High-Performance Operating System Primitives for Robotics and Real-Time Control Systems

KARSTEN SCHWAN, TOM BIHARI, BRUCE W. WEIDE, and GREGOR TAULBEE
The Ohio State University, Columbus

To increase speed and reliability of operation, multiple computers are replacing uniprocessors and wired-logic controllers in modern robots and industrial control systems. However, performance increases are not attained by such hardware alone. The *operating software* controlling the robots or control systems must exploit the possible parallelism of various control tasks in order to perform the necessary computations within given real-time and reliability constraints. Such software consists of both control programs written by application programmers and operating system software offering means of task scheduling, intertask communication, and device control.

The Generalized Executive for real-time Multiprocessor applications (GEM) is an operating system that addresses several requirements of operating software. First, when using GEM, programmers can select one of two different types of tasks differing in size, called processes and microprocesses. Second, the scheduling calls offered by GEM permit the implementation of several models of task interaction. Third, GEM supports multiple models of communication with a parameterized communication mechanism. Fourth, GEM is closely coupled to prototype real-time programming environments that provide programming support for the models of computation offered by the operating system. GEM is being used on a multiprocessor with robotics application software of substantial size and complexity.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*process control systems, real-time systems*; C.4 [**Computer Systems Organization**]: Performance of Systems—*design studies*; D.4.1 [**Operating Systems**]: Process Management—*multiprocessing/multiprogrammming, scheduling*; D.4.4 [**Operating Systems**]: Communication Management—*message sending*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems*; D.4.8 [**Operating Systems**]: Performance—*measurements*; J.7 [**Computer Applications**]: Computers in Other Systems—*real time, military, process control*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Light-weight processes, operating software, parallelism, robotics

This research was sponsored by National Science Foundation grant ECS-8307216 and by the Defense Advanced Research Projects Agency (DARPA) under contracts MDA903-82-K-0058 and DAAE07-84-K-R001 monitored by the Army Tank Automotive Command. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Author's Address: Department of Computer and Information Science, Ohio State University, 2036 Neil Avenue Mall, Columbus, OH 43210.

1. INTRODUCTION

1.1 Parallel, Real-Time Operating Software

To increase the reliability and speed of operation, the embedded computer hardware controlling modern robots and industrial control systems is becoming increasingly complex. Typically, it consists of many interconnected computers operating at multiple levels of control or supervising different mechanical or electronic system peripherals. However, increased performance is not attained by hardware improvements alone. The operating software controlling such robots or control systems must exploit the possible parallelism of various control tasks in order to perform the necessary computations within real-time and reliability constraints.

Operating software has two attributes distinguishing it from other multicomputer or multiprocessor applications [16, 30, 32, 73, 86]. First, it consists of both (a) control programs written by application programmers and (b) operating system utilities as an integral part of the program. Such utilities are tailored to the application's needs. Such tailoring is due in part to the embedded hardware's limitations in memory and processing capacity. More important, it is necessitated by the stringent real-time and reliability constraints that can be met only if (a) and (b) are tailored jointly with respect to the target hardware and the control tasks to be performed. Second, since embedded systems often are highly experimental and are also subject to dynamic change, operating software changes must be static, during software development and dynamic, at run time. For example, dynamic software reconfiguration is necessary to attain high reliability and continued high performance in the face of dynamically varying operating modes [2, 50] or hardware configurations [26, 37, 59, 85].

1.2 Operating System Primitives for Real-Time Robotics Software

The efficient execution of operating software requires that programmers deal with a variety of issues that arise for any parallel application, including resource allocation [13, 46, 69], task scheduling [76], load balancing [12], and programmed dynamic reconfiguration [2, 31, 37]. Similarly, operating system primitives for task control, intertask communication, and device operation are needed. However, additional operating system support is required:

—Operating software exhibits multiple grains of parallelism [32]. Therefore, the operating system must support parallel application tasks of differing *sizes*, ranging from small tasks executed at high rates and by necessity consisting of a small number of instructions, to large tasks executed infrequently.
—Tasks must be schedulable periodically or sporadically [53], and they must be scheduled, synchronized, and executed within strict time constraints.
—Task communication is time-critical, and tasks may make different assumptions regarding the model of communication used [84]. For example, some tasks may tolerate the loss of individual readings from a sensor in order to perform an operation asynchronously at the highest rate possible, whereas other tasks may assume that individual messages never get lost. Thus, the operating system must support multiple models of task communication.

—Communication and scheduling mechanisms must be configurable for a variety of real-time hardware, possibly exhibiting hierarchies [51] and inhomogeneities [24, 49, 62] in communication links and processors.
—For performance reasons, most current real-time systems contain minimal operating system software. However, substantial operating system support must be provided for long-lived, highly complex systems [26, 51, 62], since static configuration methods [36] must be supplemented with methods for and therefore operating system support for, dynamic adaptations of software to varying hardware configurations and performance requirements.

The GEM operating system was constructed for robot-operating software. It runs on special-purpose multiprocessors, including one embedded in a complex robot, a six-legged mobile Adaptive Suspension Vehicle (ASV) able to navigate rough terrain [50, 51, 52]. GEM's design and implementation are an attempt to apply two principles to attain high run-time performance of operating software:

(1) development of somewhat application-specific operating system primitives [23]; and
(2) design and implementation of these primitives for their efficient, shared use.

Specifically, regarding (2) and in contrast to the basic, multipurpose mechanisms offered by other contemporary operating systems [15, 31, 42], the sharing of GEM's utilities is made efficient by their *selection* or *parameterization* for each specific use.

—Programmers can *select* one of two different sizes of tasks, called processes and microprocesses. For each task size GEM offers scheduling and task-switching operations with costs commensurate with their sizes and acceptable to relatively high-speed control tasks.
—GEM supports multiple models of communication with a mechanism that is *parameterized* with respect to different interprocess communication characteristics.
—GEM's mechanisms can be configured for a variety of real-time hardware.

GEM is being used with ASV application software and has been tested with industrial control software [67] and with a synthetic workload generator [71]. Thus GEM's implementation addresses the control of individual, complex robots. However, its mechanisms and design should also be useful for the operating software of multiple robots [21, 24, 35] or manufacturing subsystems [67].

1.3 The Programming of Operating Software

In addition to the construction of GEM, the research reported in this paper also explores two facets of the programming of operating software. First, since the programming of parallel control systems is quite difficult, programmers are given domain-specific programming support tools for writing parallel software. Specifically, the STILE [78, 80] graphical program editor supports the generation of real-time programs from high-level graphical descriptions, and GEM can directly execute the programs generated with the STILE editor.

Second, since the need for adaptations of operating software adds complexity to the programming process, the ISSOS [68] parallel programming system offers,

high-level, non-graphical programming primitives for the specification, enactment, and partial automation of program adaptations. Such *adaptations* are defined as small software changes made statically during software development or dynamically during software execution in order to realize performance or reliability improvements. The aim of the ISSOS system is to facilitate program adaptation and to hide selected adaptation decisions from application programmers so that they need not be experts in the target parallel hardware and its operating system.

In the remainder of this paper, the ASV's operating software (Section 3) motivates GEM's alternative models of, and constructs for, task scheduling and communication (Sections 4 and 5). In Section 6, the multiprocessor hardware and GEM's implementation and performance are discussed. The programming environments are described in Section 7.

2. RELATED RESEARCH

This work touches upon four different areas of research: operating systems as being investigated by computer scientists in general and by researchers in the real-time domain specifically; other research regarding legged vehicles; research regarding the design and implementation of specific real-time systems in the domains of robotics, manufacturing, and aerospace; and the programming of real-time software.

With other operating systems written for multiprocessors or computer networks, GEM shares the notion that kernel-level system mechanisms should be simple and efficient [5, 11, 58], yet parameterized [15, 31, 42] so each mechanism can satisfy a variety of application requirements. For example, GEM's parameterized communication facility directly supports multiple models of message communication. This idea is related to the V system's notion of k-reliable message communications [11] between process groups and to other reported parameterizations [15, 31, 42]. However, it originated from our observations regarding the programming and operation of robotic control software, which are similar to those of Lee and Shin [41, 74].

As in other operating systems for multiprocessors [15, 17, 31, 75], GEM allows processes to communicate both via shared memory and via messages. The use of messages leads to increased portability [10] of operating software, but their exclusive use as in local area networks [11] is not acceptable in GEM. That would preclude very low-latency process interactions on shared information, such as those required for the on-board terrain scanner of the ASV robot vehicle (see the description of the ASV application in the next section) or those required for servicing high-rate actuators.

GEM's concepts are similar to those of other real-time operating systems [10, 20, 49], including the operating systems of the HXDP multiprocessor [4], the MAFT multicomputer [81], and SIFT [85]. However, GEM's primitives focus on its primary real-time application, which is the control of the ASV legged vehicle.[1] For example, while recent research in operating systems has been addressing the general notions of lightweight user or kernel processes [7], GEM's idea of a

[1] Other examples of parallel, real-time operating software appear in [67, 71].

microprocess is less general and addresses its specific use for synchronous device control. Namely, a *microprocess* is a small section of code that can be activated and executed at high rates; it cannot be interrupted or suspended and therefore, resembles an event handler as used in traditional operating systems.

Most research regarding legged vehicles and research in the ALV program [62] has dealt with mechanical designs, computer architectures [34, 51], and control and planning algorithms [50, 52, 66]. Programming and operating system support have not yet been studied methodically, despite the fact that future plans for such robots state that their computer hardware will consist of both shared memory machines and networked computers, including inhomogeneous machines (such as LISP machines used in high-level planning). One exception is Donner's [18, 19] design of a special-purpose programming language called Owl[2] for Sutherland's six-legged walking machine [79]. Owl supports the definition of processes and subprocesses that execute in sequence or concurrently and interact by mutual invocation and shared memory. Such hierarchical process structures are not supported in GEM, but are the topic of our current research [67]. Another interesting aspect of the Owl language not addressed in GEM is its support of multiple semantics of parameter passing. A comparative evaluation of the performance of Owl with GEM programs is not appropriate, since Owl's run-time system assumes the use of a single processor for program execution.

Robot manipulators research concerns the design of control algorithms and their inherent parallelism. Such work is relevant because it describes potential applications for the GEM machine and contains ample evidence of the inherent parallelism in robotics software. However, it has focused on the design of special-purpose VLSI or board-based architectures for such algorithms. Such designs include architectures for (a) the small grain parallelism in computationally expensive operations (such as inverse kinematics and Jacobian computations) [39, 57] or in data-intensive operations (such as vision processing [28]); (b) the medium grain parallelism inherent in entire robotics systems or subsystems for controlling a manipulator [1, 3, 43, 45, 82], a hand [55, 61], or subsystems consisting of multiple actuators and sensors [9, 21, 26, 54, 88]; and (c) the large-grain parallelism inherent in entire manufacturing systems [24, 35]. Of the applications described above, the medium-grain parallelism inherent in those tasks can be implemented efficiently on GEM, as partly shown by implementations of a controller for a robot arm and conveyor [67] and of dynamic arm controls [43]. For high-performance robot arm control, however, special-purpose VLSI-based algorithm implementations are necessary [43].

The programming systems associated with GEM and described in Section 7 assume that real-time programs [77] consist of sets of synchronous or asynchronous interacting real-time tasks that are structured hierarchically [22, 48, 70]. However, for reasons of simplicity, lack of memory, and performance, such hierarchical structures must be mapped to the two-level structure consisting of processes and microprocesses supported by GEM. Furthermore, those programming systems cannot directly describe the higher level information concerning the parts manipulated or the terrain traversed by the robot [29, 38].

[2] This should not be confused with another language called OWL [83] that is entirely different.

In comparison, Donner's work regarding robot programming mainly addresses the ease of programming of robot programs coupled with their efficient uniprocessor execution, whereas recent research by Korein at IBM [36] mainly addresses the exploitation of parallelism in the application and hardware of real-time control programs.

3. A REAL-TIME APPLICATION PROGRAM: OPERATING A COMPLEX ROBOT

The Adaptive Suspension Vehicle (ASV) is a six-legged, three-ton vehicle powered by an internal combustion engine via hydraulic actuators (see Fig. 0). This vehicle is operated via high-level commands issued by an on-board human operator. These commands are translated into vehicle actions by extensive operating software running on an embedded multiprocessor. Additional inputs to the system's operating software include an inertial reference system on the body, pressure, velocity, and position sensors on the legs, and an optical radar terrain scanner. The following discussion provides an overview of the major subsystems of the operating software (see Figure 1).

The *Cockpit I/O* subsystem provides the link between the human operator and the ASV. It is composed of several processes, including:

—Cockpit Input (CI)—one GEM process, running at a rate determined by human response bandwidth, typically several Hertz, accepts commands from the operator, formats them appropriately, and passes them to Vehicle Motion Planning (VMP).
—Cockpit Display (CD)—one GEM process, running at a rate determined by human response bandwidth and by the display device, typically several Hertz, displays information on a periodic basis or on demand.

The *Vehicle Control* subsystem is reponsible for the stability and proper movement of the vehicle as a whole. It is composed of several processes, including:

—Vehicle Motion Planning (VMP)—one GEM process, running at a rate determined by possible rates of change in body movements (body control bandwidth), about 20 Hz, takes high-level commands from (CI), modifies them to ensure stability of the body, determines the necessary leg velocities and sends them to Leg Motion Planning (LMP), also sends commands for the desired body position, velocity, and acceleration to Body Servo (BS).
—Body Servo (BS)—one GEM process, running at a rate determined by body control bandwidth, about 20 Hz, takes information from Inertial Navigation (IN) (VMP), calculates the necessary actuator pressures when the legs are in the "support phase" on the ground and sends them to the Leg Servos.

The *Leg Control*, that is, subsystems (one for each leg) are responsible for the movement of the individual legs. They are each composed of several processes, including:

—Leg Motion Planning (LMP)—one GEM process per leg, running at about 20 Hz, determines the actual leg trajectories while they are in the air, or "transfer phase," and sends commands to corresponding (LS).

Fig. 0. The Adaptive Suspension Vehicle (ASV).

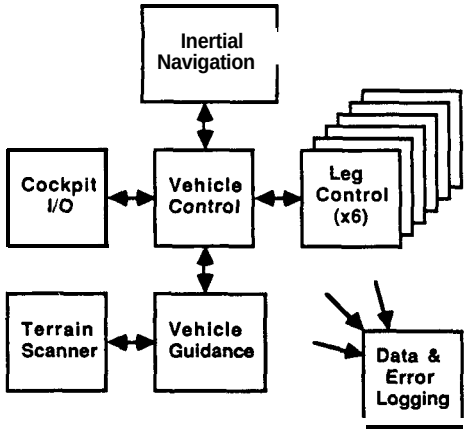


Fig. 1. The ASV Robot's operating software-subsystems.

-Leg Servo (LS)-one process per leg, each running at a rate determined by the bandwidth of the hydraulic/mechanical system, typically about 100 Hz. The Leg Servo processes form the feedback loops in the leg control systems, combining the requested actuator pressures with feedback information from leg sensors, and issuing commands to the hydraulic system.

The *Inertial Navigation (IN)* subsystem is composed of two GEM processes: a high-frequency process (INH) running at about 20 Hz and a low-frequency

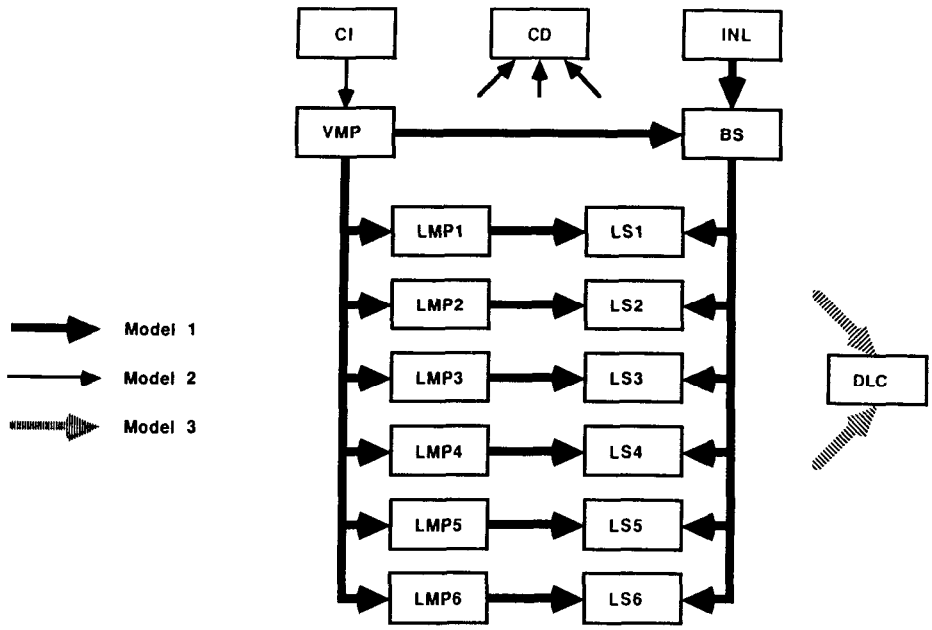


Fig. 2. The ASV Robot's operating software—interaction examples.

process (INL) running at about 5 Hz. These processes jointly read the inertial reference system, reconcile it with the position obtained by dead reckoning (i.e., the position deduced from previous commands to the machine), format the information, and send it to (BS).

The *Terrain Scanner* subsystem consists of three GEM processes, each running at about 2 Hz. These processes accept data from the optical radar terrain scanner, convert it to fixed-earth coordinates, and store it in a terrain map. The data stored are used by the Vehicle Guidance subsystem.

The *Vehicle Guidance* subsystem is composed of four GEM processes, each running at about 1 Hz. These processes control body velocities and select footholds when the vehicle is in terrain-following mode.

In addition to the *periodic* [53] processes above, several *sporadic* processes are activated occasionally to support the computation, to collect test data for analysis, and to handle errors, including:

—Data Log Collection (DLC)—one or more GEM processes that execute on demand from an outside observer, collect traces of data from certain processes, and pass these traces to an attached computer for analysis.
—Initialization (I)—one global GEM process and one GEM process local to each multiprocessor node, each of which executes only once at start-up time. These processes initialize application-specific data structures.

Figure 2 shows a selected subset of the processes comprising the robot's operating software as well as their communication and control relationships. The arrows represent data communication. While the precise meaning of each type

of arrow is elaborated in Section 5, here they indicate that processes interact using elements of two distinct models of communication:

—A process may use as input the *current values* of various hardware sensors and software variables and produce as output a set of current values. Inputs are assumed to be always present. Output values overwrite the previous values. Such processes are typically executed periodically, where the period is determined by the rates of change of the inputs. In Figure 2 for example, the LS processes run periodically, read the current values of the leg position, velocity and pressure sensors, and read the current values of the desired leg position, velocity and pressure issued by the LMP and BS processes. The LS processes then update the commanded positions, velocities, and pressures for the leg actuators.

—A process may use as input *discrete messages* containing commands or data values and produce such messages as output. Messages are queued up and never overwritten. Such processes are typically executed when new input messages arrive. In Figure 2, the CI process accepts commands from the operator and sends discrete command messages to VMP. The CD process receives discrete messages from other processes and displays the information in them on the cockpit display.

In reality most of the processes use both models, as demonstrated by the VMP process. It normally runs periodically, updating current values of commanded position, velocity, and so on. When changing from one walking gait to another, the VMP process sends discrete gait-change commands to many of the other processes for synchronization purposes.

4. PARALLELISM AND SCHEDULING

The subsystems in the ASV application and the control tasks in those subsystems differ widely in size. For example, because the Terrain Scanner (TS) subsystem performs multiple control tasks, including interactions with the terrain scanner and transformation and storage of the terrain data, it is represented as several independently schedulable, asynchronous activities (GEM processes). These GEM processes execute and communicate at moderate rates (2 Hz execution rate) and jointly require the processing power of at least two Intel 8086 CPUs. In contrast, the leg servo control (LS) task can be represented as a single activity (GEM process) scheduled to execute at a very high rate (100 Hz) on a single CPU. Internally, LS consists of two separable control tasks that share extensive amounts of code and data. Only one of the control tasks is active at a given time, depending on whether the leg is on the ground or in the air.

The coexistence of multiple, loosely interacting, larger activities (as in TS) with tightly interacting, small activities (as in LS) motivates the support of two different activity sizes in GEM—processes and microprocesses. A GEM *process* may represent a larger control activity. It can be scheduled for execution at a moderate cost in time and typically interacts loosely [32] with other processes. For representation of activities that are activated frequently and interact tightly with each other, a programmer may select a GEM *microprocess*, which is a

separable control activity within a single GEM process. It may interact with other microprocesses in the same or in a different process, and it can be activated at a low cost in time.

GEM does not support the dynamic creation of processes, which is of limited usefulness for the ASV and many other robotics application. Specifically, given the high rate of execution (100 Hz) of the LS process, driving a single leg in the robot vehicle suggests that the overhead of replacement of an unreliable or crashed process by dynamic process creation (estimated at more than 2 milliseconds[3]) is too large. Such dynamic reconfiguration may be performed using one or several statically created "buddy" processes [58] that sleep in operating system queues until they are needed.

4.1 Processes

Process definition—A GEM *process* defines a single address space for a statically defined set of control tasks. A process may be in any one of the following states.[4] The various **Asleep** states are defined so that process actions can be time- and/or event-driven:

Running	currently executing
Ready	available for execution but not currently executing
Asleep-T	waiting until a time at which to resume execution
Asleep-W	waiting until a "wakeup" call is received
Asleep-WorT	waiting until either of the two previous events
Asleep-WandT	waiting until both of the two previous events

The initial state of a process can be any of the states listed above (except **Running**); the specific time until which a process initially sleeps must be specified, if applicable; the final state of a process is a sleep state from which it cannot be awakened.

The following *scheduling information* is maintained in the process control block (PCB) of each process:

Deadline	Specifies the time interval, measured from the time at which it is made ready [53], within which the process should complete the execution of all of its microprocesses that are ready to run (if such microprocesses exist—see Section 4.2 for the definition of microprocess); this is used when the "Deadline" mode is selected for the process scheduler (see Section 6.2).
Period	Specifies the process' period of execution, that is, the number of time units between the times at which the process is made ready to run; its value may again depend on the values of its microprocesses' periods; it is undefined if the process is sporadic [53].
Priority	The scheduling priority of the process (higher priority processes always run before lower priority processes).

[3] This estimate is derived from timings of static process creation.

[4] Actually, several other process states exist, but they are not visible to the user. See the Appendix.

TimeSlice The number of time units this process should be run before it loses the CPU and another process of the same priority is run. This is used when the RoundRobin mode is selected for the process scheduler (see Section 6.2).

Process scheduling operations—The scheduling operations provided by GEM allow processes to control themselves, and they allow higher level system or application-dependent "scheduler" processes to control the actions of other processes. Specifically, a process (when **Running**) can place itself into any of the **Asleep** states by use of the operation:

GoToSleep (State, WakeTime)

"State" specifies the specific waiting state to be used and "WakeTime" specifies the time at which the process should be awakened; if "State" is **Asleep-W**, WakeTime is ignored. The counterpart of this operation is executed by other processes:

WakeUp (ProcessId)

This operation has the following effects:

—If the given process is currently **Asleep-W** or **Asleep-WorT**, it will be made **Ready**.
—If the given process is currently **Asleep-WandT**, it will be made **Asleep-T**.
—If the given process is currently **Running**, **Ready** or **Asleep-T**, the process state is not changed.

The first WakeUp performed on a **Running**, **Ready**, or **Asleep-T** process is not lost—GEM remembers this event until the next execution of GoToSleep by the target process. However, subsequent WakeUps do not pend, they are discarded. The purpose of the pending first WakeUp is to eliminate the possibility of lost WakeUps due to race conditions. Appendix B contains a complete description of GEM's state transitions.

In addition to the GoToSleep operation executed by a process on itself, application-dependent or operating system schedulers can control the execution of a process by setting its scheduling parameters (timeslice, priority, deadline etc.), by awakening it via the WakeUp operation and by forcing it to enter an **Asleep** state using the operation:

PutToSleep (ProcessId, Mode, State, WakeTime)

where "ProcessId" is the name of the target process, "State" specifies the sleeping state into which the target process should be placed, and "WakeTime" specifies the time at which the target process should be awakened, if applicable.

"Mode" denotes the time at which the PutToSleep action should become effective. Specifically, if "Mode" has the value "immediate," then the PutToSleep action becomes effective at the next possible scheduling point on the target processor, regardless of the current state of the target process. That *scheduling point* is defined as either the time of the next clock interrupt on the target processor or the next execution of GoToSleep by any process on the target

processor, whichever is first.[5] If "Mode" is "deferred," then PutToSleep does not affect the target process until the target process executes its next GoToSleep.

Use of the simple scheduling operations—The ASV operating software makes use of the GoToSleep, WakeUp, and PutToSleep scheduling primitives and of the *Period*, *Priority*, and *TimeSlice* values in each process' PCB. Deadline scheduling is not currently being used in the ASV application, but is available in GEM and is used in other real-time applications [76].

GoToSleep(**Asleep-T**, . . .) is used in the processes leg motion planning (LMP) and leg servo control (LS). Both are periodic processes with period values adjusted to the minimum frequencies with which leg actuators must be driven or commands must be sent to LS, 100 and 20 Hz, respectively. In the outline of those processes below, "Time" is a local variable recording current time set with the GetTime system call. The value of the "Period" parameter is the value in the **Period** field of the process control block. Comments are enclosed in curly brackets. Additional detail concerning the computations in both processes is given in Section 4.2.

```
LS Process:
  {This process is scheduled at priority 6}
  {Its period value is initialized to 10 milliseconds}
  GetTime(Time);
  while true do
    begin
    {main computation}
    Time := Time + Period;
    GoToSleep(Asleep-T, Time);
    end
LMP Process:
  {This process is scheduled at priority 5}
  {Its period value is initialized to 50 milliseconds}
  GetTime(Time);
  while true do
    begin
    {main computation}
    Time := Time + Period;
    GoToSleep(Asleep-T, Time);
    end
```

The LS process is run at a higher priority than LMP. Therefore, LS, which is a fairly small task with a short period, may interrupt the execution of LMP, which is a larger task with a longer period, if both processes are on the same processor. In the current process-to-processor assignment, this is the case.

Regarding the PutToSleep operation, GEM's distinction of "immediate" from "deferred" scheduling actions is motivated by the volatile nature of time-critical control actions involving mechanical or electronic equipment. Their interruption at arbitrary times by "immediate" PutToSleep actions can result in equipment damage or data inconsistencies between cooperating processes. Usually "deferred" and "immediate" actions are not mixed; "deferred" actions occur during normal

[5] Since processors cannot interrupt each other, the scheduling point is the next time that the process scheduler on the target processor can gain control conveniently.

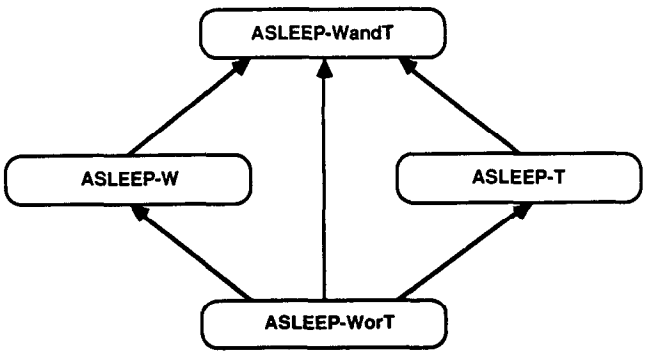


Fig. 3. The Asleep State lattice.

system operation, whereas "immediate" actions are used to handle exceptional conditions.

The effect of multiple PutToSleep actions in the time between two scheduling points of a single target process is an accumulation of the "State" and "WakeTime" parameter values stated in those actions. As with WakeUp operations, actions are accumulated rather than discarded in order to eliminate inconsistent state changes due to race conditions.

Currently the ASV application does not use the PutToSleep operation while in normal operating modes. During testing however, the PutToSleep operation is used to activate and deactivate particular subsystems for debugging.

Scheduling operations and state changes—Figure 3 shows the relationships between the various **Asleep** states that form a *lattice*.[6] When combining the current requested **Asleep** state of a process with a new **Asleep** state requested via the PutToSleep operation, the resulting state is the *join* of the two states. The resulting WakeTime is the maximum of the two original WakeTimes. The result of this "join/maximum" accumulation of states is that the target process always sleeps *at least as long* as has been requested by the callers of PutToSleep (or GoToSleep).

For example, two "State" values **Asleep-T** (with WakeTime = 1000) and **Asleep-W** specified in successive, "immediate" PutToSleeps on the same target accumulate to **Asleep-WandT** (with WakeTime = 1000). However, "deferred" and "immediate" PutToSleeps on the same target process are accumulated separately, since those actions must become effective at different times. A complete description of the effects of accumulated PutToSleep actions in connection with the various states of target processes appears in Appendix II.

4.2 Microprocesses

Microprocess definition—Much as multiple operations can be defined and invoked for single abstract objects [40], multiple streams of execution, called microprocesses, can be defined and activated in a single GEM process.

[6] A *lattice* is a set (e.g., set of states) with a partial ordering (shown by the arrows in Figure 3) which has certain properties [65]. The *join* of two states is the unique element found by following the arrows upward from the two states until they meet. For example, the join of **Asleep-T** and **Asleep-W** is **Asleep-WandT**, while the join of **Asleep-WorT** and **Asleep-T** is **Asleep-T**.

A GEM *microprocess* is small compared to concurrent objects in other systems [40, 44]. It consists of a sequence of instructions and associated data structures and shares address space with other microprocesses in the same process. In contrast to notions of lightweight processes now being developed for computer workstations [7], a microprocess resembles an event-handler in that it is always run to completion. As a result the microprocesses within a single process are executed sequentially.

A microprocess is executed when the single microscheduler defined statically as part of the process selects it for execution. Such selections (scheduling decisions) are made using the following information:

Local variables	The current values of local variables maintained by the microscheduler.
Inputs/Outputs	The current values of inputs and outputs associated with each microprocess and known to the microscheduler.
Scheduling information	A *Priority* indication, a *Deadline* that is the time interval within which the execution of the microprocess should be completed and a *Period* of execution.
Execution states	Microprocesses may be marked ready to run if they have been **Poked**, that is, if some external indication of need for their activation has been shown.

Microprocess scheduling operations.—Current implementations of abstract objects in computer networks [40, 44] typically assume that the processes providing an object's operations are scheduled independently of the processes using these operations. As a result, an invocation of an operation in an abstract object is typically implemented as the deposit of an invocation message in a buffer associated with the appropriate operation. This model of autonomously executing processes and microprocesses can be implemented in GEM using appropriate microschedulers and the data communication facilities described in Section 5. However, for prompt real-time response to requests for services implemented by microprocesses, GEM provides a simple, high-performance mechanism similar to "blocking" mechanisms in other multicomputer operating systems [31]. Namely, the services of a microprocess can be requested by poking its unique "control" input. Such "Poke" operations will wake up the microprocess' microscheduler if it is not currently running.

In contrast to invocations of abstract objects, the "control" action associated with the poking of a microprocess is separated from the communication of data values. This decoupling is intentional and increases the performance of the Poke operation, allowing microprocesses to cause others to perform computations or update output values with or without new input data values.

The control inputs and outputs of microprocesses are entities that contain the values "on" or "off." Each microprocess refers to its control and data (see Section 5) inputs and outputs with local names for *ports*. The interconnections of ports are specified separately so that they can be changed without changes to

port names. As a result, microprocesses can be assembled into cooperating groups and dissembled easily. Each output control port [84] of a microprocess can be connected to multiple input control ports of other microprocesses, and a single input control port can receive "Pokes" from any number of output control ports. The operation

```
Poke (PortId)
```

updates a data structure in the target GEM process that keeps track of the **Poked** control ports of its microprocesses. If the GEM processes containing the microprocesses to whose control input ports the output control port "PortId" is connected are not currently **Ready** or **Running**, then the Poke operation also performs WakeUp operations on the target processes. As with the WakeUp scheduling function for processes, multiple Poke operations on a single control port are lost if they appear with a frequency higher than their frequency of service. Only one Poke is remembered.

While the value in a control port is set to "on" with the Poke operation, the port "PortId" is read and reset by the microscheduler with the nonblocking operation

```
Value := Test-and-Reset (PortId)
```

When this operation completes, "Value" contains the current value in the port, and the port has been reset to "off" if it was set to "on."

If there is no work to be done by any of the microprocesses within a process, then the process' microscheduler may use the following operation to allow the process to go to sleep until there is more work to be done:

```
Sleep-If-Not-Poked (State, WakeTime)
```

This operation checks to see if a Poke has occurred since the process was last awakened. If so, it returns immediately, allowing the process' microscheduler to continue scheduling microprocesses. If no Poke has occurred, the operation executes a GoToSleep using the State and WakeTime parameters.

Sample microscheduler with microprocesses—Consider a slightly revised version of the LS process in Section 4.1. In this version LS has two data input ports: the first input contains commands issued by LMP; the second input contains commands issued by BS. If the leg is on the ground, then LS accepts commands from BS. If the leg is in the air, then LS accepts commands from LMP. If neither BS nor LMP have issued commands, then LS proceeds with its default action. LS always issues output commands to its leg's actuator.

The default subtask and the two subtasks in LS that interaction with LMP and BS can be described as three microprocesses with a microscheduler. These microprocesses are activated by the microscheduler. Specifically, the microscheduler makes scheduling decisions using the *Period* values of the microprocesses and the "Poke" signals generated by LMP and BS each time a command is issued. Upon selection of a microprocess using the algorithm shown below, the microscheduler removes the **Poked** indication and runs the microprocess to completion.

Upon microprocess completion, control returns to the microscheduler which continues operating until all **Poked** microprocesses have been run. Then the microscheduler suspends its process for at most the interval of time indicated by the value of *Period* in its process control block. This value may be set, for example, to the minimum of the microprocesses' *Period* [53] values, so the LS process will issue commands to the leg actuators with the minimal frequency required, even in the absence of commands from BS or LMP.

The use of **Asleep-WorT** as the "State" parameter in the SleepIfNotPoked operation allows Micro-Process-1 and Micro-Process-2 to be scheduled by Poke operations and Micro-Process-3 to be scheduled periodically (in the absence of Poke operations). "Port-*i*" is a boolean-valued variable set by the Poke operations addressed to control port *i*; the variable "NextTime" is used for time maintenance:

```
LS Process:
  {This process is scheduled at priority 6}
  GetTime(NextTime);
  while true do
    begin
      if test-and-reset (Port-1)
        then Micro-Process-1
      else if test-and-reset (Port-2)
        then Micro-Process-2
      else
        Micro-Process-3;
      NextTime := NextTime + Period;
      Sleep-If-Not-Poked (Asleep-WorT, NextTime);
    end
```

Micro-Process-1 and Micro-Process-2 perform calculations and activate the appropriate actuators (for brevity, the default actions taken by Micro-Process-3 are not shown below):

Micro-Process-1:
{leg is on the ground}
Get commanded actuator pressures from BS command data port and activate the appropriate hydraulic actuators.

Micro-Process-2:
{leg is in the air}
Get commanded leg acceleration, velocity and position from LMP command data port. Activate the appropriate hydraulic actuators using feedback from actual leg velocity, position and actuator pressure sensors.

The performance of microprocesses and "Poke" operations are compared to the performance of processes and "WakeUp" operations in Section 6.2.

5. TASK COMMUNICATION: THREE MODELS

Independent of task size, three different models of task interaction can be identified in the ASV software and in other real-time software [41, 71, 74, 84]. The "control" interactions between tasks occurring in those models can be implemented using GEM's process-scheduling primitives described in Section 4.1. The "data" interactions required for each model are implemented using a single message communication facility adaptable by parameterization. The three models and GEM's communication primitives are described next.

Model 1: *Asynchronous execution with data loss*—In this model, tasks execute asynchronously with respect to each other. Tasks generate outputs continuously based on their inputs, which are always assumed present. Communications between tasks can occasionally be lost, but tasks operate correctly as long as their inputs have not aged beyond statically defined tolerances known to the application programmer. In the ASV software, this model of communication is used for the sharing of data between asynchronously executing control tasks within subsystems, such as body and leg servo control (BS and LS), as illustrated by the thick black arrows in Figure 2. In addition, the model is used to decouple the operation of independent subsystems. Other researchers have suggested the use of this model at a low level of control in real-time software [8, 41, 84].

Model 2: *Synchronous execution without data loss*—In this model, tasks execute synchronously, where execution is driven by the acceptance of individual items of input (e.g., a receipt of a service request and its parameters), which are not always assumed present. Inputs and outputs cannot be lost without jeopardizing the correctness of system operation. This model of communication is one of the most frequently used models supported by the communication facilities of network and multiprocessor operating systems [11, 31]. In the ASV software, it is used to communicate commands and associated data to various processes, during initialization or during execution when such commands should not be lost. In Figure 2, the use of this communication model is indicated by the thin black lines.

Model 3: *Synchronous or asynchronous operation with possible loss of aged data*—A hybrid of models 1 and 2, this model of task interaction assumes that a fixed-size set of recent output items of one task is available as input to other tasks. This set of output items is explicitly ordered in time. Synchronous operation permits prevention of loss of such aged data by disallowing outputs issued by sending tasks, whereas asynchronous task execution results in loss of aged data if instantaneous execution rates of communicating tasks differ too much. In the ASV software, this model is used for data logging, as indicated by the thick gray arrows in Figure 2.

5.1 The GEM Communication System

Basic facilities—In GEM the three models of communication are implemented for processes and microprocesses using mailboxes and envelopes. A *mailbox* consists of a pool of envelopes (in implementation, buffers of equal size), some of which are free and some of which are in a queue of envelopes currently in use. *Envelopes* are reusable containers for messages and are explicitly acquired and returned as part of the message-passing paradigm. A *letter* is an envelope into which a message has been written. Each envelope pool is associated with exactly one mailbox for reasons of buffer addressing and performance in queue management. Prior to sending a message, a process acquires an envelope from the mailbox using the operation:

GetEnvelope (MailboxId, Envelope, Status)

The "MailboxId" is a unique name for the mailbox. "Envelope" is the buffer returned as a result of the call. "Status" reports the success or failure of envelope acquisition.

Once an envelope is acquired, a user writes into it to construct a letter and then sends the result to its mailbox:

SendLetter (MailboxId, Envelope, Status)

"Status" returns error or completion codes regarding the successful deposit of the "Envelope" in the mailbox "MailboxId".

An envelope containing a letter is received from a mailbox with

GetLetter (MailboxId, Envelope, Status)

The letter's envelope must be released explicitly after reading with the operation

DiscardEnvelope (MailboxId, Envelope, Status)

The communication operations listed above copy references to envelopes (sending "by reference") rather than copying their letter contents. The operations assume that the readers and writers can share memory. The operations are therefore relatively fast and are particularly useful for transferring large messages.

For communication "by value," the following composite operations assume that writers possess "SourceEnvelope"s whose contents are first copied into intermediate envelopes automatically acquired and released as part of the mailbox operations and then copied into readers' "DestinationEnvelope"s:

SendLetterCopy (MailboxId, SourceEnvelope, Status)
GetLetterCopy (MailboxId, DestinationEnvelope, Status)

These operations send (get) a single letter to (from) the mailbox "MailboxId" from (into) the envelope "SourceEnvelope" ("DestinationEnvelope") explicitly acquired by the user. When readers and writers cannot share memory, these "by value" operations must be used.

Implementation of the communication models—GEM provides three models of communication by parameterizing the mailbox communication facility described above. A parameterized GEM mailbox consists of two data structures with access parameters that describe the actions taken when those data structures are manipulated. The two data structures are

—*Free Pool*—a pool of envelopes not currently in use, and
—*Letter Queue*—a queue of filled envelopes (letters) sent to this mailbox but not yet received.

The mailbox parameters are as follows:

—*Sticky*—This parameter controls the action taken when an empty queue of letters is read using the GetLetter operation; either the most recent letter written with the SendLetter operation is returned, even if it has been read before, or a failure status is issued. In other words this parameter controls whether the most recently sent letter "sticks" to the mailbox regardless of the number of times the mailbox is read.

—*QueueFull*—If the letter queue is full, then the writer of a letter is either issued a failure status, or the oldest letter in the letter queue is discarded and replaced with the new letter.
—*PoolEmpty*—If a new envelope is acquired using the operation GetEnvelope when the pool of available envelopes is empty, then one of two actions can be taken: either a failure status is returned, or the oldest letter in the letter queue is discarded in order to reuse that envelope.
—*MaximumQueueLength*—The maximum length of the letter queue.
—*EnvelopeSize*—The size in bytes of each envelope in the pool.
—*NumberOfEnvelopes*—The total number of envelopes in the mailbox.

The three models of communication are implemented as follows:

—*Model* 1—The queue is made "sticky" (controlled by the parameter Sticky), so a reader always receives the most recent letter written. To ensure that new information can always be entered into the mailbox, the oldest letter is discarded if a writer cannot find an unfilled envelope (PoolEmpty) or if the mailbox queue is full (QueueFull). MaximumQueueLength is set to 1. Neither readers nor writers will ever be denied envelopes (letters) if NumberOfEnvelopes is at least NumberOfReaders + NumberOf Writers + 1.
—*Model* 2—Failure status is returned in all "special" cases, namely, when an empty queue is read or when GetEnvelope is performed on an empty pool, and when writers find a full queue. MaximumQueueLength is generally set to NumberOfEnvelopes.
—*Model* 3—Readers receive failure status when the queue is empty. In contrast to Model 2, the mailbox can always be written by discarding the oldest letter in the mailbox, both if the envelope pool is empty or if the letter queue is full. MaximumQueueLength is set to the maximum number of most recent letters to be saved. Typical interactions with a mailbox in this model consist of writers continuously attempting to write into the mailbox while readers inhibit writes by locking the mailbox occasionally (using operations not explained in this paper) in order to retrieve a consecutive set of letters from the mailbox; such a consecutive set of letters provides a history of the activities of writers within a period of time.

5.2 Microprocess Communication

While communications between microprocesses can potentially follow any of the models above, Model 1 has proven most useful because it approximates the "analog" model of computation shown useful for low-level control [8]. In this model, asynchronously executing microprocesses continuously use the current values on their input "lines" to compute new values for their output "lines," and a changed signal on any "line" must be visible to all connected microprocesses.

The mailbox paradigm does not seem directly suitable for implementing "lines" between microprocesses because the implementation of a microprocess should not change when its interconnections with other microprocesses change. For this reason data ports [75, 78, 80, 84] are introduced. Data ports can connect multiple microprocesses residing in the same or in different processes in the same fashion as the control ports used by the Poke operation described in Section 4.2. This

extension of the mailbox mechanisms automatically maps the local name of a data port in a microprocess to the global names of mailboxes that represent this port or the data ports connected to it. In the current implementation, a single "sticky" mailbox is associated with each input data port of a microprocess.

The operations defined for port-based communication are Read and Write. The operation

```
Read (PortId, Data)
```

reads the value in the data input port "PortId" into the variable called "Data". Since port-based communication uses Model 1, a data value is always assumed present in the port and will not be destroyed as a result of the read operation. Therefore, a data input port can be implemented using a "sticky" GEM mailbox, regardless of the number of outputs to which this input is connected and regardless of the number of microprocesses reading from this port.

The operation

```
Write (PortId, Data)
```

writes the value in the variable "Data" to the output data port "PortId". As required in Model 1, the operation uses the value in "Data" to overwrite the contents of all input ports bound to this output. We are considering the optimization of port-based communication so that it performs better than mailbox-based communication for small fan-outs for input ports.

5.3 Shared Memory

In addition to the mailbox and port communication mechanisms previously discussed, GEM allows users to define and access areas of shared memory (where the hardware topology permits). Such areas of shared memory are always created as lockable objects and are addressed in the same fashion as mailboxes. Synchronization of access may be done using the GEM operation:

```
TestAndSetFlag (SharedObjectName)
```

Alternatively and for improved performance, users can make direct use of TestandSet operation available in assembly language. Shared memory is particularly useful when working with large arrays of data, for example, image data from the terrain scanner in the ASV.

6. IMPLEMENTATION OF GEM

This section addresses implementation issues relating to the computer architecture for which GEM is used and to GEM's application domain, real-time control programs.

6.1 Multiprocessing Hardware

The GEM operating system has been implemented on a hierarchical multiprocessor system. This hardware is typical of computers embedded in electromechanical, real-time systems. Program development tools reside on a separate, attached machine offering secondary storage and user interface facilities, and the interconnection structure of the embedded system is inhomogeneous [67] and is tailored to the specific application. In the current multiprocessor, processors are

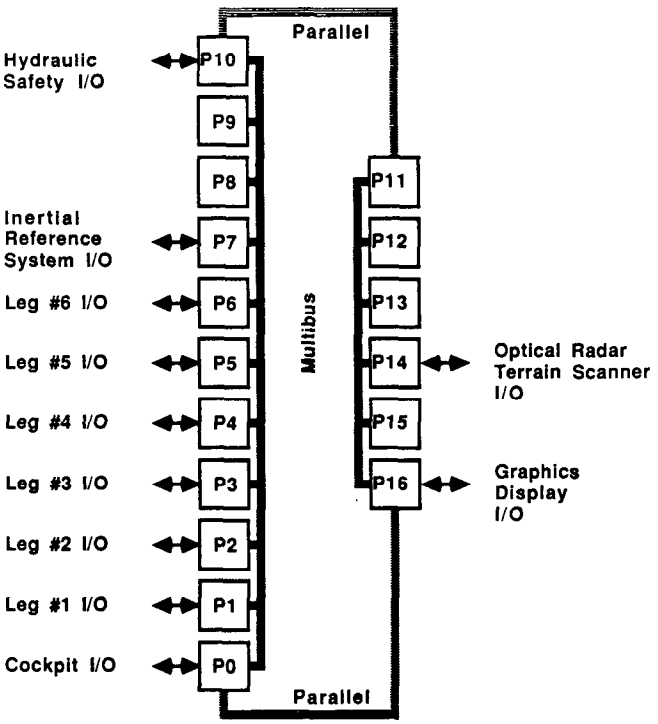


Fig. 4. The ASV Robot's computer hardware.

partitioned into two clusters connected with at least one fully duplexed (8-bits each direction) parallel link. Within each cluster shared memory is accessed via a Multibus (see Figure 4). Thus the interconnection topology has three levels: *local* for processes on the same processor; *intracluster* for processes in the same cluster (able to share memory); and *intercluster* for processes on different clusters (unable to share memory).

The processors are Intel 86/30 single-board computers, each containing an Intel 8086 microprocessor (8 MHz clock, 750 ns Basic Instruction Cycle), an 8087 floating point coprocessor, and 128K-256K bytes of memory (750 ns cycle time) partitioned into locally and globally accessible sections. Devices (displays, sensors, actuators) are attached to individual computers.

A 6-processor cluster is dedicated to the terrain scanner, graphics display, and vehicle guidance, and an 11-processor cluster runs vehicle control, leg control, and inertial navigation. The sample devices shown in Figure 4 are the optical radar terrain scanner, the graphics display, the inertial reference system, the hydraulic safety system, and the cockpit displays and controls.

6.2 Operating System Implementation and Performance

Distribution of GEM—The *GEM machine* is defined by the operating system operations accessible to application programs. It consists of a small, replicated kernel implementing basic operations invoked synchronously, and shared operating system utilities invoked asynchronously and executed concurrently with

the invoking processes [31]. The basic operations of GEM are the process and microprocess scheduling and communication facilities described above and some small, fast utilities such as GetTime. Including default-interrupt handlers, the kernel occupies about 20K bytes of code and data.

The shared utilities include user interface, I/O, and monitoring facilities, which are implemented as processes, instances of which may reside on one or more processors. One important example of such shared utilities are the intercluster communication servers which are attached to each end of the intercluster parallel link(s). These servers act as surrogates for processes in each cluster wishing to schedule, or communicate with, processes in the other cluster.

Processes and process scheduling—Process scheduling is done on a per processor basis using priority levels. Specifically when a processor's currently running process executes a GoToSleep operation (or is PutToSleep), the next process to be run is chosen by that processor's scheduler from its own ReadyQueue. The choice made depends on the scheduler's *mode*, which is RoundRobin or Deadline. In *RoundRobin* mode, the next process is the one with the highest priority; it is run for an interval of size TimeSlice, as specified in its control block, and it is placed at the end of the ReadyQueue (at its priority level) after its execution. The running process is preempted only if a higher priority process arrives in the ReadyQueue during its execution. The preempted process is placed back on the ReadyQueue ahead of all other processes at its priority level. If preemption does not occur and if at the end of its timeslice a running process remains the only one at its priority in the ReadyQueue, then its execution is continued for another TimeSlice interval.

A process scheduler operating in *Deadline* mode performs priority-based scheduling as well. However, in this mode all processes in each priority level are run in "shortest deadline first" order, and they are run to completion (their TimeSlice parameter, if any, is ignored).

The data structures used for process scheduling are designed to minimize the execution times of commonly performed operations. For example, to minimize search time within a processor's ReadyQueue, it is implemented as an array of circular queues—one for each priority level. This allows the queue operations Insert, Delete, and RoundRobin to be done in constant time in RoundRobin mode and in linear time in Deadline mode. The resulting low latency of ReadyQueue operations is shown by measurement of the "Delete" operation, cf. below. This measurement is taken (using a logic analyzer) in RoundRobin mode on a single processor:[7]

Operation	Local time
ReadyQueue dequeue	260 μs
Restore state	105 μs
Total **Ready → Running**	365 μs

[7] As with all other timings appearing in this paper, the granularity of the timings is 5 microseconds for times less than a millisecond and 50 microseconds for longer times. Furthermore, timings were performed under "low load" conditions. That is, no extraneous processing or I/O was being done. For comparison, execution times for "typical" 8086 instructions (jmp, mov) are approximately 2 microseconds; a procedure call without parameters requires 15 microseconds.

Table I. Cost of Process Switching

Operation	Local time
EventTable/ReadyQueue	305 μs
Restore State	105 μs
Asleep-W → Running	410 μs
Save State	140 μs
ReadyQueue dequeue	260 μs
Running → Asleep-W	400 μs

As with the ReadyQueue, each processor's SleepQueue is organized such that its latency of access is minimized too. Specifically, each SleepQueue is a heap, with processes nearest their wake time at the root, that allows queue operations to be done in logarithmic time. In addition, a link to the ReadyQueue and an index into the SleepQueue maintained in each PCB permit rapid access to those queues when processes are blocked or awakened.

The processing done for each scheduling call is distributed among the processors involved by use of an EventTable residing in each machine and recording the scheduling actions to be taken for all processes on that machine. This table consists of a queue and an array. When performing a scheduling action, the (intracluster or local) process writes into the array indexed by ProcessId. If this is the first event for the process since the last scheduling point, then the ProcessId is also placed into the queue. At this point, the operation completes in the originating process, which results in very low latencies for performing WakeUp operations:

Operation	Local	Intracluster
WakeUp	165 μs	180 μs

The remaining processing required for WakeUp operations is done by the target processor's kernel. At each scheduling point (when its clock interrupt occurs or when one of its processes performs a GoToSleep operation), the kernel checks the EventTable and updates the states of all processes whose ProcessId's are in the queue. As a result, the cost of a GoToSleep operation is higher than that of a WakeUp, as demonstrated by the following measurement of a GoToSleep attempted by a process when a WakeUp is pending:

Operation	Local time
Attempted GoToSleep	580 μs

This time is a composite of the time required to save the state of the currently running process (140 microseconds), process the local EventTable with a nonlocal entry, and perform an aborted enqueue on the ReadyQueue (335 microseconds), and restore process state (105 microseconds).

Similarly, overheads stemming from EventTable manipulation ensue for a process' state change from **Asleep-W** to **Running**, but not for a state change from **Running** to **Asleep-W**: (see Table I).

Table II. Process versus Microprocess Overheads

	Process	Microprocess
Scheduling and Descheduling		
Respond to WakeUp\Poke	580 μs (810 μs)	140 μs (950 μs)
Execution	—	—
Output		
Data	—	—
Control	WakeUp: 180 μs	Poke: 245 μs (425 μs)
Total	760 μs (975 μs)	395 μs (1375 μs)

Note that a processor's EventTable does not record the current states of its processes. Instead, the table records the states in which the processes would be if scheduling actions were performed immediately rather than at discrete scheduling points. As a result, the actual states of the processes will "catch up" with the EventTable's "requested states" only at those points in time.

Comparison of processes and microprocesses—To understand the trade-offs of using processes versus microprocesses for representing real-time control tasks, the differences in functionality and performance of process activation by use of a WakeUp operation and microprocess activation by use of a Poke operation have to be considered.

Assuming the existence of multiple microprocesses that interact with Poke operations, the total time spent by a target microprocess when responding to a Poke operation can be decomposed into the following components: (1) *scheduling time*—activation of the corresponding microprocess by the target's microscheduler in response to a Poke operation; (2) *execution time* of the target microprocess; (3) *output time*—activation of the next microprocess (which may include execution of a WakeUp operation on the process containing that microprocess); and (4) *descheduling time*—deactivation of the target microprocess.

In GEM as in other process-based systems [11], having ignored the fact that Poke operations carry an indication of the type of service desired, such functionality can also be attained by (1) process activation in response to a WakeUp performed by another process, (2) execution of the target process, (3) execution of a WakeUp operation on the next process, followed by (4) deactivation of the target process.

The following table presents the costs of Poke operations and microprocess activation and deactivation overheads and contrasts them with the respective costs incurred for processes. In the table, the cost of deactivation of the previous activity is added to the cost of activation of the next activity, thereby combining (1) and (4). The scheduling and output times each are listed for processes and microprocesses. In the larger process scheduling time, the target process is not currently running. Similarly, the larger microprocess scheduling time includes the cost of waking the microprocess' parent process. The larger Poke time includes the cost of performing a WakeUp on the microprocess' parent process. In all cases the use of intracluster operations is assumed (see Table II).

As demonstrated by the measurements above, a Poke operation is more expensive for the originating microprocess than a WakeUp. However, the total

overheads associated with "poking" a microprocess in a **Running** process (395 microseconds) differ substantially compared to waking up a **Running** process (760 microseconds). Thus the use of microprocesses has substantial benefits when the programmer can determine a high likelihood of encountering a target process in a **Running** state, as is the case with bursty task interaction or if the start times and periods of interacting real-time tasks overlap due to the requirements of the sensors or actuators controlled by those tasks.

The following simple analysis derives "p," the minimum probability of encountering a **Running** target process that is required to make the use of microprocesses preferable to the use of processes. Again ignored is the fact that the Poke operation has additional functionality compared to the WakeUp operation such that it carries an indication of the type of service desired:

(1) Costs using processes:
$180 + 580 \times p + 810 \times (1 - p)$
(2) Costs using microprocesses:
$245 \times p + 395 + (1 - p) + 140 \times p + 950 \times (1 - p)$

The solution to this equation is about $p = 0.49$. Therefore if there is at least a 50 percent chance of finding the microprocess' scheduler running, then a microprocess implementation is faster on the average than an equivalent process implementation.

Given this low required value of "p," it appears that even programmers with only scant knowledge of the application's run-time characteristics should be able to make good use of microprocesses.

Task communication—naming—Two naming problems arise for task communication in the multiprocessor architecture. First at the hardware level, no two processors have the same "view" of the address space of the shared memory. As a result, the shared memory on each board has two sets of addresses: one when accessed by the local processor and another when accessed from other processors. Second, the data ports and their interconnections used by microprocesses must be mapped to GEM mailboxes, which are the underlying communication facility. Both problems are handled in a similar fashion, by maintaining addressing and connection tables in shared memory. For the first problem, the mailbox mechanism maintains a MailboxAddressTable in each processor that contains the appropriate addresses of all mailboxes accessible to processes on the processor. For the second problem, the binding of each input port is recorded to a mailbox. In addition, the port-addressing mechanism maintains a list of the input mailbox addresses connected to each output port, so Write operations can be mapped properly.

Task communication—comparison to shared memory—Mailbox-based communication is not appropriate for sharing large amounts of information, as with the terrain map shared by different processes in the Terrain Scanner subsystem in the ASV.

The basic overheads for sharing information via shared data versus messages are given by descriptions of the costs of (1) GEM's lock and unlock operations

Table III. Trade-offs in Mailbox Location—Intracluster

Operation	Local	Intracluster
GetEnvelope	155 μs	160 μs
Transfer of 87 bytes	260 μs	335 μs
SendLetter	180 μs	190 μs
Other processing	95 μs	95 μs
SendLetterCopy (87 bytes)	690 μs	780 μs
GetLetter	185 μs	200 μs
Transfer of 87 bytes	260 μs	335 μs
DiscardEnvelope	160 μs	165 μs
Other processing	80 μs	80 μs
GetLetterCopy (87 bytes)	685 μs	780 μs

on shared memory with (2) the TestAndSet and Write operations available in assembly language with (3) a single zero-length message:

Operation	Cost
TestAndSetFlag	80 μs
ResetFlag	80 μs
TestAndSet	10 μs
Write	5 μs
GetLetterCopy (0 bytes)	445 μs[8]

In the best case, the direct use of shared memory improves access to shared data by a factor of thirty. Furthermore, in the ASV application, shared memory must be used in some places due to restrictions in available memory that preclude the copying of data.

Task communication—mailbox location—The trade-offs regarding the locations of processes and mailboxes are demonstrated by the timings below, comparing messages being sent (received) to (from) a local and an intracluster mailbox. Bus contention[9] and memory contention are not considered in these timings, measured in microseconds (see Table III).

As evident from the measurements in Table III, mailbox locations within a single cluster of the GEM hardware do not affect performance severely if contention does not exist and message sizes are relatively small. But intercluster operations across the parallel link connecting the two clusters carry significant overheads due to the low speed of the parallel bus (0.1 Mbps) and the extra processing and buffering overhead incurred by use of link server processes in each cluster (times are in microseconds). (See Table IV.)

The comparatively good performance of the intercluster WakeUp operation is due to the small amount of information sent across the link.

[8] This number is computed by deducting from the total time for GetLetterCopy (780 μs) by the time required to transfer its contents across the intracluster bus (335 μs).

[9] Bus contention appears insignificant for the hardware and application programs described in this paper.

Table IV. Trade-offs in Mailbox Location—Intercluster

Operation	Intracluster	Intercluster
WakeUp	180 μs	1500 μs
Link transfer (87 bytes)	335 μs	6500–7000 μs
SendLetterCopy (87 bytes)	780 μs	8150 μs
GetLetterCopy (87 bytes)	780 μs	10550 μs

Portability of the GEM machine and of its programs—GEM and its applications can be fairly easily ported to multicomputer hardware with the following characteristics:

—GEM and its applications can execute on multiple, linked, shared-memory multiprocessors, where each multiprocessor must offer sufficient amounts of global memory directly accessible to all of its processors (required for representations of mailboxes and other data structures shared by the GEM kernels on each processor), in addition to the private, local memory each processor may possess.
—Machine code instructions equivalent to indivisible "TestAndSet" must be available within each multiprocessor so that GEM's synchronization constructs can be implemented.
—Timing devices providing a somewhat accurate notion of global time must exist within each multiprocessor (the current architecture uses a single global clock signal).
—Processes executing under GEM on different multiprocessors can only assume the availability of mailbox-based communications and process-to-process scheduling primitives. Shared memory across multiprocessors is not supported.

GEM does not make any other hardware assumptions. In particular interprocessor interrupts are not assumed to exist. When ported, the following machine-dependent components of GEM have to be rewritten [10]:

—Interrupt handling, including the access functions to the timing device.
—I/O instructions implemented as interrupt handlers for specific devices—in GEM, byte-serial and block-serial interfaces exist for such I/O instructions, where the latter are implemented using GEM's message communication mechanism.
—The saving and restoring of process state.
—The invocation of operating system operations by application code. In the current version of GEM, the operating system and all application code execute in the same protection domain. As a result, all synchronous operating system operations are invoked by simple procedure calls. Future versions of GEM may be stored in ROM or PROM memory, thereby protecting the operating system code.

7. SOFTWARE DEVELOPMENT SUPPORT FOR GEM

Operating software should be constructed from reusable "software piece-parts" that can be "snapped together" to form more complex parts [84] and adapted to fit the changing requirements of the parts being assembled [71]. While the

potential benefits of this methodology have been recognized, their realization is difficult due to a lack of computer-based programming tools that support software assembly and adaptation.

Two specific characteristics of GEM's application domain add to the need for support tools. For one, programmers of robotics applications should not be required to become expert users of parallel machines. They should instead be given domain-specific tools that concentrate on the functional descriptions of their programs and not on descriptions catering to the parallel hardware or its operating system. At the same time, those software descriptions must result in efficient, executable operating software. Therefore, hardware-specific implementation decisions and optimizations must be done—but with minimal programmer interaction. Second, software adaptations [2, 37] must be supported because the performance requirements of different versions of the operating software vary and reconfigurations of the experimental hardware and the attached mechanical or electronic systems occur. Categorized as dynamic and static, such program changes include static changes in the allocation of hardware resources given a fixed implementation of parallel operating software, and changes in implementation of operating software given a fixed-program specification. Examples of dynamic changes in the ASV software are the adaptation of operating software when operating modes are changed or when exceptional conditions occur.

The STILE graphical programming system [78, 84] and the ISSOS programming environment [68] are being constructed to test hypotheses regarding the utility of domain-specific programming tools and of support for program adaptation. At their lowest levels, both systems use the same basic model of concurrent software. They assume that software is constructed from components called *basic* objects that contain procedures and data structures written in a sequential language [70] (currently Pascal and C). The systems diverge in their manipulation of these components and in the user interfaces presented.

7.1 STILE—Domain-Specific Tools

The STILE system interacts with users graphically. In this case, *basic* objects contain programmer-defined application code and generic communication code. In terms of GEM, each basic object describes the set of microprocesses (see Section 4.2) in a GEM process. These microprocesses communicate with other basic objects using data and control ports.

Basic objects are separately compiled and placed into a library in object code format. Separate testing of basic objects and their reusability when snapped together to form complex parts are possible because their contents and their communication are designed according to a precise methodology.[10] The interpretation of port-to-port connections and the protocols for using them to those defined by Model 1 in Section 5. All code within a basic object is written regardless of that with which the object interacts, implying that all port identifiers are local names, and the interconnection language is separate from the language used within the basic object.

[10] In this sense, STILE's approach follows the spirit of UNIX—UNIX is a Trademark of AT&T Bell Laboratories—pipes and filters.

The STILE graphical programming system is the basis for building a variety of domain-dependent abstract concurrency and communication models on top of the basic model provided by GEM (i.e., Model 1 of Section 5). The goal is that systems-oriented programmers provide application-domain experts—in this case, control engineers—with models that closely resemble familiar ones such as the analog model mentioned in Section 5.2. The STILE system is designed to present other models to application programmers such that they are unaware of something more basic as a foundation. In this way STILE provides true *abstraction* of communication models, and not simply *aggregation* or *hierarchy* as available in other superficially similar approaches [14, 47, 63, 64, 87].

In sum, the graphical programming of application software for GEM proceeds as follows:

—A graphical editor, used to design operating software components allows traversal of the hierarchy of components and facilitates abstraction of the communication protocols among them.
—Components are designed, tested individually, and finally compiled and placed into a library. Each component also contains a microscheduler, the code for which can be generated automatically, since the underlying abstract model of a component is consistent.
—The interconnection information is augmented with run-time details such as processor assignments for the processes and time-slices.
—Command files and initialization code are created and executed automatically to link, load, and initialize the operating software built from the primitive components.
—The program executes on the target machine.

The present status of STILE is this: two prototype graphical editing and program development environments have been completed and a third is underway. The first was built before GEM and is no longer in use, but did provide some insights into how the system should ultimately work. The second [80] was integrated with GEM and used as outlined above, but does not support many of the features needed for design of large systems.

The current effort seriously attempts to construct a practically useful environment. Its user interface is faithful to what we have termed the *engineering design metaphor* [78]. A software system is composed from parts that are described by catalog pages and blueprints. A catalog page contains specifications of the external behavior of each part organized into catalogs for easy reference and use. A blueprint (STILE graph) explains the implementation of the part by describing the relationships among component parts.

STILE understands there are three kinds of parts. It does not know what they mean, only about how they can interconnected; its understanding is purely syntactic.

—*Boxes* represent the active agents in a program, in this case the GEM processes or groups of them.
—*Ports* are the connection points on boxes for binding to other boxes, in this case GEM's control and data ports.
—*Links* are extension cords to connect pairs of ports together.

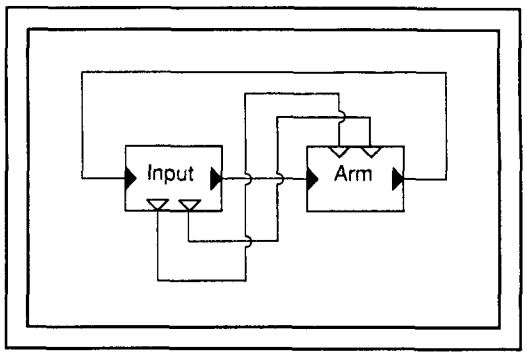


Fig. 5. Blueprint for robot control.

No inherent meaning is assigned to blueprints, so the STILE environment can be used to design software using a wide variety of methodologies and graphical languages. What the parts mean and how they work when "compiled" into executable code is up to a postprocessing program that takes the structural description of the parts (blueprints) as input. When the blueprints describe software for GEM, this postprocessor interprets interconnections among the data and control ports of the basic building blocks as described in Section 5.2.

If communication models other than those provided by GEM are used in blueprints, they may be constructed purely syntactically from basic objects using the STILE editor. Any of the components, including ports and links, can be built up from other parts using a variety of syntactic constructs, in which case the translation to an executable GEM program can be performed with a generic postprocessor that "expands" composite parts down to the level of basic objects. But for efficiency it may be better to offer additional low-level communication primitives in GEM and write a special-purpose graph compiler for blueprints having interpretations other than GEM's Model 1.

The simple problem described next [33, 80] provides an introduction to STILE and illustrates the potential value of STILE in building complex systems from simpler parts.[11] It has a hierarchy of components but does not show the abstraction of communication protocols.

Consider a robot arm with two degrees of freedom X and Y where stepper motors control the motion along each axis. The arm is controlled by the user's input of absolute coordinates for the arm's new position. The user must wait until the arm is at that position before entering another destination.

Figure 5 shows the topmost level of the robot controller hierarchy. The system contains two major subsystems, an input subsystem and a robot arm subsystem. For brevity, this discussion focuses on details of the "Arm" box. Briefly, the "Input" box allows a user to enter new absolute arm coordinates. When the box's input control port is poked, it prompts the user for new X and Y coordinates for the arm. It has two data outputs, one for the X and one for the Y coordinate. In addition, it has an output control line it pokes when the data ports have their new values.

[11] We have used the methodology suggested by this example to design more complex real-time control software, and have investigated at least a half-dozen other interpretations of STILE graphs ranging from analog computation to queueing networks to module interconnection.

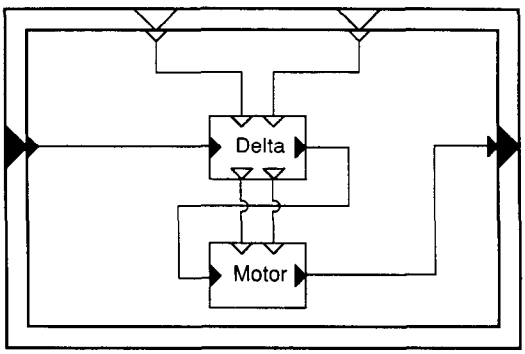


Fig. 6. Blueprint for arm.

The "Arm" box has an input control port and an output control port. When "Arm's" input control port has been poked, it reads the coordinate values on its input data ports and positions the robot arm there, then pokes its output control port.

"Input's" data ports are connected to "Arm's" data ports. Further, "Input's" output control port is connected to "Arm's" input control port and vice versa. Given this configuration (shown in Figure 5), the robot control system works as follows. After its input control port is poked, "Input" fetches new arm coordinates from the user. The coordinates are written to the output data ports and the output control port is poked. Since these are connected to the input ports on "Arm," the robot arm will be moved to the requested position. Finally, "Arm" pokes its output control port. Because this port is connected to "Input's" input control port, "Input" fetches a new position for the robot arm from the user and the process repeats itself.

Figure 6 shows the blueprint of "Arm"; notice the ports along the box's exterior correspond to its icon in Figure 5. When its input control port is poked, "Delta" converts the values on its input data ports from absolute coordinates to relative coordinates, writes the relative coordinates to its output data ports, and pokes its output control port. When "Motor's" input control port is poked, it gets the distances to move the arm from its input data ports, moves the arm to the specified location, and pokes its output control port upon completion of the move.

For brevity, the details of "Motor" (Figure 7) are not described. Figure 8 shows a basic part, "Delta" that corresponds directly to a GEM process and is not further decomposed into more primitive parts. It contains a single microprocess; in general there is one for each input control port. The "init" code is executed only once at system startup, and initializes the robot arm and the internal variables $x0$ and $y0$.[12] The "calc" microprocess performs the translation from absolute coordinates to relative coordinates, writes the results to its output data ports, and pokes its output control port.

The STILE graphical programming system is not a necessary part of the GEM operating system. Its parts are built on top of GEM, and use GEM's process

[12] This design is not the best strategy for initialization because "Delta" is not reusable. It has been chosen for simplicity of explanation.

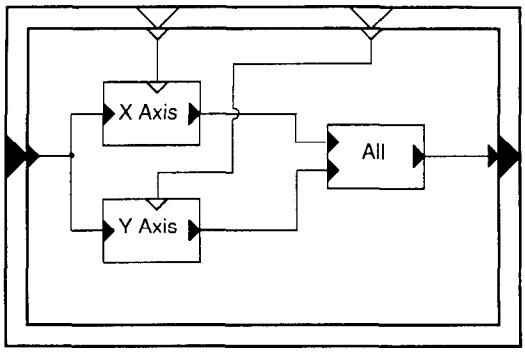


Fig. 7. Blueprint for motor.

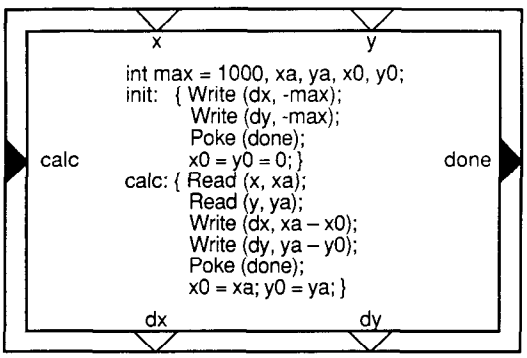


Fig. 8. Blueprint for delta.

scheduling and port communication primitives. Therefore, modification and augmentation of the programming support system can be done independently of GEM. The implementation of the run-time system has been decoupled from that of the programming support system, even though conceptually the two are highly intertwined. Decoupling has been achieved by cooperative design of GEM, during which attention was paid to the aims of the programming system as well as run-time efficiency and the other traditional objectives.

7.2 ISSO—Tools for Program Adaptation

Static or dynamic program tuning for performance improvement has been addressed by other researchers in real-time or parallel systems, most notably by past work regarding multicomputer program construction [6, 25, 72] and by recent work regarding the dynamic reconfiguration of real-time systems [2, 37]. Furthermore, research concerning reusable software [27] and software transformation [60] are related to this topic.

Our research differs from such work in that we are assuming programmers are explicitly involved in the process of program tuning [69] due to the complexity of the tuning actions performed. Specifically, the support for program adaptations offered by the ISSOS system consists of (1) language primitives with which adaptations may be stated, (2) a descriptive model and database suitable for representation of the program being adapted and the adaptations being performed, and (3) compile-time and run-time support for making decisions regarding adaptations and performing them [89].

Since static and dynamic adaptations of operating software can involve changes to code, data, and resource allocation, the descriptive model and database must contain (1) information about software structure and semantics typically maintained at compile-time, and (2) information regarding software execution typically available at run-time. This requires the integration of programming environment and operating system to a degree not present in current systems. A novel software tool in the ISSOS system called an *adaptation controller* provides such integration by defining and controlling the interface between operating system and programming environment. The resulting system for program adaptation consists of three parts:

—A *programming environment* supporting the design and implementation of adaptable operating software; this environment has knowledge of the parallel structure and detailed syntax and semantics of the operating software.
—An *adaptation controller* able to cooperate with both the programming environment and the operating system to effect static or dynamic software adaptations.
—*Run-time and monitoring system* extensions of the operating system providing the basic information and mechanisms for static and dynamic software adaptation.

As with STILE, multiple prototypes of the ISSOS system have been constructed. The initial prototype which predated GEM, was constructed using the experimental syntax-directed editor generator systems developed as part of the Gandalf project at Carnegie-Mellon University [56]. Its run-time system was built for a network of UNIX™ machines [68]. A more sophisticated system is now being constructed that will interface to both a redesigned network run-time system and an extended version of GEM.

A subset of the new ISSOS system for use with GEM is now in the final stages of construction [89]. Descriptions of the program components being adapted in this subset are maintained in a database by the programming environment and used by the adaptation controller. This database contains information about the software and hardware *entities* (e.g., processes, microprocesses, mailboxes, processors, memories and communication links) and about the *relationships* between them (e.g., access of processes to mailboxes, mapping of processes to processors, mapping of mailboxes to memories). It has been implemented in LISP and runs on a SUN™ workstation attached to the GEM machine, the multiprocessor.

Parallel, adaptable programs for GEM currently are written on the workstation. Their construction involves the creation of modules of sequential code for the process and mailbox entities and their placement into a library. In addition entities and relationships describing the desired program configuration are entered into the database, along with information about the current hardware configuration.

The adaptation controller, also running on the workstation, takes this information and generates commands to link the appropriate program components into a set of loadable modules. The multiprocessor is then loaded and the program is run.

While the program is running, data concerning its execution such as processor loads, process execution-times, mailbox access frequencies are collected by the

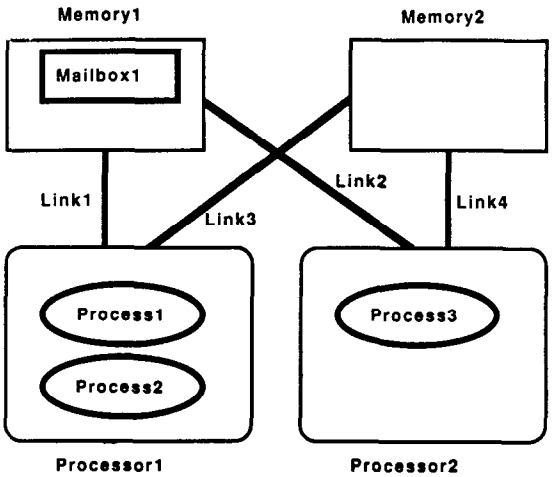


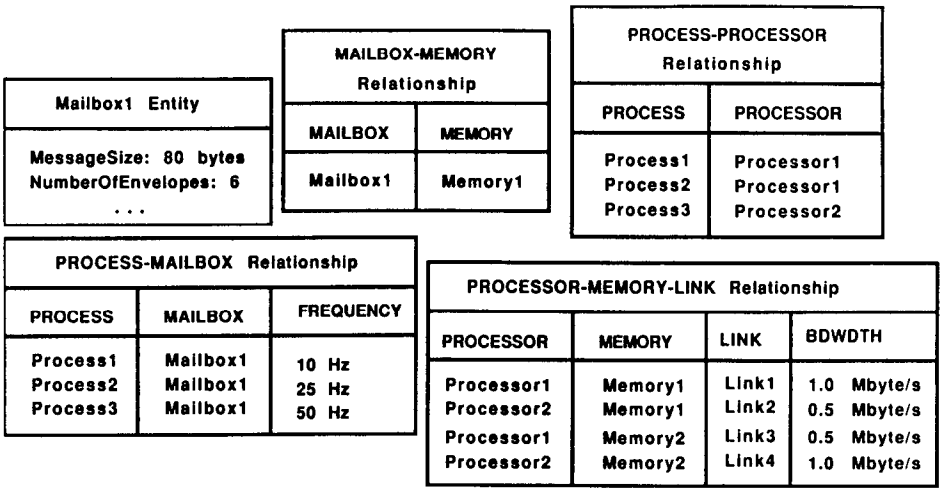
Fig. 9. A simple software/hardware configuration.

monitoring system embedded in GEM and are stored as entities/relationships in the database. The adaptation controller first makes adaptation decisions based on analyses of monitoring data, for example, recognition of unbalanced processor loads and then *enacts* those decisions. Program adaptations may include changes in the mapping of software components to hardware components and alterations of the software structure itself (see the example below).

Adaptation *enactment* involves making changes to the entities and relationships in the database. These changes activate *action routines* [56] attached to the entities and relationships that perform the actual modifications to the software components and interconnections described by the entities and relationships.

The adaptation controller has compile-time and run-time access to the entity/relationship database and may execute complex queries on the data, producing specific *views* [71] of the concurrent program. These views contain exactly the information required for performing particular types of adaptations. For example, Figure 9 shows a small software/hardware system. An adaptation of Mailbox1 that decides whether to move it from Memory1 to Memory2 may take the information shown in Figure 10 and compile the views shown in Figure 11. The first view shows the current loads on the links (with Mailbox1 and Memory1). The second view shows hypothetical loads assuming Mailbox1 is moved to Memory2. From these views it can be seen that total data access time for Mailbox1 would be lower if it were moved to Memory2.

Entities may be classified by *type*. For example, BubbleSortType and ShellSortType may be subtypes of SortType. There may be multiple *instances* of a type in the actual application program; ShellSort1 and ShellSort2 may be two processes of type ShellSortType. An entity may describe groups of components; Figure 12 shows diagrams of an entity representing a process and of an entity representing a triple-modular-redundant software subsystem providing the same functionality. The adaptation controller may choose to replace a single copy of the process with the TMR subsystem if higher reliability is needed.



Mailbox1 Entity
MessageSize: 80 bytes NumberOfEnvelopes: 6 . . .

MAILBOX-MEMORY Relationship	
MAILBOX	MEMORY
Mailbox1	Memory1

PROCESS-PROCESSOR Relationship	
PROCESS	PROCESSOR
Process1 Process2 Process3	Processor1 Processor1 Processor2

PROCESS-MAILBOX Relationship		
PROCESS	MAILBOX	FREQUENCY
Process1 Process2 Process3	Mailbox1 Mailbox1 Mailbox1	10 Hz 25 Hz 50 Hz

PROCESSOR-MEMORY-LINK Relationship			
PROCESSOR	MEMORY	LINK	BDWDTH
Processor1	Memory1	Link1	1.0 Mbyte/s
Processor2	Memory1	Link2	0.5 Mbyte/s
Processor1	Memory2	Link3	0.5 Mbyte/s
Processor2	Memory2	Link4	1.0 Mbyte/s

Fig. 10. Sample entity and relationships.

Link Loads (with Mailbox1 on Memory1)		
LINK	LOAD	BDWDTH
Link1	28000 byte/s	1.0 Mbyte/s
Link2	40000 byte/s	0.5 Mbyte/s
Link3	0 byte/s	0.5 Mbyte/s
Link4	0 byte/s	1.0 Mbyte/s

Link Loads (with Mailbox1 on Memory2)		
LINK	LOAD	BDWDTH
Link1	0 byte/s	1.0 Mbyte/s
Link2	0 byte/s	0.5 Mbyte/s
Link3	28000 byte/s	0.5 Mbyte/s
Link4	40000 byte/s	1.0 Mbyte/s

Fig. 11. Multiple views of mailbox1.

At the current time, the prototype system described here performs adaptations directed toward two goals: fault-tolerance and real-time response, that is, meeting deadlines. Application programmers may provide multiple versions of software subsystems, as in the TMR example above, as well as reliability and deadline constraints. Application-independent heuristics provided by the system choose the appropriate versions for inclusion into the software configuration. The system also provides heuristics to map the software components to the appropriate hardware components. Provision has not yet been made for allowing application programmers to specify application-dependent adaptations. That problem is being studied with the extended system now being constructed [68].

As with the graphical tools, the ISSOS system is built on top of the basic GEM primitives. Specifically, its monitoring and run-time system employs GEM's basic communication primitives and its code implementing dynamic program adaptations resides in GEM processes. However, a few low-level monitoring and load-balancing primitives had to be added to the GEM system. In addition GEM has been extended to allow the direct, run-time representation of higher-level entities in terms of objects [40, 67].

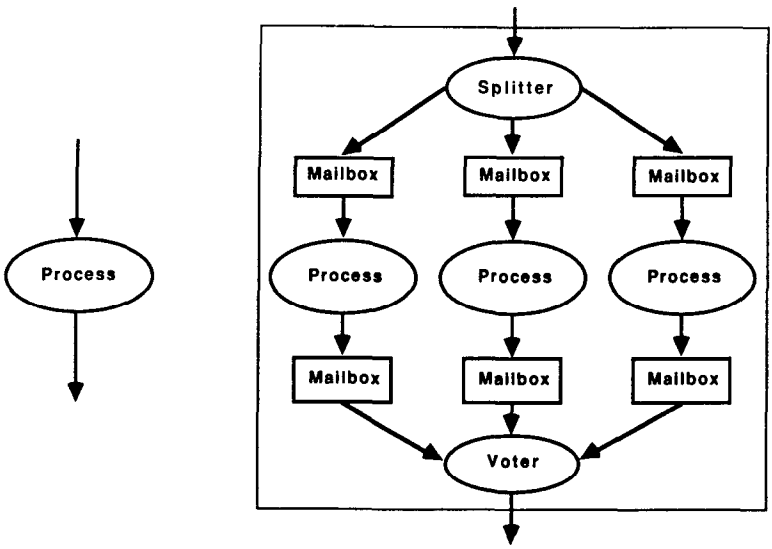


Fig. 12. Single process and corresponding TMR subsystem.

8. CONCLUSIONS, STATUS OF RESEARCH, AND FUTURE RESEARCH

The GEM system described in this paper has been implemented and is being used with the operating software of the ASV vehicle. Prototypes of both programming environments described in Section 7 have been implemented. As with most operating systems in active use, additions to GEM are being constructed, such as enhanced monitoring facilities, new device drivers, and most important, more direct support for the object model used in the ISSOS system [67, 68].

Interesting topics in operating systems research regarding GEM include:

—Determination of the different representations of objects useful in real-time applications, including representations that use single versus multiple processes or lightweight processes, and the semantics of object invocation required for such applications [67].
—The appropriate use of the inhomogeneous communication links existing in embedded computer systems, as exemplified by the multibus and parallel bus connections coexisting in the ASV hardware. As shown by the high overheads of the intercluster operations (see Section 6.2), it is doubtful that the network server paradigm commonly used in inhomogeneous computer networks will result in acceptable real-time performance.
—The support of inhomogeneous computer systems within real-time systems, as exemplified by the LISP machines now being used for high-level planning tasks in the ALV project [62] and by special vision-processing equipment now being constructed for the ASV.
—The automation of task scheduling, both statically and dynamically. Algorithms improving the results of Stankovic [76] are being developed.

Other research with GEM concerns its use for additional real-time applications, such as hand control, now being investigated jointly with the Department of

Electrical Engineering at The Ohio State University, and its use for nonreal-time applications, such as computer graphics. We are currently implementing a parallel scanline conversion algorithm on GEM's multiprocessor to which a framebuffer has been connected. We intend to investigate the changes in GEM's operating system primitives that might be required for such applications.

Future research concerning STILE and ISSOS mainly concerns the investigation of a common metaphor for graphical and adaptable programs. Toward that end we are experimenting with the graphical display of alternative or multiple program views.

Current experimentation with GEM and with operating software is being conducted in the Parallel Real-Time Systems Laboratory (PARTS) at The Ohio State University. This laboratory consists of several SUN™ workstations connected via an EtherNet network, an eight- and a six-node multiprocessor running GEM, and an Intel iPSC hypercube [73]. The real-time equipment in this laboratory now includes a robot arm, a conveyor, and a vision system. To investigate the use of parallelism in Artificial Intelligence applications, we are now adding a BBN Butterfly shared memory multiprocessor to the laboratory.

APPENDIX

Process State Transitions

The GEM process scheduling mechanism is implemented as an event-driven finite state machine. Each process is in one of several states (shown below). Events cause a process state to change. Since processes may change state only at specific scheduling points (See Section 4.1), the "current state" mentioned below is actually an idealized state. The actual states of all processes are updated to their idealized states at scheduling points.

A process may be in any one of the following states:

1. **Running**
2. **Running** And WakeUp Pending
3. **Ready**
4. **Ready** And WakeUp Pending
5. **Asleep-T**
6. **Asleep-T** And WakeUp Pending
7. **Asleep-W**
8. **Asleep-WorT**
9. **Asleep-WandT**

The "WakeUp Pending" states correspond to the "remembered" WakeUp described in Section 4.1.

The events recognized by GEM are

1. WakeTime Reached—The time a sleeping process is waiting for has arrived.
2. WakeUp(ProcessId)
3. GoToSleep(**Asleep-T**, WakeTime)
4. GoToSleep(**Asleep-W**, —)
5. GoToSleep(**Asleep-WorT**, WakeTime)

Table V. Process State Transitions

EVENT \ CURRENT STATE	Running&WakeUpPending (RuP)	Ready&WakeUpPending (ReP)	Running (Ru)	Ready (Re)	Asleep-T (T)	Asleep-W (W)	Asleep-WorT (WoT)	Asleep-WandT (WaT)	Asleep-T&WakeUpPending (TP)
WakeTime Reached					Re		Re	W	ReP
WakeUp(...)	RuP	ReP	RuP	ReP	TP	Re	Re	T	TP
GTS(...,Asleep-T,...)	TP		T						
GTS(...,Asleep-W,...)	Ru		W						
GTS(...,Asleep-WorT,...)	Ru		WoT						
GTS(...,Asleep-WandT,...)	T		WaT						
PTS(...,Asleep-T,...)	TP	TP	T	T	T	WaT	T	WaT	TP
PTS(...,Asleep-W,...)	Ru	Re	W	W	WaT	W	W	WaT	T
PTS(...,Asleep-WorT,...)	Ru	Re	WoT	WoT	T	W	WoT	WaT	TP
PTS(...,Asleep-WandT,...)	T	T	WaT	WaT	WaT	WaT	WaT	WaT	T
Context Switch From CPU	ReP		Re						
Context Switch To CPU		RuP		Ru					

NOTE: PTS = PutToSleep, GTS = GoToSleep
Blanks represent transitions that cannot occur.

6. GoToSleep(**Asleep-W and T**, WakeTime)
7. PutToSleep(ProcessId, Mode, **AsleepT**, WakeTime)
8. PutToSleep(ProcessId, Mode, **Asleep-W**, —)
9. PutToSleep(ProcessId, Mode, **Asleep-WorT**, WakeTime)
10. PutToSleep(ProcessId, Mode, **Asleep-WandT**, WakeTime)
11. Swap Process Out Of CPU—A running process is made ready (e.g., when a time-slice is up).
12. Swap Process Into CPU—A ready process is made running (when some other process has been swapped out).

The actions taken when servicing an event are shown in Table V. A new state is determined from the current state of the process and the event. In cases where there are two WakeTimes (e.g., if the current state is "**Asleep-T** until T1" and the event is "PutToSleep(. . . , **Asleep-WandT**, T2)"), the resulting WakeTime is the maximum of T1 and T2. Immediate and Deferred PutToSleeps are

accumulated separately, as described in Section 4.1. When a process executes a GoToSleep, the single accumulated Deferred PutToSleep is executed.

ACKNOWLEDGMENTS

When GEM was being constructed, the ASV robotics project was led by Professor Robert McGhee in the Department of Electrical Engineering at The Ohio State University. The robot's computer hardware and application software were and are being implemented by Dennis Pugh, Eric Ribble, and Mark Patterson of Adaptive Machine Technologies Inc., to whom we are thankful for many informative discussions regarding robotics software. D. Orin in the Department of Electrical Engineering and his graduate students have been helpful to us regarding the development of our notions concerning operating software and its use in robotics. Many graduate student members of the ISSOS research group in Computer and Information Science helped in the design and implementation of the GEM operating system, including Prabha Gopinath, Ben Blake, Win Bo, Sharad Rastogi, Sanjiv Taneja, Jim Matthews, R. Subramanian, and V. Jawantheeswaran.

REFERENCES

1. AHMAD, S. Real-time multiprocessor based robot control. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, pp. 858–863.
2. BARHEN, J. Hypercube concurrent computation and virtual time architecture for robotic applications. In *Workshop on Special Computer Architectures for Robot Control, Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, Extended abstract.
3. BARHEN, J. Robot inverse dynamics on a concurrent computation ensemble. In *Proceedings of the ASME Computers in Engineering* (Boston, Mass., Aug. 1985). ASME, pp. 415–429.
4. BHATT, D. Bus performance experiments on a real-time distributed computer system. In *Proceedings of the Real-Time Systems Symposium* (Arlington, Va., Dec. 1983). IEEE, New York, pp. 41–50.
5. BIRREL, A. D., AND NELSON. B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.
6. BLOOM, T. Dynamic module replacement in a distributed programming system. PhD thesis, MIT/LCS/TR-303. Laboratory for Computer Science, Massachusetts Institute of Technology. Mar. 1983.
7. BROOKS, E. D. A multitasking kernel for the C and Fortran programming languages. Tech. Rep. UCID-20167, Lawrence Livermore National Laboratory, Sept. 1984.
8. BROWN, M. E., AND WEIDE, B. W. Automating process-to-processor mapping under real-time constraints. In *Proceedings of the 1984 Real Time Systems Symposium* (Austin, Tex., Dec. 1984). IEEE, New York, 1984, pp. 145–150.
9. CARLOW, G. D. Architecture of the space shuttle primary avionics software system. *Commun. ACM 27*, 9 (Sept. 1984), 926–936.
10. CHERITON, D. R., MALCOLM, M. A., MELEN, L. S., AND SAGER, G. R. Thoth, a portable real-time operating system. *Commun. ACM 22*, 2 (Feb. 1979), 105–115.
11. CHERITON, D. R., AND ZWAENEPOL, W. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th Symposium on Operating System Principles* (Bretton Woods, N. H., Oct. 1983). ACM SIGOPS, 1983, pp. 128–139.
12. CHOU, T. C. K., AND ABRAHAM, J. A. Load balancing in distributed systems. *IEEE Trans. Softw. Eng. SE-8*, 4 (July 1982), 401–412.
13. CHU, W. W., HOLLOWAY, L. J., LAN, M.-T., AND EFE, K. Task allocation in distributed data processing. *Computer Magazine 13*, 11 (Nov. 1980), 57–70.

14. CONCEPCION, A. I., AND ZEIGLER, B. P. Distributed simulation of distributed system models. Tech. Rep. CSC-82-016, Wayne State Univ., Dept. of Computer Science, Detroit, Mich., Dec. 1982.
15. COX, G., CORWIN, W. M., LAI, K. K., AND POLLACK, F. J. A unified model and implementation for interprocess communication in a multiprocessor environment. In *Proceedings of the 8th Symposium on Operating System Principles* (Asilomar, Calif., Dec. 1981). ACM, New York, 1981, pp. 44–53.
16. CROW, F. C. A more flexible image generation system. *Computer Graphics 16*, 3 (July 1982), 9–18.
17. CROWTHER, W. ET AL. The butterfly parallel processor. *IEEE Computer Architecture Technical Committee Newsletter* (Dec. 1985), 18–45.
18. DONNER, M. D. The design of Owl: a language for walking. In *Proceedings of the SIGPLAN Symposium on Programming Language Issues in Software Systems* (June 1983). ACM, New York, 1983, pp. 158–165.
19. DONNER, M. D. Control of walking: local control and real-time systems. PhD thesis, CMU-CS-84-121, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., May 1984.
20. DRUMMOND, M., MCMULLEN, C., AND VASUDEVAN, R. Packrat—a real time kernel for distributed systems. In *Proceedings of the 2nd Real-Time Systems Symposium* (Dec. 1982). IEEE, New York, 1982, pp. 151–154.
21. DUPOURQUE, V., GUIOT, H., AND ISHACIAN, O. Towards multiprocessor and multi-robot controllers. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, 1986, pp. 864–870.
22. FAVERJON, B. Object-level programming of industrial robots. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, 1986, pp. 1406–1411.
23. FOX, G. C., AND KOLAWA, A. Implementation of the high performance crystalline operating system in Intel IPSC hypercube. Tech. Rep. Hm247, Caltech Concurrent Computational Program and Physics Dept., Jan. 1986.
24. GAGLIANELLO, R. D., AND KATSEFF, H. P. A distributed computing environment for robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, 1986, pp. 1890–1896.
25. GEHRINGER, E. F., JONES, A. K., AND SEGALL, Z. Z. The Cm* testbed. *IEEE Computer Magazine 15*, 10 (Oct. 1982), 40–53.
26. GIFFORD, D., AND SPECTOR, A. EDS. The space shuttle primary computer system. *Commun. ACM 27*, 9 (Sept. 1984), 874–900.
27. GOGUEN, J. A. Parameterized programming. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept. 1984), 528–543.
28. GROSS, T., KUNG, H. T., LAM, M., AND WEBB, J. WARP as a machine for low-level vision. In *Proceedings of the IEEE International Conference on Robotics and Automation* (St. Louis, Mo., Mar. 1985). IEEE, New York, 1985, pp. 790–800.
29. HOPCROFT, J. E. The impact of robotics on computer science. *Commun. ACM 29*, 6 (June 1986), 487–498.
30. JEFFERSON, D. R. Synchronization in distributed simulation. In *Proceedings of the ASME Computers in Engineering* (Boston, Mass., Aug. 1985). ASME, 1985, pp. 407–413.
31. JONES, A. K., CHANSLER, R. J., DURHAM, I., MOHAN, J., SCHWAN, K., AND VEGDAHL, S. StarOS, a multiprocessor operating system. In *Proceedings of the 7th Symposium on Operating System Principles* (Asilomar, Calif., Dec. 10–12, 1979). ACM, 1979, pp. 117–127.
32. JONES, A. K., AND SCHWARZ, P. Experience using multiprocessor systems: a status report. *ACM Comput. Surv. 12*, 2 (June 1980), 121–166.
33. KERRIDGE, J. M., AND SIMPSON, D. Three solutions for a robot arm controller using Pascal-Plus, Occam, and Edison. *Softw. Pract. Exper. 14*, 1 (Jan. 1984), 3–15.
34. KLEIN, C. A., AND WAHAWISAN, W. Use of a multiprocessor for control of a robotic system. *Int. J. Robot. Res. 1*, 2 (Summer 1982), 45–59.
35. KLOTZ, T. H., PHILLIPS, R. G., SPRATTLING, R. L., AND SUTHERLAND, H. A. Real-time performance evaluation of local area networks used in automated manufacturing systems. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, 1986, pp. 1723–1730.

36. KOREIN, J. U., MAIER, G. E., TAYLOR, R. H., AND DURFEE, L. F. A configurable system for automation programming and control. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, 1986, pp. 1871–1877.
37. KRAMER, J., AND MAGEE, J. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr. 1985) 424–436.
38. KRIEGMAN, D. J. ET AL. Computational architecture for the Utah/MIT hand. In *Proceedings of the IEEE International Conference on Robotics and Automation* (St. Louis, Mo., Mar. 1985). IEEE, New York, 1985, pp. 918–924.
39. LATHROP, R. H. Parallelism in manipulator dynamics. *Int. J. Robot. Res. 4*, 2 (Summer 1985), 80–102.
40. LAZOWSKA, E. D., LEVY, H. M., ALMES, G. T., FISHER, M. J., FOWLER, R. J., AND VESTAL, S. C. The architecture of the Eden system. In *Proceedings of the 8th Symposium on Operating System Principles* (Dec. 1981). ACM, 1981, pp. 148–159.
41. LEE, I., AND GEHLOT, V. Language constructs for distributed real-time programming. In *Proceedings of the 6th Real-Time Systems Symposium* (San Diego, Calif., Dec. 1985). IEEE, New York, 1985, pp. 57–66.
42. LEVIN, R., COHEN, E., CORWIN, W., POLLACK, F., AND WULF, W. Policy/mechanism separation in Hydra. In *Proceedings of the 5th Symposium on Operating System Principles.* (Austin, Tex., Nov. 1975). ACM, New York, 1975.
43. LILLY, K. W., AND ORIN, D. E. Multiprocessor implementation of dynamic control schemes for robot manipulators. In *Proceedings of the ASME Computers in Engineering Conference* (Chicago, Ill, July, 1986).
44. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983) 381–404.
45. LIU, C.-H., AND CHEN, Y.-M. Multi-microprocessor-based Cartesian space control techniques for a mechanical manipulator. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, 1986, pp. 823–827.
46. LO, V. M. Task assignment to minimize completion time. In *Proceedings of the 5th International Conference on Distributed Computing Systems* (Denver, Colo., May 1985). IEEE, New York, 1985, pp. 329–336.
47. MACQUEEN, D. B. Models for distributed computing. Tech. Rep. 351, IRIA, Apr. 1979.
48. MCCAIN, H. G. A hierarchically controlled, sensory interactive robot in the automated manufacturing research facility. In *Proceedings of the IEEE International Conference on Robotics and Automation* (St. Louis, Mo., Mar. 1986). IEEE, New York, 1986, pp. 931–940.
49. MCDONALD, W. C., AND SMITH, R. W. A flexible distributed testbed for real-time applications. *IEEE Computer Magazine 15*, 10 (Oct. 1982), 25–39.
50. MCGHEE, R. B., AND ISWANDHI, G. I. Adaptive locomotion of a multilegged robot over rough terrain. *IEEE Trans. Syst. Man Cyber. SMC-9*, 4 (Apr. 1979), 176–182.
51. MCGHEE, R. B., ORIN, D. E., PUGH, D. R., AND PATTERSON, M. R. A hierarchically-structured system for computer control of a hexapod walking machine. In *Proceedings of the 5th IFTOMM Symposium on Robots and Manipulator Systems*, (Udine, Italy, June 1984). IFTOMM, 1984.
52. MCGHEE, R. B. Vehicular legged locomotion. In *Advances in Automation and Robotics.* Jai Press Ltd., New York, 1985, pp. 259–284.
53. MOK, A. K.-L. The decomposition of real-time system requirements into process models. In *Proceedings of the 5th Real-Time Systems Symposium* (Austin, Tex., Dec. 1984), IEEE, New York, pp. 125–134.
54. MONTEMERLO, M. D. NASA's automation and robotics technology development program. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif. Apr. 1986). IEEE, New York, 1986, pp. 977–986.
55. NARASIMHAN, S., SIEGEL, D., HOLLERBACH, J. M., BIGGERS, K., AND GERPHEIDE, G. Implementation of control methodologies on the computational architecture of the Utah/MIT hand. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif. Apr. 1986). IEEE, New York, 1986, pp. 1884–1889.
56. NOTKIN, D. The GANDALF project. *J. Syst. Softw. 5*, 2 (May 1985), 91–106.
57. ORIN, D. E., AND SCHRADER, W. W. Efficient computation of the jacobian for robot manipulators. *Int. J. Robot. Res. 4*, 1 (Spring 1985), 2–15.

58. OUSTERHOUT, J. K., SCELZA, D. A., AND SINDHU, P. Medusa: an experiment in distributed operating system structure. *Commun. ACM 23*, 2 (Feb. 1980), 92–104.
59. OZGUNER, F., AND KAO, M. L. A reconfigurable multiprocessor architecture for reliable control of robotic systems. In *Proceedings of the IEEE International Conference on Robotics and Automation* (St. Louis, Mo., Mar. 1985). IEEE, New York, 1985, pp. 802–806.
60. PARTSCH, H., AND STEINBRUEGGEN, R. Program transformation systems. *ACM Comput. Surv. 15*, 3 (Sept. 1983), 199–236.
61. PAUL, R. P., ZHANG, H. Design of a robot force motion server. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, 1986, pp. 1878–1883.
62. PAYTON, D. W. An architecture for reflexive autonomous vehicle control. In *Proceedings of the IEEE International Conference on Robotics and Automation* (San Francisco, Calif., Apr. 1986). IEEE, New York, 1986, pp. 1838–1845.
63. PONG, M.-C., AND NG, N. PIGS—a system for programming with interactive graphical support. *Softw. Pract. Exper. 13*, 9 (Sept. 1983), 847–856.
64. PONG, M.-C. A graphical language for concurrent programming. In *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages* (Dallas, Tex., June 1986) IEEE, New York, 1986, pp. 25–33.
65. PREPARATA, F. P., AND YEH, R. T. *Introduction to Discrete Structures*. Addison-Wesley, Reading, Mass., 1973.
66. RAIBERT, M. H. Legged robots. *Commun. ACM 29*, 6 (June, 1986), 499–514.
67. SCHWAN, K., GOPINATH, P., AND BO, W. CHAOS—Kernel support for objects in the real-time domain. To appear in *IEEE Trans. Comput.* (July 1987).
68. SCHWAN, K., RAMNATH, R., VASUDEVAN, S., AND OGLE, I. A system for parallel programming. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, Calif., Mar. 1987), IEEE, New York, 1987, pp. 270–282.
69. SCHWAN, K., AND JONES, A. K. Specifying resource allocation for the Cm* multiprocessor. *IEEE Softw. 3*, 3 (May 1984), 60–70.
70. SCHWAN, K., AND JONES, A. K. Flexible software development for multiple computer systems. *IEEE Trans. Softw. Eng. SE-12*, 3 (Mar. 1986), 385–401.
71. SCHWAN, K., AND RAMNATH, R. Adaptable operating software for manufacturing systems and robots: a computer science research agenda. In *Proceedings of the 5th Real-Time Systems Symposium* (Austin, Tex., Dec. 1984). IEEE, New York, 1984, pp. 255–262.
72. SEGALL, Z., AND RUDOLPH, L. PIE: a programming and instrumentation environment for parallel processing. *IEEE Softw. 2*, 6 (Nov. 1985), 22–37.
73. SEITZ, C. L. The cosmic cube. *Commun. ACM 28*, 1 (Jan. 1985), 22–33.
74. SHIN, K. G., AND EPSTEIN, M. E. Communication primitives for a distributed multi-robot system. In *Proceedings of the IEEE International Conference on Robotics and Automation* (St. Louis, Mo., Mar. 1986). IEEE, New York, 1986, pp. 910–917.
75. SOLOMON, M. H., AND FINKEL, R. A. The Roscoe distributed operating system. In *Proceedings of the 7th Symposium on Operating System Principles* (Asilomar, Calif., Dec. 10–12, 1979). ACM, New York, 1979, pp. 108–114.
76. STANKOVIC, J. A., AND SIDHU, I. S. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Proceedings of the 4th International Conference on Distributed Computing Systems* (San Francisco, Calif., May 1984). IEEE, New York, 1984, pp. 49–59.
77. STEUSLOFF, H. U. Advanced real-time languages for distributed industrial process control. *IEEE Comput. Mag. 17*, 2 (Feb. 1984), 37–47.
78. STOVSKY, M. P., WEIDE, B. W., AND HARMS, D. STILE: a graphical design and development environment. Tech. Rep. OSU-CISRC, Dept. of Computer and Information Science, Ohio State Univ., Columbus, Aug. 1986. Submitted for publication.
79. RAIBERT, M. H., AND SUTHERLAND, I. E. Machines that walk. *Sci. Am. 248* (Jan. 1983), 44–53.
80. TANEJA, S., AND WEIDE, B. W. Graphical description and run-time environments for real-time software. In *Proceedings of the 14th Annual Computer Science Conference* (Cincinnati, Oh., Feb. 1986). ACM, New York, 1986, pp. 205–211.
81. WALTER, C. J., KIECKHAFER, R. M., AND FINN, A. M. MAFT: a multicomputer architecture for fault-tolerance in real-time control systems. In *Proceedings of the 6th Real-Time Systems Symposium* (San Diego, Calif., Dec. 1985). IEEE, New York, 1985, pp. 133–140.

82. WAMPLER, C. Multiprocessor control of a telemanipulator with optical proximity sensors. *Int. J. Robot. Res. 3*, 1 (Spring 1984), 52–61.
83. WEIDE, B. W. Design and specification of abstract data types using OWL. Tech. Rep. OSU-CISRC-TR-86-1, Dept. of Computer and Information Science, Ohio State Univ., Columbus, Jan. 1986.
84. WEIDE, B. W., BROWN, M. E., ALEGRIA, J. A. S., AND MEYER, G. R. A graphical interconnection language and its application to concurrent and real-time programming. In *Proceedings of the 20th Annual Allerton Conference on Communication Control, and Computing* (Univ. of Illinois, Oct. 1982). IEEE, New York, 1982, pp. 567–576.
85. WENSLEY, J. H., LAMPORT, L., GOLDBERG, J., GREEN, M. W., LEVITT, K. N., MELLIARD-SMITH, P. M., SHOSTAK, R. E., AND WEINSTOCK, C. B. SIFT: design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE 66*, 10 (Oct. 1978), 1240–1268.
86. WULF, W. A., LEVIN, R., AND HARBISON, S. R. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill Advanced Computer Science Series, McGraw-Hill, New York, 1981.
87. ZEIGLER, B. P. *Theory of Modelling and Simulation.* Wiley, New York, 1976.
88. ZHENG, Y. F., AND CHEN, B. R. A multiprocessor for dynamic control of multilink systems. In *Proceedings of the IEEE International Conference on Robotics and Automation* (St. Louis, Mo., Mar. 1985). IEEE, New York, 1985, pp. 295–300.

Received August 1985; revised July 1986; accepted January 1987