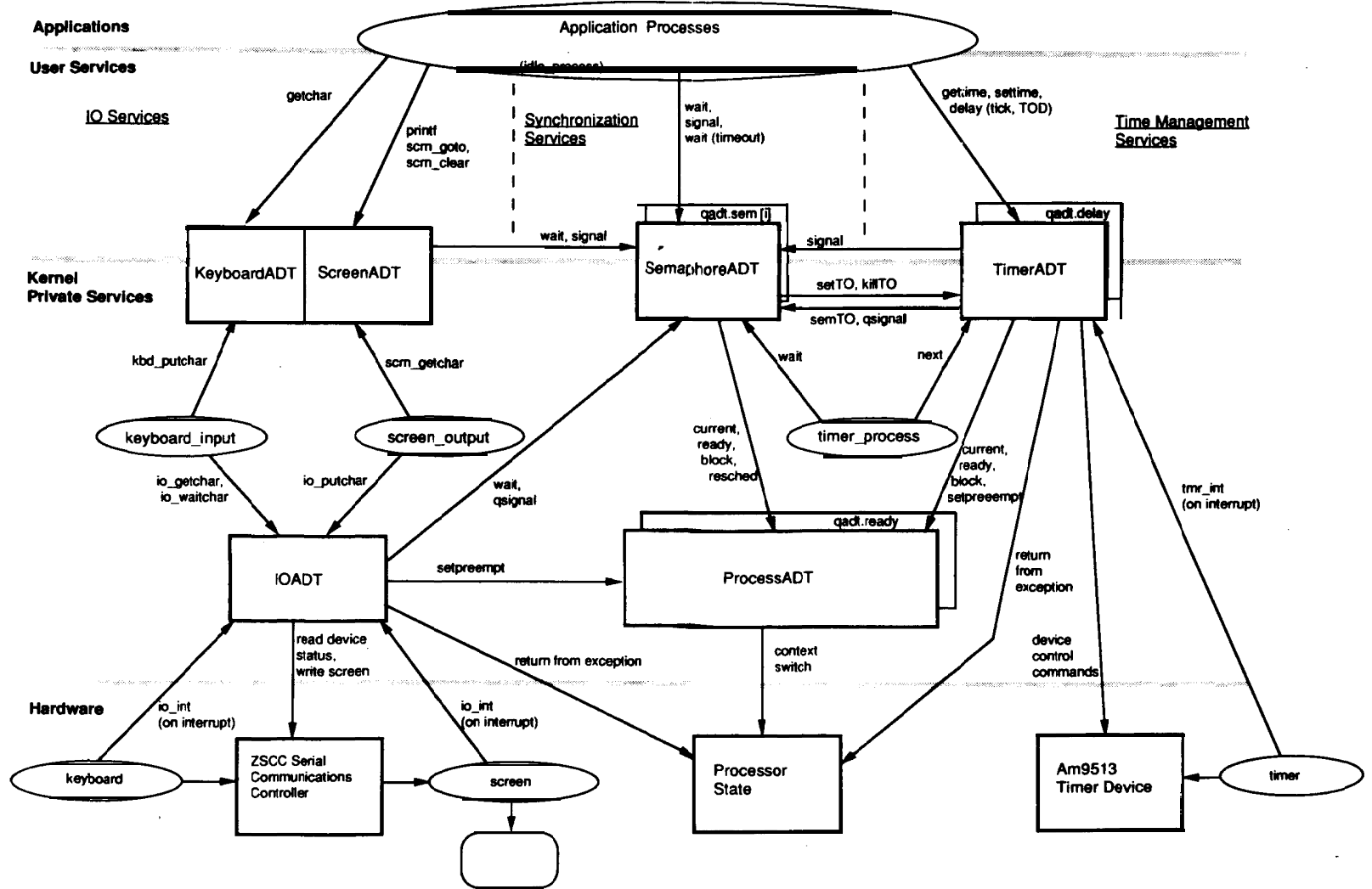


Figure 5. Process/adt graph of kernel (real-time execution phase)

The QueueADT utility is available to all kernel adts and is one example of an adt which is multiply instantiated. It manages queue instances for the SemaphoreADT (one instance per semaphore), the ProcessADT (the ready queue) and the TimerADT (the delay queue).

EXPERIMENTS AND EXPERIENCES

Through our kernel design and implementation we evaluated the process/adts model in four areas: as a framework for reconfigurable systems, as a generator of software with deterministic timing characteristics, as a means for producing efficient implementations and as a design and documentation scheme.

Reconfiguration experiment

To test the hypothesis that our model would produce software which is easily tailored to different environments, we experimentally moved the boundary between hardware and software. The layered kernel architecture of Figure 3 was adapted to run under UNIX by replacing several modules formerly implemented in hardware with emulation software (Figure 6). The timer interrupt has been replaced with the UNIX SIG_ALARM signal, and direct access to processor state is emulated through manipulation of process context as preserved on the UNIX signal stack. Another modification replaces the entire kernel IO subsystem with UNIX IO facilities at the call compatible interfaces.

Adaptation between these two configurations involved only 163 lines of code in seven procedures. Although these 163 lines constitute approximately 13 per cent of the kernel code, this percentage of change is consistent with the fact that much of the kernel is low-level hardware/software interface code. We believe that the ease

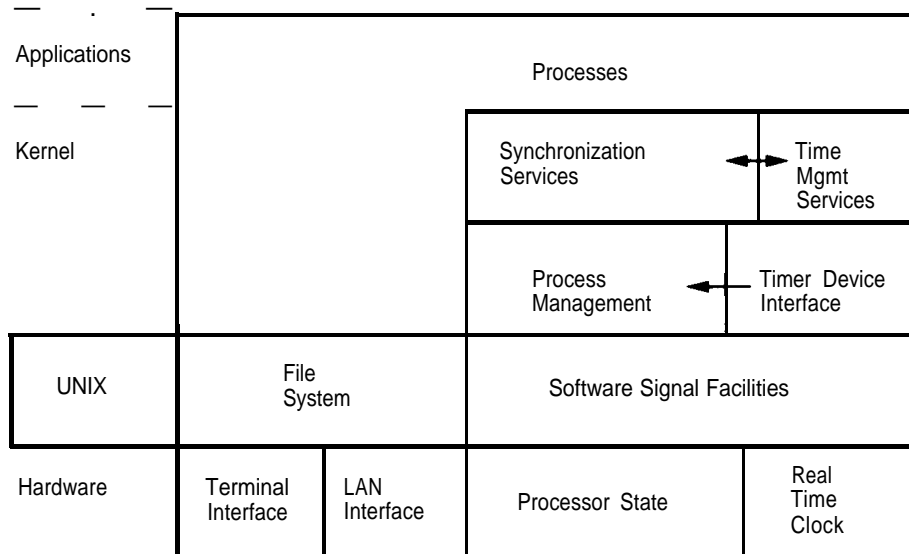


Figure 6. Adaptation of the kernel for UNIX emulation mode

of this adaptation resulted in large part from the encapsulation provided in the simple process/adt model.

The practical result is a very useful piece of software that can be adapted to different execution environments through selection of alternative build procedures (UNIX make files). All software, except for the direct interface to the target hardware, can be debugged under UNIX, taking advantage of standard tools such as source level debuggers, core dump and post-dump analysis tools, and even profiling and timing facilities. This capability has proved exceedingly useful in implementing software for a target system with primitive debugging tools.

Deterministic timing analysis

Deterministic timing performance was a major design objective of the kernel. We ran a series of timing tests on the completed system to measure the actual performance of the software and to attempt to substantiate that execution times were deterministic. Because there is little information published on performing this kind of timing analysis and because it was surprisingly difficult, we have provided some detail on our techniques and results.

The execution of the kernel in an idle system was the subject of our timing tests. As depicted in [Figure 7](#), a single application process, the `idle_process`, is periodically interrupted by the activation of the `timer_process` in response to interrupts from the

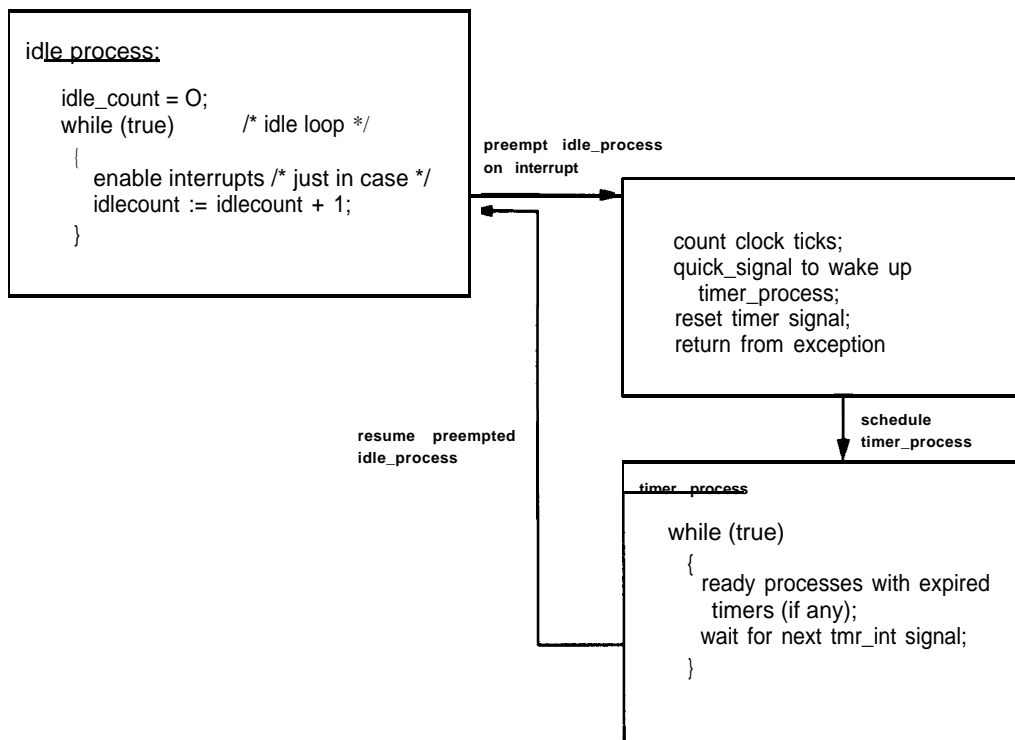


Figure 7. Kernel processing cycle in an idle system

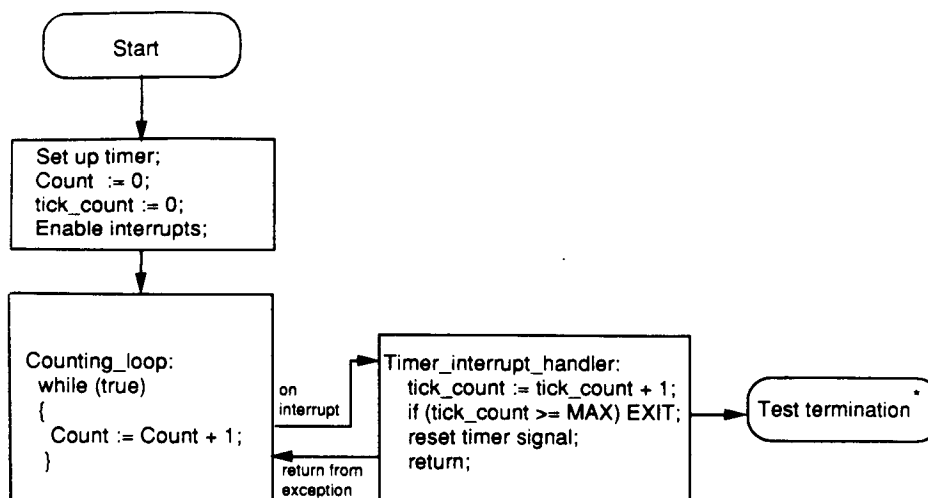
timer. We refer to this complete processing cycle (idle_process, timer interrupt handler tmr_int, timer_process and supporting kernel services) as the kernel processing cycle.

The idle_process executes a non-terminating loop, the *idle loop*, in which interrupts are enabled and a counter incremented (Figure 7). At each real-time clock interrupt, the timer_process pre-empts the active process to check for the expiration of process delays (including time-outs). In this test there are no processes with outstanding delay requests, so the idle_process resumes execution each time the timer_process relinquishes control.

The kernel processing cycle has two interesting characteristics: (1) the configuration of hardware and software during execution is completely known, leading to expectation of deterministic timing results; and (2) although simple in function, this basic processing cycle, in fact, exercises a significant fraction of the total code of the kernel. We hoped that each procedure would have precise timing behaviour and that by building a tree-structured execution graph, we could accurately predict high level timing.

Step 1: measuring the timer interrupt handler

Timing test strategy involved use of the Am9513 timer device, a clock of known accuracy, for measurement of execution times. In order to employ this external timing signal to measure internal execution times, it was necessary to know how long it took to handle each timer interrupt. A simple test set-up consisting of the interrupt handler and a two-instruction counting loop that executed between interrupt handler activations, as shown in Figure 8 , was used to measure this time.



* MAX = 50,000: test termination condition (50,000 interrupts)

Figure 8. Calibration test code

The approximate time of the interrupt handler execution was determined from observation using an analogue oscilloscope, by observing the interval between raising of the interrupt signal and its reset, the last step before exiting the handler. This measurement was approximate because it included simple interrupt latency* and excluded the time to execute a return-from-exception (rte).

The interval between interrupts was then stepped programmatically over a range around this approximate value. The interval at which the fewest counting loop cycles (see [Figure 8](#)) was executed during the handling of a fixed number (MAX) of interrupts was determined to be the interrupt handler time within one increment of the timer clock. † Table I records the number of counting loop cycles executed at each interrupt interval in this range. The execution time of the handler was determined to be 24.36 μ s. This value exceeded a predicted execution time of 21.012 μ s, calculated from manufacturer's timing data, by 16 per cent. We discuss reasons for this discrepancy at the end of this section.

Step 2: measuring procedure times in isolation

The next step of the timing tests consisted of measuring a stand-alone execution time for each of the procedures invoked in the kernel processing cycle and for the *idle loop* itself. The test set-up for each measurement followed this general form: (1) a measurement was made for a test loop containing a sequence of null operations

Table 1. Counting cycles versus interrupt interval

| Timer count | Interval between interrupts | |
|-------------|-----------------------------|----------------------|
| | Microseconds | Loop cycles executed |
| 116 | 23.548 | 90,520 |
| 117 | 23.751 | 68,234 |
| 118 | 23.954 | 68,029 |
| 119 | 24.157 | 27,964 |
| 120 | 24.360 | 8486* |
| 121 | 24.563 | 10,947 |
| 122 | 24.766 | 11,053 |
| 123 | 24.969 | 14,858 |
| 124 | 25.172 | 17,535 |
| 125 | 25.375 | 20,920 |
| 126 | 25.578 | 23,792 |
| 127 | 25.781 | 26,476 |
| 128 | 25.984 | 27,612 |

*Minimum 'work'

*Simple interrupt latency is viewed in the time between the assertion of the interrupt signal and the beginning of interrupt processing by the processor.

†When the time to handle an interrupt exceeded the interval between interrupts, the processor periodically failed to reset the interrupt before the interval expired. As a result, an interrupt was lost, the actual time of the test was erroneously extended, and 'false work', in significant amounts, was registered by the counting loop. This technique was derived from techniques for measuring interrupt latency described in [Reference 15](#).

‡A *sequence* of null operations ensured that the null operation loop exceeded the size of the MC68010 instruction loop buffer.

(2) the time to call a dummy procedure with parameters identical to that of the procedure to be timed was measured, and (3) the time to call and execute the real procedure of interest was measured. The time to make the call (time for (2) minus time for (1)) was allocated to the calling procedure and the time for execution of the procedure body (time for (3) minus time for (2)) was allocated to the called procedure. A timing estimate for all the processing triggered by each timer interrupt was accumulated by summing the measured execution times for each node of the call tree. The results appear in Table II. In this situation, average execution time was expected to equal both best and worst case execution times, because the exact context of execution was known. However, there were slight and unexpected variations on the order of 0.1 per cent between measured times in repeated tests.

Step 3: measuring the kernel processing cycle

Finally, the software for the kernel processing cycle was run for a measured period of time and the number of executions of each major component was observed. The estimated time spent in the interrupt handler, timer_process and scheduling oper-

Table II. Execution time of processing activated by timer interrupt

| Procedure name | Measured execution time (in μ s) |
|-----------------------|--------------------------------------|
| TimerADT.tmr_Int | 164.3 |
| SemaphoreADT.qsignal | 83.0 |
| SemaphoreADT.bad | 10.3 |
| QueueADT.look | 37.1 |
| QueueADT.first | 63.5 |
| ProcessADT.ready | 39.3 |
| QueueADT.insert | 65.4 |
| ProcessADT.setpreempt | 33.8 |
| ProcessADT.resched | 70.1 |
| QueueADT.look | 37.1 |
| ProcessADT.ready | 39.3 |
| QueueADT.insert | 94.5 |
| ProcessADT.block | 61.0 |
| QueueADT.first | 63.5 |
| ProcessADT.switch | 58.6 |
| process_timer | 25.6 |
| TimerADT.next | 22.7 |
| QueueADT.look | 15.0 |
| SemaphoreADT.wait | 94.3 |
| SemaphoreADT.bad | 10.3 |
| ProcessADT.current | 60.7 |
| QueueADT.insert | 65.4 |
| ProcessADT.set | 36.1 |
| ProcessADT.block | 77.2 |
| QueueADT.first | 63.5 |
| ProcessADT.switch | 58.6 |
| Total | 1450.2 |

*Different execution times for a single procedure, e.g. two timings for QueueADT.look, result from taking different execution paths through the module on different calls.

ations, plus the amount of time spent in the idle loop, was expected to sum to the elapsed time of the test. Over a series of ten test runs of varying duration, the predicted execution time averaged 99.98 per cent of the observed time with a standard deviation of 0.18 per cent. The predicted and observed values were sufficiently close, given the presence of known timing variations as described below, to indicate that all significant factors affecting the timings had been identified and taken into account.

In general, we are satisfied that use of the process/adts model did help us build procedures and processes with reasonably predictable performance. Encapsulation of data structures and standardization of operations forced programs into consistent access techniques. This standardized framework resulted in simple, effective timing analysis: times for high level functions are the sum of procedure body time plus parameterized times for lower level operations, given known workloads. Although this work focused on precise measurement of a completely defined workload, the method is suitable for prediction of worst case performance of the kernel in a more complex process/adts environment.

Differences between expected and measured results at each step of the test, however, prompted a search for the source of discrepancy. A lengthy study yielded the following contributing factors:

1. Loss of execution cycles due to lack of synchronization between the CPU cycle and the memory refresh cycle. Related research efforts found that up to 7 per cent of the available CPU cycles could be effectively lost as a result of this asynchrony.¹⁶ Empirical data indicated that for this particular instruction mix about 5 per cent of CPU cycles were forfeit.
2. Low-level, but unpredictable, activity of the PROM monitor software. This source of variation in timing measurements accounted for discrepancies between test repetitions and could be eliminated completely in timing tests by disarming the Am9513 timer which served as a source of unmaskable interrupt to the 68010 processor. Since the availability of the PROM monitor was essential to control of the target system under realistic run-time conditions, however, tests were run with interrupts to the PROM monitor enabled, tacitly accepting the variations in average timing introduced by retaining this function. When enabled, PROM monitor activity added about 3 per cent to measured execution times and introduced variations in average execution time of as much as 0.1 per cent between test repetitions.
3. Known non-deterministic features in the target architecture such as the instruction loop buffer. Because deterministic timing was a goal, we strove diligently to produce code which failed to take advantage of this buffer, but we never eliminated it absolutely as a source of possible execution time variance.
4. Interrupt latency: There still remained the problem referred to in Step 1 above of the 16 per cent discrepancy for the interrupt handler, however. Precise measurement using a digital oscilloscope showed that the context switch to the interrupt handler introduced variable delays due to uncertainties in the synchronization of bus transactions involving the timer device. This delay accounted for the additional cycles in the measured time.

One conclusion, confirmed by some study of other architectures, is that most modern computer systems are *not* built for deterministic timing performance ! Note, however, that this kind of deterministic timing prediction is very much a research area and that our results for part of the kernel are just a start.

Performance overheads

A by-product of the timing tests was some basic analysis of performance overheads associated with our strict application of the process/adts model. Running with clock interrupts at 10 millisecond intervals, the kernel itself consumes 14.5 per cent of available CPU cycles (Table II). Note that timing predictability was a goal in this implementation, not necessarily efficiency. The kernel code was in no way optimized for performance, and the timings presented are not intended to indicate an acceptable level of performance for a 'real world' application. Even in this less than optimal implementation, we found that 29 per cent of kernel cycles were spent executing procedure call/return sequences.

Each procedure was classified primarily on the efficiency of the implementation, i.e. as a function of the overhead of the procedure call mechanism compared to the execution time of the body of the procedure. Many of the 'inefficient' calls are to procedures which provide simple access to encapsulated data structures, calls which with less stringent application of our model might have been implemented as direct access to shared data. We determined analytically that we could reduce the execution overhead of the kernel by 11 per cent simply by replacing this type of procedure call with either compile-time macros or in-line expansion of the procedure body. * Through such implementation as *lightweight adts*, negative performance impacts of the process/adts approach can be minimized while retaining the benefits of the information-hiding concepts inherent in the model.

Design and documentation scheme

The process/adts model certainly provided a clean, simple description of our software, a small operating system that embodied concurrency but had no critical real-time constraints. For a system of this scale, it is possible to describe the major modules of the system and their relationships on a single diagram, a property that was useful for design presentations and discussions.

The simplicity of the primitives does appear to be a good solution for real-time systems. All interactions between modules are explicitly represented in the model, so that it is easy to locate, investigate and understand implementation decisions and execution time characteristics, especially timing behaviour. The insistence on encapsulation of shared data within adts should enhance system reliability, although we have not explicitly assessed issues of reliability in our work to date.

The approach is particularly applicable at the interface of hardware and software. Modelling an interrupting device as a process invoking a procedure of an adts† accurately captures the asynchronous operation of the external device and lends itself to replacement of hardware with simulation software at the interface.

We used C as our implementation language, even though it has little direct support for the encapsulation of data within adts. Adhering to the necessary discipline was facilitated by using C capabilities for structures, pointers, user-defined types and

*The former approach is suitable for simple, often-repeated operations. The latter is appropriate for more complex operations for which separate compilation during debugging is desirable; the in-line expansion can then be invoked late in integration and test when code is stable and achieving performance becomes a primary objective.

†Some contemporary programming languages, notably Ada, employ a similar model.

statically scoped variables, but we see no practical barriers to use of the model even with implementation languages as hostile as, say, Fortran, given adequate motivation on the part of the developer.

There are some obvious deficiencies in the representation of real-time systems using the process/adts model. Although the presence of multiple processes indicates the existence of concurrency relationships, the model conveys no explicit information regarding these relationships, particularly the temporal relationships among processes. (This liability is a property of all approaches of which we are aware.) In addition, the notation requires some grouping extension to abstract and refine modules in order to handle large systems.

CONCLUSIONS

Our use of a pure process/adts model has been a success. However, the experiments have also pointed out several problems requiring further work.

We have found that constant, careful attention to timing predictability at all levels of design, including especially hardware architecture and the first levels of software, is necessary to achieve deterministic timing predictability. The precise timing characteristics of modern general-purpose computers seem particularly difficult to obtain, but this may be because hardware designers have emphasized stochastic performance metrics. A second point concerns intra-module interactions. The model describes essentially hierarchical use relationships among modules; it does not capture explicitly many other relationships among processes and types that would be necessary for a complete analysis. One direction worth pursuing is to try to incorporate information on procedure parameters and on process and procedure semantics, particularly those related to time.

Plans for the kernel include further use as a test-bed for reusability experiments, extensions to support more services, and the construction of several applications. We also hope to adapt our kernel to some multiprocessor and distributed parallel processing environments.

Much remains to be done in the area of predictable timing before complete analyses are possible. In related research, we are developing techniques and tools to predict deterministic timing performance from source code.^{16,17} We expect to continue to use our real-time operating system test-bed in support of this work.

Our initial experiments indicate that the process/adts framework does, indeed, provide a simple, convenient and practical design and programming model for real-time systems. The model has proved particularly useful for capturing essential relationships at the hardware/software interface. We have used this feature to advantage when replacing hardware with simulation software to facilitate design, integration and testing. The model has also contributed to our ability to design software with predictable timing performance. We have been able, through careful attention to hardware and software architecture, to build system services with deterministic timing characteristics necessary for real-time systems.

ACKNOWLEDGEMENTS

We are grateful to Kevin Jeffay, Jihong Kim and ChangYun Park for many discussions on this research and for a critical reading of an earlier version of the

paper. This research was supported in part by the U.S. National Science Foundation under grant CCR-8700435 and by the U.S. Office of Naval Research under grant number N00014-89-J-1040.

REFERENCES

1. A. C. Shaw and K. Jeffay, 'Software engineering of real-time operating systems', *TR-88-01-01*, Department of Computer Science, University of Washington, Seattle, WA, January 1988.
2. A. C. Shaw, 'Real time systems = processes + abstract data types', *TR-88-12-07*, Department of Computer Science, University of Washington, Seattle, WA, 1988; also published in *Proceedings of the EUROMICRO Workshop on Real-Time*, IEEE Computer Society Press, June 1989, pp. 188–197.
3. B. Lampson, 'Hints for computer system design', *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, October 1983; in *ACM Operating Systems Reviews*, **17**, (5) (Special Issue), 33–48.
4. D. L. Parnas, P. C. Clements and D. M. Weiss, 'The modular structure of complex systems', *IEEE Trans. Software Engineering*, **SE-11**, (3), 259–266 (1985).
5. N. Wirth, 'Toward a discipline of real-time programming', *Communications of the ACM*, **20**, (8), 577–583 (1977).
6. R. H. Campbell, G. M. Johnston, P. W. Madany and V. F. Russo, 'Principles of object-oriented operating system design', *UIUCDCS-R-89-1510*, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1989.
7. *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, U.S. Department of Defense, January 1983.
8. B. Lampson and D. Redell, 'Experience with processes and monitors in Mesa', *Communications of the ACM*, **23**, (2), 105–117 (1980).
9. N. Wirth, *Programming in Modula-2*, 2nd edn, Springer-Verlag, NY, 1983.
10. Motorola, *M68000 16/32-Bit Microprocessor Programmer's Reference Manual*, 4th edn, Prentice-Hall, Englewood Cliffs, NJ, 1984.
11. *Am9513 System Timing Controller Technical Manual*, Advanced Micro Devices, 1983.
12. *Zilog Components/Data Book*, Zilog, Inc., Campbell, CA, 1985.
13. 'Sun Workstation CPU PROM Monitor', *System Internals Manual for the Sun Workstation*, Rev. C, Sun Microsystems, Inc., Mountain View, CA, 1984.
14. C. Lin, S. Mann, C. McCann, W. F. Mershon and B. Roberson, 'Popcorn real-time kernel and applications', *Class Projects in CSC1590R and 551*, Winter/Spring 1988, Department of Computer Science, University of Washington, Seattle, WA, 1988.
15. M. Borger, 'VAXELN experimentation: programming a real-time clock and interrupt handling using VAXELN Ada 1. 1', *CMU/SEI-87-TR-29*, Carnegie Mellon University/Software Engineering Institute, 1987.
16. C. Y. Park and A. C. Shaw, 'Experiments with a program timing tool based on source-level timing schema', *Proceedings of the IEEE 11th Real-time Systems Symposium*, IEEE Computer Society, December, 1990.
17. A. C. Shaw, 'Reasoning about time in higher-level language software', *IEEE Trans. Software Engineering*, **SE-15**, (7), 875–889 (1989).